Algorithms for Graphs of Small Treewidth

Algoritmen voor grafen met kleine boombreedte

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. J.A. van Ginkel,
ingevolge het besluit van het College van Decanen
in het openbaar te verdedigen
op woensdag 19 maart 1997 des middags te 2:30 uur

door

Babette Lucie Elisabeth de Fluiter

geboren op 6 mei 1970
te Leende

Promotor: Prof. Dr. J. van Leeuwen
Co-Promotor: Dr. H.L. Bodlaender

Faculteit Wiskunde & Informatica

ISBN 90-393-1528-0

These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO). They have been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

Chapter 1

Introduction

This thesis is concerned with the design of efficient sequential and parallel algorithms for problems on graphs of bounded treewidth. Many real-life problems can be modeled as optimization or decision problems on graphs. Consider for instance the problem in which a courier has to deliver a number of packages at different addresses, and the courier's company wants him to follow the shortest route visiting all addresses, starting and ending at the company's address. This problem is known as the *traveling salesman problem*. The input can be modeled as a weighted graph in which the vertices represent the addresses that have to be visited, including the company's address, and an edge between two vertices represents the road between the corresponding addresses. Each edge has a weight that corresponds to the length of the road between the corresponding addresses. The problem is then to find a cycle in the graph which contains all vertices and has minimum weight.

Unfortunately, many graph problems that model real-life problems are hard in the sense that there are (probably) no efficient algorithms which solve these problems. More formally, these problems are NP-hard. The traveling salesman problem is such a problem. A way of overcoming this disadvantage is to discover a special structure in the graphs modeling the real-life problem which may help in finding a more efficient algorithm for the problem. For instance, the input graphs may have a special structure that assures that the problem at hand is easy to solve. Another possibility is that the problem can be decomposed into subproblems, and that the structure of the input graphs assures that some of these subproblems are easy to solve. This might help in finding a more efficient algorithm that computes an optimal solution for the complete problem, or in finding an efficient algorithm that computes a good approximation of the optimal solution.

One suitable structure is the tree-structure: it appears that many graph problems that are hard in general, are efficiently solvable on trees. As an example, consider the maximum independent set problem, in which we search for a subset I of the vertices of G for which there is no edge between any two vertices of I, and the cardinality of I is as large as possible. This problem is NP-hard, but if the input graph is a tree, then we can easily solve the problem to optimality as follows. Let T be a rooted tree with root r. For each node v of T, let T_v denote the subtree of T rooted at v. For each node v, we compute integers n_v and m_v which denote the size of a maximum independent set of T_v that contains v, and the size of a maximum independent set of T_v that does not contain v, respectively. It follows that the size of a maximum independent set of T is the maximum of n_r and m_r. A particular instance is

shown in part I of Figure 1.1: each node v of the tree is labeled with the pair n_v, m_v. The size of a maximum independent set in T is ten.

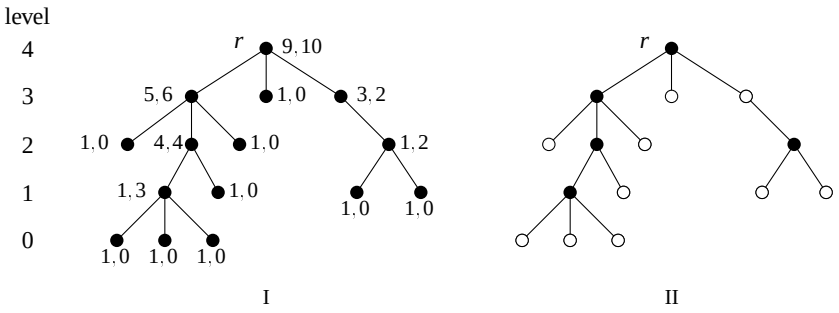


Figure 1.1. A rooted tree in which each node v is labeled with the pair n_v, m_v (part I), and a maximum independent set of the tree, denoted by the white nodes (part II).

For each node v of T, we can compute n_v and m_v from the values of n_u and m_u of all children u of v. This implies that we can perform dynamic programming on T to find the size of a maximum independent set of T: first compute the numbers of the nodes that are at the lowest level of the tree (these nodes have no children). Next rise one level in the tree, and compute the numbers for the nodes on that level, by using the numbers for the nodes one level down. This step is repeated until the numbers n_r, m_r of the root are computed. We can also *find* a maximum independent set of T, by using the computed numbers. Part II of Figure 1.1 shows a maximum independent set of the tree in part I (the nodes in the independent set are white). Sequentially, the above algorithm can be made to run in $O(n)$ time, where n denotes the number of nodes of the tree. In a parallel algorithm, we can perform the computations of different nodes on the same level in parallel, which gives a faster algorithm.

A similar dynamic programming approach as described above can be applied for many problems if the input graph is a tree. For most practical cases however, the class of trees is too limited. Therefore, we consider extensions of the class of trees which are more useful in practice, namely the classes of graphs of *treewidth* at most k and *pathwidth* at most k, for any positive integer k. Intuitively, the treewidth of a graph measures the resemblance of the graph to a tree: a graph has treewidth at most k if one can associate a tree T with G in which each node represents a subgraph of G with at most $k+1$ vertices, such that all vertices and edges of G are represented in at least one of the nodes of T, and for each vertex v of G, the nodes of T in which v is represented form a subtree of T. Such a tree associated with a graph is called a *tree decomposition* of the graph, of width k. As an example, consider the graph G depicted in part I of Figure 1.2. Part III of this figure gives a tree decomposition TD of G. Part II shows the correspondence between the nodes of TD and some subgraphs of G.

The width of a tree decomposition is the maximum number of vertices occuring in any node minus one (the 'minus one' has been introduced to obtain the fact that the class of

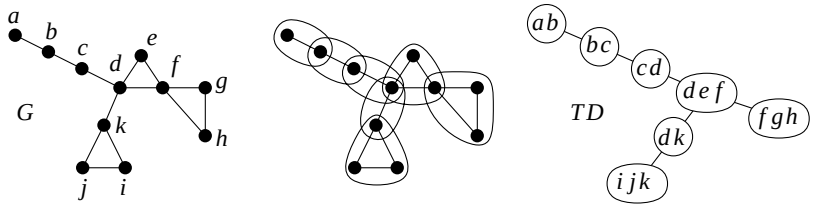


Figure 1.2. A graph G (part I), a tree decomposition TD of width two of G (part III), and the correspondence between the nodes of TD and subgraphs of G (part II).

connected graphs of treewidth one is exactly the class of trees). The tree decomposition in Figure 1.2 for instance has treewidth two. A *path decomposition* of a graph is a tree decomposition with the extra restriction that the tree is a path. A graph has pathwidth at most k if there is a path decomposition of the graph of width at most k.

Many (hard) problems can be solved efficiently on graphs of small treewidth, using the tree-like structure of the graphs. For instance, a large class of problems can be solved efficiently by applying dynamic programming on a tree decomposition of small width of the graph, in a way similar to the dynamic programming for finding a maximum independent set in a tree (this algorithm is described in Section 2.2.3). These algorithms usually work on rooted, binary tree decompositions of small width with $O(n)$ nodes, and for parallel algorithms, we additionally require that the tree decomposition has height $O(\log n)$. Examples of problems that can be solved efficiently on graphs of small treewidth by using the dynamic programming approach are the maximum independent set problem and the traveling salesman problem: both problems can be solved in $O(n)$ time sequentially and in $O(\log n)$ time in parallel with $O(n/\log n)$ processors (the algorithms are exponential in the treewidth of the graph). To solve problems this way, it is necessary to find a tree decomposition of small width of the given graph first. Fortunately, for each positive integer k, there is a linear time algorithm which, given a graph, finds a tree decomposition of width at most k of the graph, if one exists (this algorithm is again exponential in k) [Bodlaender, 1996a]. In parallel, the problem can be solved in $O(\log^2 n)$ time with $O(n/\log^2 n)$ processors, and the result is a binary tree decomposition with height $O(\log n)$ [Bodlaender and Hagerup, 1995].

It appears that many graph problems have practical instances in which the input graphs have small treewidth. For example, it has been shown that graphs modeling special types of expert systems have small treewidth, which helps in solving statistical problems for reasoning with uncertainty in expert systems [Lauritzen and Spiegelhalter, 1988; van der Gaag, 1990]. Also, in natural language processing, it has been shown that dependency graphs encoding the major syntactic relations among words have pathwidth at most six [Kornai and Tuza, 1992]. Thorup [1995] has shown that control-flow graphs of structured programs have treewidth at most six, which helps in finding good register allocations. Cook and Seymour [1993] have

shown that graphs modeling certain telephone networks have bounded treewidth. They have used this for solving a variant of the traveling salesman problem in these graphs (they actually use branchwidth, but this concept is closely related to treewidth).

Also, many (practical) graph problems require that the treewidth or pathwidth of the input graph is small. Examples of such problems are the gate matrix layout problem that occurs in VLSI design [Fellows and Langston, 1992; Deo, Krishnamoorty, and Langston, 1987; Möhring, 1990; Ramachandramurthi, 1994], Cholesky factorization for sparse matrices [Bodlaender, Gilbert, Hafsteinsson, and Kloks, 1995], the perfect phylogeny problem that occurs in evolutionary theory [Agarwala and Fernández-Baca, 1993; Bodlaender, Fellows, and Warnow, 1992; Bodlaender and Kloks, 1993; Kannan and Warnow, 1990; Kannan and Warnow, 1992; McMorris, Warnow, and Wimer, 1994], the DNA physical mapping problem occuring in molecular biology [Golumbic, Kaplan, and Shamir, 1994; Fellows, Hallett, and Wareham, 1993] (see also Chapter 4 of this thesis), and interval routing problems in networks [Bodlaender, Tan, Thilikos, and van Leeuwen, 1995].

In all the problems described above, the fact that the treewidth of the input graph is or should be small helps to find more efficient algorithms to solve them. It does not ensure the existence of efficient algorithms, see e.g. Section 4.2.2.

Unfortunately, many algorithms solving problems on graphs of small treewidth are only efficient in theory: the running time of the algorithms is usually exponential in the treewidth of the graph. This means that if the input graph is only of moderate size, and the bound on the treewidth is six or more, then in the running time of the algorithm, the factor that is exponential in the treewidth is likely to overtake the factor that is polynomial in the size of the graph. For example, consider the problem of finding a tree or path decomposition of width at most k of a given graph, if one exists (k constant). The sequential algorithms of Bodlaender [1996a] solve these problems in $O(n)$ time. These algorithms use an algorithm of Bodlaender and Kloks [1996] which, given a graph G and a tree decomposition of width at most l of G (for any fixed integer l), finds a tree (path) decomposition of width at most k of G, if one exists (for any fixed integer k with $k < l$). This algorithm runs in $O(n)$ time sequentially, but the constants hidden in the O-notation make the algorithm only practical for $k \leq 5$. Also, the algorithm of Bodlaender and Kloks [1996] is rather complicated, and thus not easy to implement. This makes the algorithms of Bodlaender [1996a] for finding a tree or path decomposition of small width of a graph inefficient in practice for $k \geq 6$. Furthermore, for $k < 6$, the algorithm is still hard to implement, and algorithms that are tailor-made for a specific treewidth are probably more efficient in practice. (For treewidth 2, 3, and 4, such algorithms exist [Arnborg and Proskurowski, 1986; Matoušek and Thomas, 1991; Sanders, 1996].) The parallel algorithms of Bodlaender and Hagerup [1995] for finding a tree or path decomposition of small width of a graph use a parallel version of the algorithm of Bodlaender and Kloks [1996], and thus have the same drawback.

The goal of this thesis is to give efficient sequential and parallel algorithms for several problems on graphs of small treewidth or pathwidth. We consider both graph problems which require that the treewidth or pathwidth of the input graph is bounded by some constant, and graph problems which are hard on general graphs, but have more efficient solutions on graphs

of small treewidth or pathwidth. The aim is to design algorithms which are not only theoretically efficient, but are also efficient in practical applications.

This thesis is organized as follows. We start with preliminary results in Chapter 2. In this chapter, we introduce the terminology used throughout the thesis, and we give formal definitions of tree and path decompositions and of the treewidth and pathwidth of graphs. Furthermore, we present a number of well-known properties of tree and path decompositions and of graphs of bounded treewidth or pathwidth which will prove useful in the remainder of the thesis. Most of the results presented here include a proof to give the reader a feeling for the concepts of treewidth and pathwidth. We also present an overview of the most powerful algorithmic results on graphs of bounded treewidth and pathwidth. Finally, we discuss some graph notions that are closely related to the notion of treewidth or pathwidth and that are used in the remainder of the thesis.

In Chapter 3, we give a complete characterization of graphs of pathwidth at most two. This characterization is then used for the design of a linear time algorithm that checks whether a given graph has pathwidth at most two, and if so, builds a path decomposition of minimum width of the graph. Although Bodlaender's algorithm [1996a] can be used to solve this problem in $O(n)$ time, our algorithm is probably more efficient and useful in practice, as it does not use the theoretical result of Bodlaender and Kloks [1996] and is much easier to implement. The characterization of graphs of pathwidth two that is presented in Chapter 3 is used for the algorithms presented in Chapter 4.

Chapter 4 is concerned with two decision problems that occur in DNA physical mapping, namely k-INTERVALIZING SANDWICH GRAPHS (k-ISG) and k-UNIT-INTERVALIZING SANDWICH GRAPHS (k-UISG). In both problems, the input consists of k copies of a DNA string that are fragmented, and for each pair of fragments, either it is known that they overlap, or it is known that they do not overlap, or nothing is known about their overlap. In k-UISG, we additionally have the information that all fragments have the same length. There is no explicit information on the order of the fragments in the DNA string, or on the copy from which each fragment originates. The problem is to recover the complete overlap information of the fragments, and with this, the order of the fragments in each copy of the DNA string. The input of the problems is modeled as graph $G = (V,E)$ and an extra set of edges F: the vertices of the graph represent the fragments, and for each two vertices in V, there is an edge between u and v in E if we know that the corresponding fragments overlap, and there is an edge between u and v in F if the corresponding fragments possibly overlap, i.e. are not known not to overlap. The problem is then to find a set E', $E \subseteq E' \subseteq F$, such that the graph (V,E') represents the complete overlap information. Both for k-ISG and k-UISG, there is such a set E' only if the input graph G has pathwidth at most $k \Leftrightarrow 1$.

In Chapter 4 we resolve the complexity of k-ISG for all fixed integers $k \geq 2$: we give a linear time algorithm for 2-ISG, a quadratic algorithm for 3-ISG, and we show that k-ISG is NP-complete if $k \geq 4$. Furthermore, we give an $O(n+m)$ time algorithm for 3-UISG (where m denotes the number of extra edges that is part of the input). This algorithm improves on the $O(n^2)$ algorithm of Kaplan and Shamir [1996]. The algorithms for 3-ISG and 3-UISG heavily rely on the characterization of graphs of pathwidth at most two, as described

in Chapter 3: it consists of a large case analysis based on the structure of the input graph. We present the most instructive part of the algorithms for 3-ISG and 3-UISG in Chapter 4, namely the algorithms for the case that the input graph is biconnected. The remaining part of the algorithm consists of a lot of technical details, mostly based on the same principles as the algorithm for biconnected graphs. This part is not included in the thesis.

In Chapters 5 – 9, we discuss a special type of algorithms, namely *reduction algorithms*. A reduction algorithm is an algorithm which applies a sequence of reductions to the input graph: in each reduction, a small part of the graph is replaced by a smaller part, thus reducing the size of the graph. In a sequential algorithm, all reductions are performed subsequently, but in a parallel reduction algorithm, non-interfering reductions can be performed at the same time. The reduction behavior is described by a set of reduction rules, which is problem specific. It turns out that for many decision and optimization problems, such a set of reduction rules can be constructed, and with this constructed set, the problem can be solved efficiently on graphs of small treewidth, both sequentially and in parallel.

An advantage of reduction algorithms is that they are easy to implement: the difficulty of a problem is hidden in the design of the problem specific set of reduction rules, and not in the reduction algorithm itself. Another advantage of reduction algorithms over other algorithms on graphs of small treewidth is that a reduction algorithm works directly on the input graph, and hence no tree decomposition of small width of the graph is needed. As the running times of the algorithms for finding a tree decomposition of small width of a graph are not efficient in practice, this makes reduction algorithms potentially more practical (if the set of reduction rules is not too large). In Chapter 5, we present the basic theory on reduction algorithms and we show that reduction algorithms can be used to solve large classes of decision and optimization problems on graphs of bounded treewidth. The chapter is meant as a comprehensive overview of results presented by Arnborg, Courcelle, Proskurowski, and Seese [1993], Bodlaender [1994], and Bodlaender and Hagerup [1995]. It gives an introduction into the theory of reduction algorithms and their applications to decision and optimization problems on graphs of bounded treewidth. In effect, it provides the basic terminology and results that are used in Chapters 6 – 9.

One drawback of the reduction algorithms presented in Chapter 5 is that they only solve decision and optimization problems. For decision problems, the algorithms only return 'yes' or 'no', but they do not return a solution for the problem if the answer is 'yes'. Similarly, for optimization problems, only the optimal value is returned, but no optimal solution of the problem is returned. In Chapter 6, we extend the theory of reduction algorithms to *constructive* reduction algorithms, which also return an (optimal) solution for the problem at hand, if one exists. We show that this theory can be applied to a large class of constructive decision and optimization problems on graphs of bounded treewidth.

In Chapter 7, a number of well-known optimization problems is listed on which the theory of reduction algorithms that is presented in Chapters 5 and 6 can be applied. This result implies that the listed problems can be solved efficiently on graphs of bounded treewidth using reduction algorithms. We also give a number of optimization problems for which the theory can not be applied. These problems, however, can be solved efficiently if a tree de-

composition of the graph is given.

In Chapters 8 and 9 we show that the following two problems can be solved by efficient parallel reduction algorithms:

- given a graph G, check whether G is series-parallel, and if it is, construct an 'sp-tree' for G (see Section 2.3.3 for definitions), and
- given a graph G, check whether G has treewidth at most two, and if it does, construct a tree decomposition of width at most two of the graph.

The two problems are closely related: a series-parallel graph has treewidth at most two, and a biconnected graph of treewidth at most two is series-parallel. Despite this resemblance, it turns out that the algorithm for solving the 'treewidth two' problem is more complicated than the algorithm for recognizing series-parallel graphs. In Chapter 8, we present an (almost) logarithmic parallel algorithm for recognizing series-parallel graphs; in Chapter 9, we modify this algorithm to obtain a parallel algorithm for graphs of treewidth at most two with the same resource bounds. Both algorithms are applications of the general theory of constructive reduction algorithms as presented in Chapter 6, but they do not fit in the framework of constructive reduction algorithms for graphs of bounded treewidth that are presented in that chapter. For both problems, the set of reduction rules is described completely. These sets of reduction rules are quite small, which means that there are no large constants in the running time of the algorithms. Hence the algorithms are probably also efficient in practice.

The parallel algorithm for series-parallel graphs presented in Chapter 8 improves in efficiency on the parallel algorithms of He and Yesha [1987], He [1991], and Eppstein [1992]. The parallel algorithm for treewidth at most two presented in Chapter 9 improves in efficiency on the parallel algorithms for treewidth at most k for any fixed k that are given by Bodlaender and Hagerup [1995].

In Appendix A, we give definitions of a number of well-known graph problems that are used throughout this thesis.

This thesis comprises, among other things, the work that has been published in Bodlaender and de Fluiter [1995, 1996b, 1996c, 1996a].

Chapter 2

Preliminaries

In this chapter, we give a number of definitions and preliminary results. We start in Section 2.1 by presenting the terminology on graphs and algorithms as it is used in this thesis. Section 2.2 provides an introduction to the notions of treewidth and pathwidth and discusses some results related to these notions. In Section 2.3, we define a number of graph classes that are used in this thesis, and we indicate their role in the theory of treewidth and pathwidth.

2.1 Graphs and Algorithms

We assume that the reader is familiar with graph theory and algorithms, but we give an overview of the terminology that is used in this thesis. More background information can be found in e.g. Harary [1969] for graph theory and Cormen, Leiserson, and Rivest [1989] for algorithms.

2.1.1 Graphs

Definition 2.1.1 (Graph). A *simple graph* G is a pair (V,E), where V is a set of *vertices*, and E is a set of *edges*. Each edge is an unordered pair of distinct vertices u and v, denoted by $\{u,v\}$. A *multigraph* G is a pair (V,E), where V is a set of vertices, and E is a multiset of edges. A *graph* is either a simple graph or a multigraph.

In this chapter, the term 'graph' refers to both simple graphs and multigraphs. In the remaining chapters of the thesis, we use the term graph for one of the two, and we state which one is meant at the beginning of each chapter. In some cases, we use *directed* graphs (either simple graphs or multigraphs): in a directed graph, each edge is an ordered pair of vertices, and an edge from vertex u to v is denoted by (u,v). The sets of vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The cardinality of $V(G)$ is usually denoted by n, the cardinality of $E(G)$ by m.

Let $G=(V,E)$ be a graph. For any edge $e=\{u,v\}\in E$, u and v are called the *end points* of e, and e is called an edge between u and v, or connecting u and v. Two vertices $u,v\in V$ are *adjacent* if there is an edge $\{u,v\}\in E$. If two vertices u and v are adjacent, we also say that u is a *neighbor* of v, and vice versa. A vertex $v\in V$ and an edge $e\in E$ are called *incident* if $e=\{u,v\}$ for some $u\in V$. The *degree* of a vertex v in G is the number of edges that are incident with v, and is denoted by $\deg(v)$ (note that for simple graphs, the degree of a vertex equals the number of its neighbors, whereas for multigraphs this does not necessarily hold).

If G is a multigraph and e_1 and e_2 are distinct edges with end points u and v, then we say that e_1 and e_2 are *parallel* to each other, and there are *multiple* edges between u and v.

A graph G' is a *subgraph* of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. If G' is a subgraph of G, then G is called a *supergraph* of G'. A graph G' is the subgraph of G *induced* by W, where $W \subseteq V(G)$, if $V(G') = W$ and $E(G') = \{\{u,v\} \in E \mid u,v \in W\}$. We also say G' is an *induced subgraph* of G. For any $W \subseteq V(G)$, the subgraph induced by W is denoted by $G[W]$.

A *walk* W in a graph G is an alternating sequence $(v_1, e_1, v_2, e_2, \ldots, e_p, v_{p+1})$ of vertices and edges ($p \geq 0$), starting and ending with a vertex, such that for each i, $v_i \in V$, and $e_i \in E$, and $e_i = \{v_i, v_{i+1}\}$. The walk W is also called a walk from v_1 to v_{p+1}, or a walk between v_1 and v_{p+1}. Vertices v_1 and v_{p+1} are called the end points of the walk, all other vertices are inner vertices. We also call v_1 the first vertex and v_{p+1} the last vertex of the walk. The *length* of a walk is the number of edges in the walk. We say a walk W *uses* a vertex v if $v = v_i$ for some i with $1 \leq i \leq p+1$, and W *avoids* v if W does not use v. If only the sequence of vertices in a walk W is of importance, then W is also denoted as a sequence $(v_1, \ldots, v_p)$ of vertices, such that for each i with $1 \leq i < p$, $\{v_i, v_{i+1}\} \in E(G)$. Note that if G is a simple graph, then this sequence determines the edges of the walk as well.

A *path* in a graph G is a walk in which all vertices are distinct (and hence all edges are distinct). A *cycle* C in G is a walk in which all edges are distinct, and all vertices are distinct, except for the first and the last vertex, which are equal. A walk, path or cycle H in a graph can also be seen as a subgraph of G, and we denote the set of vertices in H by $V(H)$, and the set of edges by $E(H)$. The *distance* between two vertices v and w in G is the length of the shortest path from v to w in G.

Two vertices are *connected* in a graph G if there is a path between them. A graph G is *connected* if every pair of vertices of G is connected. A *(connected) component* C of G is a maximal connected subgraph of G, i.e. C is a subgraph of G which is connected, and there is no subgraph of G which properly contains C and is also connected. A set $W \subseteq V$ is a *separator* of G if there are two vertices $u,v \in V \Leftrightarrow W$, such that u and v are connected in G and not connected in $G[V \Leftrightarrow W]$. A *cut vertex* of G is a vertex $v \in V(G)$ for which $\{v\}$ is a separator of G; we also say v *separates* the graph G. A graph G is *biconnected* if G is connected and contains no cut vertices. A *biconnected component* or *block* of G is a maximal biconnected subgraph of G. It can be seen that the blocks of G partition the set E of edges of G, each block is an induced subgraph of G, and a vertex $v \in V$ is a cut vertex of G if and only if v is contained in two or more blocks of G. An edge $e \in E$ is called a *bridge* of G if there are two vertices $u,v \in V$ that are connected in G, but that are not connected in the graph $(V, E \Leftrightarrow \{e\})$. A block B of G consisting of one edge with its two end points is called a *trivial block*. If a block B of G contains two or more edges, it is called a *non-trivial* block of G.

A *tree* is a simple connected graph without cycles. A *forest* is a simple graph without cycles, i.e. a graph is a forest if and only if each of its components is a tree. Note that in a tree, there is a unique path between each pair of vertices.

A *rooted tree* is a tree T with a distinguished vertex $r \in V(T)$ called the *root* of T. In a rooted tree T, the *descendants* of a vertex $v \in V(T)$ are the vertices of which the path to the

root uses v. The *children* of v are the descendants of v which have distance one to v. If v is not the root, then the parent of v is the unique vertex of which v is a child. The *leaves* of a rooted tree are the vertices without children (i.e. each leaf has degree one, unless if it is the root, in which case it has degree zero). (The term leaf is sometimes used in trees that are not rooted: in this case, it refers to a vertex of degree one.) The vertices which are not leaves are called *internal vertices*. It can be shown by induction that the number of internal vertices with two or more children is at most equal to the number of leaves of the tree. A *rooted binary* tree is a rooted tree in which each internal vertex has two children.

The *depth* of a rooted tree T is the maximum distance of any vertex in T to the root. The *level* of a vertex v in a rooted tree T equals the depth of T minus the distance of v to the root. Hence the root has level d, where d is the depth, and the vertices on level zero are leaves which have distance d to the root.

A *complete graph* or *clique* is a simple graph in which every two vertices are adjacent. The complete graph on n vertices is denoted by K_n. A clique in a graph G is a subgraph of G which is a clique. The maximum clique size of a graph G is the maximum number of vertices of any clique in G.

Let $G = (V,E)$ be a graph. Let $C = (v_1, e_1, \ldots, e_p, v_1)$ be a cycle in G. A *chord* of C in G is an edge $e \in E$ such that $e = \{v_i, v_j\}$ for some i and j, and there is no edge between v_i and v_j in C. If C has no chords in G, then C is called a *chordless cycle* of G. In other words, a cycle C in a simple graph G is a chordless cycle if the vertices of C induce a cycle in G. A graph which contains no chordless cycles of length four or more is called a *chordal* or *triangulated* graph.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be *isomorphic* if there are bijections $f : V_1 \to V_2$ and $g : E_1 \to E_2$ such that for each $v \in V_1$ and $e \in E_1$, v is incident with e in G_1 if and only if $f(v)$ is incident with $g(e)$ in G_2. The pair (f, g) is called an *isomorphism* from G_1 to G_2. If G_1 and G_2 are simple graphs, then it suffices to give the bijection between the vertices of G_1 and G_2, i.e. G_1 and G_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ such that for each $u, v \in V_1$, $\{u,v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$. For simple graphs, we also say that f is an isomorphism from G_1 to G_2.

Let G be a simple graph, and let $e \in E(G)$ with $e = \{u,v\}$. Furthermore, let $N \subseteq V(G)$ denote the union of the sets of neighbors of u and v, except for u and v themselves. The *contraction* of e in G is the operation that removes u and v and their incident edges from G, and adds a new vertex w to the graph which is exactly adjacent to the vertices in N. An *edge contraction* in G is the contraction of some edge $e \in E(G)$. A *minor* of G is a simple graph G' which is obtained from a subgraph of G by applying a sequence of edge contractions.

2.1.2 Graph Problems and Algorithms

This thesis is concerned with algorithms for graph problems. A graph problem usually consists of a description of an arbitrary instance of the problem, and the problem that has to be solved for this instance. For graph problems, each instance contains at least a graph. In Appendix A, we give definitions of a number of well-known graph problems that are considered in this thesis, or that act as illustrative examples throughout.

We distinguish between two types of graph problems, namely *decision problems* and *optimization problems*. Both types have a *non-constructive* version and a *constructive* version. In a non-constructive decision problem, or simply decision problem, the question is whether a certain property holds for an instance. An algorithm solving a decision problem has as input an instance of the problem, and as output the answer to the question, which may be either 'yes' (true) or 'no' (false). An example of a decision problem is HAMILTONIAN CIRCUIT, in which the question is whether a given graph contains a Hamiltonian circuit (see also Appendix A). A decision problem P of which each instance is a single graph can also be seen as a *graph class*: take the class G of all graphs for which P has 'yes' as an answer. The problem of checking whether a given graph is in a graph class G is also called a *recognition problem*. An algorithm solving this problem is an algorithm that recognizes graphs in the class G.

In a constructive decision problem, or *construction problem*, it should not only be decided whether a certain property holds for a given instance, but if the property holds, a concrete solution for the instance should be constructed. For example, in the constructive version of HAMILTONIAN CIRCUIT, the problem is to construct a Hamiltonian circuit in a given graph, if one exists. Hence an algorithm solving a construction problem has as input an instance of the problem, and as output a solution to the problem if there is one, and 'no' or false otherwise.

In a non-constructive optimization problem, or simply optimization problem, the problem is to find the *value* of some optimal solution of the instance: an optimal solution is a solution with optimal value, where optimal can be either maximum or minimum. An example of an optimization problem is MAX INDEPENDENT SET (see Appendix A for a definition): in this problem, the solutions are independent sets of the given graph, and the value of an independent set is its cardinality. The problem is to find the maximum size of any independent set in the given graph. An algorithm solving an optimization problem has as input an instance of the problem, and returns the value of an optimal solution of this instance, if there is a solution, 'no' or false otherwise.

In a constructive optimization problem, the problem is not only to find the value of an optimal solution for a given instance, but also to construct an optimal solution. For example, in the constructive version of MAX INDEPENDENT SET, we ask for an independent set in the given graph of maximum cardinality. Hence an algorithm solving a constructive optimization problem has as input an instance of the problem, and returns an optimal solution and its value if there is a solution, 'no' or false otherwise.

With an optimization problem, we can usually associate a number of decision problems. Suppose for instance that we have a maximization problem called MAX PROBLEM, in which the problem is to find the maximum value of any solution in a given graph. Then we define the decision problem PROBLEM as follows: given a graph G and an integer k, does G contain a solution of value at least k? We can also assume that the integer is not part of the input, but is fixed to some value k. Then the problem is denoted by k-PROBLEM. For minimization problems, we can apply the same technique. (See also Appendix A.) Consider for example MAX INDEPENDENT SET. Then INDEPENDENT SET is the problem in which a graph G and a non-negative integer k are given, and the question is whether G contains an independent

set of size k or more. For each fixed k, k-INDEPENDENT SET asks whether a given graph contains an independent set of size k or more.

In an algorithm solving a graph problem, the input graph must be stored in memory. There are many ways to represent graphs in memory. We make use of an *adjacency list representation*. This representation contains a doubly linked list of all vertices in the graph. For each vertex v, a doubly linked, cyclic list is maintained which contains an entry for each edge that is incident with v. This list is called the adjacency list of v; an entry for an edge $e = \{u,v\}$ in the adjacency list of v contains the edge e, a pointer to its two end points, and a pointer to the entry for edge e in the adjacency list of u.

In this thesis, we give sequential and parallel algorithms for different graph problems. The algorithms are usually described in a rather informal way, the details are left out. For sequential algorithms, we use the random access machine or RAM with uniform cost measure. A RAM consists of a single processor with a random access memory [Mehlhorn, 1984]. Each basic instruction, like reading from and writing to a memory location, and arithmetic or logic operations, uses one time unit. For parallel algorithms, the model of computation we use is the parallel RAM, or PRAM: in this model, we have a number of processors (RAMs) and a global memory. Each processor can read to and write from the global memory at the same time. In this thesis, we use two models for parallel computation, namely the CRCW PRAM and the EREW PRAM: in the first model, different processors may read from or write to the same memory location at the same time, whereas in the latter model, at most one processor may read from or write to the same memory location at any point in time.

In the analysis of a sequential algorithm, we describe the running time of the algorithm, and sometimes the amount of memory-space used by the algorithm, as a function of the size of the input. The running time is estimated by the number of basic instructions. In analyzing a parallel algorithm, we describe the amount of time, the number of operations, and the amount of space that the algorithm uses, all as a function of the input size. By the number of operations, we mean the product of the amount of time and the number of processors that is used. If the number of operations of a parallel algorithm for some problem equals the best known running time of a sequential algorithm for this problem, we say that parallel algorithm has optimal speedup. For more background information on (the analysis of) sequential or parallel algorithms, see e.g. Cormen, Leiserson, and Rivest [1989] and JáJá [1992].

2.2 Treewidth and Pathwidth

In this section, we give some background information on the treewidth and pathwidth of a graph. The notions of treewidth and pathwidth were introduced by Robertson and Seymour [1983, 1986a].

Definition 2.2.1 (Tree Decomposition & Treewidth). Let $G = (V,E)$ be a graph. A *tree decomposition* TD of G is a pair (T,X), where $T = (I,F)$ is a tree, and $X = \{X_i \mid i \in I\}$ is a family of subsets of V, one for each node (vertex) of T, such that

- $\bigcup_{i \in I} X_i = V$,

- for every edge $\{v,w\} \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and
- for all $i,j,k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *treewidth* or *width* of a tree decomposition $((I,F),\{X_i \mid i \in I\})$ is $\max_{i\in I}|X_i| \Leftrightarrow 1$. The treewidth of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum width over all possible tree decompositions of G.

The vertices of a tree in a tree decomposition are usually called *nodes* to avoid confusion with the vertices of a graph. If a vertex v or the end points of an edge e are contained in X_i for some node i of a tree decomposition, we also say node i *contains* v or e. An example of a graph of treewidth two and a tree decomposition of width two of the graph is given in Figure 2.1. A tree decomposition is usually depicted as a tree in which each node i contains the vertices of X_i.

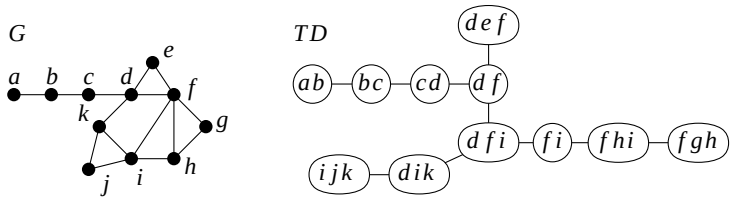


Figure 2.1. A graph G of treewidth two, and a tree decomposition TD of width two of G.

Definition 2.2.2 (Path Decomposition & Pathwidth). A *path decomposition PD* of a graph G is a tree decomposition (T,X) of G in which the tree T is a path (i.e. the nodes of T have degree at most two). The *pathwidth*of a graph G is the minimum width over all possible path decompositions of the graph, and is denoted by $\mathrm{pw}(G)$.

Let $PD = (T,X)$ be a path decomposition of a graph G with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. We usually represent PD by the sequence $(X_{i_1},X_{i_2},\ldots,X_{i_t})$, where $(i_1,i_2,\ldots,i_t)$ is the path representing T. Note that the pathwidth of a graph is at least equal to the treewidth of the graph, and there are graphs of which the pathwidth is larger than the treewidth. The graph of Figure 2.1 for example, has pathwidth three. A path decomposition of width tree of this graph is depicted in Figure 2.2.

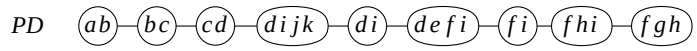


Figure 2.2. A path decomposition PD of width three of the graph G depicted in Figure 2.1.

Let k be a positive integer. Graphs of treewidth and pathwidth at most k are also called *partial k-trees* and *partial k-paths*, respectively (as they are exactly the subgraphs of k-trees or k-paths, see e.g. Kloks [1994] for definitions and proofs). In the literature, many other notions have been defined which turned out to be equivalent to the notions of treewidth or pathwidth. Bodlaender [1996b] gave a list of these notions. There are also many classes of graphs which have a constant bound on the treewidth or pathwidth, or which are closely related to classes of graphs of bounded treewidth or bounded pathwidth. For example, the forests are exactly the simple graphs of treewidth at most one. Series-parallel graphs (see Definition 2.3.3) have treewidth at most two, k-outerplanar graphs have treewidth at most $3k \Leftrightarrow 1$ [Bodlaender, 1996b]. In Section 2.3, we introduce some of these classes. For a complete overview, see Bodlaender [1996b].

In the remainder of this section, we show some properties of tree and path decompositions of graphs (Section 2.2.1), we discuss the complexity of the problems of computing the treewidth and pathwidth of a graph (Section 2.2.2), and we describe two major algorithmic results for graphs of bounded treewidth (Sections 2.2.3 – 2.2.5).

2.2.1 Properties of Tree and Path Decompositions

We give a number of well-known properties of tree and path decompositions and of graphs of bounded treewidth or pathwidth in this section. Most of these properties have already been noted by many authors (see e.g. Robertson and Seymour [1983, 1986a], Scheffler [1989] and Bodlaender [1996b]). To give some feeling for the concepts of tree and path decompositions, we give a proof for some of these results.

Lemma 2.2.1 [Scheffler, 1989; Bodlaender, 1996b]. *Let G be a graph.*

1. *The treewidth or pathwidth of any subgraph of G is at most the treewidth or pathwidth of G.*
2. *The treewidth of G is the maximum treewidth over all components of G.*
3. *The pathwidth of G is the maximum pathwidth over all components of G.*
4. *The treewidth of G is the maximum treewidth over all blocks of G.*

Proof.

1. Let D be a tree or path decomposition of G of minimum width, and let G' be a subgraph of G. Turn D into a tree or path decomposition of G' by removing all vertices of $V(G) \Leftrightarrow V(G')$ from nodes of D. The width of the resulting tree or path decomposition is at most the width of D.

2. By part 1 of the lemma, each component of G has treewidth at most $\mathrm{tw}(G)$. Suppose G has t components, and let $TD_1, \ldots, TD_t$ be minimum width tree decompositions of the components of G. Connecting the tree decompositions $TD_1, \ldots, TD_t$ in an arbitrary way without introducing any cycles results in a tree decomposition of G. The width of this tree decomposition equals the maximum width of $TD_1, \ldots, TD_t$.

3. Each component of G has pathwidth at most $\mathrm{pw}(G)$. Let $PD_1, \ldots, PD_t$ be minimum width path decompositions the components of G. The concatenation of $PD_1, \ldots, PD_t$ in arbitrary order results in a path decomposition of G. The width of this path decomposition equals the maximum width of $PD_1, \ldots, PD_t$.

4. Suppose G is connected. By part 1, each block of G has treewidth at most $\mathrm{tw}(G)$. Let $B_1, \ldots, B_s$ denote the blocks of G and let C be the set of cut vertices of G. Let $TD_1, \ldots, TD_s$ be minimum width tree decompositions of the respective blocks of G, with $TD_i = (T_i, X_i)$ for each i. Let T be the disjoint union of $T_1, \ldots, T_s$ and let X be the disjoint union of $X_1, \ldots, X_s$. For each cut vertex v of G, add a new node i_v to T and a new set X_{i_v} to X with $X_{i_v} = \{v\}$. For each block B_j of G which contains v, add an edge between node i_v, and a node of T_j which contains v. The result is a tree decomposition (T, X) of G, and its width is the maximum width of all blocks of G. Hence the treewidth of G is at most the maximum treewidth of a block of G. If G is not connected, then the same construction can be performed for all components of G, and we get the result from part 2. □

Lemma 2.2.2. *Let G be a graph and $TD = (T, X)$ a tree decomposition of G.*

1. *Let $u, v \in V(G)$, and let $i, j \in I$ be such that $u \in X_i$ and $v \in X_j$. Then each node on the path from i to j in T contains a vertex of every path from u to v in G.*
2. *For each connected subgraph G' of G, the nodes in T which contain a vertex of G' induce a subtree of T.*

Proof.
1. Let $u, v \in V(G)$, and let $P = (u, e_1, w_1, e_2, w_2, \ldots, v)$ be a path from u to v in G. We use induction on the length of P. If P has length zero, then $u = v$ and the result holds by property 3 of a tree decomposition.

Suppose P has length one or more. Let $i, j \in I$ be such that $u \in X_i$ and $v \in X_j$. Let P' be the subpath of P from w_1 to v. Let $l \in I$ such that $u, w_1 \in X_l$. By the induction hypothesis, each node on the path from l to j in T contains a vertex of P'. If i is on the path from l to j in T, then this proves part 1 of the lemma. If i is not on the path from l to j, then each node on the path from i to l in T contains u, and hence each node on the path from i to j either contains u or a vertex of P'. This proves part 1 of the lemma.

2. Suppose that there is a connected subgraph G' of G which does not induce a subtree of T. Then there are nodes $i, j \in I$ such that X_i contains a vertex v of G', X_j contains a vertex w of G', and there is a node l on the path from i to j which does not contain a vertex of G'. As there is a path from v to w in G', and hence in G, each node on the path from i to j in T contains a vertex of G' (by part 1 of this lemma). This gives a contradiction. □

The following lemma is proved in e.g. Bodlaender and Möhring [1993].

Lemma 2.2.3 (Clique Containment). *Let $G = (V, E)$ be a graph, let $TD = (T, X)$ be a tree decomposition of G with $T = (I, F)$ and $X = \{X_i \mid i \in I\}$, and let $W \subseteq V$ be such that W induces a clique in G. There is an $i \in I$ such that $W \subseteq X_i$.*

Proof. We prove this by induction on $|W|$. If $|W| = 1$, then there is an $i \in I$ with $W \subseteq X_i$ by definition. Suppose $|W| > 1$. Let $v \in W$. By the induction hypothesis, there is a node $i \in I$ such that $W \Leftrightarrow \{v\} \subseteq X_i$. Let $T' = (I', F')$ be the subtree of T induced by the nodes containing v. If $i \in I'$, then $W \subseteq X_i$. Suppose $i \notin I'$. Let $j \in I'$ be such that j is the node of T' that has the shortest distance to i. We show that $W \subseteq X_j$. Let $w \in W \Leftrightarrow \{v\}$. Note that each path from a node in T' to node i in T uses node j. As there is an edge $\{v, w\} \in E(G)$, there is a node $j' \in I'$ such that $v, w \in X_{j'}$. The path from j' to i uses node j, and hence $w \in X_j$. □

Lemma 2.2.4. *Let G be a connected partial k-path, $k \geq 1$, and $W \subseteq V$ such that $G[W]$ is connected. At most two of the connected components of $G[V \Leftrightarrow W]$ have pathwidth k.*

Proof. Suppose there are three components G_1, G_2 and G_3 of $G[V \Leftrightarrow W]$ which have pathwidth k. Let $PD = (X_1, \ldots, X_t)$ be a path decomposition of G of width k. For $i = 1, 2, 3$ let $(X_{j_i}, \ldots, X_{l_i})$ denote the subsequence of PD consisting of all nodes that contain vertices of G_i. Note that for each i, if we remove the vertices of $V(G) \Leftrightarrow V(G_i)$ from $(X_{j_i}, \ldots, X_{l_i})$, then we get a path decomposition of width k of G_i. Suppose w.l.o.g. that $j_1 \leq j_2 \leq j_3$. If $l_1 > l_2$, then each node in $(X_{j_2}, \ldots, X_{l_2})$ contains a vertex of G_1. This is not possible, since G_2 has pathwidth k and $V(G_1) \cap V(G_2) = \emptyset$. Hence $l_1 \leq l_2$ and analogously $l_2 \leq l_3$. Let $G' = G[V(G_1) \cup V(G_3) \cup W]$. Note that G' is a connected subgraph of G which has no vertices in common with G_2. Hence, by Lemma 2.2.2, each X_i, $j_1 \leq i \leq l_3$, contains at least one vertex of G'. But $j_1 \leq j_2 \leq l_2 \leq l_3$ and G_2 has pathwidth k, which gives a contradiction. □

A *rooted binary* tree decomposition of a graph G is a tree decomposition (T, X) of G in which T is a rooted binary tree.

Lemma 2.2.5. *Let G be a graph. There is a rooted binary tree decomposition of minimum width of G with $O(n)$ nodes.*

Proof. Let $k = \mathrm{tw}(G)$, and let $TD = (T, X)$ be a tree decomposition of width k of G, with $T = (I, F)$ and $X = \{X_i \mid i \in I\}$. We turn TD into a rooted binary tree decomposition of width k of G. Take an arbitrary node $r \in I$ as the root. Repeat the following as long as possible. For each leaf node $i \in I$ with $i \neq r$, if $X_i \subseteq X_j$, where j is i's parent, then remove node i. For each node $i \in I$, $i \neq r$, with degree two, do the following. Let j be i's parent, and l be i's only child. If $X_i \subseteq X_j$, then 'splice out' i, i.e. remove i and let j be l's new parent. The result is still a tree decomposition of G of the same width. We show that the number of nodes of TD is $O(n)$.

The number of internal nodes with two or more children is at most equal to the number of leaves. Let $i \in I$ be a node with at most one child, suppose $i \neq r$ and j is i's parent. As $X_i \not\subseteq X_j$, X_i contains a vertex which is not in X_j. Let v_i denote this vertex. For every two distinct nodes i and j with at most one child that are not the root, $v_i \neq v_j$, otherwise property 3 of a tree decomposition is violated. Hence there are at most n nodes with at most one child (except for the root). This implies that the total number of nodes is at most $O(n)$.

We next show how TD can be transformed into a rooted binary tree decomposition. To this end, we apply the following transformations to each node. Let $i \in I$ be an internal node of T. If i has two children, do nothing. If i has one child, then add a new leaf node j to T which

is the second child of i, and let $X_j = X_i$. Suppose i has $d \geq 3$ children, and let $j_1, \ldots, j_d$ denote the children of i. We split i into nodes $i_1, \ldots, i_{d-1}$, and let $X_{i_a} = X_i$ for each a, $1 \leq a \leq d \Leftrightarrow 1$. The new nodes are connected as follows. The parent of i_1 is the former parent of i. For each a, $1 \leq a < d \Leftrightarrow 1$, i_a has children j_a and i_{a+1}, and i_{d-1} has children j_{d-1} and j_d. See also Figure 2.3. It can be seen that the resulting tree is a rooted tree in which each internal node has exactly two children, and hence we have a rooted binary tree decomposition of minimum width of G. Furthermore, we have added at most $O(n)$ nodes to the tree decomposition, which means that the total number of nodes is $O(n)$. □

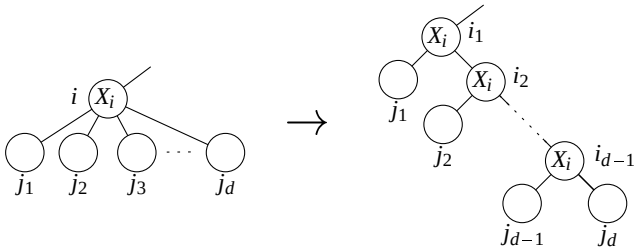


Figure 2.3. The splitting step performed on each node i with three or more children in order to get a rooted binary tree decomposition.

The last part of the construction of the proof of Lemma 2.2.5 shows that, if we have a tree decomposition of a graph G with l nodes, then we can transform it in $O(l)$ time into a rooted binary tree decomposition of the same width of G with $O(l)$ nodes.

Lemma 2.2.6. *Let $G = (V,E)$ be a simple graph, let $k \geq 1$, and suppose* $\mathrm{tw}(G) = k$. *Then* $|E| \leq k|V| \Leftrightarrow \frac{1}{2}k(k+1)$.

Proof. We prove this by induction on $|V|$. If $|V| < k+1$, then $\mathrm{tw}(G) < k$. If $|V| = k+1$, then $|E| \leq \frac{1}{2}k(k+1) = k|V| \Leftrightarrow \frac{1}{2}k(k+1)$. Suppose $|V| > k+1$. Let $TD = (T,X)$ be a tree decomposition of G of width at most k with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. Assume that for each node $i \in I$ with neighbor $j \in I$, $X_i \not\subseteq X_j$ (it is shown in the proof of Lemma 2.2.5 that there is such a tree decomposition). Note that T contains at least two nodes. Let i be a node of degree one of T and let j be the neighbor of i. Note that there is a vertex $v \in X_i$ with $v \notin X_j$. This implies that v is adjacent to at most k vertices, as $|X_i| \leq k+1$. The graph $G' = G[V \Leftrightarrow \{v\}]$ has treewidth at most k, and has $|V| \Leftrightarrow 1$ vertices, hence $|E(G')| \leq k|V(G')| \Leftrightarrow \frac{1}{2}k(k+1)$. This implies that $|E| \leq |E(G')| + k \leq k|V(G')| \Leftrightarrow \frac{1}{2}k(k+1) + k = k|V| \Leftrightarrow \frac{1}{2}k(k+1)$. □

Lemma 2.2.7. *Let G be a graph and let H be a minor of G. Then* $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ *and* $\mathrm{pw}(H) \leq \mathrm{pw}(G)$.

Proof. Let D be a tree or path decomposition of minimum width of G. We transform D into a tree or path decomposition of H without increasing the width. We first show how D is

transformed if H is obtained from G by one edge contraction. Let $e = \{u, v\}$ be the contracted edge of G. Suppose w is the new vertex that is added in order to get H. In D, we replace all occurrences of u and v by w. It is easy to see that the result is a tree or path decomposition of H and that the width has not increased.

Suppose H is obtained from G by applying a sequence of edge contractions on the subgraph G' of G. We first transform D into a tree or path decomposition of G' by removing all occurrences of vertices in $V(G) \Leftrightarrow V(G')$ from D. After this, we repeatedly apply the transformation as described above for each contraction in the sequence. □

2.2.2 Complexity Issues of Treewidth and Pathwidth

The treewidth and pathwidth optimization problems are defined as follows (see also Appendix A).

MIN TREEWIDTH (MIN PATHWIDTH)
Instance: A graph $G = (V, E)$.
Find: The treewidth (pathwidth) of G.

In the constructive versions of MIN TREEWIDTH and MIN PATHWIDTH, we additionally ask for a tree or path decomposition of minimum width of the graph. We also define the associated decision problems TREEWIDTH and PATHWIDTH, and k-TREEWIDTH and k-PATHWIDTH for any fixed integer $k \geq 1$.

Arnborg, Corneil, and Proskurowski [1987] showed that both MIN TREEWIDTH and MIN PATHWIDTH are NP-hard. Polynomial time approximation algorithms were found by Bodlaender, Gilbert, Hafsteinsson, and Kloks [1995]. They gave polynomial time algorithms which, given a graph G, find a tree decomposition of width at most $O(\text{tw}(G) \cdot \log n)$, and a path decomposition of width at most $O(\text{pw}(G) \cdot \log^2 n)$ of G (for all logarithms used in this thesis, the base is two). For many graph classes, the treewidth and pathwidth can be found more efficiently. Examples are chordal graphs, interval graphs (see Definition 2.3.1), permutation graphs [Bodlaender, Kloks, and Kratsch, 1993] and cographs [Bodlaender and Möhring, 1993]. For an overview, see Bodlaender [1993].

For fixed k, both k-TREEWIDTHand k-PATHWIDTH can be solved in polynomial time, which was first proved by Arnborg et al. [1987]: they gave an $O(n^{k+2})$ algorithm for k-TREEWIDTH. This algorithm actually solves the constructive version of this problem, i.e. it returns a tree decomposition of width at most k of the graph, if one exists. Many people have worked on the problem to find efficient algorithms for k-TREEWIDTH and k-PATHWIDTH [Robertson and Seymour, 1986a; Robertson and Seymour, 1990c; Lagergren, 1996; Reed, 1992], which resulted eventually in $O(n)$ time algorithms by Bodlaender [1996a] for the constructive versions of both problems. The algorithms given by Bodlaender [1996a] are not very practical, as they have large hidden constants. For k-TREEWIDTH with $k \leq 3$, more practical algorithms are given by Matoušek and Thomas [1991], using results of Arnborg and Proskurowski [1986]. For $k = 4$, Sanders [1996] has given a more practical algorithm using similar, but more detailed techniques.

Parallel algorithms for (the constructive version of) k-TREEWIDTH are given by Bodlaender [1988b], Chandrasekharan and Hedetniemi [1988], Lagergren [1996], and Bodlaender and Hagerup [1995]. The algorithm of Bodlaender and Hagerup [1995] is the only one with optimal speedup; it has running time $O(\log^2 n)$ and uses $O(n)$ operations and space on an EREW or CRCW PRAM. Bodlaender and Hagerup [1995] also solve the constructive version of the k-PATHWIDTH problem within the same time and resource bounds. In Chapter 9, we improve on this result for 2-TREEWIDTH and 2-PATHWIDTH: these algorithms use $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM ($\log^* n$ denotes the amount of times we have to replace n by the value of $\log n$ in order to get a value that is at most one. For all practical values of n, $\log^* n \leq 5$ (note that $\log^* 2^{65536} = 5$).)

2.2.3 Dynamic Programming on Tree Decompositions

Many, even NP-hard graph problems can be solved in polynomial time if we know a bound on the treewidth of the input graph. One technique that is applicable to a large class of problems is dynamic programming on a tree decomposition of the graph. We sketch the basic approach of this technique.

Suppose we have some graph problem P, we want to solve P on a simple graph $G = (V, E)$ of treewidth at most k for some constant k, and we have a rooted binary tree decomposition $TD = (T, X)$ of width k of G with $T = (I, F)$ and $X = \{X_i, \mid i \in I\}$. Let r denote the root of T. For each i, let

$$Y_i = \{v \in X_j \mid j = i \vee j \text{ is a descendant of } i \text{ in } T\},$$

and let $G_i = G[Y_i]$. Note that $G_r = G$. For each $i \in I$, a table S_i is computed which contains information about the graph G_i with respect to problem P. These tables must have the following properties.

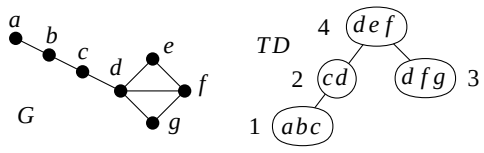
1. For each node $i \in I$, problem P can be solved for G_i solely from the information in table S_i.
2. For each leaf node $i \in I$, S_i can be computed from $G[X_i]$.
3. For each internal node $i \in I$, S_i can be computed from $G[X_i]$ and the tables of i's children in the tree.

If these properties hold, then dynamic programming on the tree decomposition T can be used to compute S_r. First compute the tables S_i for all nodes i on level zero in T (the nodes on level zero are the nodes with largest depth). Next, use these tables to compute the tables of all nodes on level one, and so on until, finally, table S_r is computed. Once S_r is computed, the problem can be solved from the information in S_r. In order to obtain an efficient algorithm, it must be the case that each table can be computed efficiently from the tables of the children. For instance, if for each node i table S_i can be computed in polynomial time in the size of the graph, given $G[X_i]$ and the tables of i's children, then it takes polynomial time to compute the

table of the root node. If each table can be computed in constant time, then it takes $O(n)$ time to compute the table of the root.

The important property of tree decompositions that is used to achieve the design of tables that satisfy the properties described above is the following. For each node $i \in I$, the vertices of G_i that are adjacent to vertices outside G_i must be contained in node X_i. In other words, consider a node $i \in I$, and let $v \in V(G)$ be such that $v \notin Y_i$. Suppose there is a vertex $u \in Y_i$ which is adjacent to v. Then $u \in X_i$ (this follows from part 1 of Lemma 2.2.2, and the fact that there is a node j with $u \in X_j$ and j a descendant of i, and there is a node l with $v \in X_l$ and l not a descendant of i). Hence in G, the graphs G_i and $G[V \Leftrightarrow Y_i]$ are only 'connected' via the vertices in X_i.

As an example, consider MAX INDEPENDENT SET (see also Arnborg [1985] or Bodlaender [1993]). For each $i \in I$, we let the table S_i contain the following information: for each $Z \subseteq X_i$, $S_i(Z)$ is the size of the largest independent set S in G_i with $S \cap X_i = Z$ (let $S_i(Z) = \Leftrightarrow\infty$ if there is no such independent set). It is easy to see that for every $i \in I$, the maximum size of any independent set of G_i is $\max\{S_i(Z) \mid Z \subseteq X_i\}$. This implies that the maximum size of an independent set of G can easily be obtained from the information in table S_r. Figure 2.4 shows an example of a graph, a binary tree decomposition, and the tables corresponding to each node: in each table, only the values which are larger than $\Leftrightarrow\infty$ are given. The size of a maximum independent set in the depicted graph is 4, as this is the value of $S_4(e)$.



T_1		T_2		T_3		T_4	
$\emptyset$	0	$\emptyset$	1	$\emptyset$	0	$\emptyset$	3
a	1	c	2	d	1	d	3
b	1	d	2	f	1	f	3
c	1			g	1	e	4
a,c	2						

I II III

Figure 2.4. A graph G (part I), a binary rooted tree decomposition TD of G (part II), and the tables S_i for each node i of TD (part III).

For each leaf node $i \in I$ and each $Z \subseteq X_i$, we have

$$S_i(Z) = \begin{cases} |Z| & \text{if } \forall_{v,w \in Z} \{v,w\} \notin E \\ \Leftrightarrow\infty & \text{otherwise.} \end{cases}$$

Let $i \in I$ be an internal node, and let j and l be the children of i. For each $Z \subseteq X_i$, we have

$$S_i(Z) = \begin{cases} \max\{S_j(Z_1) + S_l(Z_2) \Leftrightarrow |Z_1 \cap Z| \Leftrightarrow |Z_2 \cap Z| + |Z| \mid (Z_1 \subseteq X_j) \\ \quad \wedge (Z_2 \subseteq X_l) \wedge (Z_1 \cap X_i = Z \cap X_j) \wedge (Z_2 \cap X_i = Z \cap X_l)\} & \text{if } \forall_{v,w \in Z} \{v,w\} \notin E \\ \Leftrightarrow \infty & \text{otherwise.} \end{cases}$$

We show that the latter expression is correct. Let i be an internal node with children j and l, and let $Z \subseteq X_i$. Clearly, there is an independent set in G_i that contains Z if and only if no two vertices of Z are adjacent. Suppose no two vertices of Z are adjacent. Consider an independent set IS of G_i of size $S_i(Z)$ such that $IS \cap X_i = Z$. Let $IS_1 = Y_j \cap IS$ and $IS_2 = Y_l \cap IS$, and let $A_1 = IS_1 \cap X_j$ and $A_2 = IS_2 \cap X_j$. Note that IS_1 is an independent set of G_j and hence $|IS_1| \leq S_j(A_1)$. Similarly, IS_2 is an independent set of G_l and $|IS_2| \leq S_l(A_2)$. Note also that $A_1 \cap X_i = Z \cap X_j$ and $A_2 \cap X_i = Z \cap X_l$. Furthermore,

$$\begin{aligned} S_i(Z) &= |IS| \\ &= |IS_1| + |IS_2| \Leftrightarrow |IS_1 \cap Z| \Leftrightarrow |IS_2 \cap Z| + |Z| \\ &\leq S_j(A_1) + S_l(A_2) \Leftrightarrow |A_1 \cap Z| \Leftrightarrow |A_2 \cap Z| + |Z| \\ &\leq \max\{S_j(Z_1) + S_l(Z_2) \Leftrightarrow |Z_1 \cap Z| \Leftrightarrow |Z_2 \cap Z| + |Z| \mid \\ &\quad (Z_1 \subseteq X_j) \wedge (Z_2 \subseteq X_l) \wedge (Z_1 \cap X_i = Z \cap X_j) \wedge (Z_2 \cap X_i = Z \cap X_l)\}. \end{aligned}$$

On the other hand, let $A_1 \subseteq X_j$ and $A_2 \subseteq X_l$ such that $A_1 \cap X_i = Z \cap X_j$, $A_2 \cap X_i = Z \cap X_l$, and $S_j(A_1) + S_j(A_2) \Leftrightarrow |A_1 \cap Z| \Leftrightarrow |A_2 \cap Z| + |Z|$ is maximum. Note that $S_j(A_1) \geq 0$ and $S_l(A_2) \geq 0$. Let IS_1 and IS_2 be independent sets of G_j and G_l, respectively, such that $IS_1 \cap X_j = A_1$, $IS_2 \cap X_l = A_2$, $|IS_1| = S_j(A_1)$ and $|IS_2| = S_l(A_2)$. Then $IS = IS_1 \cup IS_2 \cup Z$ is an independent set of G_i with $IS \cap X_i = Z$. Furthermore,

$$\begin{aligned} S_i(Z) &\geq |IS| \\ &= |IS_1| + |IS_2| \Leftrightarrow |IS_1 \cap Z| \Leftrightarrow |IS_2 \cap Z| + |Z| \\ &= S_j(A_1) + S_l(A_2) \Leftrightarrow |A_1 \cap Z| \Leftrightarrow |A_2 \cap Z| + |Z| \\ &= \max\{S_j(Z_1) + S_l(Z_2) \Leftrightarrow |Z_1 \cap Z| \Leftrightarrow |Z_2 \cap Z| + |Z| \mid \\ &\quad (Z_1 \subseteq X_j) \wedge (Z_2 \subseteq X_l) \wedge (Z_1 \cap X_i = Z \cap X_j) \wedge (Z_2 \cap X_i = Z \cap X_l)\}. \end{aligned}$$

This shows that the recursive definition of $S_i(Z)$ is correct.

With the two expressions given above, the table S_r can be obtained by computing the tables of all nodes in a bottom-up way: first compute the tables of all nodes on level zero in the tree, then the tables of all nodes on level one, and so on. Each table S_i has size $O(2^{k+1})$ (as $|X_i| \leq k+1$ and T_i contains one entry for each subset of X_i). Therefore, if adjacency of two vertices can be checked in constant time, then each table of a leaf node can be computed in $O(2^{k+1})$ time, and each table of an internal node can be computed in $O(2^{3k+3})$ time from the tables of its children. Note that, with an adjacency list representation, it is not possible to

check in constant time whether two vertices are adjacent. However, as the graph has treewidth at most k, we can modify the adjacency lists in such a way that each edge $e = \{u, v\}$ occurs either in the adjacency list of u or in the adjacency list of v, and each adjacency list has length at most k. It can be seen that we can build such a representation in $O(kn)$ time (we omit the details of this construction). Furthermore, with this representation we can check in $O(k)$ time whether two vertices are adjacent.

The discussion above implies that for any constant $k \geq 1$ and any simple graph G with $\mathrm{tw}(G) \leq k$, the size of a maximum independent set of G can be computed in $O(n)$ time if a tree decomposition of bounded width of G with $O(n)$ nodes is given: turn the given tree decomposition into a rooted binary tree decomposition with $O(n)$ nodes, and apply the algorithm as described above. If we are also interested in a maximum independent set of the graph, then we can compute one from the information in the tables in $O(n)$ time in a top-down manner. For example, in the graph G of Figure 2.4, this gives the following maximum independent set $\{a, c, e, g\}$.

We can also use parallel dynamic programming to compute the size of a maximum independent set in G: for each level l in the rooted tree T of the rooted binary tree decomposition, the tables of the nodes at level l in T can be computed independently of each other, from the tables of their respective children. This suggests the following parallel algorithm, consisting of $d+1$ rounds, where d denotes the depth of T. In round l, $0 \leq l \leq d$, the tables S_i of all nodes i on level l are computed. Each node is handled by a different processor, which needs $O(1)$ time to compute the table. It can easily be seen that this algorithm takes $O(d)$ time with $O(n)$ processors on an EREW PRAM. As the total number of nodes that has to be handled is $O(n)$, standard techniques show that this algorithm can be made to run in $O(d)$ time with $O(n)$ operations, and thus $O(n/d)$ processors (in round i, $0 \leq i \leq d$, let each processor handle $n_i d/n$ nodes on level i, where n_i denotes the number of nodes on level i; summed over all rounds, this takes $O(d)$ time per processor).

Bodlaender and Hagerup [1995] have shown that, given a tree decomposition of width at most k with $O(n)$ nodes of a graph, a rooted binary tree decomposition of the graph of width at most $3k+2$ with depth $O(\log n)$ can be constructed. This transformation can be done in $O(\log n)$ time with $O(n)$ operations and space on an EREW PRAM. On this tree decomposition of depth $O(\log n)$, the parallel dynamic programming algorithm as described above takes $O(\log n)$ time with $O(n)$ operations on an EREW PRAM, if the input of the algorithm is a graph G and a tree decomposition of bounded width of G.

It turns out that the dynamic programming technique described above can be applied to many problems on graphs of bounded treewidth (see e.g. Arnborg [1985], Johnson [1985] and Johnson [1987] for an overview). More systematic attempts have led to linear time algorithms that can solve classes of graph problems on graphs of bounded treewidth [Takamizawa, Nishizeki, and Saito, 1982; Wimer, 1987; Scheffler, 1987; Bodlaender, 1988a; Bern, Lawler, and Wong, 1987; Courcelle, 1990; Borie, Parker, and Tovey, 1991; Arnborg, Lagergren, and Seese, 1991; Abrahamson and Fellows, 1993; Courcelle and Mosbah, 1993]. A very general class of problems for which this has been shown is the class of recognition problems of *finite state* graph classes. These problems include all graph problems that can be defined in

Monadic Second Order Logic.

2.2.4 Finite State Problems and Monadic Second Order Logic

A *graph property* is a function P which maps each graph to the value true or false. We assume that isomorphic graphs are mapped to the same value. A graph property P holds for graph G or $P(G)$ holds, if $P(G) = \text{true}$. An *extended graph property* is a function P for which there are domains $D_1, \ldots, D_t$ ($t \geq 0$), such that for each graph G and each $X_i \in D_i$, $1 \leq i \leq t$, $P(G, X_1, X_2, \ldots, X_t)$ is mapped to the value true or false. (Note that for fixed X_i, $1 \leq i \leq t$, $P(G, X_1, X_2, \ldots, X_t)$ is a graph property.)

A graph property P corresponds directly to a decision problem: given a graph G, does P hold for G? An algorithm decides a property P if it solves the corresponding decision problem.

Definition 2.2.3 (Terminal Graph). A *terminal graph* G is a triple (V, E, X) with (V, E) a simple graph, and $X \subseteq V$ an ordered subset of $l \geq 0$ vertices. We denote X by $\langle x_1, \ldots, x_l \rangle$. Vertices in X are called *terminals* or *terminal vertices*. Vertices in $V \Leftrightarrow X$ are called *inner vertices*.

Figure 2.5 gives an example of a terminal graph. Although a terminal graph with zero terminals is not exactly an ordinary simple graph, we sometimes use it in that way.

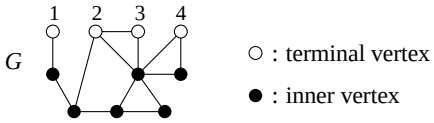


Figure 2.5. Example of a terminal graph G with four terminals.

A terminal graph with l terminals ($l \geq 0$) is also called an l-terminal graph. Let $G = (V, E, X)$ be an l-terminal graph, $l \geq 0$, with $X = \langle x_1, \ldots, x_l \rangle$. For each i, $1 \leq i \leq l$, we call x_i the ith terminal of G. A terminal graph (V, E, X) is said to be *open* if there are no edges between terminals.

Terminal graphs are also called *sourced* graphs (e.g. in Arnborg et al. [1993], Lagergren and Arnborg [1991]), in which case the terminals are called the *sources* of the graph, or *boundaried* graphs (e.g. in Fellows and Langston [1989]), in which case the set of terminals is called the *boundary*.

Definition 2.2.4. The operation $\oplus$ maps two terminal graphs G and H with the same number l of terminals to a simple graph $G \oplus H$, by taking the disjoint union of G and H, then identifying corresponding terminals, i.e., for $i = 1, \ldots, l$, identifying the ith terminal of G with the ith terminal of H, and removing multiple edges.

For an example of the $\oplus$-operation, see Figure 2.6. Note that the result of an $\oplus$ operations is a simple graph, and not a terminal graph.

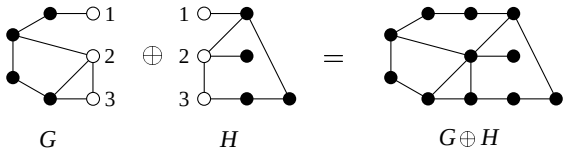


Figure 2.6. Example of operation $\oplus$ applied to two three-terminal graphs.

Definition 2.2.5. Let P be a graph property, and l a non-negative integer. For l-terminal graphs G_1 and G_2, we define the equivalence relation $\sim_{P,l}$ as follows:

$$G_1 \sim_{P,l} G_2 \quad \Leftrightarrow \quad \text{for all } l\text{-terminal graphs } H\text{: } P(G_1 \oplus H) \Leftrightarrow P(G_2 \oplus H).$$

Property P is of *finite index* if for all $l \geq 0$, $\sim_{P,l}$ has finitely many equivalence classes.

There are many equivalent terms for a graph class of which the corresponding graph property is of finite index: such a graph class is *recognizable*, [Courcelle, 1990], *finite state* or *(fully) cutset regular* [Abrahamson and Fellows, 1993], or *regular* [Bern, Lawler, and Wong, 1987; Borie, Parker, and Tovey, 1991] (equivalence has been shown by Courcelle and Lagergren [1996]). We use the term finite state.

The dynamic programming approach described in Section 2.2.3 can be used to recognize graphs from any finite state graph class in linear time, as long as there is a bound on the treewidth of the graph. Therefore, it is again assumed that the input of the algorithm consists of a graph G and a binary rooted tree decomposition of bounded width of G with $O(n)$ nodes. The idea of this algorithm is as follows. Let P be a graph property of finite index. Note that for each equivalence class C of $\sim_{P,0}$, either P holds for each graph in C, or P does not hold for any graph in C. We call a class of $\sim_{P,0}$ of the first type an *accepting class*.

Let $G = (V,E)$ be a simple graph of width at most k, for some constant $k \geq 1$, and let (T,X) be a rooted binary tree decomposition of width at most k of G with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. Suppose r is the root of T, and suppose w.l.o.g. that $X_r = \emptyset$. For each i, let H_i denote the terminal graph obtained from G_i by letting the vertices in X_i be the terminals (recall that G_i is the subgraph of G induced by the vertices in nodes X_j, where j is a descendant of i). For each node $i \in I$, the information that is computed is the equivalence class of $\sim_{P,l}$ that contains H_i (where $l = |X_i|$). Let C_i denote this equivalence class. Now, $P(G)$ holds if and only if C_r is an accepting class. Furthermore, if i is a leaf node, then C_i only depends on $G[X_i]$. If i is an internal node with children j and l, then C_i only depends on the graphs $G[X_i]$, $G[X_j]$ and $G[X_l]$, and the equivalence classes C_j and C_l. Hence we can use this information in a dynamic programming algorithm. We briefly show how this is done.

The number of different graphs with at most $k+1$ vertices is bounded. Furthermore, for each $l \geq k+1$, the number of equivalence classes of $\sim_{P,l}$ is bounded. From this, it can be seen that there is a table T_1 of bounded size in which, for any leaf node i with $|X_i| \leq k+1$, we can find the equivalence class of H_i, given $G[X_i]$. Furthermore, there is a transition table

T_2 of bounded size in which, for each internal node i with children j and l, we can find the equivalence class of H_i, given $G[X_i]$, $G[X_j]$, $G[X_l]$, and the equivalence classes of H_j and H_l. Tables T_1 and T_2 can be effectively constructed if we have an algorithm which decides $\sim_{P,l}$ for all $l \geq 0$. With table T_1, C_i can be computed in constant time for each leaf node i of the tree decomposition. With table T_2, C_i can be computed in constant time for each internal node i, by only using nodes j and l, where j and l are the children of i.

This implies that the dynamic programming approach can be used to recognize graphs of any finite state class, if the input consists of a graph of treewidth at most k for some fixed integer k, and a binary rooted tree decomposition of width at most k of the graph. Sequentially, this algorithm takes $O(n)$ time, if the tree decomposition contains $O(n)$ nodes. In the parallel case, the algorithm takes $O(\log n)$ time with $O(n)$ operations on an EREW PRAM, if the tree decomposition contains $O(n)$ nodes (see also Lagergren [1991]). (In Courcelle [1990], Arnborg, Lagergren, and Seese [1991], and Abrahamson and Fellows [1993], the algorithm is described in terms of finite state tree-automata, but this boils down to the same principle.)

Courcelle [1990] has given a large class of graph properties which are finite state. We define this class here. The *Monadic Second Order Logic* or MSOL for graphs $G = (V, E)$ consists of a language in which predicates can be built with

- the logic connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$ (with their usual meanings),
- individual variables which may be vertex variables (with domain V), edge variables (with domain E), vertex set variables (with domain $P(V)$, the power set of V), and edge set variables (with domain $P(E)$),
- the existential and universal quantifiers ranging over variables ($\exists$ and $\forall$, respectively), and
- the following binary relations:
  - $v \in W$, where v is a vertex variable and W a vertex set variable,
  - $e \in F$, where e is an edge variable and F an edge set variable,
  - 'v and w are adjacent in G', where v and w are vertex variables,
  - 'v is incident with e in G', where v is a vertex variable, and e an edge variable, and
  - equality for variables.

A predicate that is defined in MSOL for graphs is also called an *MSOL predicate*. Let R be an MSOL predicate such that R has no free variables. Then a graph G *satisfies* R if R evaluates to true for G with the common interpretations of the language elements. A graph property P is *MS-definable* if there is a predicate R defined in MSOL for graphs, such that R has no free variables and for each graph $G = (V, E)$, $P(G)$ holds if and only if G satisfies R. A graph class or decision problem is MS-definable if the corresponding graph property is MS-definable.

As an example, we show that the graph property P with $P(G) =$ 'G is two-colorable' is MS-definable (a graph is two-colorable if there is a partition of the vertices in two sets such that each set is an independent set of the graph). For a graph $G = (V, E)$, we define the MSOL

predicate R as follows:

$$\begin{aligned} R \quad = \quad & \exists_{U \subseteq V} \, \exists_{W \subseteq V} \, (U \cap W = \emptyset) \wedge (U \cup W = V) \\ & \wedge (\forall_{u \in V} \forall_{v \in V} \, (u \text{ and } v \text{ are adjacent}) \Rightarrow (u \in U \Leftrightarrow v \in W)\,). \end{aligned}$$

Note that $U \cap W = \emptyset$ can be defined in MSOL as $\forall_{v \in V} \neg (v \in U \wedge v \in W)$, and that $U \cup W = V$ can be defined as $\forall_{v \in V} (v \in U \vee v \in W)$. Now, a graph $G = (V, E)$ is two-colorable if and only if G satisfies predicate R. Hence property P is MS-definable.

Let P be an extended graph property P with variables $G, X_1, X_2, \ldots, X_t$, where $G = (V, E)$ is a graph and for each i, $1 \leq i \leq t$, $X_i \in D_i$ for some domain D_i. Then P is MS-definable if there is a predicate $R(Y_1, Y_2, \ldots, Y_t)$ that is defined in MSOL for graphs, with free variables $Y_1, Y_2, \ldots, Y_t$, such that for each graph G and every $X_1, X_2, \ldots, X_t$ with $X_i \in D_i$ for each i, $P(G, X_1, X_2, \ldots, X_t)$ holds if and only if G satisfies $R(X_1, X_2, \ldots, X_t)$.

As an example, consider the extended graph property Q with for each graph G and every subsets V_1 and V_2 of $V(G)$, $Q(G, V_1, V_2)$ holds if and only if (V_1, V_2) is a two-coloring of G. Let $R(Y_1, Y_2)$ be the MSOL predicate defined as follows:

$$\begin{aligned} R(Y_1, Y_2) \quad = \quad & (Y_1 \cap Y_2 = \emptyset) \wedge (Y_1 \cup Y_2 = V) \\ & \wedge (\forall_{u \in V} \forall_{v \in V} \, (u \text{ and } v \text{ are adjacent}) \Rightarrow (u \in Y_1 \Leftrightarrow v \in Y_2)\,). \end{aligned}$$

Clearly, for each graph G and each two subsets V_1 and V_2 of $V(G)$, (V_1, V_2) is a two-coloring of G if and only if G satisfies $R(V_1, V_2)$. Hence Q is MS-definable.

Courcelle [1990] has shown that MS-definable graph properties are finite state, and thus decidable in linear time for graphs of bounded treewidth. There are many decision problems which are MS-definable, even many NP-complete decision problems, including HAMILTONIAN CIRCUIT and (for fixed k) k-COLORABILITY (see e.g. Arnborg et al. [1991] for a list).

Arnborg et al. [1991] gave an alternative proof of the fact that all MS-definable graph classes are recognizable in $O(n)$ time, if a tree decomposition of bounded width of the input graph is given. They extended MS-definability to, among others, construction problems and (constructive) optimization problems (definitions are given below). They have shown that MS-definable (constructive) decision and (constructive) optimization problems can be solved in linear time, given a tree decomposition of bounded width of the input graph. We describe some of these results.

A construction problem is MS-definable if there is an MS-definable extended graph property $Q(G, X_1, X_2, \ldots, X_t)$, such that the construction problem is to find, for a given graph G, values of $X_1, \ldots, X_t$ for which $Q(G, X_1, \ldots, X_t)$ holds. For instance, the constructive version of k-COLORABILITY is MS-definable, as the extended graph property $Q(G, V_1, V_2, \ldots, V_k)$ which holds if $(V_1, V_2, \ldots, V_k)$ is a k-coloring of G is MS-definable. The constructive version of HAMILTONIAN CIRCUIT is also MS-definable.

An optimization problem is MS-definable if there is an MS-definable extended graph property $Q(G, X_1, \ldots, X_t)$, and there are constants $\alpha_1, \ldots, \alpha_t$ such that the problem is to find

for a given graph G the maximum value of $\alpha_1|X_1| + \alpha_2|X_2| + \cdots + \alpha_t|X_t|$. For instance, MAX INDEPENDENT SET is MS-definable: let $Q(G,W)$ be the extended graph property which holds if W is an independent set of G. It is easy to see that Q is MS-definable. The problem of finding the maximum size of an independent set of $G = (V,E)$ is then the problem to find the maximum value of $|W|$ for any set $W \subseteq V$ for which $Q(G,W)$ holds. Other examples of MS-definable optimization problems are MAX CUT, LONGEST PATH and LONGEST CYCLE.

A constructive optimization problem is MS-definable if there is an MS-definable extended graph property $Q(G,X_1,\ldots,X_t)$, and there are constants $\alpha_1,\ldots,\alpha_t$ such that the problem is to find for a given graph G values of $X_1,\ldots,X_t$, such that $Q(G,X_1,\ldots,X_t)$ holds and $\alpha_1|X_1| + \alpha_2|X_2| + \cdots + \alpha_t|X_t|$ is maximum. For instance, the constructive versions of MAX INDEPENDENT SET, MAX CUT, LONGEST PATH and LONGEST CYCLE are MS-definable.

Borie et al. [1991] have used a different approach to show similar results, using the results of Bern et al. [1987].

The disadvantage of the algorithms presented in this section is that a tree decomposition of bounded width of the input graph is needed. Although there is a linear time algorithm that finds a tree decomposition of small width of a given graph if one exists [Bodlaender, 1996a], this algorithm is not practical, as argued before. In parallel, the best known algorithm to find a tree decomposition of small width of a graph uses $O(\log^2 n)$ time with $O(n)$ operations, even on a CRCW PRAM [Bodlaender and Hagerup, 1995]. This slows down the computations by a factor $\log n$. A way to overcome this disadvantage is to use *reduction algorithms*. These algorithms work directly on the graph: to decide whether a given graph is in some graph class, a reduction algorithm reduces small parts of the graph into smaller parts, thereby preserving membership of the graph class. If no more reductions can be performed, either the graph is reduced to a small graph, which is easy to handle, or the graph is not in the graph class. Reduction algorithms are more thoroughly discussed in Chapter 5.

2.2.5 Forbidden Minors Characterization

Let G and H be graphs. We say G contains H as a minor or H is a minor of G if G has a minor that is isomorphic to H. A graph class $\mathcal{G}$ is *minor-closed* if for every graph G and every minor H of G, if $G \in \mathcal{G}$ then $H \in \mathcal{G}$. Note that for each integer $k \geq 1$, the class of graphs of treewidth or pathwidth at most k is minor-closed, by Lemma 2.2.7. Robertson and Seymour have established deep results on graph minors in their series of papers [1983 – 1996]. An overview of these results can be found in Robertson and Seymour [1985]. The most important of these results in our context are the following.

Theorem 2.2.1 (Graph Minor Theorem). *Let $G_1, G_2, \ldots$ be a countable sequence of graphs. Then there are indices $1 \leq i < j$ such that G_i is a minor of G_j.*

Theorem 2.2.1 was formerly known as Wagner's Conjecture. Let $\mathcal{G}$ be a minor-closed graph class. Let H be a graph which is not in $\mathcal{G}$. Each graph which has H as a minor is not in $\mathcal{G}$, otherwise H would be in $\mathcal{G}$. We call H a *forbidden minor* of $\mathcal{G}$. A *minimal forbidden minor* of $\mathcal{G}$ is a forbidden minor of G of which each proper minor is in $\mathcal{G}$. A minor-closed graph class $\mathcal{G}$ is completely characterized by the set of all minimal forbidden minors $\mathcal{O}$, called

the *obstruction set* of G: a graph is in G if and only if it does not contain a minor in O. Theorem 2.2.1 immediately implies the following result.

Corollary 2.2.1. *For each minor-closed class of graphs G, the obstruction set has finite cardinality.*

Note that Corollary 2.2.1 shows that for each fixed $k \geq 1$, the class of graphs of treewidth at most k or pathwidth at most k has a finite obstruction set.

Robertson and Seymour [1985] have also shown that for a fixed graph H, one can check whether H is a minor of a graph G in $O(n^3)$ time (where $n = |V(G)|$). If the input graph is known to have a bound on the treewidth, then we can even do a minor test in $O(n)$ time, since the corresponding decision problem is MS-definable [Arnborg, Lagergren, and Seese, 1991].

These results imply that there exist $O(n^3)$ time recognition algorithms for all minor-closed graph classes: test for a given graph whether it has a minor in the obstruction set of the graph class. For classes of graphs which have a bound on the treewidth, there even exist $O(n)$ time recognition algorithms. Unfortunately, the results of Robertson and Seymour are non-constructive in the sense that they only prove existence of a finite obstruction set, but provide no method to obtain the obstruction set. Also, the $O(n^3)$ minor testing algorithm has large hidden constants, which makes this algorithm rather impractical. Furthermore, the size of an obstruction set can be very large. For the class of graphs of pathwidth at most k for example, the obstruction set contains among others $(k!)^2$ trees, each having $(5 \cdot 3^k \Leftrightarrow 1)/2$ vertices [Takahashi, Ueno, and Kajitani, 1994].

Many efforts have been made to actually find obstruction sets of minor-closed graph classes. For example, Arnborg and Proskurowski [1986] have given the obstruction sets of the classes of graphs of treewidth at most one, two and three; Bryant, Fellows, Kinnersley, and Langston [1987] have given the obstruction sets for pathwidth at most one and Kinnersley and Langston [1994] for pathwidth at most two. More general approaches have been taken by Fellows and Langston [1989] and Lagergren and Arnborg [1991], who have given a number of ingredients that have to be present in order to be able to compute an obstruction set for a given graph class. A more practical approach, based on the result of Fellows and Langston [1989], is taken by Dinneen [1995].

We explicitly mention the following result, as it will be used in this thesis.

Lemma 2.2.8 [Arnborg and Proskurowski, 1986]. *A graph has treewidth at most one if and only if it does not have K_3 as a minor, and treewidth at most two if and only if it does not have K_4 as a minor.*

2.3 Related Graph Classes

In this section, we define a number of graph classes and graph problems which are related to treewidth and pathwidth.

2.3.1 Chordal Graphs and Interval Graphs

A chordal or triangulated graph is a graph which does not contain any induced cycles of length four or more. Let $G = (V,E)$ be a graph. A *triangulation* of G is a supergraph G' of G with $V(G') = V$, such that G' is chordal.

The following result relates chordal graphs to treewidth.

Lemma 2.3.1 [Robertson and Seymour, 1986a]. *Let $G = (V,E)$ be a graph and let $ct(G)$ denote the least maximum clique size of any triangulation of G. Then* $\text{tw}(G) \leq ct(G) \Leftrightarrow 1$.

Proof. We use the following result of Gavril [1974]: a graph G is a chordal graph if and only if G is the intersection graph of a family of subtrees of a tree (the intersection graph G of a family F of subtrees of a tree is the graph $G = (V,E)$ which contains a vertex for each tree in F, and edge between two vertices if and only if the corresponding trees intersect).

We first show that $\text{tw}(G) \geq ct(G) \Leftrightarrow 1$. Suppose $\text{tw}(G) = k \Leftrightarrow 1$ and let $TD = (T,X)$ be a tree decomposition of width $k \Leftrightarrow 1$ of G, with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. Let E' be the set $\{\{u,v\} \mid \exists_{i \in I}\, u,v \in X_i\}$ and let $G' = (V,E')$. TD is a tree decomposition of width $k \Leftrightarrow 1$ of G', and thus, by Lemma 2.2.3, the clique size of G' is at most k. We show that $G' = (V,E')$ is a chordal graph. For each $v \in V(G)$, let T_v denote the subtree of T induced by the nodes containing v. Then $F = \{T_v \mid v \in V\}$ is a family of subtrees of T, and G' is the intersection graph of F. Hence $\text{tw}(G) \leq ct(G) \Leftrightarrow 1$.

We next show that $\text{tw}(G) \leq ct(G) \Leftrightarrow 1$. Suppose G' is a triangulation of G with maximum clique size k. Let F be a family of subtrees of a tree $T = (I,F)$ such that G' is the intersection graph of F. For each $v \in V(G')$, let $T_v \in F$ be the tree corresponding to vertex v. For each $i \in I$, let $X_i = \{v \in V(G') \mid i \in V(T_v)\}$. Then (T,X) is a tree decomposition of the graph G', and hence of G. Furthermore, as G' has clique size k, each node i contains at most k vertices. Thus (T,X) has width $k \Leftrightarrow 1$, and hence $\text{tw}(G) \leq ct(G) \Leftrightarrow 1$. □

Simple chordal graphs can be recognized in $O(n+m)$ time [Rose, Tarjan, and Lueker, 1976].

For the pathwidth of a graph there is a similar result, which we describe after a few more preliminaries.

Definition 2.3.1 (Interval Graph). A graph $G = (V,E)$ is an *interval graph* if there is a function ϕ which maps each vertex of V to an interval of the real line, such that for each $u,v \in V$ with $v \neq u$,

$$\phi(u) \cap \phi(v) \neq \emptyset \Leftrightarrow \{u,v\} \in E.$$

The function ϕ is called an *interval realization* for G.

It can be seen that interval graphs are chordal. An example of an interval graph and an interval realization of the graph is given in Figure 2.7.

Simple interval graphs can be recognized in $O(n+m)$ time [Booth and Lueker, 1976; Hsu, 1993; Korte and Möhring, 1989], and with these algorithms it is also possible to find an interval realization of the given graph (if it is an interval graph). The relation between interval graphs and pathwidth is expressed in the following results.

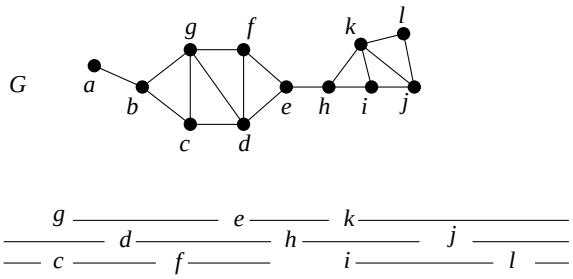


Figure 2.7. An interval graph G and an interval realization ϕ of G.

Lemma 2.3.2. *Let $G = (V,E)$ be a graph and let $PD = (X_1, \ldots, X_t)$ a path decomposition of G. Let $G' = (V,E')$ be the supergraph of G with $E' = E \cup \{ \{u,v\} \mid \exists_{1 \leq i \leq t}\ u,v \in X_i \}$. The graph G' is an interval graph.*

Proof. Let $\phi : V \to \{1, \ldots, n\}$ be defined as follows. For each $v \in V$, if the subsequence of PD consisting of all nodes containing v is $(X_j, \ldots, X_l)$, then $\phi(v) = [j,l]$. Then for each $u,v \in V$, $\{u,v\} \in E'$ if and only if $\phi(u)$ and $\phi(v)$ overlap. □

The graph G' as defined in Lemma 2.3.2 is called the *interval completion* of G for PD.

Lemma 2.3.3 [Möhring, 1990]. *Let $G = (V,E)$ be a graph and let $ci(G)$ denote the least maximum clique size of any interval graph which is a supergraph of G. Then* $\mathrm{pw}(G) = ci(G) \Leftrightarrow 1$.

Proof. We first show that $\mathrm{tw}(G) \leq ci(G) \Leftrightarrow 1$. Suppose $G' = (V', E')$ is an interval graph such that $E \subseteq E'$, the clique size of G' is $ci(G)$, and there is no other interval supergraph of G with smaller clique size. Let $k = ci(G)$. Note that if $V' \neq V$, then the subgraph of G' induced by V is an interval with maximum clique size at most $ci(G)$. Hence we may assume that $V' = V$. Let $\phi : V \to I$ be an interval realization for G', and suppose w.l.o.g. that for each vertex v, $\phi(v) = [l_v, r_v]$ for some integers l_v and r_v. Let $(u_1, \ldots, u_n)$, $n = |V|$, be an ordering of V in such a way that for all i,j with $1 \leq i < j \leq n$, $l_{u_i} \leq l_{u_j}$. For each i with $1 \leq i \leq n$, let $X_i = \{ v \in V \mid l_{u_i} \in \phi(v) \}$. Then $PD = (X_1, \ldots, X_n)$ is a path decomposition of G' and hence of G. Furthermore, each node contains at most k vertices, since the clique size of G is k. Hence PD has pathwidth at most $k \Leftrightarrow 1$.

The proof that $\mathrm{tw}(G) \geq ci(G) \Leftrightarrow 1$ follows directly from Lemmas 2.2.3 and 2.3.2. □

2.3.2 Bandwidth

Definition 2.3.2 (Layout and Bandwidth). Let $G = (V,E)$ be a graph. A *layout* of G is a function $\ell : V \to \mathbb{Z}^+$, such that for each $v \neq w$, $\ell(v) \neq \ell(w)$. The *bandwidth* of a layout ℓ is defined to be $\max\{ \ell(v) \Leftrightarrow \ell(w) \mid \{v,w\} \in E \}$. The *bandwidth* of G is the minimum bandwidth of all layouts of G.

Figure 2.8 shows a layout ℓ of a graph G. This layout has bandwidth two, and this is also the bandwidth of G. The layout is depicted by drawing the vertices in a sequence in the order in which they appear in the layout.

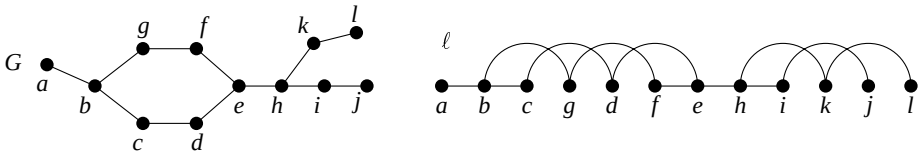


Figure 2.8. A graph G of bandwidth two and a layout ℓ of bandwidth two of G.

Lemma 2.3.4 [Bodlaender, 1996b]. *If G is a graph of bandwidth at most k, then G has pathwidth at most k.*

Proof. Suppose G is a graph of bandwidth at most k. Let ℓ be a layout of G of bandwidth at most k. Then we can make a path decomposition of width at most k as follows. Order the vertices of G as $v_1, \ldots, v_n$, such that for each $i < j$, $\ell(v_i) < \ell(v_j)$. Now for each i, $1 \leq i \leq n \Leftrightarrow k$, make a node i with $X_i = \{v_i, v_{i+1}, \ldots, v_{i+k}\}$. It can be seen that $PD = (X_1, \ldots, X_{n-k})$ is a path decomposition of G and has pathwidth k. □

The BANDWIDTH problem (given a graph G, integer k, does G have bandwidth at most $k \Leftrightarrow 1$?) is NP-complete [Garey, Graham, Johnson, and Knuth, 1978], even when the input graph is a tree [Monien, 1986]. For any fixed k, there is an $O(n^k)$ algorithm that solves k-BANDWIDTH [Gurari and Sudborough, 1984], and there is an $O(n)$ time algorithm for 2-BANDWIDTH [Garey, Graham, Johnson, and Knuth, 1978]. Bodlaender, Fellows, and Hallett [1994] have shown that k-BANDWIDTH is hard for $W[2]$. We do not give an exact definition of $W[i]$-hardness ($i \in \mathbb{N}$) here [Downey and Fellows, 1995], but the idea is that if a parameterized graph problem called k-PROBLEM is hard for $W[i]$, where $i \in \mathbb{N}$, then it is unlikely that k-PROBLEM is *fixed parameter tractable*, i.e. it is unlikely that there exists a constant c such that for any fixed number k, k-PROBLEM is solvable in time $O(f(k)n^c)$.

2.3.3 Series-Parallel Graphs

Series-parallel graphs appear in several applications. For example, if we want to compute the resistance of an electrical network of resistors using Ohm's laws, then the underlying graph of the network must be a series-parallel graph.

A *source-sink labeled graph* is a triple (G, s, t), where G is a multigraph and s and t are distinct vertices of G, called the *source* and *sink* of the graph, respectively.

The *series composition* of two or more source-sink labeled graphs is the operation which takes $r \geq 2$ source-sink labeled graphs $(G_1, s_1, t_1), \ldots, (G_r, s_r, t_r)$ and returns a new source-sink labeled graph (G, s, t) that is obtained by taking the disjoint union of $G_1, \ldots, G_r$, identifying s_{i+1} with t_i for all i, $1 \leq i < r$, and letting $s = s_1$ and $t = t_r$. Figure 2.9 shows the series composition of three source-sink labeled graphs.

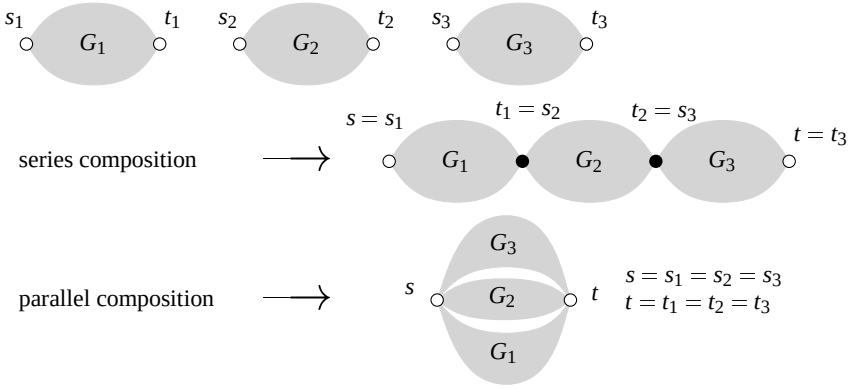


Figure 2.9. A series and a parallel composition of three source-sink labeled graphs $(G_1,s_1,t_1),(G_2,s_2,t_2),(G_3,s_3,t_3)$.

The *parallel composition* of two or more source-sink labeled graphs is the operation which takes $r \geq 2$ source-sink labeled graphs $(G_1,s_1,t_1),\ldots,(G_r,s_r,t_r)$ and returns a new source-sink labeled graph (G,s,t) that is obtained by taking the disjoint union of $G_1,\ldots,G_r$, identifying all vertices $s_1,\ldots,s_r$ into the new source s, and identifying all vertices $t_1,\ldots t_r$ into the new sink t. Figure 2.9 shows the parallel composition of three source-sink labeled graphs.

Definition 2.3.3 (Series Parallel Graphs). A source-sink labeled graph (G,s,t) is a *series-parallel graph* if and only if one of the following holds.

- (G,s,t) is a *base series-parallel graph*, consisting of two vertices s and t with one edge between s and t.
- (G,s,t) is obtained by a series or parallel composition of $r \geq 2$ series-parallel graphs.

Part I of Figure 2.10 shows a series-parallel graph with source s and sink t. An equivalent definition which is often used only involves series and parallel compositions with two series-parallel graphs. A multigraph G is said to be series-parallel if and only if there are vertices $s,t \in V(G)$ such that (G,s,t) is a series-parallel graph.

The 'decomposition' of a series-parallel graph (G,s,t) into series and parallel compositions is expressed in an *sp-tree* T_G of the graph. An sp-tree is a rooted tree, in which each node has one of the types *p-node*, *s-node* and *leaf node*, and has a label. A label of a node is an ordered pair (u,v) of vertices of G. Every node of an sp-tree corresponds to a unique series-parallel graph (G',a,b), where G' is a subgraph of G, and (a,b) is the label of the node. The root of the tree has label (s,t), and corresponds to the graph (G,s,t). The leaves of the tree are of type leaf node, and correspond to the base series-parallel graphs that represent the edges of G: there is a one-to-one correspondence between leaves of T_G and edges $e \in E(G)$.

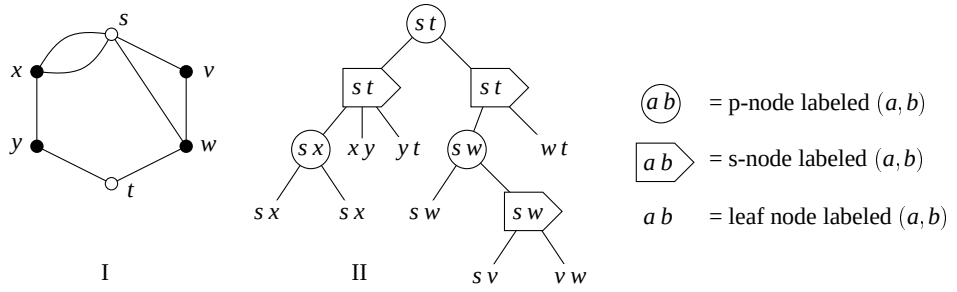


Figure 2.10. A series-parallel graphs and its minimal sp-tree.

Internal nodes are of type s-node (series node) or p-node (parallel node). The children of a series node are ordered, while the children of a parallel node are not ordered. The series-parallel graph associated to an s-node α is the graph that is obtained by a series composition of the series-parallel graphs associated to the children of α, where the order of the children gives the order in which the series composition is applied. The series-parallel graph associated to a p-node β is the graph that is obtained by a parallel composition of the series-parallel graphs associated to the children of β. Note that the children of a p-node have the same label as their parent. Part II of Figure 2.10 shows an sp-tree of the series-parallel graph given in part I.

Note that a series-parallel graph can have different sp-trees. An sp-tree is called a binary sp-tree if each internal node has two children. It can be seen that any series-parallel graph has a binary sp-tree. A *minimal sp-tree* of a series-parallel graph (G,s,t) is an sp-tree of the graph in which p-nodes only have s-nodes and leaf nodes as children, and s-nodes only have p-nodes and leaf nodes as children. Note that the sp-tree in part II of Figure 2.10 is minimal. For each series-parallel graph (G,s,t) there is a unique minimal sp-tree which can be obtained from any sp-tree of (G,s,t) as follows: for any s-node α with another s-node β as child, contract the edge between α and β, and adapt the label. Do the same for any p-node with another p-node as child.

We can also define directed series-parallel graphs. These are defined in the same way as undirected series-parallel graphs, with the sole exception that a base series-parallel graph is a directed graph with two vertices s and t and a directed edge from the source s to the sink t. As a result, directed series-parallel graphs are acyclic, and every vertex lies on a directed path from the source to the sink.

Lemma 2.3.5. *If a multigraph G is series-parallel, then the treewidth of G is at most two.*

Proof. Let $T=(N,F)$ be a binary sp-tree of G. We make a tree decomposition $TD=(X,T)$ of width at most two of G from T with $X=\{X_\alpha \mid \alpha \in N\}$. For each p-node α with label (v,w), let $X_\alpha=\{v,w\}$, and for each s-node α with label (v,w) and labels of its two children (v,x) and

(x,w), let $X_\alpha = \{v,w,x\}$. One can verify that (X,T) is a tree decomposition of G of treewidth at most two. □

If an sp-tree of a series-parallel graph is given, many graph problems can be solved in linear time (in the number of edges) by doing dynamic programming on the sp-tree [Bern et al., 1987; Borie et al., 1992; Kikuno et al., 1983; Takamizawa et al., 1982]. These results also follow from the fact that series-parallel graphs have treewidth at most two.

A *series reduction* in a source-sink labeled graph (G,s,t) is the operation which removes a vertex $v \in V(G)$ of degree two of G, $v \neq s,t$, and adds an (extra) edge between the neighbors of v. A *parallel reduction* in a source-sink labeled graph (G,s,t) is the operation which removes an edge e between two vertices u and v which are connected by two or more edges. The rules for series and parallel reduction are depicted in Figure 2.11.

Figure 2.11. Series and parallel reduction rules.

Duffin [1965] has shown that a source-sink labeled graph (G,s,t) is series-parallel if and only if any sequence of series and parallel reductions eventually leads to a base series-parallel graph. Valdes et al. [1982] have given an $O(m)$ time algorithm for recognizing series-parallel graphs which is based on this characterization. This algorithm also builds an sp-tree of the input graph if it is series-parallel.

In Chapter 8 we give an additional set of reduction rules with which series-parallel graphs can be recognized in parallel in $O(\log m \log^* m)$ time with $O(m)$ operations on an EREW PRAM. This algorithm also returns an sp-tree of the input graph, if it is series-parallel.

Chapter 3

The Structure of Partial Two-Paths

In this chapter, we give a complete characterization of the structure of partial two-paths. This characterization is presented in three steps: we first describe the structure of biconnected partial two-paths (Section 3.2), then the structure of trees of pathwidth two (Section 3.3), and finally the structure of partial two-paths in general (Section 3.4). In Section 3.5, we give a linear time algorithm which, given a graph G, checks whether G has pathwidth at most two, and if so, finds the structure of the graph as described in this chapter. This algorithm is used in Chapter 4. We start with some definitions and preliminary results in Section 3.1.

3.1 Preliminaries

In this section, we give some terminology and preliminary results that are used in this chapter and in Chapter 4.

The graphs we consider in this chapter are simple. Let G be a graph, and $PD = (V_1, \ldots, V_t)$ a path decomposition of G. Let G' be a subgraph of G. The *occurrence* of G' in PD is the subsequence $(V_j, \ldots, V_{j'})$ of PD in which V_j and $V_{j'}$ contain an edge of G', and no node V_i, with $i < j$ or $i > j'$ contains an edge of G', i.e. $(V_j, \ldots, V_{j'})$ is the shortest subsequence of PD that contains all nodes of PD which contain an edge of G'. We say that G' *occurs* in $(V_j, \ldots, V_{j'})$. The vertices of G' are said to occur in $(V_l, \ldots, V_{l'})$ if this sequence is the shortest subsequence of PD containing all vertices of G'.

Let G be a graph and $PD = (V_1, \ldots, V_t)$ a path decomposition of G. Let $1 \leq j \leq t$. We say that a node V_i is on the *left side* of V_j if $i < j$, and on the *right side* of V_j if $i > j$. Let G' be a connected subgraph of G, suppose G' occurs in $(V_l, \ldots, V_{l'})$. We say that G' occurs on the left side of V_j if $l' < j$, and on the right side of V_j if $l > j$. In the same way, we speak about the left and right sides of a sequence $(V_j, \ldots, V_{j'})$, i.e. a node is on the left side of $(V_j, \ldots, V_{j'})$ if it is on the left side of V_j, and a node is on the right side of $(V_j, \ldots, V_{j'})$ if it is on the right side of $V_{j'}$.

The following definition only makes sense if the graph G has pathwidth at most two. An edge e (or vertex v) is an *end edge* (or *end vertex*) of G' if in each path decomposition of width two of G, e (or v) occurs in the leftmost or rightmost end node of the occurrence of G'. An edge e (or vertex v) is a *double end edge* (or *double end vertex*) of G' if in each path decomposition of width two of G, e (or v) occurs in both end nodes of the occurrence of G'.

Let G be a graph, let $PD = (V_1, \ldots, V_t)$ be a path decomposition of G, and let $V' \subseteq V$. Suppose $G[V']$ occurs in $(V_j, \ldots, V_{j'})$, $1 \leq j \leq j' \leq t$. The path decomposition of $G[V']$

induced by PD is denoted by $PD[V']$ and is obtained from the sequence $(V_j \cap V', \ldots, V_{j'} \cap V')$ by deleting all empty nodes and all nodes $V_i \cap V'$, $j \leq i < j'$, for which $V_i \cap V' = V_{i+1} \cap V'$.

Let G be a graph, and let G_1 and G_2 be subgraphs of G such that the union of G_1 and G_2 equals G. Let $PD_1 = (V_1, \ldots, V_t)$ and $PD_2 = (W_1, \ldots, W_{t'})$ be path decompositions of G_1 and G_2. The *concatenation* of PD_1 and PD_2 is denoted by $PD_1 +\!\!+ PD_2$ and is defined as follows.

$$PD_1 +\!\!+ PD_2 = (V_1, \ldots, V_t, W_1, \ldots, W_{t'})$$

Note that $PD_1 +\!\!+ PD_2$ is a path decomposition of G if and only if the vertices of $V(G_1) \cap V(G_2)$ occur in V_t and in W_1.

Lemma 3.1.1. *Let $G = (V,E)$ be a connected partial two-path and let $V' \subseteq V$. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G such that the vertices of V' occur in $(V_j, \ldots, V_{j'})$. On each side of $(V_j, \ldots, V_{j'})$, edges of at most two components of $G[V \Leftrightarrow V']$ occur.*

Proof. Suppose there are edges of at least three components of $G[V \Leftrightarrow V']$ on the left side of V_j. Let G_1, G_2, G_3 be three of these components. Let V_l, $1 \leq l < j$, be the rightmost node on the left side of V_j containing an edge of one of the components G_1, G_2 and G_3, say G_1. V_l contains a vertex of G_2 and of G_3. Hence $|V_l| \geq 4$. □

3.2 Biconnected Partial Two-Paths

We only consider non-trivial biconnected graphs in this section. For the characterization of biconnected partial two-paths, we make use of a result of Bodlaender and Kloks [1993] (see also Kloks [1994]), who gave a characterization of biconnected partial two-trees.

Definition 3.2.1. Given a biconnected graph $G = (V,E)$, the *cell completion* $\bar{G}$ of G is the graph which is obtained from G by adding an edge $\{u,v\}$ for all pairs u,v of vertices in V, $u \neq v$, for which $\{u,v\} \notin E(G)$ and $G[V(G) \Leftrightarrow \{u,v\}]$ has at least three connected components.

Lemma 3.2.1. *Let G be a biconnected graph, let $\bar{G}$ denote the cell completion of G. Let C_1 and C_2 be distinct chordless cycles of $\bar{G}$.*

1. *C_1 and C_2 have at most two common vertices.*
2. *If C_1 and C_2 have two common vertices u and v, then they have the edge $\{u,v\}$ in common.*

Proof. We first prove the following claim: if C_1 and C_2 have two vertices u and v in common, then $\{u,v\}$ is an edge in $\bar{G}$, and furthermore, $\{u,v\} \in E(C_1)$ and $\{u,v\} \in E(C_2)$.

Let P_1 and P_2 be the two paths from u to v in C_1 which are internally vertex-disjoint. There is a path P_3 from u to v in C_2 which is internally vertex-disjoint from both P_1 and P_2. By the definition of cell completion, $\{u,v\} \in E(\bar{G})$. Since C_1 and C_2 are both chordless cycles, it must be the case that $\{u,v\} \in E(C_1)$ and $\{u,v\} \in E(C_2)$. This proves the claim.

Suppose C_1 and C_2 have a set W of vertices in common with $|W| \geq 3$. By the previous claim, W induces a clique in C_1 and in C_2. This is only possible if $|W| = 3$ and C_1 and C_2 are both cycles on the vertices of W. But that means that $C_1 = C_2$, contradiction. Hence C_1 and C_2 have at most two vertices in common, and if they have two vertices in common, then they have the edge between these vertices in common. □

The following lemma has been proved by Bodlaender and Kloks [1993] in the setting of partial two-trees. For the sake of completeness, we give an alternative proof here.

Lemma 3.2.2. *Let G be a biconnected partial two-path. Each path decomposition of width two of G is a path decomposition (of width two) of the cell completion $\bar{G}$ of G.*

Proof. Let $u,v \in V(G)$, $u \neq v$, and suppose $\{u,v\} \notin E(G)$ and $G[V(G) \Leftrightarrow \{u,v\}]$ has at least three connected components. Since G is biconnected, there are three internally vertex-disjoint paths P_1, P_2, P_3 from u to v in G (i.e. for $i \neq j$, P_i and P_j only have vertices u and v in common). We show that in each path decomposition of width two of G, there is a node which contains u and v. Suppose not. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G such that V_i is the rightmost node containing u, V_j is the leftmost node containing v, and $i < j$. Node V_{i+1} contains at least one vertex of each P_l, $1 \leq l \leq 3$, but it does not contain u. But each of these vertices has a path to u, and $u \notin V_{i+1}$. This means that, for each l, $1 \leq l \leq 3$, there is a vertex $v_l \in V(P_l) \Leftrightarrow \{u,v\}$ such that $v_l \in V_i$. This means that $|V_i| \geq 4$, a contradiction. □

Bodlaender and Kloks [1993] gave a linear time algorithm for finding the cell completion of a biconnected partial two-path. In the cell completion of a graph, each two distinct chordless cycles have at most one edge in common. Bodlaender and Kloks [1993] have shown that the cell completion of a biconnected partial two-tree is a 'tree of cycles'. We show that the cell completion of a biconnected partial two-path is a 'path of cycles'. First, we give some definitions and prove a number of lemmas.

Definition 3.2.2 [Bodlaender and Kloks, 1993]. The class of *trees of cycles* is the class of graphs recursively defined as follows.

- Each cycle is a tree of cycles.
- For each tree of cycles G and each cycle C, the graph obtained from G and C by taking the disjoint union and then identifying an edge and its end vertices in G with an edge and its end vertices in C, is a tree of cycles.

Note that two different chordless cycles in a tree of cycles have at most one edge in common.

Definition 3.2.3. A *path of cycles* is a tree of cycles G for which the following holds.

1. Each chordless cycle of G has at most two edges which are contained in other chordless cycles of G.
2. If an edge $e \in E(G)$ is contained in $m \geq 3$ chordless cycles of G, then at least $m \Leftrightarrow 2$ of these cycles have no other edges in common with other chordless cycles, and consist of three vertices.

For an example of a path of cycles, see Figure 3.1. With each path of cycles G, we can associate a sequence $(C_1, \ldots, C_p)$ of all chordless cycles of G and a sequence $(e_1, \ldots, e_{p-1})$ of edges of G, such that for each i, $1 \leq i < p$, cycles C_i and C_{i+1} have edge e_i in common, and furthermore, if $i < p \Leftrightarrow 1$ and $e_i = e_{i+1}$, then C_{i+1} has three vertices.

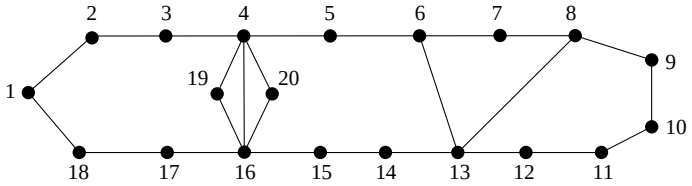


Figure 3.1. A path of cycles.

Definition 3.2.4 (Cycle Path). Let G be path of cycles, let $C = (C_1, \ldots, C_p)$ be a sequence of chordless cycles as defined above, and let $E = (e_1, \ldots, e_{p-1})$ be the corresponding set of common edges. The pair (C, E) is called a *cycle path* for G.

For the path of cycles shown in Figure 3.1, a cycle path consists of 6 cycles. A possible cycle path is (C, E), with $C = (C_1, \ldots, C_6)$ and $E = (e_1, \ldots, e_5)$:

$$V(C_1) = \{1,2,3,4,16,17,18\}, \quad V(C_2) = \{4,16,19\}, \quad V(C_3) = \{4,16,20\},$$
$$V(C_4) = \{4,5,6,13,14,15,16\}, \quad V(C_5) = \{6,7,8,13\}, \quad V(C_6) = \{8,9,10,11,12,13\},$$

and furthermore, $e_1 = e_2 = e_3 = \{4,16\}$, $e_4 = \{6,13\}$, and $e_5 = \{8,13\}$.

Consider a path of cycles G. Suppose there are two distinct cycle paths (C, E) and (C', E') for G, where $C = (C_1, \ldots, C_p)$, $E = (e_1, \ldots, e_{p-1})$ and $C' = (C'_1, \ldots, C'_p)$, $E' = (e'_1, \ldots, e'_{p-1})$. Then either for each i, $1 \leq i < p$, $e_i = e'_i$ or for each i, $e_i = e'_{p-i}$. If the first condition holds, then for each i, $1 \leq i \leq p$, if $C_i \neq C'_i$, then C_i and C'_i both consist of three vertices, and if $1 < i < p$, then $e_{i-1} = e_i$. If the latter condition holds, a similar property can be derived. In other words: two cycle paths for the same path of cycles can only differ in the ordering of cycles which consist of only three vertices, and have the same (only one) edge in common with any other cycles.

In the remainder of this section, we show that a biconnected graph G is a partial two-path if and only if its cell completion is a path of chordless cycles. We first show one side of this equivalence.

Lemma 3.2.3. *Let G be a biconnected graph. If $\bar{G}$ is a path of cycles, then G is a partial two-path.*

Proof. Suppose $\bar{G}$ is a path of cycles and (C, E) is a cycle path for $\bar{G}$, with $C = (C_1, \ldots, C_p)$ and $E = (e_1, \ldots, e_{p-1})$, $p \geq 1$. Then we can make a path decomposition of width two of G as follows. Let e_0 be an arbitrary edge in C_1 with $e_0 \neq e_1$, and let e_p be an arbitrary edge in C_p with $e_p \neq e_{p-1}$. For each i, $1 \leq i \leq p$, we make a path decomposition PD_i of C_i as follows. If $|V(C_i)| = 3$, let $PD_i = (V(C_i))$. Otherwise, do the following. Let $e_{i-1} = \{u, v\}$ and $e_i = \{u', v'\}$ such that there is a path from u to u' which does not contain v or v', Let $P_1 = (u_1, \ldots, u_q)$ denote the path in C_i from u to u' which avoids v and v' (i.e. $u = u_1$ and $u' = u_q$), and let $P_2 = (v_1, \ldots, v_r)$ denote the path in C_i from v to v' avoiding u and u' (i.e.

$v = v_1$ and $v' = v_r$). For each j, $1 \leq j < q$, let $V_j = \{u_j, u_{j+1}, v_1\}$, and for each j, $1 \leq j < r$, let $V_{j+q-1} = \{u_q, v_j, v_{j+1}\}$. Let $PD_i = (V_1, \ldots, V_{q+r-2})$. Note that PD_i is a path decomposition of width two of C_i with $e_{i-1} \subseteq V_1$ and $e_i \subseteq V_{q+r-2}$, and hence $PD = PD_1 +\!\!+ PD_2 +\!\!+ \cdots +\!\!+ PD_p$ is a path decomposition of width two of G, and thus of G. □

As an example, consider the path of cycles of Figure 3.1. Figure 3.2 shows a path decomposition of width two of this graph, made according to the construction of the proof of Lemma 3.2.3, where $e_0 = \{1, 18\}$ and $e_p = \{9, 10\}$.

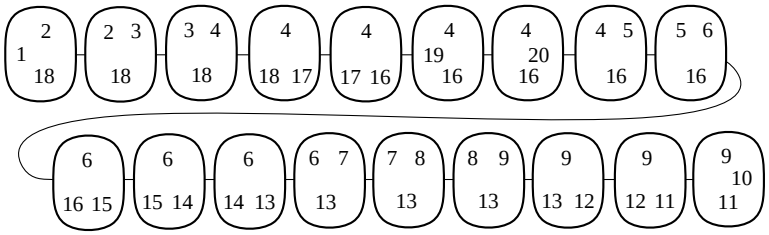


Figure 3.2. A path decomposition of width two for the graph of Figure 3.1.

We now give three technical lemmas in order to prove that if a biconnected graph G has pathwidth two, then its cell completion $\bar{G}$ is a path of cycles. These lemmas show that, in a path decomposition of width two of a biconnected partial two-path, the occurrences of two chordless cycles can overlap in only a very small part.

Lemma 3.2.4. *Let G be a biconnected partial two-path, C a cycle of $\bar{G}$, and $PD = (V_1, \ldots, V_t)$ a path decomposition of G of width two. Suppose C occurs in $(V_j, \ldots, V_{j'})$, $\{u, v\}$ is an edge of C occurring in V_j, and $\{u', v'\}$ an edge occurring in $V_{j'}$. The following holds.*

1. *If $|V(C)| > 3$, then $\{u, v\} \neq \{u', v'\}$.*
2. *For each i, $j \leq i \leq j'$, $|V_i \cap V(C)| \geq 2$ and for each edge $e \in E(C)$ there is an i, $j \leq i \leq j'$, such that $e \subseteq V_i$ and $|V_i \cap V(C)| = 3$.*

Proof.

1. Suppose $|V(C)| > 3$, but $u = u'$ and $v = v'$. There is an edge $\{w, x\}$ in C with $\{w, x\} \cap \{u, v\} = \emptyset$. Because of the definition of path decomposition, there is an i, $j \leq i \leq j'$, such that $w, x \in V_i$, and also $u, v \in V_i$. Hence $|V_i| \geq 4$, contradiction.

2. Suppose w.l.o.g. that u and u' are connected by a path in C which avoids v and v'. Denote this path by P_1. Denote the path between v and v' which avoids u and u' by P_2. See also Figure 3.3. According to Lemma 2.2.2, each V_i, $j \leq i \leq j'$, contains a vertex of P_1. Analogously, each V_i contains a vertex of P_2. Since P_1 and P_2 are vertex-disjoint, $|V_i \cap V(C)| \geq 2$ for each i, $j \leq i \leq j'$. Suppose P_1 contains at least one edge. Let e be an edge of P_1. Let V_l, $j \leq l \leq j'$, be such that $e \subseteq V_l$. Node V_l also contains a vertex of P_2, hence there is an i such that $e \subseteq V_i$ and $|V_i \cap V(C)| \geq 3$ for each edge e on P_1 and P_2. Now consider edge $\{u, v\} \subseteq V_j$.

If there is another vertex of C in V_j, then the lemma holds for $\{u,v\}$. If $V_j \cap V(C) = \{u,v\}$, then there must be an i, $j < i \leq j'$, such that $\{u,v\} \subseteq V_i$ and V_i contains a neighbor of u or v. Hence $|V_i \cap V(C)| = 3$. A similar argument establishes that there is a node V_i with $\{u',v'\} \subseteq V_i$ and $|V_i \cap V(C)| = 3$. □

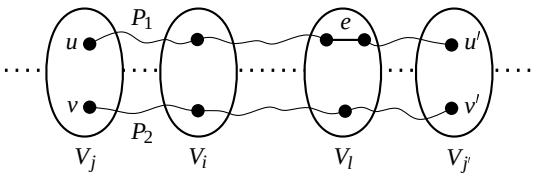


Figure 3.3. The occurrence of chordless cycle C as described in part 2 of the proof of Lemma 3.2.4.

Let G be a biconnected partial two-path. Lemma 3.2.4 implies that the occurrences of two chordless cycles of G that do not have a vertex in common can not overlap in any path decomposition of width two of G.

Lemma 3.2.5. *Let G be a biconnected partial two-path with cycles C and C' which have one vertex u and no other vertices in common. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of G of pathwidth two in which no consecutive nodes are the same. Suppose C occurs in $(V_j, \ldots, V_{j'})$ and C' occurs in $(V_l, \ldots, V_{l'})$. Then either $j' \leq l$ or $l' \leq j$.*

Proof. Assume $j \leq l$. Cycle C contains an edge of which both end points are not vertices of C', and similarly, C' contains an edge of which both end points are not vertices of C. Hence $j < l$ and $j' < l'$.

If $j' < l$, then clearly $j' \leq l$ holds. Suppose $l \leq j'$. For each i, $l \leq i \leq j'$, V_i contains two vertices from C and two vertices from C' (Lemma 3.2.4, part 1), hence $u \in V_i$. Let $\{u,v\} \in E(C)$ such that $u,v \in V_{j'}$. There is a node i', $j \leq i' \leq j'$, which contains u, v and another vertex from C (Lemma 3.2.4, part 2). This is only possible if $i' < l$, since $j' < l'$. Hence, for each i, $l \leq i \leq j'$, V_i contains u and v. In the same way, we can prove that, if $\{u,w\}$ is the edge of C' which occurs in V_l, then $w \in V_i$ for each i, $l \leq i \leq j'$. Hence for each i, $l \leq i \leq j'$, $V_i = \{u,v,w\}$. Since PD contains no consecutive nodes that are the same, this means that $l = j'$. Hence if $j \leq l$, then $j' \leq l$.

By symmetry, $l \leq j$ implies $l' \leq j'$. □

Lemma 3.2.5 shows that, if two cycles C and C' have one vertex in common, then in any path decomposition of pathwidth two their occurrences can overlap in at most one node. We say that C occurs on the left side of C' or C occurs on the right side of C'.

Lemma 3.2.6. *Let G be a biconnected partial two-path with cycles C and C' which have one edge $\{u,v\}$ and no other vertices in common. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of G of pathwidth two. Suppose C occurs in $(V_j, \ldots, V_{j'})$ and C' occurs in $(V_l, \ldots, V_{l'})$. Then the following holds.*

1. *$j \leq l$ and $j' \leq l'$ or $j \geq l$ and $j' \geq l'$. If $j = l$ and $j' = l'$, then $|V(C)| = |V(C')| = 3$.*
2. *Edge $\{u,v\}$ is an end edge of C and C'. If $j \leq l$ and $j' \leq l'$, then*
   - *$j' \geq l$,*
   - *$\{u,v\}$ occurs in $V_{j'}$ and in V_l, and*
   - *there is an i, $l \leq i < j'$, such that $V(C) \cap (V_{i+1} \cup \ldots \cup V_t) = \{u,v\}$ and $V(C') \cap (V_1 \cup \ldots \cup V_i) = \{u,v\}$ (or possibly vice versa, if $j = l$ and $j' = l'$).*

Proof.
1. Suppose $j < l$ and $j' > l'$. Then $|V(C')| = 3$, say $V(C') = \{u,v,w\}$, since each node in $V_j, \ldots, V_{j'}$ contains two vertices of C. Let $j < i < j'$, such that $V_i = \{u,v,w\}$. Suppose $\{x,y\}, \{x',y'\} \in E(C)$ and $\{x,y\} \subseteq V_j$, $\{x',y'\} \subseteq V_{j'}$, such that there is a path from x to x' which avoids y and y'. Let P_1 denote this path, and let P_2 denote the path from y to y' which avoids x and x'. $\{x,y\} \neq \{u,v\}$ and $\{x',y'\} \neq \{u,v\}$, so suppose $\{u,v\} \in E(P_1)$. Node V_i contains a vertex of P_2, which is not u, v or w. Hence $|V_i| \geq 4$, which is a contradiction. So either $j \leq l$ and $j' \leq l'$ or $j \geq l$ and $j' \geq l'$. If $j = l$ and $j' = l'$, then $|V(C)| = |V(C')| = 3$, since each V_i, $j \leq i \leq j'$, contains two vertices of C and two vertices of C'.

2. Suppose that $j \leq l$ and $j' \leq l'$. It is clear that $j' \geq l$, since $\{u,v\}$ is an edge of both C and C'. There are nodes V_m and $V_{m'}$ such that $V_m = \{u,v,w\}$ for some $w \in V(C)$ with $w \neq u,v$, and $V_{m'} = \{u,v,w'\}$ for some $w' \in V(C')$ with $w' \neq u,v$. Note that $l \leq m, m' \leq j'$. Suppose first that $l \leq m < m' \leq j'$. We show that all vertices of $V(C) \Leftrightarrow \{u,v\}$ occur only on the left side of $V_{m'}$. Suppose there is a vertex $x \in V(C) \Leftrightarrow \{u,v\}$ which occurs on the right side of $V_{m'}$. There is a path from x to w in C which avoids u and v. Node $V_{m'}$ contains a vertex of this path. Hence $|V_{m'}| \geq 4$. This is a contradiction. Since each V_i, $m \leq i \leq m'$, contains u and v, this means that there is an i, $m \leq i < m'$, such that all vertices of $V(C) \Leftrightarrow \{u,v\}$ occur only in $(V_1, \ldots, V_i)$, and the vertices of $V(C') \Leftrightarrow \{u,v\}$ occur only in $(V_{i+1}, \ldots, V_t)$. Furthermore, since $i < j'$ and $V_{j'}$ contains an edge of C, $V_{j'}$ contains u and v. Similarly, V_l contains u and v.

Now suppose $l \leq m' < m \leq j'$. In the same way as before, we can show that the vertices of $V(C) \Leftrightarrow \{u,v\}$ occur only on the right side of $V_{m'}$, and the vertices of $V(C') \Leftrightarrow \{u,v\}$ occur only on the left side of V_m. Hence there is an i, $m' \leq i < m$, such that all vertices of $V(C) \Leftrightarrow \{u,v\}$ occur only in $(V_{i+1}, \ldots, V_t)$ and all vertices of $V(C') \Leftrightarrow \{u,v\}$ occur only in $(V_1, \ldots, V_i)$. Furthermore, V_l is the leftmost node which contains an edge of C', which means that $j = l$. In the same way, we can prove that $j' = l'$, and V_l and $V_{j'}$ both contain u and v. Hence $\{u,v\}$ is an end edge. □

Note that in part 2 of the lemma, the part $(V_j, \ldots, V_i)$ of PD restricted to $V(C)$ is a path decomposition of C, and $(V_{i+1}, \ldots, V_{l'})$ restricted to $V(C')$ is a path decomposition of C'. We say that C occurs on the left side of C'. In other words, Lemma 3.2.6 says that, if there are two cycles which have one edge in common, then in each path decomposition of width two one occurs on the left side of the other one.

Let G be a biconnected partial two-path, and let C and C' be two distinct chordless cycles of G. Lemmas 3.2.1, 3.2.4, 3.2.5 and 3.2.6 show that in any path decomposition of width two of G, either C occurs on the left side of C', or C occurs on the right side of C'.

Lemma 3.2.7. *Let G be a biconnected graph. If G is a partial two-path, then $\bar{G}$ is a path of cycles.*

Proof. Suppose G is a biconnected partial two-path. It follows from the result of Bodlaender and Kloks [1993] that $\bar{G}$ is a tree of cycles. That means that we only have to show that conditions 1 and 2 of Definition 3.2.3 hold.

Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G (and hence of $\bar{G}$). By the previous discussion, there is a sequence $(C_1, \ldots, C_p)$ of the chordless cycles of $\bar{G}$, such that for each $1 \leq i < j \leq p$, C_i occurs on the left side of C_j in PD. For each i, j, $1 \leq i < j \leq p$, let $W_{ij} = V(C_i) \cap V(C_j)$.

Claim. *For each i, $1 \leq i \leq p$, at most two edges of C_i are contained in other chordless cycles of $\bar{G}$.*

Proof. Consider a cycle C_j with $j > i$. Note that, by the definition of cell completion, $|W_{ij}| \leq 2$, and if $|W_{ij}| = 2$, then W_{ij} is an edge of both C_i and C_j. Hence each chordless cycle that occurs between C_i and C_j (in PD) also contains the vertices from W_{ij}: for each l, $i \leq l \leq j$, $W_{ij} \subseteq V(C_l)$. This means that $W_{ij} \subseteq V(C_{i+1})$ and hence $W_{ij} \subseteq W_{i(i+1)}$. Hence if C_i has an edge e in common with C_j, then it has this edge in common with C_{i+1}. But C_i has at most one edge in common with C_{i+1}, which means that at most one edge of C_i is contained in any chordless cycle that occurs on C_i's right side. By symmetry, C_i has at most one edge in common with any chordless cycle that occurs on its left side. Hence C_i has at most two edges in common with any other chordless cycle. □

Claim. *If an edge e occurs in $m \geq 3$ chordless cycles, then at least $m \Leftrightarrow 2$ of these cycles have no other edges in common with any other chordless cycle, and consist of only three vertices.*

Proof. Suppose edge e occurs in $m \geq 3$ chordless cycles. Let i, j, $1 \leq i < j \leq p$, be such that C_i is the leftmost cycle containing e and C_j is the rightmost cycle containing e. Then for each l, $i < l < j$, edge e occurs in each node of the occurrence of C_l, and thus, by part 2 of Lemma 3.2.4, $e \in E(C_l)$, and furthermore, by part 1 of Lemma 3.2.4, $|V(C_l)| = 3$. □

This proves the lemma. □

The main result of this section now follows.

Theorem 3.2.1. *Let G be a biconnected graph. G is a partial two-path if and only if $\bar{G}$ is a path of cycles.*

3.3 Trees of Pathwidth Two

The following result, describing the structure of trees of pathwidth k, is similar to a result of Ellis, Sudborough, and Turner [1994].

Lemma 3.3.1. *Let H be a tree and let $k \geq 1$. H is a tree of pathwidth at most k if and only if there is a path $P = (v_1, \ldots, v_s)$ in H such that $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$, i.e. if and only if H consists of a path with trees of pathwidth at most $k \Leftrightarrow 1$ connected to it.*

Proof. If H consists of a path $P = (v_1, \ldots, v_s)$ with trees of pathwidth at most $k \Leftrightarrow 1$ connected to it, then we can make a path decomposition of H as follows. For each i, $1 \leq i \leq s$, make a path decomposition of width at most $k \Leftrightarrow 1$ of all components of $H[V \Leftrightarrow V(P)]$ which are connected to v_i, and add vertex v_i to all nodes in this path decomposition. Let PD_i denote this path decomposition. Now let

$$PD = PD_1 +\!\!+ (\{v_1, v_2\}) +\!\!+ PD_2 +\!\!+ (\{v_2, v_3\}) +\!\!+ \cdots +\!\!+ \{(v_{s-1}, v_s\}) +\!\!+ PD_s.$$

Then PD is a path decomposition of width at most k of H.

Suppose $(V_1, \ldots, V_t)$ is a path decomposition of H of width at most k. Select $v, w \in V$ such that $v \in V_1$ and $w \in V_t$. Let P be the path from v to w in H. Then each V_i, $1 \leq i \leq t$, contains a vertex of P. Hence $PD[V(H) \Leftrightarrow V(P)]$ is a path decomposition of width at most $k \Leftrightarrow 1$ of $H[V \Leftrightarrow V(P)]$. □

A graph has pathwidth zero if and only if it consists of a set of isolated vertices. Because graphs of pathwidth one do not contain cycles, each component of a graph of pathwidth one is a tree which consists of a path with 'sticks', which are vertices of degree one adjacent only to a vertex on the path ('caterpillars with hair length one'). An example of a partial one-path is shown in Figure 3.4.

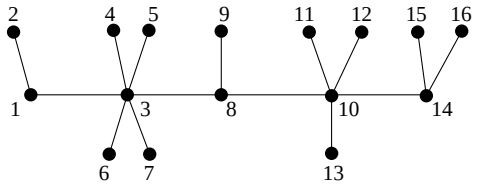


Figure 3.4. Example of a partial one-path.

Lemma 3.3.2. *Let H be a tree of pathwidth k, $k \geq 1$, and suppose there is no vertex $v \in V(H)$ such that $H[V \Leftrightarrow \{v\}]$ has pathwidth $k \Leftrightarrow 1$ or less. Then there is a unique shortest path P in H such that $H[V \Leftrightarrow V(P)]$ has pathwidth $k \Leftrightarrow 1$ or less. Furthermore, P is a subpath of each path P' in H for which $H[V \Leftrightarrow V(P')]$ has pathwidth at most $k \Leftrightarrow 1$.*

Proof. If P is a path in H such that $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$, then all the paths in H containing P have that same property. Suppose there are two distinct paths P and P', such that $H[V \Leftrightarrow V(P)]$ and $H[V \Leftrightarrow V(P')]$ have pathwidth at most $k \Leftrightarrow 1$. We first show that $V(P) \cap V(P') \neq \emptyset$. Suppose $V(P) \cap V(P') = \emptyset$. Let H' be the component of $H[V \Leftrightarrow V(P)]$ which contains P', let H'' be the component of $H[V \Leftrightarrow V(P')]$ which contains P, and let $v \in V(P)$ be the vertex to which H' is connected, i.e. there is a $w \in V(H')$ such that $\{v, w\} \in E(H)$. See Figure 3.5. Consider the components of $H[V \Leftrightarrow \{v\}]$. H' is one of these components, and has pathwidth at most $k \Leftrightarrow 1$. All other components contain no vertex of P', and hence are

subgraphs of H'', which also has pathwidth at most $k \Leftrightarrow 1$. Hence $H[V \Leftrightarrow \{v\}]$ has pathwidth at most $k \Leftrightarrow 1$, a contradiction.

Let P'' be the intersection of P and P', which is again a (non-empty) path. The forest $H[V \Leftrightarrow V(P'')]$ has pathwidth at most $k \Leftrightarrow 1$, since each component of $H[V \Leftrightarrow V(P'')]$ contains no vertices of P or no vertices of P', hence is a component or a subgraph of a component of either $G[V \Leftrightarrow V(P)]$ or $G[V \Leftrightarrow V(P')]$.

This means that the intersection P' of all paths P for which $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$ also has the property that $H[V \Leftrightarrow V(P')]$ has pathwidth at most $k \Leftrightarrow 1$, and it is unique and shorter than all other paths having this property. □

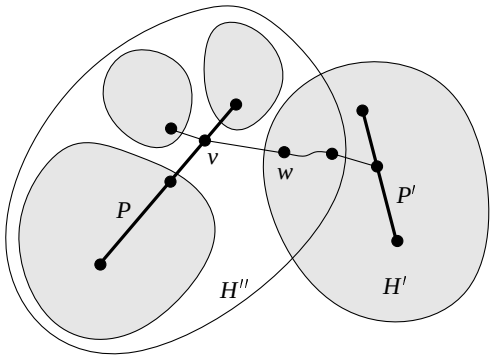


Figure 3.5. Example of a tree of pathwidth k for the proof of Lemma 3.3.2.

Let H be a tree of pathwidth k. In the next two lemmas, we show that for $k = 1$ and $k = 2$, there are at most a constant number of vertices $v \in V(H)$ for which $H[V \Leftrightarrow \{v\}]$ has pathwidth at most $k \Leftrightarrow 1$.

Lemma 3.3.3. *Let H be a tree of pathwidth one, let $W \subseteq V(H)$ consist of all vertices $v \in V(H)$ for which $H[V \Leftrightarrow \{v\}]$ has pathwidth zero, and suppose that $|W| \geq 1$. Then $|W| \leq 2$, and if $|V(H)| > 2$, then $|W| = 1$.*

Proof. Let $v \in W$. Then $H[V \Leftrightarrow \{v\}]$ consists of single vertices. If $|V| = 2$, then G consists of one edge, so $|W| = 2$. If $|V| > 2$, then all (at least two) edges of G are incident with v. Hence for each $w \in V \Leftrightarrow \{v\}$, $H[V \Leftrightarrow \{w\}]$ contains at least one edge incident with v, and does not have pathwidth zero. So if $|V| > 2$, then $|W| = 1$. □

Lemma 3.3.4. *Let H be a tree of pathwidth two and let $W \subseteq V(H)$ consist of all vertices $v \in V(H)$ for which $H[V \Leftrightarrow \{v\}]$ has pathwidth at most one. Suppose $|W| \geq 1$. The following holds.*

1. *$H[W]$ is a connected graph.*

2. *If there is a $v \in W$ such that $H[V \Leftrightarrow \{v\}]$ has four or more components of pathwidth one, then $|W| = 1$.*
3. *There is a vertex $v \in W$ such that $H[V \Leftrightarrow \{v\}]$ has two or more components of pathwidth one.*
4. $|W| \leq 7$.

Proof.
1. Suppose $|W| \geq 2$. Let $v, v' \in W$ be distinct vertices. Let w be a vertex on the path from v to v' in H. Then each component of $H[V \Leftrightarrow \{w\}]$ does not contain v or does not contain v'. Hence each component is a subgraph of a component of $H[V \Leftrightarrow \{v\}]$ or of $H[V \Leftrightarrow \{v'\}]$, so $w \in W$.

2. Let $v \in W$, let H_i, $1 \leq i \leq s$, be the components of $H[V \Leftrightarrow \{v\}]$ which have pathwidth one. Suppose $s \geq 4$. Let $w \in V(H)$ for some $w \neq v$, and let H' be the component of $H[V \Leftrightarrow \{w\}]$ containing v. If $w \in V(H_j)$ for some j, then H' contains all H_i with $i \neq j$. Otherwise, H' contains all H_i. In both cases, H' has pathwidth two, according to Lemma 2.2.4, since v separates H' in three or more components of pathwidth one. Hence $|W| = 1$.

3. Suppose W does not contain a vertex $v \in W$ such that $H[V \Leftrightarrow \{v\}]$ has two or more components of pathwidth one. Let $v \in W$. There is one component of $H[V \Leftrightarrow \{v\}]$ which has pathwidth one, otherwise, H has pathwidth at most one. Let H' be this component, and let $w \in V(H')$ such that $\{v, w\} \in E(H)$. It must be the case that H' contains three or more vertices: otherwise, H has pathwidth one. Let P denote the unique shortest path in H' for which $H'[V \Leftrightarrow V(P)]$ has pathwidth zero (P exists by Lemmas 3.3.2 and 3.3.3). There are two possibilities for w. Either w is an inner vertex of P, or w is a stick of an inner vertex w' of P. In all other cases, H has pathwidth one. If w is an inner vertex of P, then $H[V \Leftrightarrow \{w\}]$ has at least two components of pathwidth one, namely the two components which contain vertices of P. Furthermore, all components of $H[V \Leftrightarrow \{w\}]$ have pathwidth at most one, since all neighbors of v except w have degree one. Hence the component containing v has pathwidth one. If w is a stick of inner vertex w' of P, then $H[V \Leftrightarrow \{w'\}]$ has at least two components of pathwidth one for the same reason, and all components of $H[V \Leftrightarrow \{w'\}]$ have pathwidth one.

4. If W contains a vertex v for which $H[V \Leftrightarrow \{v\}]$ has four or more components of pathwidth one, then $|W| = 1$.

Consider the case that for all $v \in W$, $H[V \Leftrightarrow \{v\}]$ has at most three components of pathwidth one. First suppose W contains a vertex v such that $H[V \Leftrightarrow \{v\}]$ has three components of pathwidth one. Let H_1, H_2 and H_3 denote these components. For all $w \in V$ such that $w \neq v$ and $w \notin V(H_1) \cup V(H_2) \cup V(H_3)$, $H[V \Leftrightarrow \{w\}]$ has a component of pathwidth two, namely the component containing v. Let $w \in H_1$ and suppose $H[V \Leftrightarrow \{w\}]$ has pathwidth at most one. Let H' be the component of $H[V \Leftrightarrow \{w\}]$ containing v. Note that H' contains both H_2 and H_3 as a subgraph, and hence H' has pathwidth one. As both H_2 and H_3 have pathwidth one, $|V(H')| \geq 5$. Hence there is a unique shortest path P in H' for which $H'[V(H') \Leftrightarrow V(P)]$ has pathwidth zero. This path contains at least one vertex of H_2 since H_2 has pathwidth one.

Similarly, it contains at least one vertex of H_3. This implies that $v \in V(P)$ and that v is an inner vertex of P. Then either w is adjacent to v or there is a vertex $w' \in V(H_1)$ such that w' has degree two in H and w and v are its only neighbors. In both cases, there are at most two possible vertices $u \in V(H_1)$ for which $H[V \Leftrightarrow \{u\}]$ has pathwidth one. For H_2 and H_3 the same holds, and hence $|W| \leq 7$.

Now suppose W contains no vertex $v \in W$ such that $H[V \Leftrightarrow \{v\}]$ has three components of pathwidth one. Let $v \in W$ such that $H[V \Leftrightarrow \{v\}]$ has two components of pathwidth one. Let H_1 and H_2 be the components of $H[V \Leftrightarrow \{v\}]$ which have pathwidth one, and let $w_1 \in V(H_1)$ and $w_2 \in V(H_2)$ such that $\{v, w_1\}, \{v, w_2\} \in E(H)$. It must be the case that $|V(H_1)| \geq 3$ and $|V(H_2)| \geq 3$, otherwise H has pathwidth one. For $i = 1, 2$, let P_i denote the unique shortest path in H_i for which $H_i[V(H_i) \Leftrightarrow V(P_i)]$ has pathwidth zero. Then for $i = 1, 2$, w_i is either an inner vertex or a stick adjacent to an inner vertex of P_i, since otherwise either H does not have pathwidth two, or W contains a vertex w such that $H[V \Leftrightarrow \{w\}]$ has three components of pathwidth one. For each $w \in W$ with $w \neq v$ and $w \notin V(H_1) \cup V(H_2)$, $H[V \Leftrightarrow \{w\}]$ has pathwidth two. If w_1 is inner vertex of P_1 and v has degree two, then w_2 is the only vertex in H_2 for which $H[V \Leftrightarrow \{w_2\}]$ has pathwidth one, otherwise, there is no such vertex in H_2. Similar for w_1. Hence $|W| \leq 3$. This completes the proof. □

Note that the bound $|W| \leq 7$ is sharp: in Figure 3.6, the tree H has pathwidth two and for each vertex $v \in V(H)$ it holds that $H[V \Leftrightarrow \{v\}]$ has pathwidth one.

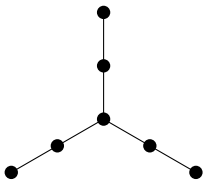


Figure 3.6. A tree of pathwidth two. Removing any vertex results in a graph of pathwidth one.

For $k \geq 3$, the number of vertices in a tree of pathwidth k whose removal decreases the pathwidth by at least one, is not necessarily bounded. For instance, Figure 3.7 shows a tree of pathwidth three. For each i, $1 \leq i \leq m$, the removal of vertex v_i leaves a forest of pathwidth two. But m may be arbitrarily large.

Definition 3.3.1. Let H be a tree and let $k \geq 1$. $P_k(H)$ denotes the set of all paths P in H for which $H[V \Leftrightarrow V(P)]$ is a partial $(k \Leftrightarrow 1)$-path, and there is no strict subpath P' of P for which $H[V \Leftrightarrow V(P')]$ is a partial $(k \Leftrightarrow 1)$-path. If $|P_k(H)| = 1$, then $P_k(H)$ denotes the unique element of $P_k(H)$.

Let H be a tree and let $k \geq 1$. Note that if H has pathwidth more than k, then $P_k(H) = \phi$. If H has pathwidth less than k, then $|P_k(H)| = 1$ and $P_k(H) = ()$. If H has pathwidth exactly

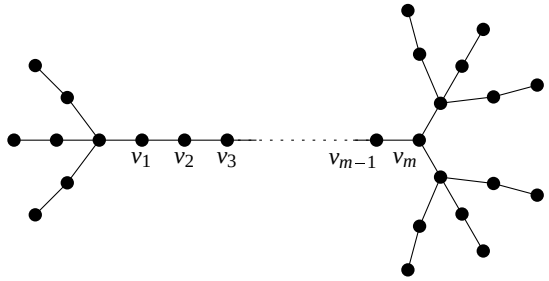


Figure 3.7. A tree of pathwidth three. Removing any vertex v_i, $1 \leq i \leq m$, results in a forest of pathwidth two.

k then $|P_k(H)| \geq 1$ and all paths in $P_k(H)$ contain at least one vertex. If $P_k(H)$ contains more than one element, then its elements are all paths consisting of one vertex.

For a tree of pathwidth one, all path decompositions of width one are essentially the same.

Lemma 3.3.5. *Let $H = (V,E)$ be a tree of pathwidth one and let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width one of H. Suppose $|V(H)| > 2$, and let $P_1(H) = (v_1, \ldots, v_s)$. For each $e \in E(H)$, let $f(e)$ be such that $V_{f(e)}$ is the leftmost node containing e. If $s \geq 3$, then either for each i, $1 \leq i < s \Leftrightarrow 1$, $f(\{v_i, v_{i+1}\}) < f(\{v_{i+1}, v_{i+2}\})$, or for each i, $f(\{v_i, v_{i+1}\}) > f(\{v_{i+1}, v_{i+2}\})$. Suppose the first case holds. Then for each i, $1 \leq i \leq s$, and each $w \in V(H) \Leftrightarrow V(P_1(H))$ such that $\{v_i, w\} \in E(H)$, the following holds. If $i < s$, then $f(\{v_i, w\}) < f(\{v_i, v_{i+1}\})$, and if $i > 1$, then $f(\{v_i, w\}) > f(\{v_{i-1}, v_i)$.*

Proof. Straightforward from the definition of path decomposition. □

In Figure 3.8, a path decomposition of the partial one-path of Figure 3.4 is given.

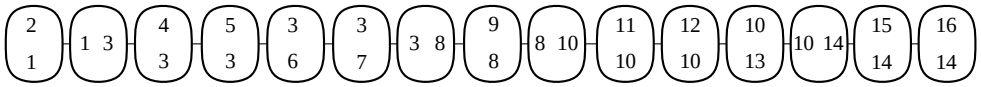


Figure 3.8. A path decomposition of width one of the partial one-path of Figure 3.4.

Lemma 3.3.6. *Let $k \geq 1$ and let H be a tree of pathwidth k such that there is a $v \in V(H)$ for which $H[V \Leftrightarrow \{v\}]$ has pathwidth at most $k \Leftrightarrow 1$. For each path P in H for which $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$, there is a $v \in V(P)$ such that $H[V \Leftrightarrow \{v\}]$ has pathwidth at most $k \Leftrightarrow 1$.*

Proof. Let P be a path in H for which $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$. Let $v \in V(H)$ be such that $H[V \Leftrightarrow \{v\}]$ has pathwidth $k \Leftrightarrow 1$. Suppose $v \notin V(P)$. Let H' denote the component of $H[V \Leftrightarrow V(P)]$ containing v. Let $v' \in V(P)$ be such that there is a $w \in V(H')$

for which $\{v', w\} \in E(H)$. We show that $H[V \Leftrightarrow \{v'\}]$ has pathwidth at most $k \Leftrightarrow 1$. The components that do not contain a vertex of P have pathwidth at most $k \Leftrightarrow 1$ because they are components of $H[V \Leftrightarrow V(P)]$. All other components are subgraphs of the component of $H[V \Leftrightarrow \{v\}]$ which contains P. Hence these components also have pathwidth at most $k \Leftrightarrow 1$. □

Together with Lemma 3.3.2, Lemma 3.3.6 implies that if $|P_k(H)| = 1$, then $P_k(H)$ is the intersection of all paths P for which $H[V \Leftrightarrow V(P)]$ has pathwidth at most $k \Leftrightarrow 1$. Furthermore, it implies the following result, which will be frequently used in the next section.

Corollary 3.3.1. *Let $k \geq 1$, let H be a tree of pathwidth k, and let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width k of H. Let $v \in V_1$ and $v' \in V_t$. Then the path P from v to v' contains one of the paths in $P_k(H)$ as a subpath.*

3.4 General Graphs

In this section, we denote by a block a non-trivial block. A graph may contain a number of blocks. If all edges which are contained in a block are removed, then the resulting graph is a forest.

Definition 3.4.1. Let G be a graph. The subgraph G_{tr} is the graph obtained from G by deleting all edges of blocks of G. The components of G_{tr} are called the trees of G.

The cell completion of a graph G is the graph obtained from G by replacing each block by its cell completion. It is denoted by $\overline{G}$. Let G be a partial two-path. Clearly, the cell completion of each block of G is a path of cycles, and each tree of G is a tree of pathwidth at most two. The number of possible ways in which blocks and trees of G can be connected to each other is large. In this section, we give a complete description of this structure. First we show that for each tree H of G, the vertices of H which are contained in a block of G all lie on one path, which also contains a path of $P_2(H)$. After that, we give for each block of $\overline{G}$ all possible interconnections with other blocks and trees of $\overline{G}$.

Lemma 3.4.1. *Let G be a partial two-path and H a tree of G. Let $V' \subseteq V(H)$ be the set of vertices which are vertices of blocks of $\overline{G}$. There is a path in H which contains all vertices of V' and a path of $P_2(H)$.*

Proof. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G, and suppose the vertices of H occur in $(V_j, \ldots, V_{j'})$. Select $v \in V_j$ and $v' \in V_{j'}$ such that $v, v' \in V(H)$. Let P denote the path from v to v'. All vertices of V' are on P, since for each $w \in V'$, there is a cycle C which contains w, hence there is a node V_i, $j \leq i \leq j'$, such that V_i contains w and two other vertices of C, so $V_i \cap V(H) = \{w\}$. Furthermore, there is a path in $P_2(H)$ which is a subpath of P. □

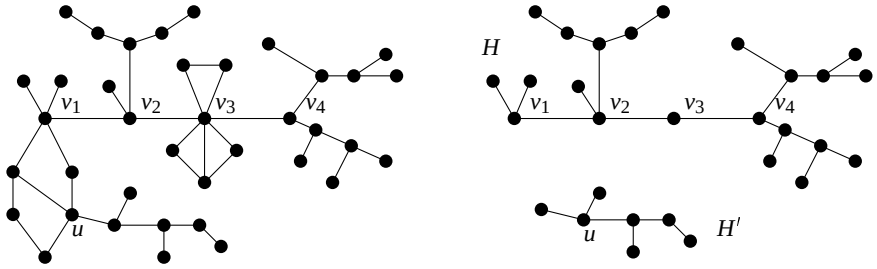
Definition 3.4.2. Let G be a partial two-path and H a tree of G. Let $V' \subseteq V(H)$ be the set of all vertices of H which are contained in a block of G. P_H denotes the set of all paths P in H for which the following conditions hold:

1. there is a path in $P_2(H)$ which is a subpath of P (if $P_2(H) \neq \phi$),
2. $V' \subseteq V(P)$, and
3. there is no strict subpath P' of P for which conditions 1 and 2 hold.

If $|P_H| = 1$, then P_H denotes the unique element of P_H, and P_H is called *the path* of H.

Let G be a partial two-path and H a tree of G. If $|P_2(H)| = 1$, then $|P_H| = 1$. If $|P_2(H)| > 1$, then all elements of $P_2(H)$ are paths consisting of one vertex, and all these vertices form a connected subgraph H' of H (Lemma 3.3.4). This means that if there is one vertex $v \in V(H)$ for which v is contained in a block, then there is a unique shortest path containing v and a path from $P_2(H)$, since one of the vertices of H' is closer to v than the others. If there are two or more vertices of H which are contained in a block, then a similar argument holds. Hence $|P_H| = 1$ if there is at least one vertex of H which is contained in a block of G.

Figure 3.9 shows a partial two-path G in which G has one tree H of pathwidth two, one tree H' of pathwidth one, and a number of trees of pathwidth zero, i.e. isolated vertices. For H, $P_2(H) = (v_2, v_3, v_4)$ and $P_H = (v_1, \ldots, v_4)$, and for H', $P_2(H') = \phi$ and $P_{H'} = (u)$.



G G_{tr}, after deletion of isolated vertices

Figure 3.9. A partial two-path G and its two trees of pathwidth at least one.

From the proof of Lemma 3.4.1 it can be seen that an analog of Corollary 3.3.1 also holds for P_H.

Corollary 3.4.1. *Let G be a connected partial two-path which is not a tree and let H be a tree of G. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G and suppose H occurs in $(V_j, \ldots, V_{j'})$. There is a $v \in V_j \cap V(H)$ and a $v' \in V_{j'} \cap V(H)$ such that the path from v to v' contains P_H.*

The following lemma shows some conditions for the structure of blocks of a partial two-path G.

Lemma 3.4.2. *Let G be a connected partial two-path which is not a tree, let H be a component of G_{tr}, and let $P_H = (v_1 \ldots, v_s)$ be the path of H. Let $G' = G[V \Leftrightarrow V(P_H)]$. At most two*

components of G' have pathwidth two. For each component G'' of G' of pathwidth two, there is a $v \in V(G')$ such that either $\{v, v_1\} \in E(G)$ or $\{v, v_s\} \in E(G)$, i.e. G'' is connected to v_1 or v_s. If $s > 1$, then at most one component of pathwidth two is connected to v_1, and at most one to v_s.

Proof. Because of Lemma 2.2.4, at most two components of G' may have pathwidth two. If there is a component of width two adjacent to v_i, $1 < i < s$, then v_i is a vertex which separates G into three or more components of width two, and hence G has pathwidth three. If $s \neq 1$ and there are two or more components of width two adjacent to v_1, or if $s = 1$ and there are three or more components of width two adjacent to v_1, then v_1 separates G into three components of width two, and hence G has pathwidth three. □

For the vertices of each block of a partial two-path, we define states, which reflect the structure of the subgraphs which are connected to them. In Figure 3.10, an example is given for all possible states.

Definition 3.4.3 (Vertex States). Let G be a partial two-path, and B a block of G. Let $v \in V(B)$, and let H denote the tree of G containing v. The *(vertex) state* of v, denoted by $st(v)$, is the element of the set $\{\mathsf{N}, \mathsf{S}, \mathsf{E1}, \mathsf{I1}, \mathsf{E2}, \mathsf{I2}\}$ defined as follows.

$st(v) = \mathsf{N}$ if v has no neighbors outside of B (vertex v_1 in Figure 3.10).

$st(v) = \mathsf{S}$ if v has only neighbors of degree one outside of B: only sticks are connected to v (vertex v_2 in Figure 3.10).

$st(v) = \mathsf{E1}$ if H has pathwidth one, $P_H = (v)$, v is adjacent to exactly one vertex $w \notin B$ which does not have degree one and $w \in V(H)$, and either v or w is end point of $P_1(H)$ (vertex v_3 in Figure 3.10).

In other words, $st(v) = \mathsf{E1}$ if B is the only block containing v, H has pathwidth one and contains at least one edge which is not incident with v (hence $|P_1(H)| = 1$), $P_H = (v)$, and v is not an inner vertex of $P_1(H)$, but there is a path in H containing v and $P_1(H)$.

$st(v) = \mathsf{I1}$ if B is the only block containing v, H has pathwidth one and contains at least one edge which is not incident with v, $P_H = (v)$, and v is an inner vertex of $P_1(H)$ (vertex v_4 in Figure 3.10).

$st(v) = \mathsf{E2}$ if at least one of the following conditions holds.

- There is another block containing v (vertex v_5 in Figure 3.10).
- Tree H has pathwidth one, $P_H = (v)$ and there is no path in H containing v and a path of $P_1(H)$ (vertex v_6 in Figure 3.10).
- Tree H has pathwidth one, $P_H \neq (v)$, and v is end point of P_H (vertex v_7 in Figure 3.10).
- Tree H has pathwidth two and v is an end point of P_H (vertex v_8 in Figure 3.10).

$st(v) = \mathsf{I2}$ if H has pathwidth at most two and v is an inner vertex of P_H (vertex v_9 in Figure 3.10).

The states are ordered in the following way: I2 $\succ$ E2 $\succ$ I1 $\succ$ E1 $\succ$ S $\succ$ N.

Note that all possibilities are covered for v, and that all states are well-defined. In the remainder of this section, we derive what combinations of states are possible for the vertices of a block.

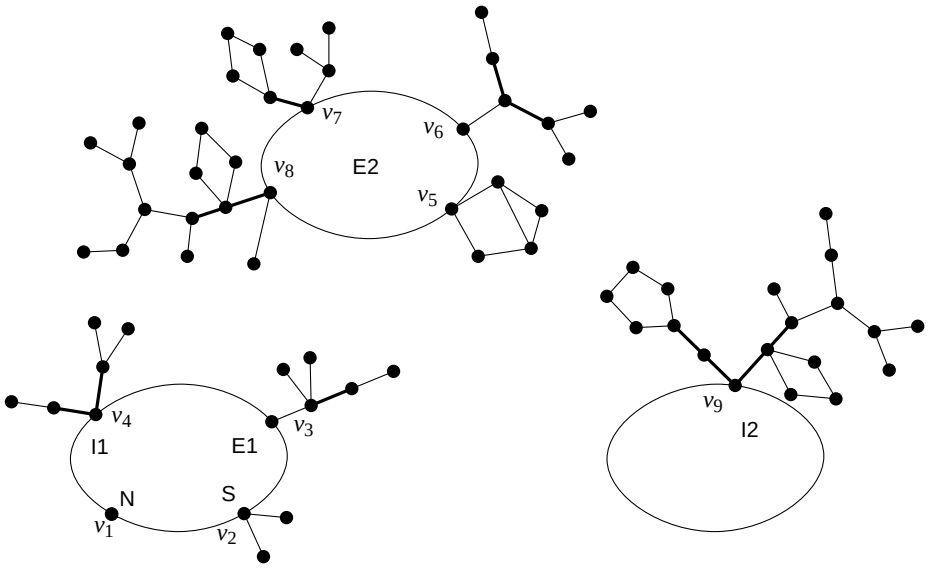


Figure 3.10. Examples of all vertex states. $st(v_1) = \mathsf{N}$, $st(v_2) = \mathsf{S}$, $st(v_3) = \mathsf{E1}$, $st(v_4) = \mathsf{I1}$, $st(v_5) = st(v_6) = st(v_7) = st(v_8) = \mathsf{E2}$ and $st(v_9) = \mathsf{I2}$. For each i, let H_i denote the component of G_{tr} which contains v_i. H_1 and H_5 consist of one single vertex. For $i \in \{3,4,6\}$, the fat edges in H_i form the path $P_1(H_i)$. For $i \in \{7,8,9\}$, the fat edges in H_i form the path P_{H_i}. For $i \in \{1,\ldots,6\}$, $P_{H_i} = (v_i)$.

Lemma 3.4.3. *Let G be a partial two-path and C a cycle in G. Let $v \in V(C)$, and let G' be a component of $G[V \Leftrightarrow V(C)]$ for which there is a vertex $u \in V(G')$ such that $\{v,u\} \in E(G)$. If G' contains at least one edge, then v is an end vertex of C.*

Proof. Let $PD = (V_1,\ldots,V_t)$ be a path decomposition of width two of G, suppose C occurs in $(V_j,\ldots,V_{j'})$, and let $\{x,y\} \in E(C)$ such that $x,y \in V_j$. Suppose $E(G') \neq \emptyset$, let $\{u,w\} \in E(G')$. Edge $\{u,w\}$ can not occur in $(V_j,\ldots,V_{j'})$, so suppose $\{u,w\}$ occurs in V_l, $l < j$. Then either $v \in V_j$ or $u \in V_j$. Consider the case that $u \in V_j$, and let V_p, $j \leq p \leq j'$, be the leftmost node containing v. Then each node in $V_j,\ldots,V_p$ contains u. Furthermore, there is a node containing x, y, and another vertex of C (Lemma 3.2.4), which means that $x,y \in V_p$. This is only possible if $v = x$ or $v = y$, which means that v is an end vertex. □

We can now show that for a block B of the cell completion of a partial two-path G, there is a cycle path (C,E) with $C=(C_1,\ldots,C_p)$ such that all vertices of B which have state E1, I1, E2 or I2, are in C_1 or C_p, and all vertices v which are in some C_i with $1<i<p$ and with $e_i=e_{i+1}$ and $v\notin e_i$ have state N.

Definition 3.4.4 (Correct Cycle Path). Let G be a partial two-path, let B be a block of $\bar{G}$, and let (C,E) be a cycle path for B with $C=(C_1,\ldots,C_p)$ and $E=(e_1,\ldots,e_{p-1})$. (C,E) is called a *correct cycle path* if

- for all $v\in V(B)$ which are not in C_1 or C_p, $st(v)\in\{\mathsf{N},\mathsf{S}\}$, and
- for all i, $1\leq i<p\Leftrightarrow 1$ and all $v\in V(C_{i+1})$, if $e_i=e_{i+1}$ and $v\notin e_i$, then $st(v)=\mathsf{N}$.

Lemma 3.4.4. *Let G be a partial two-path. There is a correct cycle path for each block B of $\bar{G}$.*

Proof. Let $PD=(V_1,\ldots,V_t)$ be a path decomposition of width two of G, and suppose B occurs in $(V_j,\ldots,V_{j'})$. As shown in Section 3.2, the chordless cycles occur in some order $C=(C_1,\ldots,C_p)$. Let $E=(e_1,\ldots,e_{p-1})$ be the sequence of edges of B for which $e_i=V(C_i)\cap V(C_{i+1})$ for each i, $1\leq i<p$. It can be seen that (C,E) is a cycle path for B.

Let C_i be such that $e_{i-1}=e_i$, let $v\in V(C_i)\Leftrightarrow e_i$. Then $st(v)=\mathsf{N}$, since e_i occurs in both end nodes of the occurrence of C_i, and hence any edge adjacent to v could not occur within the occurrence of C_i, and not within the occurrence of any other C_j.

Finally, we prove that all vertices of the component that are not in $V(C_1)$ or $V(C_p)$ are not adjacent to anything other than sticks. Let $v\in V(B)$, such that v does not have state N or S. Let C be the cycle in B with $V(C)$ the set of vertices of $V(B)$ except all $v\in V(B)$ for which $v\in V(C_i)\Leftrightarrow e_i$ for some i, $1<i<p$, for which $e_{i-1}=e_i$, and $E(C)$ the set of edges in $B[V(C)]$ except the edges e_i, $1\leq i<p$. Then v is an end vertex of C. C occurs within $(V_j,\ldots,V_{j'})$, and V_j and $V_{j'}$ can not contain any vertices of B which are not in C_1 or C_p, which is a contradiction. □

Consider a block B of the cell completion of a partial two-path G. Suppose there are two distinct correct cycle paths (C,E) and (C',E') for B, where $C=(C_1,\ldots,C_p)$, $E=(e_1,\ldots,e_{p-1})$ and $C'=(C'_1,\ldots,C'_p)$, $E'=(e'_1,\ldots,e'_{p-1})$ $(p>1)$. Suppose w.l.o.g. that for each i, $1\leq i<p$, $e_i=e'_i$. We have already seen that, for each i, $1\leq i\leq p$, if $C_i\neq C'_i$, then C_i and C'_i both consist of three vertices, and if $1<i<p$, then $e_{i-1}=e_p$ (see page 40). It also holds that both the vertex of C_i that is not in e_i (or e_{i-1} if $i=p$), and the vertex of C'_i that is not in e'_i (or e'_{i-1} if $i=p$) have state N.

From Lemma 3.1.1, we can derive that there are at most four vertices of B which have state E1, I1, E2 or I2. Furthermore, if (C,E) is a correct cycle path, then both $V(C_1)\Leftrightarrow V(C_p)$ and $V(C_p)\Leftrightarrow V(C_1)$ each have at most two vertices with state in $\{\mathsf{E1},\mathsf{I1},\mathsf{E2},\mathsf{I2}\}$.

Let G be a partial two-path, and B a block of $\bar{G}$ with $v\in V(B)$ and $st(v)\in\{\mathsf{I2},\mathsf{E2},\mathsf{I1},\mathsf{E1}\}$. Let X be a component of $G[V\Leftrightarrow V(B)]$ which is connected to v in G such that $|V(X)|>1$, and let X' denote $G[V(X)\cup\{v\}]$. Then in each path decomposition of width two of G, all edges of X' occur on the same side of the occurrence of B: if two edges $e,e'\in E(X')$ occur

on different sides of the occurrence of B, then there is a path between e and e' which avoids v, hence each node in the occurrence of B contains a vertex of this path, which is not possible since B has pathwidth two.

Lemma 3.4.5. *Let G be a partial two-path and C a cycle of $\bar{G}$. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G and suppose C occurs in $(V_j, \ldots, V_{j'})$. Let $v \in V_j$ such that $v \in V(C)$. Then V_j contains a neighbor of v in C.*

Proof. Let $\{x,y\} \in E(C)$ be such that $x,y \in V_j$. Let V_m, $j \leq m \leq j'$, be the leftmost node which contains another edge of C. Then V_m contains x, y and a neighbor z of x or y in C. Then either $m = j$ and $v = z$ or $v \in \{x,y\}$. □

In the next lemmas, we show that the vertices which have state E1, I1, E2 or I2 must have a 'small distance' to each other.

Lemma 3.4.6. *Let G be a partial two-path and B a block of $\bar{G}$. Let PD be a path decomposition of width two of G such that B occurs in $(V_j, \ldots, V_{j'})$, and let (C, E) be a correct cycle path for B such that C_1 is the chordless cycle of B that occurs leftmost in PD.*

Let $x,y \in V(B)$, and suppose $st(x), st(y) \in \{\mathsf{I2}, \mathsf{E2}, \mathsf{I1}, \mathsf{E1}\}$. Let X' be the graph consisting of all components of $G[V \Leftrightarrow V(B)]$ that are connected to x in G, and that occur on the left side of $(V_j, \ldots, V_{j'})$, and let X denote $G[V(X') \cup \{x\}]$. Similarly, let Y' be the graph consisting of all components of $G[V \Leftrightarrow V(B)]$ that are connected to y in G, and that occur on the left side of $(V_j, \ldots, V_{j'})$, and let Y denote $G[V(Y') \cup \{y\}]$ (see for example Figure 3.11). Then $x,y \in V(C_1)$ and

1. *either $\{x,y\} \in E(C_1) \Leftrightarrow \{e_1\}$ or there is a vertex $z \in V(B)$ such that $\{x,z\} \in E(C_1) \Leftrightarrow \{e_1\}$ and $\{z,y\} \in E(C_1) \Leftrightarrow \{e_1\}$ and $st(z) = \mathsf{N}$, and*
2. *either X is a partial one-path such that x is not an inner vertex of $P_1(X)$ but there is a path containing $P_1(X)$ and x, or Y is a partial one-path such that y is not an inner vertex of $P_1(Y)$ but there is a path containing $P_1(Y)$ and y.*

Proof.

1. Both x and y occur in V_j, so $x,y \in V(C_1)$. There is a neighbor of x in V_j and a neighbor of y in V_j. This means that, according to Lemma 3.4.5, either $\{x,y\} \in E(C_1)$ or there is a $z \in V(C_1)$ such that $\{x,z\} \in E(C_1)$ and $\{y,z\} \in E(C_1)$. If $\{x,y\} = e_1$, then $\{x,y\}$ is a double end edge of C_1, hence $|V(C_1)| = 3$, so there is a $z \in V(C_1)$ such that $\{x,z\}, \{y,z\} \in E(C_1) \Leftrightarrow \{e_1\}$. If there is a $z \in V(C_1)$ such that $\{x,z\} \in E(C_1)$ and $\{y,z\} = e_1$, then e_1 also is a double end vertex, hence $|V(C_1)| = 3$, and $\{x,y\} \in E(C_1) \Leftrightarrow \{e_1\}$.

   Suppose $\{x,y\} \notin E(C_1) \Leftrightarrow \{e_1\}$, and let z be the common neighbor of x and y such that $V_j = \{x,y,z\}$. Let V_i, $i < j$, be the rightmost node containing an edge of X' or Y'. Then $V_i = \{x,y,z'\}$ for some $z' \in V(X') \cup V(Y')$. This means that no edge that occurs on the left side of V_j is incident with z. In the same way, we can prove that there can be no edge incident with z which occurs on the right side of V_j.

2. Suppose X occurs in $(V_l, \ldots, V_{l'})$, $1 \leq l \leq l' \leq j$, and Y occurs in $(V_m, \ldots, V_{m'})$, $1 \leq m \leq m' \leq j$, and suppose that $m < l$. See also part II of Figure 3.11. It is clear that $x \in V_{l'}$ and

$y \in V_{m'}$, and that X has pathwidth one. Furthermore, the rightmost node containing an edge of X contains an end point v of the path $P_1(X)$ and a stick v' adjacent to it. This means that $x \in \{v, v'\}$, hence x is either an end point of $P_1(X)$ or a stick adjacent to an end point of $P_1(X)$. If $l < m$, then we get the same result for Y. □

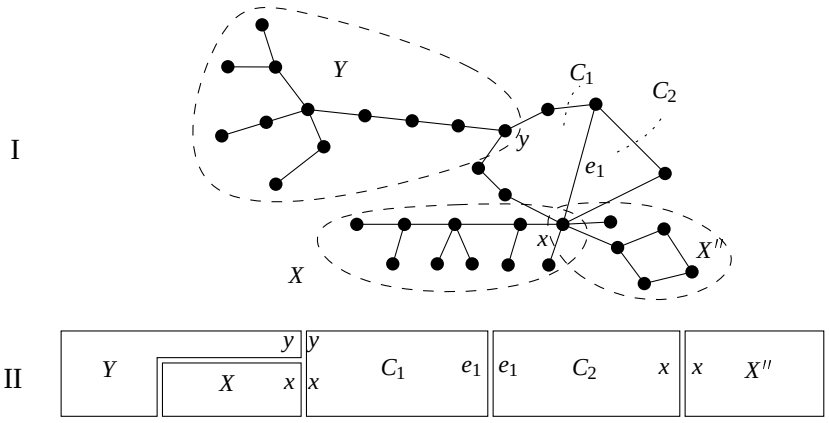


Figure 3.11. Part I is a partial two-path G which contains a path with cycles with correct cycle path (C,E) with $C = (C_1,C_2)$, $E = (e_1)$. Vertices $x,y \in V(C_1)$ both have state E2. Part II shows the order of the occurrences of C_1, C_2, X, Y and X'' in a possible path decomposition of width two of G, as it is used for the proof of Lemma 3.4.6.

Corollary 3.4.2. *Let G be a partial two-path, B a block of $\bar{G}$, and (C,E) a correct cycle path for B. Let $v_1,\ldots,v_s \in V(B)$ such that $st(v_i) \in \{\mathsf{I2},\mathsf{E2},\mathsf{I1}\}$. Then $s \leq 3$. Furthermore, if $s = 3$, there is a j, $1 \leq j \leq 3$, such that $st(v_j) = \mathsf{I1}$ and v_j is a double end vertex of B, which implies that $v_j \in e_i$ for each i.*

To be able to give the possible states for the blocks in a partial two-path, we first give a definition.

Definition 3.4.5 (Distance). Let G be a partial two-path, B a block of $\bar{G}$ and (C,E) a correct cycle path for B with $C = (C_1,\ldots,C_p)$ and $E = (e_1,\ldots,e_{p-1})$. For each $u,v \in V(B)$, $\mathrm{dst}_1(u,v) \in \{\mathsf{true},\mathsf{false}\}$ and $\mathrm{dst}_p(u,v) \in \{\mathsf{true},\mathsf{false}\}$ are defined as follows. If $p = 1$, then

$$\mathrm{dst}_1(u,v) \Leftrightarrow$$
$$u,v \in V(C_1) \wedge (\{u,v\} \in E(C_1) \vee \exists_{w \in V(C_1)} \{u,w\},\{v,w\} \in E(C_1) \wedge st(w) = \mathsf{N})$$

If $p > 1$ then

$$\begin{aligned}
&\mathrm{dst}_1(u,v) \Leftrightarrow u,v \in V(C_1) \wedge \\
&\qquad (\{u,v\} \in E(C_1) \Leftrightarrow \{e_1\} \vee \exists_{w \in V(C_1)} \{u,w\},\{v,w\} \in E(C_1) \Leftrightarrow \{e_1\} \wedge st(w) = \mathsf{N}) \\
&\mathrm{dst}_p(u,v) \Leftrightarrow u,v \in V(C_p) \wedge \\
&\qquad (\{u,v\} \in E(C_p) \Leftrightarrow \{e_{p-1}\} \vee \exists_{w \in V(C_p)} \{u,w\},\{v,w\} \in E(C_p) \Leftrightarrow \{e_{p-1}\} \wedge st(w) = \mathsf{N})
\end{aligned}$$

Figure 3.12 shows an example of dst. The picture shows a path of cycles with cycle path (C,E) with $C = (C_1,C_2)$, $E = (e_1)$. $\mathrm{dst}_2(v_2,v_3)$ and $\mathrm{dst}_2(v_3,v_4)$ hold. $\mathrm{dst}_1(v_2,v_4)$ and $\mathrm{dst}_2(v_2,v_4)$ do not hold, since the edge between v_2 and v_4 is edge e_1. $\mathrm{dst}_1(v_1,v_4)$ does not hold since the common neighbor of v_1 and v_4 has state S.

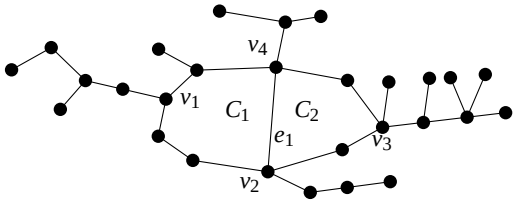


Figure 3.12. Example for the definition of $\mathrm{dst}_1(v_i,v_j)$ and $\mathrm{dst}_p(v_i,v_j)$.

In Definition 3.4.6, we are going to define the state of a block of a partial two-path. Furthermore, for each state, a necessary condition will be given which must hold for any block of that state, such that the graph can be a partial two-path. The necessity of these conditions is proved in Lemma 3.4.7.

Figures 3.14 – 3.17 symbolically depict the condition for each state. As an example, consider Figure 3.13. A cycle path (C,E) is represented by an ellipse in which the vertical lines denote the common edges of the cycles. The leftmost cycle represents C_1, the rightmost one represents C_p. The vertices that have one of the states in $\{\mathsf{I2},\mathsf{E2},\mathsf{I1},\mathsf{E1}\}$ are represented by a dot. All other vertices are not drawn. In Figure 3.13, $st(v_1) = \mathsf{I2}$, $st(v_2) = \mathsf{E2}$, $st(v_3) = \mathsf{I1}$ and $st(v_4) = \mathsf{E1}$. If $\mathrm{dst}_1(u,v)$ ($\mathrm{dst}_p(u,v)$) holds for two vertices, then the vertices are both drawn in the leftmost (rightmost) cycle, and they are connected by a fat edge. In Figure 3.13, $\mathrm{dst}_1(v_2,v_4)$ holds.

Definition 3.4.6 (Block States). Let G be a partial two-path, B a block of $\bar{G}$, and (C,E) a correct cycle path for B, $C = (C_1,\ldots,C_p)$, $E = (e_1,\ldots,e_{p-1})$. Let $v_1,\ldots,v_s$ denote the vertices of B which do not have state N or S, such that $st(v_i) \succeq st(v_{i+1})$ for each i, $1 \leq i < s$. The *state of* B is denoted by $st(B)$, and is defined as $st(B) = (st(v_1),st(v_2),\ldots,st(v_s))$. Because G is a partial two-path, the vertices $v_1,\ldots,v_s$ satisfy a number of conditions. For each value of $st(B)$, we denote these conditions by $\mathrm{cond}(st(B))$. The conditions will be defined in following tables. For $s = 0$, $\mathrm{cond}(()) = \mathsf{true}$ (Figure 3.14)

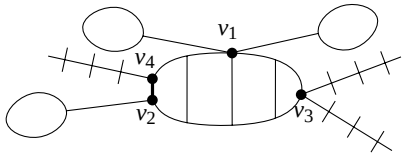


Figure 3.13. Legend for Figures 3.14 – 3.18. The states of the drawn vertices are as follows: $st(v_1) = \mathsf{I2}$, $st(v_2) = \mathsf{E2}$, $st(v_3) = \mathsf{I1}$ and $st(v_4) = \mathsf{E1}$. Furthermore, $\mathrm{dst}_1(v_2, v_4)$ holds.

$s = 1$ (Figure 3.14)	
$st(B)$	$\mathrm{cond}(st(B))$
$(\mathsf{I2})$	$\forall_{1 \le i < p}\, v_1 \in e_i$
$(\mathsf{E2})$	$v_1 \in V(C_1) \cup V(C_p)$
$(\mathsf{I1})$	$\mathrm{cond}((\mathsf{E2}))$
$(\mathsf{E1})$	$\mathrm{cond}((\mathsf{E2}))$

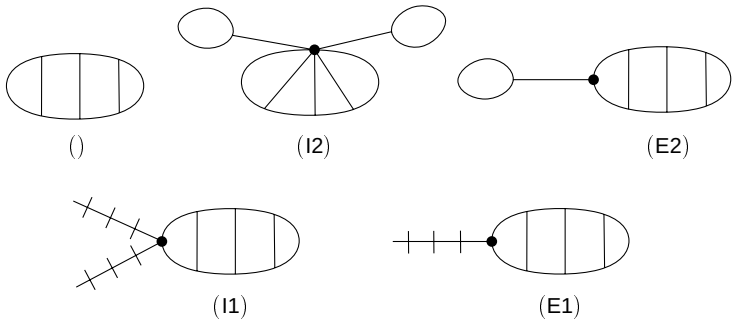


Figure 3.14. Symbolic representation of $\mathrm{cond}(S)$ for each possible block state S for $s = 0$ and $s = 1$. Cases that are symmetric in C_1 and C_p are given only once.

$s = 2$ (Figure 3.15)	
$st(B)$	$\text{cond}(st(B))$
$(\mathsf{I2},\mathsf{I1})$	$((\forall_{1\le i<p}\, v_1 \in e_i \wedge v_2 \in e_i) \wedge \text{dst}_1(v_1,v_2) \wedge \text{dst}_p(v_1,v_2))$ $\vee\, (p = 1 \wedge V(C_1) = \{v_1,v_2,u,w\}$ $\wedge E(C_1) = \{\{v_1,u\},\{v_2,u\},\{v_1,w\},\{v_2,w\}\} \wedge st(u) = st(w) = \mathsf{N})$
$(\mathsf{I2},\mathsf{E1})$	$(\forall_{1\le i<p}\, v_1 \in e_i) \wedge (\text{dst}_1(v_1,v_2) \vee \text{dst}_p(v_1,v_2))$
$(\mathsf{E2},\mathsf{E2})$	$(v_1 \in C_1 \wedge v_2 \in C_p) \vee (v_1 \in C_p \wedge v_2 \in C_1)$
$(\mathsf{E2},\mathsf{I1})$	$\text{cond}((\mathsf{E2},\mathsf{E2}))$
$(\mathsf{E2},\mathsf{E1})$	$((v_1 \in C_1 \wedge v_2 \in C_p) \vee (v_1 \in C_p \wedge v_2 \in C_1) \vee \text{dst}_1(v_1,v_2) \vee \text{dst}_p(v_1,v_2))$
$(\mathsf{I1},\mathsf{I1})$	$\text{cond}((\mathsf{E2},\mathsf{E2}))$
$(\mathsf{I1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E1}))$
$(\mathsf{E1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E1}))$

$s = 3$ (Figure 3.16)	
$st(B)$	$\text{cond}(st(B))$
$(\mathsf{I2},\mathsf{E1},\mathsf{E1})$	$(\text{dst}_1(v_1,v_2) \wedge \text{dst}_p(v_1,v_3)) \vee (\text{dst}_p(v_1,v_2) \wedge \text{dst}_1(v_1,v_3))$
$(\mathsf{E2},\mathsf{E2},\mathsf{I1})$	$(\text{dst}_1(v_1,v_3) \wedge \text{dst}_p(v_2,v_3)) \vee (\text{dst}_p(v_1,v_3) \wedge \text{dst}_1(v_2,v_3))$
$(\mathsf{E2},\mathsf{E2},\mathsf{E1})$	$(v_1 \in V(C_1) \wedge \text{dst}_p(v_2,v_3)) \vee (v_1 \in V(C_p) \wedge \text{dst}_1(v_2,v_3)) \vee$ $(v_2 \in V(C_1) \wedge \text{dst}_p(v_1,v_3)) \vee (v_2 \in V(C_p) \wedge \text{dst}_1(v_1,v_3))$
$(\mathsf{E2},\mathsf{I1},\mathsf{I1})$	$(\text{dst}_1(v_1,v_3) \wedge \text{dst}_p(v_2,v_3)) \vee (\text{dst}_p(v_1,v_3) \wedge \text{dst}_1(v_2,v_3)) \vee$ $(\text{dst}_1(v_1,v_2)) \wedge \text{dst}_p(v_3,v_2)) \vee (\text{dst}_p(v_1,v_2) \wedge \text{dst}_1(v_3,v_2))$
$(\mathsf{E2},\mathsf{I1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E2},\mathsf{E1}))$
$(\mathsf{E2},\mathsf{E1},\mathsf{E1})$	$(v_1 \in V(C_1) \wedge \text{dst}_p(v_2,v_3)) \vee (v_1 \in V(C_p) \wedge \text{dst}_1(v_2,v_3)) \vee$ $(v_2 \in V(C_1) \wedge \text{dst}_p(v_1,v_3)) \vee (v_2 \in V(C_p) \wedge \text{dst}_1(v_1,v_3)) \vee$ $(v_3 \in V(C_1) \wedge \text{dst}_p(v_1,v_2)) \vee (v_3 \in V(C_p) \wedge \text{dst}_1(v_1,v_2))$
$(\mathsf{I1},\mathsf{I1},\mathsf{I1})$	$(\text{dst}_1(v_1,v_3) \wedge \text{dst}_p(v_2,v_3)) \vee (\text{dst}_p(v_1,v_3) \wedge \text{dst}_1(v_2,v_3)) \vee$ $(\text{dst}_1(v_1,v_2) \wedge \text{dst}_p(v_3,v_2)) \vee (\text{dst}_p(v_1,v_2) \wedge \text{dst}_1(v_3,v_2)) \vee$ $(\text{dst}_1(v_2,v_1) \wedge \text{dst}_p(v_3,v_1)) \vee (\text{dst}_p(v_2,v_1) \wedge \text{dst}_1(v_3,v_1))$
$(\mathsf{I1},\mathsf{I1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E2},\mathsf{E1}))$
$(\mathsf{I1},\mathsf{E1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E1},\mathsf{E1}))$
$(\mathsf{E1},\mathsf{E1},\mathsf{E1})$	$\text{cond}((\mathsf{E2},\mathsf{E1},\mathsf{E1}))$

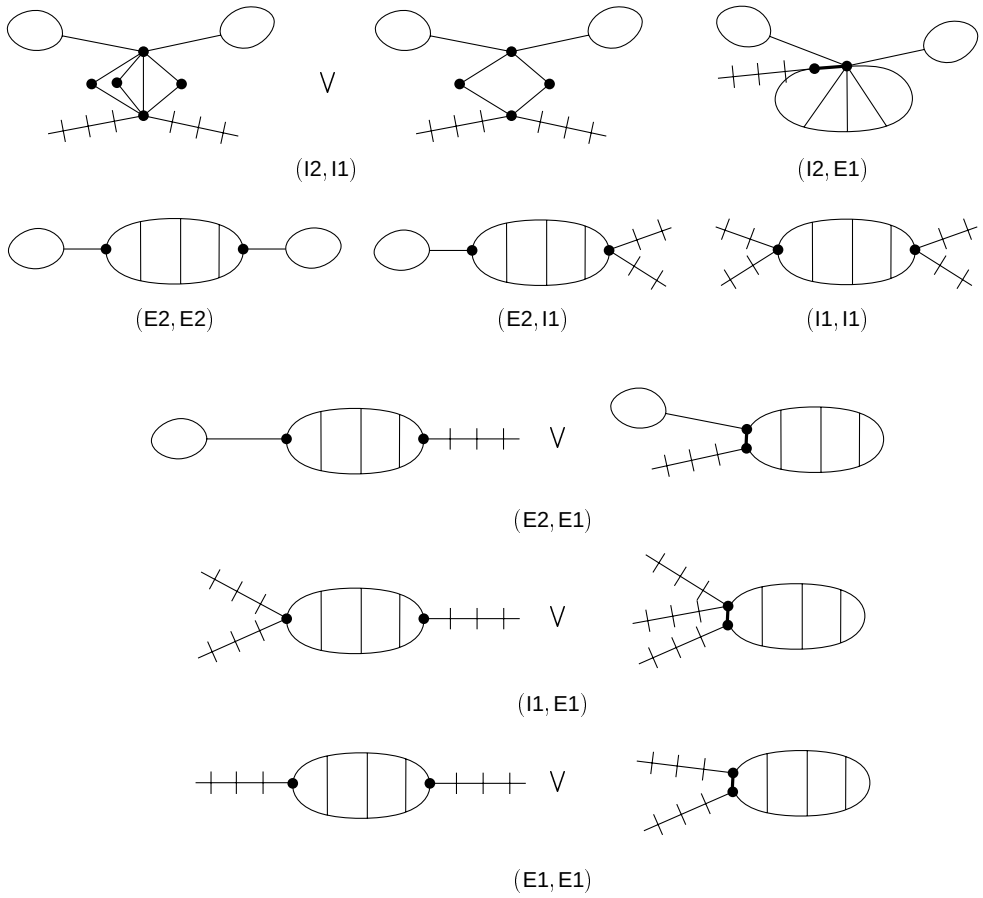


Figure 3.15. Symbolic representation of cond(S) for each possible block state S for $s = 2$. For state $(\mathsf{I2}, \mathsf{I1})$, the block is represented in its normal way. Cases that are symmetric in C_1 and C_p, or in distinct vertices v_i with the same state are given only once.

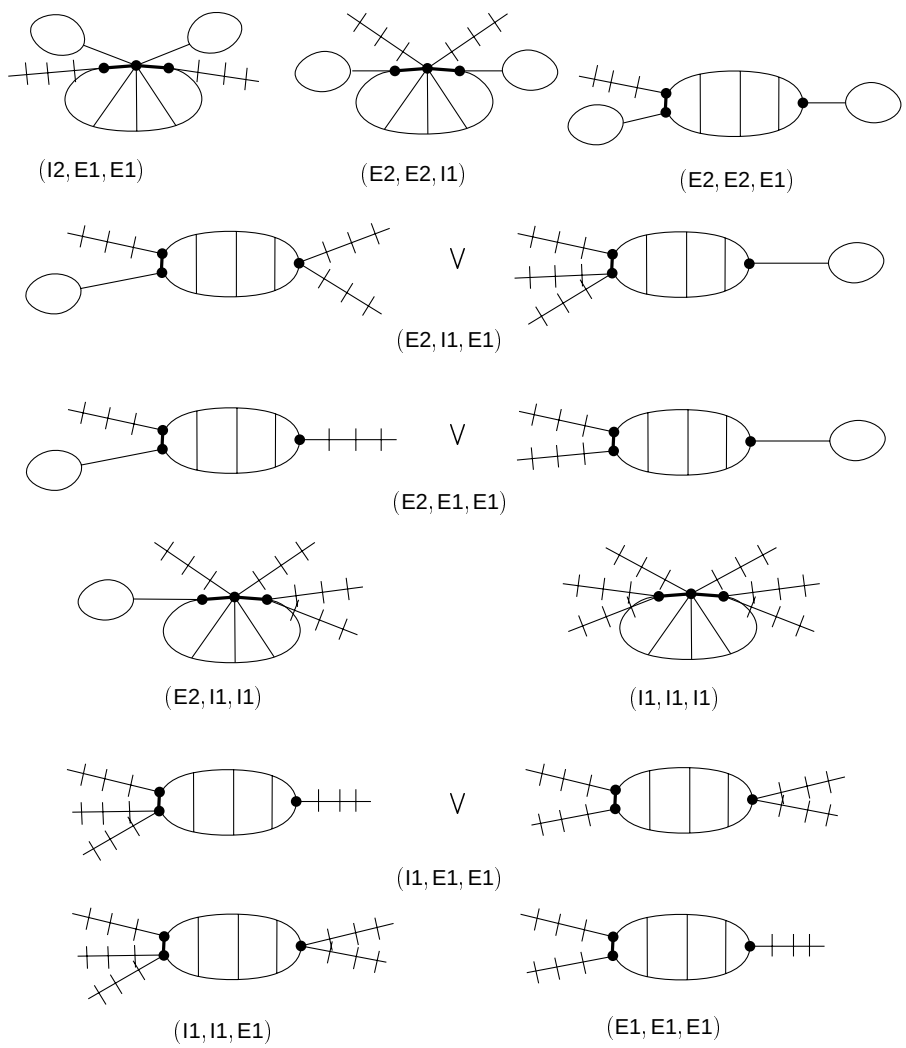


Figure 3.16. Symbolic representation of cond(S) for all block states S for $s = 3$. Cases that are symmetric in C_1 and C_p, or in distinct vertices v_i with the same state are given only once.

$s = 4$ (Figure 3.17)	
$st(B)$	cond$(st(B))$
$(\mathsf{E2},\mathsf{E2},\mathsf{E1},\mathsf{E1})$	$(\mathrm{dst}_1(v_1,v_3) \wedge \mathrm{dst}_p(v_2,v_4)) \vee (\mathrm{dst}_p(v_1,v_3) \wedge \mathrm{dst}_1(v_2,v_4)) \vee$ $(\mathrm{dst}_1(v_1,v_4) \wedge \mathrm{dst}_p(v_2,v_3)) \vee (\mathrm{dst}_p(v_1,v_4) \wedge \mathrm{dst}_1(v_2,v_3))$
$(\mathsf{E2},\mathsf{I1},\mathsf{E1},\mathsf{E1})$	cond$((\mathsf{E2},\mathsf{E2},\mathsf{E1},\mathsf{E1}))$
$(\mathsf{E2},\mathsf{E1},\mathsf{E1},\mathsf{E1})$	$(\mathrm{dst}_1(v_1,v_2) \wedge \mathrm{dst}_p(v_3,v_4)) \vee (\mathrm{dst}_p(v_1,v_2) \wedge \mathrm{dst}_1(v_3,v_4)) \vee$ $(\mathrm{dst}_1(v_1,v_3) \wedge \mathrm{dst}_p(v_2,v_4)) \vee (\mathrm{dst}_p(v_1,v_3) \wedge \mathrm{dst}_1(v_2,v_4)) \vee$ $(\mathrm{dst}_1(v_1,v_4) \wedge \mathrm{dst}_p(v_2,v_3)) \vee (\mathrm{dst}_p(v_1,v_4) \wedge \mathrm{dst}_1(v_2,v_3))$
$(\mathsf{I1},\mathsf{I1},\mathsf{E1},\mathsf{E1})$	cond$((\mathsf{E2},\mathsf{E2},\mathsf{E1},\mathsf{E1}))$
$(\mathsf{I1},\mathsf{E1},\mathsf{E1},\mathsf{E1})$	cond$((\mathsf{E2},\mathsf{E1},\mathsf{E1},\mathsf{E1}))$
$(\mathsf{E1},\mathsf{E1},\mathsf{E1},\mathsf{E1})$	cond$((\mathsf{E2},\mathsf{E1},\mathsf{E1},\mathsf{E1}))$

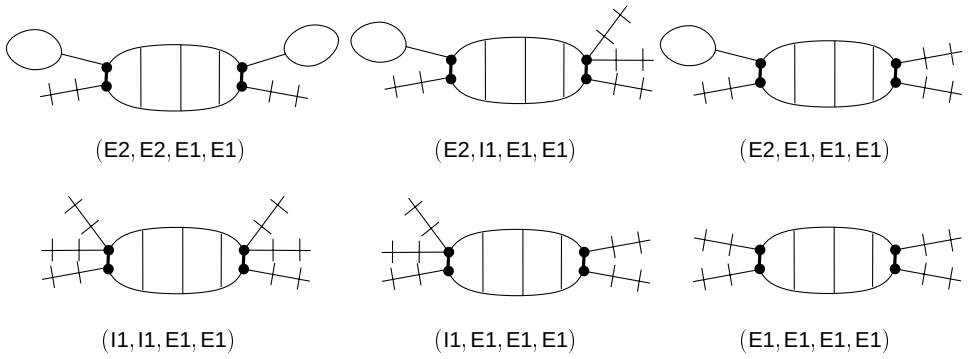


Figure 3.17. Symbolic representation of cond(S) for all possible block states for $s = 4$. Cases that are symmetric in C_1 and C_p, or in distinct vertices v_i with the same state are given only once.

Let $a \in \{\mathsf{I2},\mathsf{E2},\mathsf{I1},\mathsf{E1}\}$. We denote by S_a the set of block states for which $s \geq 1$ and $st(v_1) = a$.

Although it seems that the states and conditions that are given in Definition 3.4.6 depend on the correct cycle path that is used, this is not the case: no matter what correct cycle path (C,E) of a block B is used, the state of B and the value of cond$(st(B))$ are the same (see also page 54). Note that the block states are well-defined, i.e. each block has exactly one state.

Lemma 3.4.7. *Let G be a partial two-path. Each block B of $\bar{G}$ has one of the states in $S_{\mathsf{I2}} \cup S_{\mathsf{E2}} \cup S_{\mathsf{I1}} \cup S_{\mathsf{E1}}$, and satisfies* cond$(st(B))$.

Proof. Let B be a block of $\bar{G}$, let (C,E) be a correct cycle path of B with $C = (C_1,\ldots,C_p)$, $E = (e_1,\ldots,e_{p-1})$. Furthermore, let $v_1,\ldots,v_s$ denote the vertices of B which have one of the

states in $\{\mathsf{I2}, \mathsf{E2}, \mathsf{I1}, \mathsf{E1}\}$ such that $st(v_1) \succeq st(v_2) \succeq \cdots \succeq st(v_s)$. Then clearly $s \leq 4$. We have to show that $(st(v_1), \ldots, , st(v_s)) \in S_{st(v_1)}$ and that $\text{cond}((st(v_1), \ldots, st(v_s)))$ holds. If $s = 0$, then this is clear.

Suppose $s > 0$, let H be the tree of G which contains v_1. If $st(v_1) = \mathsf{I2}$, then v_1 is an inner vertex of the path P_H, and it follows from Lemma 3.4.2 that the component of $G[V \Leftrightarrow \{v_1\}]$ which contains vertices of B must have pathwidth one. It can easily be checked that if this is the case, then $st(B) \in S_{\mathsf{I2}}$ and $\text{cond}(st(B))$ holds.

Suppose $st(v_1) \in \{\mathsf{E2}, \mathsf{I1}, \mathsf{E1}\}$. Vertex v_1 is end point of P_H. Lemma 3.4.6 shows that $st(B) \in S_{st(v_1)}$ and that $\text{cond}(st(B))$ holds. □

Definition 3.4.7. Let G be a partial two-path, B a block of $\bar{G}$, and (C, E) a correct cycle path for B with $C = (C_1, \ldots, C_p)$ and $E = (e_1, \ldots, e_{p-1})$. Let $v_1, \ldots, v_s$ denote the vertices of B which do not have state N or S, such that $st(v_i) \succeq st(v_{i+1})$ for each i, $1 \leq i < s$. Suppose that $s \geq 1$ and $st(v_1) = \mathsf{E2}$. Let G' be the component of $G[V \Leftrightarrow \{v_1\}]$ which contains $V(B) \Leftrightarrow \{v_1\}$. $\text{cond}_1(st(B))$ is defined as follows.

$$\text{cond}_1(st(B)) \Leftrightarrow \text{cond}((\mathsf{I2}, st(v_2), \ldots, st(v_s)))$$

Note that if $st(v_1) = \mathsf{E2}$ and $\text{cond}_1(st(B))$ holds, then also $\text{cond}(st(B))$ holds. Figure 3.18 depicts $\text{cond}_1(st(B))$ for all values of $st(B)$.

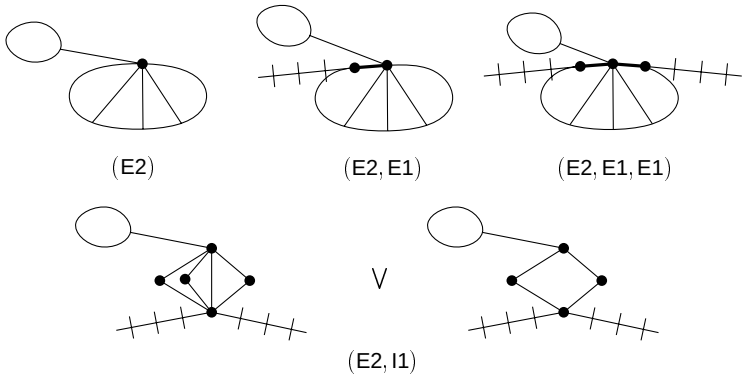


Figure 3.18. Symbolic representation of $\text{cond}_1(S)$ for possible block state $S = (st_1, \ldots, st_s)$ with $st_1 = \mathsf{E2}$. Cases that are symmetrical in C_1 and C_p, or in distinct vertices v_i with the same state are given only once.

Theorem 3.4.1. *Let G be a connected graph. G is a partial two-path if and only if the following conditions hold.*

1. *For each tree H of G, the following holds: H has pathwidth at most two, and there is a path in H which contains a path in $P_2(H)$ and all vertices that are in a block of G.*

2. *Each block B of $\bar{G}$ contains only vertices that have one of the states* I2, E2, I1, E1, S *and* N*, and at most four vertices of B do not have state* S *or* N.

3. *For each block of $\bar{G}$, there is a correct cycle path.*

4. *Each block B of $\bar{G}$ has one of the states in $S_{\mathsf{I2}} \cup S_{\mathsf{E2}} \cup S_{\mathsf{I1}} \cup S_{\mathsf{E1}} \cup \{()\}$ and satisfies* $\mathrm{cond}(st(B))$.

5. *Let H be a tree of G_{tr}, suppose $G \neq H$, let $P_H = (u_1, \ldots, u_p)$. If $p > 1$ and u_1 is a vertex of a block and $st(u_1) =$* E2*, then at most one of the blocks that contain u_1 does not satisfy* $\mathrm{cond}_1(st(B))$. *Similar for u_p.*

   If $p = 1$, u_1 is a vertex of a block and $st(u_1) =$ E2*, then at most two blocks containing u_1 do not satisfy* $\mathrm{cond}_1(st(B))$.

Proof. We first prove the 'only if' part. Suppose G is a partial two-path, then it follows directly from Lemmas 3.4.1, 3.4.4, and 3.4.7 that 1, 2, 3 and 4 hold.

We now prove 5. Let H be a tree of G, suppose $G \neq H$, let $P_H = (u_1, \ldots, u_p)$. If $p > 1$, then, according to Lemma 3.4.2, there is at most one component in $G' = G[V(G) \Leftrightarrow V(P_H)]$ that has pathwidth two and is adjacent to u_1 in G. This means that at most one block B containing u_1 does not satisfy $\mathrm{cond}_1(st(B))$, since $\mathrm{cond}_1(st(B))$ holds if the component of $G[V \Leftrightarrow \{u_1\}]$ which contains $V(B) \Leftrightarrow \{u_1\}$ has pathwidth one, as is shown in the proof of Lemma 3.4.7. If $p = 1$, then in the same way, we can show that at most two blocks B containing u_1 are allowed not to satisfy $\mathrm{cond}_1(st(B))$.

Now we prove the 'if' part. Suppose G is a connected graph that satisfies conditions 1, 2, 3, 4 and 5. If G is a tree or G is biconnected, then G has pathwidth two, as is shown in Theorem 3.2.1 and Lemma 3.3.1. Suppose G contains at least one block and at least one tree with one or more edges. We construct a path decomposition of width two of G.

First consider the trees of G. Let H be a tree of G, let $P_H = (u_1, \ldots, u_p)$. If $p = 1$ and $st(u_1) =$ E1, then make a path decomposition PD_H of width one of H in which u_1 is in the rightmost node.

If $p = 1$ and $st(u_1) =$ I1, then make a path decomposition PD_H of width one of H. Let H_1 and H_2 be the components of $H[V \Leftrightarrow \{v_1\}]$ that contain edges of H, such that the leftmost node of PD_H contains vertices of H_1 and the rightmost node contains vertices of H_2. Let $PD_H^1 = PD[V(H_1) \cup \{v_1\} \cup \{\text{sticks of } v_1\}]$, and $PD_H^2 = PD[V(H_2) \cup \{v_1\}]$. Note that v_1 is in the rightmost node of PD_H^1 and in the leftmost node of PD_H^2. Furthermore, make a path decomposition PD'_H of width two of H, which is similar to PD_H, but with vertex v_1 added to each node. In the final path decomposition of G, PD'_H is used if component H may occur completely on the same side of the block which contains v_1, and PD_H^1 and PD_H^2 are used if two parts of H must occur on different sides. In this case, PD_H^1 occurs on the left side and PD_H^2 on the right side.

If $p > 1$, or $p = 1$ and $st(u_1) \succeq$ E2, then do the following. Let G_H denote the induced subgraph of G which contains H and all components of $G[V \Leftrightarrow V(P_H)]$ which have pathwidth zero or one. For each u_i, each component of $G_H[V(G_H) \Leftrightarrow V(P_H)]$ which is connected to u_i, make

a path decomposition of width zero or one, and add u_i to each node of this path decomposition. For each u_i, concatenate the obtained path decompositions of all components which are connected to u_i, and let PD_i denote this path decomposition. Now make the following path decomposition: $PD_H = PD_1 +\!\!+ (\{u_1,u_2\}) +\!\!+ PD_2 +\!\!+ \cdots +\!\!+ (\{u_{p-1},u_p\}) +\!\!+ PD_p$. See for example Figure 3.19. In this picture, $P_H = (u_1,u_2,u_3,u_4)$, and $H_1,\ldots,H_5$ are the components of G_H which have pathwidth one.

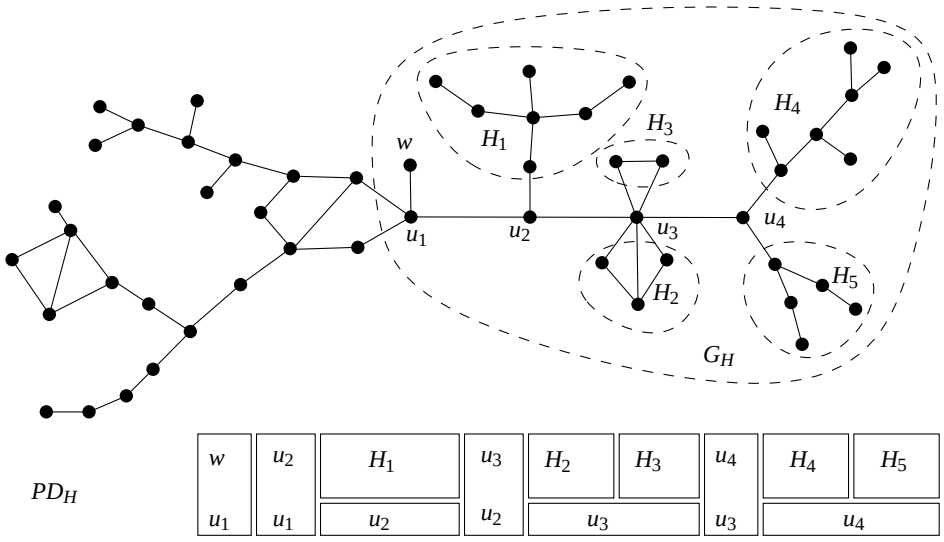


Figure 3.19. Example of the construction of PD_H for the subgraph G_H.

PD_H is a path decomposition of width two of the graph G_H. Furthermore, the leftmost node of PD_H contains u_1, the rightmost node contains u_p. There are at most two components of $G[V \Leftrightarrow V(P_H)]$ which have pathwidth two, and if $p > 1$, then at most one of these components is connected to u_1, and at most one to u_p.

Now consider the blocks which are not contained in some G_H for a tree H of G. For each block B of $\overline{G}$ for which this holds, let (C,E) be a correct cycle path with $C = (C_1,\ldots,C_p)$ and $E = (e_1,\ldots,e_{p-1})$. Let $v_1,\ldots,v_s$ denote the vertices of B which have one of the states in $\{\mathsf{E2},\mathsf{I1},\mathsf{E1}\}$. Note that B has no vertices with state $\mathsf{I2}$, since then B would be in some graph G_H. Let G_B denote the subgraph of G which contains B and all sticks of B which are adjacent to vertices with state S.

If $s = 0$, then make a path decomposition of width two of G_B as follows. First make a path decomposition PD_B of width two of B in the way that is shown in the proof of Lemma 3.2.3, but add one node on the left side which contains one of the edges in the former leftmost node, and add one node on the right side which contains one of the edges in the former rightmost node. We now extend PD_B such that it is a path decomposition of width two of G_B.

First suppose $|V(B)| > 3$. Let $v \in V(B)$ such that $st(V) = \mathsf{S}$. It can be seen that there are two nodes V_i and V_{i+1}, such that $V_i \cap V_{i+1} = \{v, u\}$ for some $u \neq v$. See e.g. Figure 3.2. For each stick w adjacent to v, we add a node $\{w, v, u\}$ between V_i and V_{i+1}. If this is done for all vertices in B which have state S, then the PD_B is a path decomposition of width two of G_B.

Suppose $V(B) = \{w_1, w_2, w_3\}$. Then $PD_B = (\{w_1, w_2, w_3\})$, and we can make a path decomposition of width two of G_B by adding on the left side for each stick w of w_1 or w_2 a node $\{w_1, w_2, w\}$, and on the right side for each stick w of w_3 a node $\{w_3, w\}$.

If $s > 1$, then make a path decomposition of G_B in the same way as for $s = 0$, but with the appropriate vertices of $\{v_1, \ldots, v_s\}$ occurring in the leftmost and rightmost node. It can be derived from the pictures of all conditions (see Figures 3.14 – 3.17 which vertex must occur on which side; e.g. if $v_1 \in V(C_1)$ and the tree H of G which contains v_1 is drawn on the left side of the block in the picture representing this case, then v_1 must occur in the leftmost node, but if $st(v_1) = \mathsf{I1}$, $v_1 \in V(C_1) \cap V(C_p)$ and part of H is drawn on left side of the block, and the other part is drawn on the right side, then v_1 must occur in both end nodes of the path decomposition. Note that this is possible, since in the conditions, the distance between two vertices v_i and v_j of which the components must occur on the same side must be sufficiently small.

If all these path decompositions are made, then they can be combined rather straightforwardly into a path decomposition of width two of G. In Figure 3.20, an example is given of how this can be done. □

Let G be a connected partial two-path which is not a tree. We now extend the definition of the path P_H for a tree H of G to a path of the graph G. Consider the set $\mathcal{H}$ of all trees of G which contain a vertex w of a block, such that w has state $\mathsf{I2}$ or $\mathsf{E2}$. Each block has at most one vertex with state $\mathsf{I2}$, at most two vertices with state $\mathsf{E2}$, and if it has a vertex with state $\mathsf{I2}$, then it has no vertices with state $\mathsf{E2}$. This means that we can give the following definition.

Definition 3.4.8. Let G be a connected partial two-path which is not a tree. Let $\mathcal{H}$ be as defined above, let $\mathcal{B}$ be the set of blocks of G. The *path* P_G of G is a graph which is defined as follows:

$$
\begin{aligned}
V(P_G) &= \bigcup_{H \in \mathcal{H}} V(P_H), \\
E(P_G) &= \{\, e \in E(G) \mid \exists_{H \in \mathcal{H}}\, e \in E(P_H) \,\} \cup \\
&\qquad \{\, \{v, v'\} \mid \exists_{H, H' \in \mathcal{H}, B \in \mathcal{B}}\, H \neq H' \wedge v \in V(P_H) \wedge v' \in V(P_H) \wedge v, v' \in V(B) \,\}.
\end{aligned}
$$

Note that P_G is unique since, if G is not a tree, then each tree H of G has at least one vertex in a block, and hence $|P_H| = 1$. The set $V(P_G)$ may be empty in the case that G contains only one block. Note furthermore that P_G is in fact a concatenation of all paths P_H of trees $H \in \mathcal{H}$, in such a way that two paths which have an end point in a common block are consecutive in P_G. P_G is not a real path of G, but it is the largest common subsequence of all paths in G between the two end points of P_G. The blocks of G which contain two vertices of P_G are called *connecting* blocks. All other blocks are called *non-connecting* blocks.

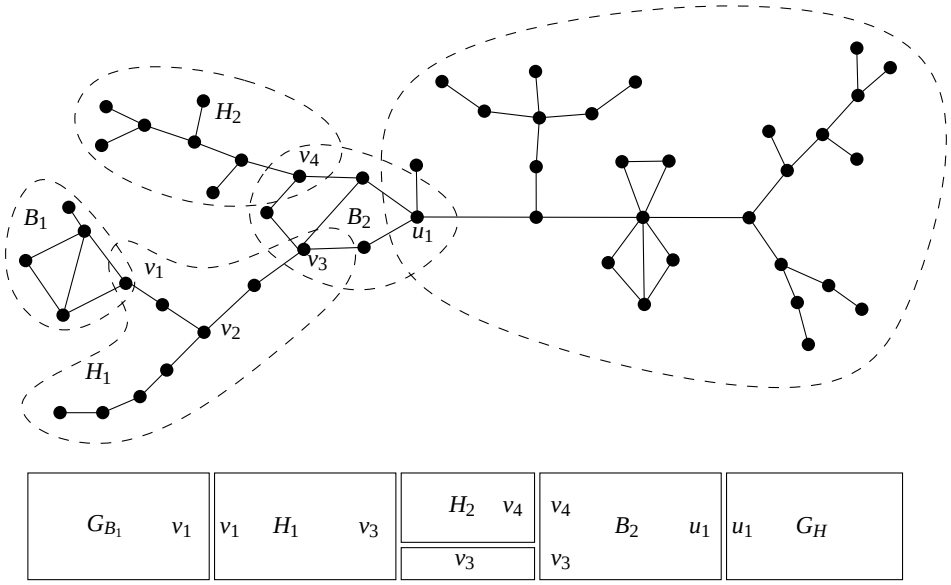


Figure 3.20. Example of the construction of a path decomposition of width two of a partial two-path G, after the path decompositions of all trees of G and all blocks, including their sticks, are constructed as in the proof of Theorem 3.4.1.

In each path decomposition $PD = (V_1, \ldots, V_t)$ of width two of G, the occurrences of the paths P_H, $H \in \mathcal{H}$, do not overlap, since they have no vertices in common. Furthermore, they occur in the same order as in P_G or in reversed order, because they are connected to each other by blocks with pathwidth two.

We show the analog of Corollary 3.3.1 for general partial two-paths.

Lemma 3.4.8. *Let G be a connected partial two-path which is not a tree. Let $P_G = (v_1, \ldots, v_s)$ and let $PD = (V_1, \ldots, V_t)$ be a path decomposition of width two of G. For each $v \in V_1$, $v' \in V_t$, any path from v to v' contains P_G as a subsequence.*

Proof. If $|V(P_G)| = 0$, the result clearly holds. Suppose $|V(P_G)| \geq 1$. Let G_1 be the subgraph of G induced by vertex v_1 and the components of $G[V \Leftrightarrow \{v_1\}]$ which do not contain vertices of P_G. Similarly, let G_2 be the subgraph of G induced by v_s and the components of $G[V \Leftrightarrow \{v_s\}]$ which do not contain vertices of P_G. We prove the lemma by proving that $V_1 \subseteq V(G_1)$ and $V_s \subseteq V(G_2)$ or vice versa, and if $s = 1$, then V_1 and V_t do not contain vertices of the same component of $G[V \Leftrightarrow \{v_1\}]$.

Suppose V_1 contains a vertex $v \notin V(G_1) \cup V(G_2)$. We distinguish two cases.

1. v is an inner vertex of P_G or there is an inner vertex v' of P_G such that v is a vertex of a component of $G[V \Leftrightarrow \{v'\}]$ which does not contain vertices of P_G.

2. $v \notin V(P_G)$ and there is a connecting block B of G such that v is in the component of $G[V \Leftrightarrow V(P_G)]$ which contains vertices of B.

First suppose case 1 holds. Let i, $1 \leq i \leq s$, be such that either $v = v_i$ or v is in a component of $G[V \Leftrightarrow \{v_i\}]$ which does not contain vertices of P_G. Let G' and G'' denote the components of $G[V \Leftrightarrow \{v_i\}]$ which contain vertices of P_G. G' and G'' have pathwidth two, hence there are nodes V_j and $V_{j'}$ in PD such that V_j contains three vertices of G' and $V_{j'}$ contains three vertices of G'. Suppose w.l.o.g. that $j < j'$. Then V_j contains a vertex of $G[V \Leftrightarrow V(G')]$, since V_1 contains v, and $V_{j'}$ contains vertices of G''. Contradiction.

Next suppose case 2 holds. Let B be the block of G for which v is in the component of $G[V \Leftrightarrow V(P_G)]$ which contains a vertex of B. Let i, $1 \leq i \leq s$, be such that $v_i, v_{i+1} \in V(B)$. Let G' be the subgraph of G induced by v_i and the component of $G[V \Leftrightarrow \{v_i\}]$ containing G_1. Similarly, let G'' be the subgraph of G induced by v_{i+1} and the subgraph of $G[V \Leftrightarrow \{v_{i+1}\}]$ containing G_2. In the same way as for case 1, we can derive a contradiction.

We next show that V_1 and V_t can not both contain a vertex of G_1, unless $s = 1$. Suppose $s > 1$ and $v \in V_1$, $v' \in V_t$ such that $v, v' \in V(G_1)$. G_2 has pathwidth two, which means that there is a node V_j, $1 \leq j \leq t$, such that V_j contains three vertices of G_2. But V_j also contains a vertex of G_1, which is a contradiction. In the same way we can prove that if $s = 1$, then V_1 and V_t can not both contain a vertex of the same component of $G[V \Leftrightarrow \{v_1\}]$. □

3.5 Finding the Structure of a Partial Two-Path

In this section we give an algorithm which, given a graph G, returns false if G has pathwidth three or more, and otherwise, constructs a correct cycle path for the cell completion of each block of G, computes the sets $P_2(H)$, P_H and $P_1(H)$ of all trees of G, computes the states of all blocks and all vertices in blocks of G, and computes the path P_G, if G is not a tree. We first give the algorithm for biconnected graphs, then for trees, and finally for general graphs.

3.5.1 Biconnected Graphs

The following algorithm has as its input a biconnected graph G, and returns false if G has pathwidth three or more, and constructs a cycle path for the cell completion of G otherwise.

Algorithm Cycle_Path(G)
Input: Biconnected graph G
Output: A cycle path for $\bar{G}$ if G has pathwidth at most two, and false otherwise
1. Find the cell completion $\bar{G}$ of G.
2. Make a list L of all chordless cycles in $\bar{G}$.
3. Check whether $\bar{G}$ is a path of cycles. If it is, construct a cycle path and **return** it. Otherwise, **return** false.

For Steps 1 and 2 we use an algorithm from Bodlaender and Kloks [1993]. This algorithm has as input a biconnected graph G, and checks if G has treewidth at most two. If so, it also computes the cell completion $\bar{G}$ of G together with a list of the chordless cycles of $\bar{G}$. It uses $O(n)$ time and space. Step 3 can be done as follows.

Algorithm Step 3

1. Check for each cycle in L whether it has at most two edges in common with any other chordless cycle. If not, **return** false.
2. Check whether there is an edge which occurs in two or more cycles. If there is no such edge, then there is only one cycle C, and **return** $((C),())$ as cycle path.
3. Take an arbitrary edge e that occurs in two or more cycles. Make a list $C = (C_1, \ldots, C_s)$ of all cycles which contain e in such a way that only C_1 and C_s may have four or more vertices or have an edge $e' \neq e$ in common with another cycle. If this is not possible, **return** false. Also, make a list $E = (e, e, \ldots, e)$, in which e occurs $s \Leftrightarrow 1$ times. Remove $C_1, \ldots, C_s$ from L.
4. Repeat the following until the first cycle in C has no edge in common with a cycle that is not in C.
   Let C_1 be the first cycle in C. If C_1 has an edge $e' \neq e$ in common with a cycle that is not in C, then do the following. Construct a list $C' = (C'_1, \ldots, C'_{s'})$ of all cycles containing e', in such a way that only C'_1 may have four or more vertices or have an edge $e' \neq e$ in common with another cycle. If there are two or more of these cycles, then **return** false. Also, build a list $E' = (e', e', \ldots, e')$, in which e' occurs s' times. Now, let the new C be the concatenation of C' and the old C, and let E be the concatenation of E' and the old E. Remove $C'_1, \ldots, C'_{s'}$ from L.
5. Repeat a modified version of step 4 for the last cycle in C, until the last cycle in C has no edge in common with a cycle that is not in C. (Directions of C' and E' must be reversed, C' is concatenated at the back of C, and similar for E').
6. **return** (C, E).

It is easy to see that the algorithm returns a cycle path for $\overline{G}$, if one exists, and that it can be made to run in $O(n)$ time.

3.5.2 Trees

Ellis et al. [1994] and Möhring [1990] have given linear time algorithms to compute the pathwidth of a tree. We can modify one of these algorithms in order to check whether a given tree H has pathwidth zero, one or two, and furthermore, if the pathwidth of H equals one, then compute $P_1(H)$, and if the pathwidth of H is two, then compute $P_2(H)$.

3.5.3 General Graphs

For general graphs, we combine the algorithms for biconnected graphs and for trees. Given a graph G, we first compute the blocks of G, and after that, the trees of G. During the computation of the trees, we mark each vertex v with a list of all blocks that contain v. Similarly, we mark each vertex v of each block with the tree containing v.

Now, for each tree H of G, we compute the set $P_2(H)$, if H has pathwidth two, and $P_1(H)$ if H has pathwidth one. After that, we compute the set P_H. With this information, we can compute the vertex states of the vertices that are in at least one block in linear time.

Next, for each block B of G, we compute the cell completion $\bar{B}$, we check if $\bar{B}$ is a path of cycles, and we compute a cycle path (C,E) for $\bar{B}$, and if possible, modify (C,E) such that it becomes a correct cycle path. If this is not possible, we return false. Now we check if B has one of the states in $S_{\mathsf{I2}} \cup S_{\mathsf{E2}} \cup S_{\mathsf{I1}} \cup S_{\mathsf{E1}} \cup \{()\}$ and satisfies cond$(st(B))$. Then, we check if condition 5 of Theorem 3.4.1 holds. Finally, if G is not a tree, we concatenate all paths P_H of trees $H \in \mathcal{H}$ into the path P_G. All steps can be done in $O(n)$ time.

We have now proved the following theorem.

Theorem 3.5.1. *There is a linear time algorithm which, given a graph G, returns* false *if G has pathwidth three or more, and otherwise, constructs a correct cycle path for the cell completion of each block of G, computes the sets $P_2(H)$, P_H and $P_1(H)$ of all trees of G, computes the states of all blocks and all vertices in blocks of G, and computes the path P_G, if G is not a tree.*

The algorithm described above can be extended to find a path decomposition of minimum width of the input graph G, if the pathwidth of G is at most two. To this end, the construction as described in the 'if' part of Theorem 3.4.1 can be used. This construction can be performed in $O(n)$ time. The algorithm thus obtained is no theoretical improvement of the linear time algorithm of Bodlaender [1996a] for finding a tree or path decomposition of width at most k of a graph, for any fixed k. However, our algorithm is tailor-made for pathwidth at most two, and does not use the theoretical result of Bodlaender and Kloks [1996] (as does the algorithm of Bodlaender [1996a]). This means that our algorithm is easier to implement, and probably more efficient in practice.

Chapter 4

DNA Physical Mapping

In this chapter we consider two graph problems which model problems arising in molecular biology. In Section 4.1 we introduce the biological problems and show how they can be modeled as graph problems. In Sections 4.2 and 4.3, we discuss the two graph problems. At the end of Section 4.1 we give a more detailed overview of Sections 4.2 and 4.3.

4.1 Introduction

The biological problems we consider are known as *sequence reconstruction* problems in molecular biology. Sequence reconstruction occurs in different levels of DNA physical mapping: it is currently not possible to find the linear structure of large parts of DNA or proteins at once [Jungck, Dick, and Dick, 1982]. Therefore, the sequence is cut into smaller parts, called *fragments*, which can then be sequenced. However, the order of the different fragments in the large sequence is lost during the fragmentation process. The reconstruction of this order is called sequence reconstruction.

There are several ways to attack the sequence reconstruction problem, many of which give rise to algorithmic (graph) problems (see e.g. Karp [1993] and Bodlaender, Downey, Fellows, Hallett, and Wareham [1995]). One way is to use, instead of one sequence of DNA or protein, a number of copies of the sequence, and to cut these copies in different ways. After that, a set of characteristics is determined for each fragment (its 'fingerprint' or 'signature'), and based on the respective fingerprints, an 'overlap' measure is computed.

The overlap measure for a pair of fragments usually consists of the probability that the two fragments overlap [Karp, 1993]. This information can be used in different ways. We use the *positive overlap information* and the *non-negative overlap information*. The positive overlap information consists of all pairs of fragments that overlap with probability one. The non-negative overlap information consists of all pairs of fragments that overlap with probability strictly larger than zero. The probabilities themselves are not used. We discuss the situation where k copies of a sequence X are fragmented, and we have positive and non-negative overlap information about the fragments. The problem is to assign each fragment to a copy of the sequence X, and for each copy, to make a linear ordering of the fragments that are assigned to that copy, in a way that is consistent with the positive and non-negative overlap information. This means that the assignment of fragments to copies and the ordering of the fragments must be such that

- two fragments overlap only if their overlap probability is strictly larger than zero, and

- if two fragments have overlap probability one then they overlap.

The positive and non-negative overlap information can be modeled by a *sandwich graph*.

Definition 4.1.1. A sandwich graph S is a triple (V, E_1, E_2) in which (V, E_1) and (V, E_2) are simple graphs, and $E_1 \subseteq E_2$.

Each vertex in the sandwich graph represents one fragment. The set E_1 represents the positive overlap information: an edge in E_1 between two vertices denotes that the corresponding fragments are known to overlap, i.e. they have overlap probability one. The set E_2 represents the non-negative overlap information: an edge in E_2 between two vertices denotes that the corresponding fragments may overlap, i.e. their overlap probability is strictly larger than zero. For each pair v, w of vertices for which $\{v, w\} \notin E_2$, it is known that the corresponding fragments do not overlap, i.e. their overlap probability is zero.

The assignment of each fragment to one of the k copies of the sequence can be modeled by a *k-coloring* of the graph (V, E_1) (we also call this a k-coloring of the sandwich graph S). (In this chapter, a k-coloring of a graph G is represented as a function $c : V(G) \to \{1, 2, \ldots, k\}$, such that for each $\{u, v\} \in E(G)$, $c(u) \neq c(v)$.) Each color represents one of the copies: if two vertices are assigned the same color, then the corresponding fragments are in the same copy, and hence they do not overlap, so there is no edge in E_1 between the vertices. The linear orderings of all the vertices of the same color can be modeled by intervals. Assign to each vertex v an interval $\phi(v)$ on the real line, such that for each pair $v, v' \in V$ of vertices, the following holds:

1. if v and v' have the same color, then $\phi(v)$ and $\phi(v')$ do not overlap,
2. if $\{v, v'\} \in E_1$ then $\phi(v)$ and $\phi(v')$ overlap, and
3. if $\{v, v'\} \notin E_2$ then $\phi(v)$ and $\phi(v')$ do not overlap.

Now the orderings on the real line of the intervals of each color give linear orderings of the fragments of each copy of the sequence which is consistent with the overlap information.

Suppose we have such an interval assignment ϕ for S. Consider the graph $G = (V, E)$, where $\{v, v'\} \in E$ if and only if the intervals corresponding to v and v' overlap. Clearly, G is an interval graph (Definition 2.3.1), the k-coloring of the sandwich graph S is also a k-coloring of G, and G is *sandwiched* in S, i.e. $E_1 \subseteq E \subseteq E_2$. Hence the maximum clique size of G is at most k.

On the other hand, suppose we have an interval graph $G = (V, E)$ which is sandwiched in sandwich graph $S = (V, E_1, E_2)$, and has clique size k for some $k \geq 1$. Then we can find an interval realization of G in linear time [Booth and Lueker, 1976; Hsu, 1993; Korte and Möhring, 1989]. Furthermore, there exists a k-coloring of G, which can easily be found in linear time from an interval realization. This means that, instead of finding a k-coloring and an interval assignment of the sandwich graph which satisfy conditions 1, 2 and 3 described above, we can find an interval graph G which is sandwiched in S and has maximum clique size at most k. We call such a graph G a *k-intervalization* of S. The decision problem can be modeled as follows [Golumbic, Kaplan, and Shamir, 1994].

INTERVALIZING SANDWICH GRAPHS (ISG)
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$
Question: Is there a k-intervalization of S?

It can be seen that there is not always a unique interval graph which is a solution to the problem. Furthermore, given an interval graph, there are usually no unique interval realizations and colorings of this graph. This means that, given a set of fragments of k copies of a sequence X, and the positive and non-negative overlap information of the fragments, there may be more than one assignment of the fragments to copies, and/or there may be more than one ordering of the fragments which satisfies the positive and non-negative overlap information, although only one of them is correct. However, when solving ISG, we only find one possibility, which is not guaranteed to be the correct one. Nevertheless, ISG, and especially the constructive version of ISG which also outputs a k-intervalization, may help to predict overlaps between fragments and to work towards reconstruction of the original sequence X.

In some applications of sequence reconstruction, the information about the copy of which fragments originate is not lost during the fragmentation process and, furthermore, only positive overlap information is used. The problem is to find, for each copy, a linear ordering of all fragments of this copy.

In this case, the input can be modeled as a graph $G = (V, E)$ and a k-coloring $c : V \rightarrow \{1, \ldots, k\}$ of G, where V denotes the set of fragments, E denotes the positive overlap information, and c represents information about the origin of each fragment: each color represents a copy of the sequence. The output of the problem can now be modeled as an interval graph $G' = (V, E')$, such that G' does not violate the positive overlap information, i.e. $E \subseteq E'$, and c is a k-coloring for G'. We call the graph G' a k-intervalization of G and c. The decision problem can be modeled as follows [Golumbic, Kaplan, and Shamir, 1994; Fellows, Hallett, and Wareham, 1993].

INTERVALIZING COLORED GRAPHS (ICG)
Instance: A simple graph $G = (V, E)$, an integer $k \geq 1$ and a k-coloring c for G
Question: Is there a k-intervalization of G and c?

This problem is a restricted version of ISG, since we can represent a graph $G = (V, E)$ and a k-coloring of G, by a sandwich graph $S = (V, E, E')$, where E' contains an edge between every two vertices which have different colors. A solution for ISG with input S and number k is also a solution for ICG with input graph G and k-coloring c, and vice versa.

In other applications, the fragmentation process generates fragments of equal length. The problem is again to find an assignment of fragments to copies and an interval assignment of the fragments which do not violate the overlap information, but an additional constraint is that the intervals must be equally long. The graph which is associated with the interval assignment is again an interval graph, but a stronger property holds: the graph is a *unit interval graph*.

Definition 4.1.2. An interval graph $G = (V, E)$ is a unit interval graph if there is an interval realization ϕ for G in which all intervals $\phi(v)$ ($v \in V$) have the same (unit) length.

Given a unit interval graph G, we can use the algorithm from Corneil, Kim, Natarajan, Olariu, and Sprague [1995] to find an interval realization of G in which all intervals have the same

length. Furthermore, we can again find a k-coloring of G in linear time, where k is the clique size of G. Hence we can restrict ourselves again to finding a unit interval graph G which is sandwiched in the input sandwich graph S and has clique size at most k. Such a graph G is called a *k-unit-intervalization* of S.

UNIT-INTERVALIZING SANDWICH GRAPHS (UISG)
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$
Question: Is there a k-unit-intervalization of S?

We can again restrict this problem to the case where we know the original copy of the sequence for each fragment, and only use positive overlap information. This gives rise to the problem: given a graph G and a k-coloring c of G, is there a unit interval graph $G' = (V, E')$, such that $E \subseteq E'$ and c is a k-coloring of G'? The graph G' is called a k-unit-intervalization of G and c.

UNIT-INTERVALIZING COLORED GRAPHS (UICG)
Instance: A simple graph $G = (V, E)$, an integer $k \geq 1$ and a k-coloring c for G
Question: Is there a k-unit-intervalization of G and c?

It has been shown that ICG, ISG, UICG and UISG are NP-complete (see Golumbic et al. [1994], Fellows et al. [1993] for ICG and ISG, and Goldberg, Golumbic, Kaplan, and Shamir [1995], Kaplan, Shamir, and Tarjan [1994] for IUCG and UISG). However, from the application it appears that the cases where k is some small given constant are of interest. For fixed k, we denote the four problems by k-ICG, k-ISG, k-UICG, and k-UISG, respectively.

Fellows et al. [1993] considered k-ICG for different fixed values of k. They showed that, although for fixed $k \geq 3$, yes-instances have bounded pathwidth (and hence bounded treewidth), standard methods for graphs with bounded treewidth will be insufficient to solve k-ICG, as the problem is not 'finite state' (see Section 2.2.4 for a definition). Also, they showed k-ICG to be hard for the complexity class $W[1]$, (which was strengthened by Bodlaender, Fellows, and Hallett [1994] to hardness for all classes $W[t]$, $t \in \mathbb{N}$). Hence k-ICG is probably not fixed parameter tractable (see also page 32). Clearly, the negative results of Fellows et al. [1993] also apply to k-ISG.

In Section 4.2 of this chapter we resolve the complexity of k-ISG and k-ICG for all constant values k. We observe that the case $k = 2$ is easy to resolve in $O(n)$ time. Then, we give an $O(n^2)$ algorithm that solves 3-ISG on *biconnected* graphs. We also show how the algorithm can be made constructive. The algorithm can be extended to an algorithm for 3-ISG on general graphs, which also runs in $O(n^2)$ time with $O(n^2)$ space. This algorithm consists of an extensive case analysis, based on the structure of partial two-paths as it is described in Chapter 3. In each case of the algorithm, a modification of our algorithm for biconnected graphs is used. As the description of the complete algorithm is very detailed and technical, it is not included in this thesis [de Fluiter and Bodlaender, 1997].

Furthermore, we show that 4-ICG is NP-complete. This implies NP-completeness of k-ICG and k-ISG for any fixed $k \geq 4$.

Kaplan et al. [1994] showed that k-UICG, and hence k-UISG, is hard for $W[1]$. Additionally, Kaplan and Shamir [1996] showed that k-UISG is solvable in polynomial time when k is fixed: they have given an $O(n^{k-1})$ time algorithm.

In Section 4.3 of this chapter we give algorithms which solve 3-UISG and 3-UICG for biconnected input graphs. The algorithm for 3-UISG runs in $O(|E_2|)$ time for sandwich graph $S = (V, E_1, E_2)$. This improves the algorithm of Kaplan and Shamir if $|E_2| = o(n^2)$. The algorithm for 3-UICG runs in $O(n)$ time, which improves the algorithm of Kaplan and Shamir by a factor n. The algorithms can be extended to obtain complete algorithms for solving 3-UISG and 3-UICG with the same time bounds. These algorithms are essentially the same as the algorithm for solving 3-ISG: the case analysis is very similar, but the building blocks are based on the linear time algorithms for biconnected graphs for 3-UISG and 3-UICG, respectively. They are not included in this thesis.

4.2 Intervalizing Sandwich Graphs

We first give a number of definitions and previously known results.

Let $S = (V, E_1, E_2)$ be a sandwich graph. For $i = 1, 2$, the graph (V, E_i) is denoted by $G_i(S)$. We call $G_1(S)$ the *underlying graph* of S. The set of vertices of S is also denoted by $V(S)$, the first edge set by $E_1(S)$ and the second edge set by $E_2(S)$. Let $W \subseteq V$. By $S[W]$ we denote the sub-sandwich graph of S induced by W, defined as follows:

$$\begin{aligned} V(S[W]) &= W \\ E_1(S[W]) &= E_1 \cap \{\{v,w\} \mid v,w \in W\} \\ E_2(S[W]) &= E_2 \cap \{\{v,w\} \mid v,w \in W\}. \end{aligned}$$

A sandwich graph is called biconnected if its underlying graph is biconnected. A biconnected sandwich graph is also called a *sandwich block*. The blocks of a sandwich graph are the blocks of its underlying graph.

The problem of k-intervalizing sandwich or colored graphs is closely related to the pathwidth problem.

Definition 4.2.1. Let $S = (V, E_1, E_2)$ be a sandwich graph. A *path decomposition* of S is a path decomposition $PD = (V_1, \ldots, V_t)$ of $G_1(S)$, such for each $v, v' \in V$, if there is a node V_i, $1 \leq i \leq t$, with $v, v' \in V_i$, then $\{v, v'\} \in E_2$. The pathwidth of S is the minimum width of any path decomposition of S.

The following lemma has been proved by Fellows et al. [1993] for intervalizations of colored graphs, and is a generalization of Lemma 2.3.3.

Lemma 4.2.1. *Let $S = (V, E_1, E_2)$ be a sandwich graph and let $k \geq 1$. Sandwich graph S has pathwidth at most $k \Leftrightarrow 1$ if and only if S has a k-intervalization.*

Proof. For the 'if' part, suppose $G = (V, E)$ is a k-intervalization of S. Let $\phi : V \to I$ be an interval realization for G. Let $(u_1, \ldots, u_n)$, $n = |V|$, be an ordering of V in such a way that for all i, j with $1 \leq i < j \leq n$, $\phi(u_i)$ starts on the left side of or at the same point as $\phi(u_j)$. For

each i let $V_i = \{ v \in V \mid \phi(v) \cap \phi(u_i) \neq \phi \}$. Then $PD = (V_1, \ldots, V_n)$ is a path decomposition of G in which there is an edge between two vertices v and v' if and only if there is a node V_i containing v and v'. Hence PD is a path decomposition of S. Furthermore, each node contains at most k vertices, since the clique size of G is k. Hence PD has pathwidth at most $k \Leftrightarrow 1$.

For the 'only if' part, suppose S has pathwidth at most $k \Leftrightarrow 1$, and let $PD = (V_1, \ldots, V_t)$ be a path decomposition of S of width at most $k \Leftrightarrow 1$. Then the interval completion G' of $G_1(S)$ for PD is an interval graph which is sandwiched in G, and has clique size most k. □

Thus, the following problem is equivalent to ISG.

SANDWICH PATHWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$
Question: Does S have pathwidth at most $k \Leftrightarrow 1$?

Note that the proof of Lemma 4.2.1 also gives an easy way to transform a solution for one problem into a solution for the other problem. Furthermore, it implies the following result.

Corollary 4.2.1. *Let $k \geq 1$ and let S be a sandwich graph. If there is a k-intervalization of S then the underlying graph of S has pathwidth at most $k \Leftrightarrow 1$.*

For the case $k = 2$, the question whether there is a path decomposition of a sandwich graph S is equal to the question whether the underlying graph of S is a partial one-path (see also Fellows et al. [1993]). This is because each path decomposition of width one of $G_1(S)$ can be transformed into a path decomposition of width one of S by simply deleting all nodes which contain no edge, and then adding a node at the right side of the path decomposition for each isolated vertex containing this vertex only. Checking whether a graph has pathwidth one can be done in linear time (Chapter 3).

Theorem 4.2.1. 2-*ISG can be solved in linear time.*

4.2.1 Three-Intervalizing Sandwich Blocks

By Corollary 4.2.1, a sandwich graph has a three-intervalization only if the underlying graph of S has pathwidth at most two. Therefore, our algorithm for finding a three-intervalization of a sandwich graph makes use of the structure of partial two-paths as described in Chapter 3. The algorithm first checks if the underlying graph $G_1(S)$ is a partial two-path and if so, finds its structure. Then this structure is used to find a three-intervalization of S.

In this section we give the algorithm for the case that the input sandwich graph is a block. The main algorithm has the following form: first, the cell completion $\bar{G}_1(S)$ of the underlying graph of S is computed. Then, a cycle path for $\bar{G}_1(S)$ is constructed if it exists (see Section 3.2). After that, this cycle path is used to check whether there is a path decomposition of S of width at most two.

Lemma 3.2.2 states that each path decomposition of width two of a partial two-path G is also a path decomposition of width two of its cell completion $\bar{G}$. With respect to intervalizations, the lemma states that each three-intervalization of a sandwich graph S is a supergraph of the cell completion $\bar{G}_1(S)$ of the underlying graph $G_1(S)$ of S.

The following lemma follows directly from the results in Section 3.2.

Lemma 4.2.2. *Let S be a sandwich block. Suppose that $G_1(S)$ is a partial two-path, $\bar{G_1}(S)$ is sandwiched in S, and (C,E) is a cycle path for $\bar{G_1}(S)$ with $C = (C_1,\ldots,C_p)$ and $E = (e_1,\ldots,e_{p-1})$. There is a path decomposition of S if and only if the following conditions hold:*

1. *there is a path decomposition of width two of $S[V(C_1)]$ with edge e_1 in the rightmost node (if $p > 1$),*
2. *there is a path decomposition of width two of $S[V(C_p)]$ with edge e_{p-1} in the leftmost node (if $p > 1$), and*
3. *for all i, $1 < i < p$, there is a path decomposition of width two of $S[V(C_i)]$ with edge e_{i-1} in the leftmost node and edge e_i in the rightmost node.*

Hence to check whether there is a path decomposition of width two of S with cycle path (C,E), the algorithm checks for each cycle C_i, $1 \leq i \leq p$, whether there is a path decomposition of $S[V(C_i)]$ with the appropriate edges in the leftmost and the rightmost node. The path decompositions of the sub-sandwich graphs induced by the cycles are then concatenated in the order in which they occur in C, and this gives a path decomposition of width two of S.

4.2.1.1 Cycles

We concentrate now on checking whether there exists a path decomposition of width two of a sandwich graph whose underlying graph is a cycle. Let S be such a sandwich graph and let $C = G_1(S)$.We denote the vertices and edges of C by $V(C) = \{v_0, v_1, \ldots, v_{n-1}\}$, and $E(C) = \{\,\{v_i, v_{i+1}\} \mid 0 \leq i < n\,\}$ (for each i, let v_i denote $v_{i \bmod n}$). For each j and l, $1 \leq l < n$, let $I(j,l)$ denote the set of vertices of $V(C)$ between v_j and v_{j+l}, when going from v_j to v_{j+l} in positive direction, i.e.,

$$I(j,l) = \{\,v_i \mid j \leq i \leq j+l\,\}.$$

Furthermore, let $C(j,l)$ denote the cycle with

$$V(C(j,l)) = I(j,l)$$
$$E(C(j,l)) = \{\{v_j, v_{j+l}\}\} \cup \{\,\{v_i, v_{i+1}\} \mid v_i \in I(j,l) \Leftrightarrow \{v_{j+l}\}\,\}$$

Note that $C(j, n \Leftrightarrow 1) = C$ for all j. For an example, consider Figure 4.1.

The following lemma is used to obtain a dynamic programming algorithm for our problem.

Lemma 4.2.3. *Let $S = (V, E_1, E_2)$ be a sandwich graph whose underlying graph is a cycle C with n vertices. Let i, j and l be integers, $2 \leq l < n$, and suppose $j \leq i < j+l$. There is a path decomposition $PD = (V_1,\ldots,V_t)$ of width two of $C(j,l)$ such that $\{v_i, v_{i+1}\} \subseteq V_1$ and $\{v_j, v_{j+l}\} \subseteq V_t$ if and only if $\{v_j, v_{j+l}\} \in E_2$ and either one of the following conditions holds:*

1. $|V(C)| = 3$,

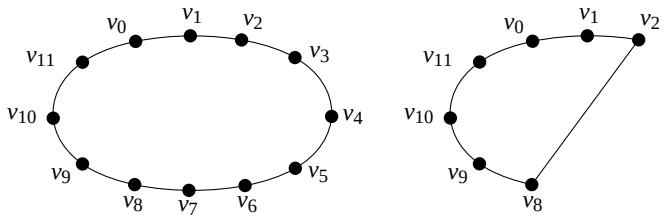


Figure 4.1. A cycle C with 12 vertices, and the cycle $C(8,6)$ derived from C.

2. *there is a path decomposition* $PD' = (V'_1, \ldots, V'_r)$ *of width two of* $S[I(j, l \Leftrightarrow 1)]$ *such that* $\{v_i, v_{i+1}\} \subseteq V'_1$ *and* $\{v_j, v_{j+l-1}\} \subseteq V'_r$, *or*
3. *there is a path decomposition* $PD'' = (V''_1, \ldots, V''_s)$ *of width two of* $S[I(j+1, l \Leftrightarrow 1)]$ *such that* $\{v_i, v_{i+1}\} \subseteq V''_1$ *and* $\{v_{j+1}, v_{j+l}\} \subseteq V''_s$.

Proof. For the 'if' part, suppose $\{v_j, v_{j+l}\} \in E_2$. If $|V(C)| = 3$, then $C(j,l) = C$, and hence $(V(C))$ is a path decomposition of width two of S. Suppose there is a path decomposition $PD' = (V'_1, \ldots, V'_r)$ of width two of $S[I(j, l \Leftrightarrow 1)]$ with $\{v_i, v_{i+1}\} \subseteq V'_1$ and $\{v_j, v_{j+l-1}\} \subseteq V'_r$. Then $PD = PD' +\!\!+ (\{v_j, v_{j+l-1}, v_{j+l}\})$ is a path decomposition of width two of $S[I(j,l)]$ which satisfies the appropriate conditions. The other case is similar.

For the 'only if' part, suppose there is a path decomposition $PD = (V_1, \ldots, V_t)$ of width two of $S[I(j,l)]$ such that $\{v_i, v_{i+1}\} \subseteq V_1$ and $\{v_j, v_{j+l}\} \subseteq V_t$. Clearly, $\{v_j, v_{j+l}\} \in E_2$, since $v_j, v_{j+l} \in V_t$. Suppose $|V(C)| > 3$. If $\{v_i, v_{i+1}\} = \{v_j, v_{j+l}\}$, then $l = n \Leftrightarrow 1$, hence $C(j,l) = C$ and $|I(j,l)| > 3$. Lemma 3.2.4 shows that the leftmost and the rightmost node of PD can not contain the same edge, contradiction. So $\{v_i, v_{i+1}\} \neq \{v_j, v_{j+l}\}$. Let V_m and $V_{m'}$, $1 \leq m, m' \leq t$, be the rightmost nodes containing edge $\{v_{j+1}, v_j\}$ and $\{v_{j+l-1}, v_{j+l}\}$, respectively.

First suppose $m' < m$. Then $V_m = \{v_{j+1}, v_j, v_{j+l}\}$, and for each k, $m < k \leq t$, $v_j, v_{j+l} \in V_k$. We claim that the path decomposition obtained from $(V_1, \ldots, V_m)$ by deleting v_j from each node is a path decomposition of width two of $S[I(j+1, l \Leftrightarrow 1)]$ with edge $\{v_{j+1}, v_{j+l}\}$ in the rightmost node and edge $\{v_i, v_{i+1}\}$ in the leftmost node.

Suppose there is a vertex $v \in V(C) \Leftrightarrow \{v_j, v_{j+1}\}$ which occurs on the right side of V_m. Vertex v has an edge to some vertex in $V(C) \Leftrightarrow \{v_j, v_{j+1}\}$, hence $v \in V_m$. But then $v = v_{j+l-1}$, which gives a contradiction. Hence all edges of $S[I(j+1, l \Leftrightarrow 1)]$ occur in $(V_1, \ldots, V_m)$. Furthermore, $\{v_{j+1}, v_{j+l-1}\}$ occurs in V_m. We only have to show $j \neq i$ and $j \neq i+1$. Node $V_{m'}$ contains v_{j+l}, v_{j+l-1}, and a vertex of the path from v_{j+1} to v_{i+1} which avoids v_j. Hence $v_j \notin V_{m'}$ and thus $v_j \notin V_1$. This proves the claim.

For the case that $m < m'$, a path decomposition of width two of $S[I(j, l \Leftrightarrow 1)]$ with $\{v_i, v_{i+1}\}$ in the leftmost node and $\{v_j, v_{j+l-1}\}$ in the rightmost node can be constructed in the same way.

If $m = m'$, then $v_{j+1} = v_{j+l-1}$, hence $|I(j,l)| = 3$. Since $\{v_i, v_{i+1}\} \neq \{v_j, v_{j+l}\}$, this means that $\{v_i, v_{i+1}\} = \{v_j, v_{j+1}\}$ or $\{v_i, v_{i+1}\} = \{v_{j+l-1}, v_{j+l}\}$. In the first case, $(\{v_i, v_{i+1}\})$ is a path decomposition of width two of $S[I(j, l \Leftrightarrow 1)]$ with edge $\{v_i, v_{i+1}\}$ in the leftmost node and edge $\{v_j, v_{j+l-1}\}$ in the rightmost node. In the latter case, $(\{v_i, v_{i+1}\})$ is a path decomposition of width two of $S[I(j+1, l \Leftrightarrow 1)]$ with edge $\{v_i, v_{i+1}\}$ in the leftmost node and edge $\{v_{j+1}, v_{j+l}\}$ in the rightmost node. □

Let S be a sandwich graph whose underlying graph is a cycle C. A *starting point* or *ending point* of S is an element of $E(C) \cup \{\mathsf{nil}\}$. Let $PD = (V_1, \ldots, V_t)$ be a path decomposition of S. We say that a starting point sp of S is in the leftmost node if either $sp \in E(C)$ and $sp \subseteq V_1$, or $sp = \mathsf{nil}$. We also denote this by $sp \in V_1$. Similarly, an ending point ep of S is in the rightmost node of PD, or $ep \in V_t$, if either $ep \in E(C)$ and $ep \subseteq V_t$, or $ep = \mathsf{nil}$.

We define $PW2$ as follows.

Definition 4.2.2. Let S be a sandwich graph of which the underlying graph is a cycle C with n vertices. Let sp be a starting point of S, and let j and l be integers, $1 \leq l < n$ and $0 \leq j < n$.

$$PW2(S, sp, j, l) = \begin{cases} \mathsf{true} & \text{if there is a path decomposition } PD = (V_1, \ldots, V_t) \\ & \text{of width two of } S[I(j,l)] \text{ with } v_j, v_{j+l} \in V_t \text{ and } sp \in V_1 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Let sp and ep be starting and ending points of a sandwich graph S of which the underlying graph is a cycle. There is a path decomposition of width two of S with sp in the leftmost node and ep in the rightmost node if and only if there is a j with $0 \leq j < n$ such that $PW2(S, sp, j, n \Leftrightarrow 1)$ holds and either $ep = \mathsf{nil}$ or $ep = \{v_{j-1}, v_j\}$.

If $n = 3$, then for any starting point sp and ending point ep, $(V(S))$ is a path decomposition of width two of S with sp in the leftmost node and ep in the rightmost node.

Suppose $n > 3$. It can be seen from the definition of $PW2$ that for all starting points sp of S, and all j, $0 \leq j < n$, $PW2(S, sp, j, 1)$ holds if and only if $sp = \mathsf{nil}$ or $sp = \{v_j, v_{j+1}\}$. We use this fact and Lemma 4.2.3 to describe $PW2$ recursively. Let sp be a starting point of S, and let j and l be integers with $1 \leq l < n$ and $0 \leq j < n$.

$$PW2(S, sp, j, l) = \begin{cases} sp = \mathsf{nil} \vee sp = \{v_j, v_{j+l}\} & \text{if } l = 1 \\ \{v_j, v_{j+l}\} \in E_2(S) \wedge & \\ (\, PW2(S, sp, j+1, l \Leftrightarrow 1) \vee PW2(S, sp, j, l \Leftrightarrow 1) \,) & \text{if } l > 1 \end{cases}$$

(Notice that $j+1$ denotes $(j+1) \bmod n$.)

We can now use dynamic programming to compute whether there is a path decomposition of width two of S with the appropriate starting and ending points as follows.

Algorithm 3-ISG_Cycle(S, sp, ep)
Input: Sandwich graph S with $G_1(S)$ a cycle C with n vertices $v_0, \ldots, v_{n-1}$,
and edges $\{\, \{v_i, v_{i+1}\} \mid 0 \leq i < n \,\}$
Starting point sp of S
Ending point ep of S

Output: ($\exists_{0\le j<n}\,(ep = \text{nil} \vee ep = \{v_{j-1}, v_j\}) \wedge PW2(S, sp, j, n \Leftrightarrow 1)$)

1. **if** $n = 3$ **then return** true
2. **if** $sp = \text{nil}$
3. **then for** $j \leftarrow 0$ **to** $n \Leftrightarrow 1$
4. **do** $P(j,1) \leftarrow$ true
5. **else for** $j \leftarrow 0$ **to** $n \Leftrightarrow 1$
6. **do** $P(j,1) \leftarrow$ false
7. Let j be such that $sp = \{v_j, v_{j+1}\} \in E(C)$
8. $P(j,1) \leftarrow$ true
9. $(* \; \forall_{0\le j<n} P(j,1) \equiv PW2(S, sp, j, 1) \; *)$
10. **for** $l \leftarrow 2$ **to** $n \Leftrightarrow 1$
11. **do for** $j \leftarrow 0$ **to** $n \Leftrightarrow 1$
12. **do** $P(j,l) \leftarrow (\{v_j, v_{j+l}\} \in E_2(S)) \wedge (P((j+1) \bmod n, l \Leftrightarrow 1) \vee P(j, l \Leftrightarrow 1))$
13. $(* \; \forall_{0\le j<n} P(j, n \Leftrightarrow 1) \equiv PW2(S, sp, j, n \Leftrightarrow 1) \; *)$
14. **if** $ep = \text{nil}$ **then return** true
15. Let j be such that $ep = \{v_{j-1}, v_j\}$
16. **return** $P(j, n \Leftrightarrow 1)$

The algorithm uses $O(n^2)$ time if we first build an adjacency matrix of the graph $G_2(S)$: this is needed in order to do the test in line 12 in constant time.

The algorithm can be made constructive in the sense that if there exists an intervalization, then the algorithm outputs one, as follows. Construct an array PP of pointers, such that for each j and l, $0 \le j < n$ and $1 \le l < n$, $PP(j,l)$ contains the nil pointer if $l = 1$ or if $P(j,l)$ is false. If $P(j,l)$ is true and $l > 1$, then $PP(j,l)$ contains a pointer to $PP(j, l \Leftrightarrow 1)$ if $P(j, l \Leftrightarrow 1)$ is true, and to $PP((j+1) \bmod n, l \Leftrightarrow 1)$ otherwise. The computation of PP can be done during the computation of P in 3-ISG_Cycle. Afterwards, if there is a three-intervalization, then one can be constructed as follows. First let G be the underlying graph of the input sandwich graph. If $ep = \text{nil}$, then start with any j, $0 \le j < n$ for which $P(j, n \Leftrightarrow 1)$ is true, otherwise. start with j for which $ep = \{v_{j-1}, v_j\}$. Then follow the pointers from $PP(j, n \Leftrightarrow 1)$ until the nil pointer is reached, and add edge $\{v_i, v_{i+l}\}$ to G for each i and l for which $PP(i,l)$ is visited. Note that the nil pointer is reached if the previous pointer pointed to $PP(i,1)$ for some i such that either $sp = \{v_i, v_{i+1}\}$ or $sp = \text{nil}$. Hence G is a three-intervalization of the input sandwich graph.

Lemma 4.2.4. *Algorithm 3-ISG_Cycle solves 3-*ISG *in $O(n^2)$ time and space for sandwich graphs of which the underlying graph is a cycle.*

4.2.1.2 Blocks

Let S be a sandwich block, suppose $G_1(S)$ is a partial two-path and $\bar{G}_1(S)$ is sandwiched in S. Let (C, E) be a cycle path for $G_1(S)$ with $C = (C_1, \ldots, C_p)$. There is a path decomposition of width two of S if and only if for each i, $1 \le i \le p$, there is a path decomposition of width two of $S[V(C_i)]$ with starting point e_{i-1} if $i > 1$, nil otherwise, and ending point e_i if $i < p$, nil otherwise (Lemma 4.2.2).

For a given sandwich block S, the following algorithm returns true if there is a three-intervalization of G, and false otherwise.

Algorithm 3-ISG_SB(S)
Input: Sandwich block S
Output: true if there is a three-intervalization of S, false otherwise
1. Check if $\bar{G_1}(S)$ is sandwiched in S, and if there is a cycle path for $\bar{G_1}(S)$. If so, construct such a path (C,E) with $C = (C_1,\ldots,C_p)$ and $E = (e_1,\ldots,e_{p-1})$. If not, **return** false.
2. **for** $i \leftarrow 1$ **to** p
3. **do** $m \leftarrow |V(C_i)|$
4. **if** $i > 1$ **then** $sp \leftarrow e_{i-1}$ **else** $sp \leftarrow$ nil
5. **if** $i < p$ **then** $ep \leftarrow e_i$ **else** $ep \leftarrow$ nil
6. **if** $\neg$3-ISG_Cycle($S[V(C_i)]$,sp,ep) **then return** false
7. **return** true

For Step 1, we can use the algorithm from Section 3.2, which takes $O(n)$ time. The loop in lines 2 – 6 runs in $O(n^2)$ time ($n = |V(G)|$) if we first make an adjacency matrix for $G_2(S)$, and then use procedure 3-ISG_Cycle.

Algorithm 3-ISG_SB can again be made constructive. To this end, the constructive version of algorithm 3-ISG_Cycle is used in line 6. After the loop has ended, the union of the graphs that are constructed by the calls to 3-ISG_Cycle form a three-intervalization of the input sandwich graph. Hence, we have proved the main result of this section.

Theorem 4.2.2. *There exists an $O(n^2)$ time algorithm that solves the constructive version* 3-ISG *for sandwich blocks.*

For three-intervalizing colored graphs we can use the same algorithm, with the only modification that in line 12 of algorithm 3-ISG_Cycle, we test whether v_j and v_{j+l} have different colors instead of testing whether $\{v_j, v_{j+l}\} \in E_2(S))$. Furthermore, we do not build an adjacency matrix of any graph, as this is not necessary.

4.2.2 Four-Intervalizing Sandwich Graphs

In this section we show that 4-ICG and 4-ISG are NP-complete. This shows that k-ICG and k-ISG are NP-complete for all $k \geq 4$. Unfortunately, in most practical cases, the number of colors is between five and fifteen. We feel however that the graphs that arise in the reduction of our NP-completeness proof will not be typical for the type of graphs that arise in sequence reconstruction applications. It may well be that special cases of ISG or ICG, which capture characteristics of the application data, have efficient algorithms.

Theorem 4.2.3. *4-ICG is NP-complete.*

Proof. Clearly, 4-ICG $\in$ NP. To prove NP-hardness, we transform from THREE-PARTITION, which is strongly NP-complete [Garey and Johnson, 1979].

THREE-PARTITION [SP15]
Instance: Integers $m \in \mathbf{N}$ and $Q \in \mathbf{N}$, a sequence $s_1, \ldots, s_{3m} \in \mathbf{N}$ such that $\sum_{i=1}^{3m} s_i = mQ$, and $\forall_{1 \le i \le 3m}\ \frac{1}{4}Q < s_i < \frac{1}{2}Q$.
Question: Can the set $\{1, \ldots, 3m\}$ be partitioned into m disjoint sets $S_1, \ldots, S_m$ such that $\forall_{1 \le j \le m} \sum_{i \in S_j} s_i = Q$?

Suppose we are given input $m, Q, s_1, s_2, \ldots, s_{3m} \in \mathbf{N}$. We define a graph $G = (V, E)$ and a four-coloring c of G, which consists of the following parts (see Figure 4.2).

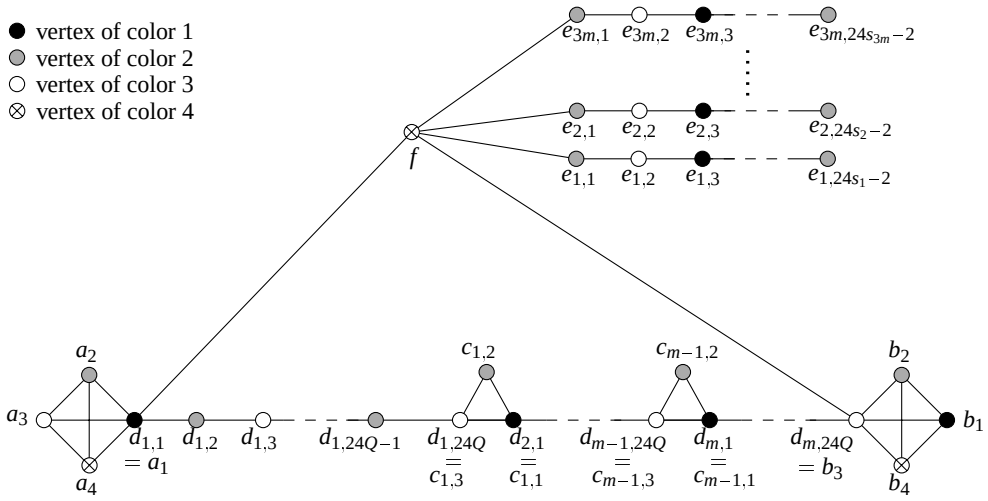


Figure 4.2. The constructed graph $G = (V, E)$ and its four-coloring.

Start clique. Take vertices $A = \{a_1, a_2, a_3, a_4\}$. Color vertex a_i with color i ($i = 1, 2, 3, 4$). Add edges between every two vertices in A.

End clique. Take vertices $B = \{b_1, b_2, b_3, b_4\}$. Color vertex b_i with color i ($i = 1, 2, 3, 4$). Add edges between every two vertices in B.

Middle cliques. Take vertices $C = \{c_{i,j} \mid 1 \le i \le m \Leftrightarrow 1,\ 1 \le j \le 3\}$. Color each vertex $c_{i,j} \in C$ with color j. Make each set $C_i = \{c_{i,1}, c_{i,2}, c_{i,3}\}$ into a clique.

Tracks. Take vertices $D = \{d_{i,j} \mid 1 \le i \le m,\ 1 \le j \le 24Q\}$. Color each vertex $d_{i,j} \in D$ with color 1 if $j \bmod 3 = 1$, with 2 if $j \bmod 3 = 2$ and with 3 if $j \bmod 3 = 0$. Identify vertex a_1 with $d_{1,1}$, vertex b_3 with $d_{m,24Q}$, and, for all i, $1 \le i \le m \Leftrightarrow 1$, identify $d_{i,24Q}$ with $c_{i,3}$, and $d_{i+1,1}$ with $c_{i,1}$. These track vertices form m paths: take edges $\{d_{i,j}, d_{i,j+1}\}$ for all i, j, $1 \le i \le m$, $1 \le j \le 24Q \Leftrightarrow 1$.

Number representing paths. Take vertices $E = \{e_{l,j} \mid 1 \leq l \leq 3m,\ 1 \leq j \leq 24s_l \Leftrightarrow 2\}$. Color each vertex $e_{l,j} \in E$ with color 2 if $j \bmod 3 = 1$, with color 3 if $j \bmod 3 = 2$, and with color 1 if $j \bmod 3 = 0$. For each l, the vertices $E_l = \{e_{l,j} \mid 1 \leq j \leq 24s_l \Leftrightarrow 2\}$ form a path: add edges $\{e_{l,j}, e_{l,j+1}\}$ for all l, j, $1 \leq l \leq 3m$, $1 \leq j \leq 24s_l \Leftrightarrow 3$.

Attachment vertex. Take one vertex f. Color f with color 4. Take edges $\{f, a_1\}$ $\{f, b_3\}$, and for all l, $1 \leq l \leq 3m$, edge $\{f, e_{l,1}\}$.

The four-colored graph, resulting from this construction, is the graph $G = (V, E)$. Note that the transformation can be done in polynomial time in Q and m.

Claim. *There exists a partition of the set $\{1, \ldots, 3m\}$ into sets $S_1, \ldots, S_m$ such that $\sum_{i \in S_j} s_i = Q$ for each j if and only if there is a four-intervalization of G and c.*

Proof. Suppose that G is a subgraph of a properly colored interval graph. So, we have a path decomposition $PD = (V_1, \ldots, V_r)$ of G, such no two vertices of the same color occur in the same node of PD. We may assume that there are no V_i, V_{i+1} with $V_i \subseteq V_{i+1}$ or $V_{i+1} \subseteq V_i$. (Otherwise, we may omit the smaller of these two sets from the path decomposition and still have a path decomposition of G.)

Note that, by the clique containment lemma (Lemma 2.2.3), there exist i_0 with $V_{i_0} = A$, and i_1 with $V_{i_1} = B$. Without loss of generality suppose $i_0 < i_1$. If $i_0 \neq 1$, then there exists a $v \in V_{i_0 - 1}$ with $v \notin A$. Note that such a vertex v has a path to a vertex in B that avoids A. It follows that V_{i_0} must contain a vertex from this path, but this will yield a color conflict with a vertex in A, contradiction. So, $i_0 = 1$. A similar argument shows that $i_1 = r$.

Also, from the clique containment lemma it follows that for each i, $1 \leq i \leq m \Leftrightarrow 1$, there is a j_i, $2 \leq j_i \leq r \Leftrightarrow 1$ with $C_i \subseteq V_{j_i}$. We have $j_1 < j_2 < j_3 < \cdots < j_{m-1}$, otherwise a color conflict will arise between a track vertex and a vertex in a set C_i. Write $j_0 = 1$, $j_m = r$. As there is a path from $d_{1,1}$ to $d_{m,24Q}$ in G that does not contain vertices with color 4 or vertices in E, it follows that each set V_i contains at least one vertex in $C \cup D$ with color 1, 2 or 3.

For each i, $1 \leq i \leq m$, call the interval $[j_{i-1} + 1, j_i \Leftrightarrow 1]$ the *ith valley*. Each vertex $d_{i,j}$ must be in one or more successive nodes V_α with α in the ith valley. It can not be in another valley, since this gives a color conflict. Note that, for each i, there are exactly $8Q$ vertices $d_{i,j}$ with color 2. For a vertex $d_{i,j}$ with color 2, we call the interval $\{\alpha \mid d_{i,j} \in V_\alpha\}$ a two-range. All two-ranges are disjoint, otherwise we have a color conflict. So, in each valley, we have exactly $8Q$ two-ranges.

For each l, $1 \leq l \leq 3m$, consider the vertices E_l. All vertices in E_l must be contained in nodes V_α with all α's in the same valley. Otherwise, the path induced by E_l will cross a middle clique, and we have a color conflict between a vertex in E_l and a vertex in C. Write $S_i = \{l \mid$ vertices in E_l are in sets V_α with α in the ith valley$\}$. We show that $S_1, \ldots, S_m$ is a partition of $\{1, \ldots, 3m\}$ such that for each j, $\sum_{i \in S_j} s_i = Q$.

For each edge $\{e_{l,j}, e_{l,j+1}\}$ with $e_{l,j}$ of color 3 (and hence, $e_{l,j+1}$ has color 1), there must be a node α with $\{e_{l,j}, e_{l,j+1}\} \subseteq V_\alpha$. α must be in a two-range, as otherwise V_α contains a one-colored or three-colored vertex from $C \cup D$, and we have a color conflict. If there exists an α with $\{e_{l,j}, e_{l,j+1}, d_{i,j'}\} \subseteq V_\alpha$, with $d_{i,j'}$ of color 2, then we say that the two-range of $d_{i,j'}$ contains the 1-3-E-edge $\{e_{l,j}, e_{l,j+1}\}$.

Claim. *No two-range contains two or more 1-3-E-edges.*

Proof. Suppose $\{e_{l_1,j_1}, e_{l_1,j_1+1}\}$ and $\{e_{l_2,j_2}, e_{l_2,j_2+1}\}$ are distinct 1-3-E-edges, and there is a $d_{i,j'}$ such that $\{e_{l_1,j_1}, e_{l_1,j_1+1}, d_{i,j'}\} \subseteq V_\alpha$, $\{e_{l_2,j_2}, e_{l_2,j_2+1}, d_{i,j'}\} \subseteq V_\beta$. Suppose w.l.o.g. that $\alpha < \beta$. Note that both $v = e_{l_1,j_1}$ and $w = e_{l_1,j_1+1}$ are adjacent to a vertex with color 2. Let $[\gamma, \delta]$ be the two-range of $d_{i,j'}$. Note that $\gamma \leq \alpha < \beta \leq \delta$. If $V_{\gamma-1}$ contains a vertex with color 1 from $C \cup D$, then consider the vertex w with color 1. It can not belong to $V_{\gamma-1}$ and it can not belong to V_β. So, if $w \in V_\varepsilon$, then $\gamma \leq \varepsilon \leq \delta$. Hence, there can not be a set V_ε that contains w and its e_{l_1,j_1+2} with color 2, contradiction. If $V_{\gamma-1}$ does not contain a vertex with color 1 from $C \cup D$, then it contains a three-colored vertex from $C \cup D$, and by considering v and using a similar argument, a contradiction also arises. □

Let $1 \leq i \leq m$. Suppose $S_i = \{l_1, l_2, \ldots, l_t\}$. Note that $E_{l_1} \cup \cdots \cup E_{l_t}$ induces $8s_{l_1} \Leftrightarrow 1 + 8s_{l_2} \Leftrightarrow 1 + \cdots + 8s_{l_t} \Leftrightarrow 1$ 1-3-E-edges. As there are $8Q$ two-ranges in a valley, we must have

$$8(s_{l_1} + s_{l_2} + \cdots s_{l_t}) \Leftrightarrow t \leq 8Q$$

By noting that each $s_l \geq Q/4 + 1/4$, it follows that $8(Q/4 + 1/4)t \Leftrightarrow t \leq 8Q$, so $t \leq 3$, and that hence also, by integrality,

$$8(s_{l_1} + s_{l_2} + \cdots s_{l_t}) \leq 8Q$$

So, we have a partition of $\{1, \ldots, 3m\}$ into sets $S_1, \ldots, S_m$, such that for all $j, 1 \leq j \leq m$, $\sum_{i \in S_j} s_i \leq Q$. As $\sum_{j=1}^{m} \sum_{i \in S_j} s_i = mQ$, it follows that for all j, $1 \leq j \leq m$, $\sum_{i \in S_j} s_i = Q$.

Now, suppose $S_1, S_2, \ldots, S_m$ is a partition of $\{1, \ldots, 3m\}$, such that for all j, $1 \leq j \leq m$, $\sum_{i \in S_j} s_i = Q$. We will give a path decomposition $PD = (V_1, \ldots, V_r)$ of $G = (V, E)$, such that no V_i contains two vertices of the same color. We leave most of the easy verification that the given path decomposition fulfills the requirements to the reader.

Take $t = 48Q$, $r = mt + 1$. Take $V_1 = A$, $V_r = B$. For each vertex $c_{i,j} \in C$, put $c_{i,j}$ in node V_{ti+1}. For each vertex $d_{i,j} \in D$, put $d_{i,j}$ in node $V_{t(i-1)+2j-1}$, $V_{t(i-1)+2j}$, and $V_{t(i-1)+2j+1}$. (Note that each vertex occurs in consecutive nodes; even vertices with two names like the vertices with names $d_{i,24Q}$ and $c_{i,3}$ for $1 \leq i < m$.)

For each i, $1 \leq i \leq m$, suppose $S_i = \{l_1, l_2, l_3\}$. Put vertex $e_{l_1,1}$ in node $V_{t(i-1)+2}$. For all j, $2 \leq j \leq 24s_{l_1} \Leftrightarrow 2$, put vertex $e_{l_1,j}$ in nodes $V_{t(i-1)+2j-2}$, $V_{t(i-1)+2j-1}$, $V_{t(i-1)+2j}$.

For all j, $1 \leq j \leq 24s_{l_2} \Leftrightarrow 2$, put vertex $e_{l_2,j}$ in nodes $V_{t(i-1)+48s_{l_1}+2j-2}$, $V_{t(i-1)+48s_{l_1}+2j-1}$, $V_{t(i-1)+48s_{l_1}+2j}$. For all j, $1 \leq j \leq 24s_{l_3} \Leftrightarrow 2$, put vertex $e_{l_3,j}$ in nodes $V_{t(i-1)+48s_{l_1}+48s_{l_2}+2j-2}$, $V_{t(i-1)+48s_{l_1}+48s_{l_2}+2j-1}$, $V_{t(i-1)+48s_{l_1}+48s_{l_2}+2j}$.

Finally, put f in all nodes $V_2, \ldots, V_{r-1}$.

A straightforward, but somewhat tedious verification shows that the resulting path decomposition PD is indeed a path decomposition of G, and that no node V_i contains two different vertices with the same color, and hence PD has pathwidth at most three. □

As three-partition is strongly NP-complete and our transformation is polynomial in Q and m, the claimed theorem now follows. □

Note that we even proved a slightly stronger result.

Corollary 4.2.2. *4-ICG is NP-complete, even if there is one color that is only given to three vertices of the input graph.*

Since ICG is a restricted version of ISG, we also have the following result.

Corollary 4.2.3. *4-ISG is NP-complete.*

4.3 Unit-Intervalizing Sandwich Graphs

In this section, we consider the problems UNIT-INTERVALIZING SANDWICH GRAPHS (or UISG) and UNIT-INTERVALIZING COLORED GRAPHS (or UICG).

Kaplan and Shamir [1996] have shown a relationship between a unit-intervalization of a sandwich graph, its bandwidth, and its *proper* pathwidth. Before citing this result, we first define the bandwidth and proper pathwidth of a sandwich graph.

Let $G = (V,E)$ be a graph and $PD = (V_1, V_2, \ldots, V_t)$ a path decomposition. Let

$$E' = E \cup \{\{u,v\} \mid \exists_i u,v \in V_i\}.$$

Recall from Lemma 2.3.2 that (V,E') is an interval graph, and is called the interval completion of G for PD.

Definition 4.3.1. A path decomposition $PD = (V_1, \ldots, V_t)$ of a (sandwich) graph is called a *proper path decomposition* if for each $v,w \in V$, the occurrence of v is not properly contained in the occurrence of w, i.e. if w occurs in all nodes containing v, then v occurs in all nodes containing w.

The *proper pathwidth* of a (sandwich) graph is defined as the minimum width of all proper path decompositions of the (sandwich) graph.

Definition 4.3.2. Let $S = (V, E_1, E_2)$ be a sandwich graph. A layout of S is a layout of $G_1(S)$ (Definition 2.3.2). A layout ℓ of S is called a *legal* layout of S if, for each $v,w,v',w' \in V$, $\ell(v) \leq \ell(v') < \ell(w') \leq \ell(w)$ and $\{v,w\} \in E_1$ implies that $\{v',w'\} \in E_2$.

The *bandwidth* of S is the minimum bandwidth of all legal layouts of S. If there is no legal layout of S, then the bandwidth of S is ∞.

The following lemma has been proved implicitly by Kaplan and Shamir [1996].

Lemma 4.3.1. *Let $G = (V,E)$ be a graph and ℓ a layout of G of bandwidth k. Let $G' = (V,E')$ be the supergraph of G with E' defined as follows.*

$$E' = \{\{v,w\} \mid v,w \in V \wedge \exists_{\{v',w'\} \in E}\ \ell(v') \leq \ell(v) < \ell(w) \leq \ell(w')\}$$

G' is an interval graph with maximum clique size $k+1$.

Proof. We construct an interval realization ϕ of G'. For each $v \in V$, let $m(v) = \max\{\ell(w) \mid \ell(w) \geq \ell(v) \wedge (\{v,w\} \in E' \vee v = w)\}$. For each $v \in V$, let $\phi(v) = [\ell(v), m(v)]$. Clearly, if there is an edge $\{v,w\} \in E'$, then $\phi(v)$ and $\phi(w)$ overlap. Now suppose $\phi(v)$ and $\phi(w)$ overlap, and w.l.o.g. suppose that $\ell(v) < \ell(w)$. Then $m(v) \geq \ell(w)$, hence there is a $u \in V$, such that $\ell(u) \geq \ell(w)$ and $\{v,u\} \in E'$. By definition of G', this means that $\{v,w\} \in E'$. This proves that G' is an interval graph.

Since each graph of bandwidth k has maximum clique size at most $k+1$, and ℓ is a layout of G', this means that G' has maximum clique size k. □

G' is called the interval completion of G for ℓ. The following theorem has been proved by Kaplan and Shamir [1996].

Theorem 4.3.1 [Kaplan and Shamir, 1996]. *Let S be a sandwich graph and let G be a graph with $V(G) = V(S)$. The following statements are equivalent.*

1. *G is a k-unit-intervalization of S.*
2. *There is a proper path decomposition PD of width $k \Leftrightarrow 1$ of S such that G is the interval completion of $G_1(S)$ for PD.*
3. *There is a legal layout ℓ of S of bandwidth $k \Leftrightarrow 1$ such that G is the interval completion of $G_1(S)$ for ℓ.*

As an example of Theorem 4.3.1, consider Figure 4.3. It shows a sandwich graph S, with $V(S) = \{1,2,\ldots,12\}$. The solid edges denote the edges in $E_1(S)$, and the dashed edges denote the edges in $E_2(S)$ which are not in $E_1(S)$. The graph G depicted in the figure is a three-unit-intervalization of S: G is sandwiched in S, and ϕ is an interval realization of G in which all intervals are of equal length, which means that G is a unit-interval graph. Furthermore, the path decomposition PD that is depicted is a proper path decomposition of width two of S, and ℓ is a legal layout of bandwidth two of S. The layout ℓ is depicted by the ordering of the vertices. The edges drawn in this layout are the edges in $E_1(S)$. It is easy to see that the unit-interval graph G is the interval completion of $G_1(S)$ for PD and for ℓ.

Thus, the following problems are equivalent to UISG, and furthermore, a solution of one problem can easily be transformed into a solution of another problem (as can be seen from the proofs of Kaplan and Shamir [1996]).

SANDWICH PROPER PATHWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, integer $k \geq 1$
Question: Does S have proper pathwidth at most $k \Leftrightarrow 1$?

SANDWICH BANDWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, integer $k \geq 1$
Question: Does S have bandwidth at most $k \Leftrightarrow 1$?

Consider the problem 2-UISG. A connected graph has bandwidth at most one if and only if it is a path. Furthermore, if the underlying graph of a sandwich graph S is a path, then any layout of bandwidth one of $G_1(S)$ is a legal layout of bandwidth one of S. Checking this can clearly be done in $O(n)$ time (although $|E_2(S)|$ may be $\omega(n)$).

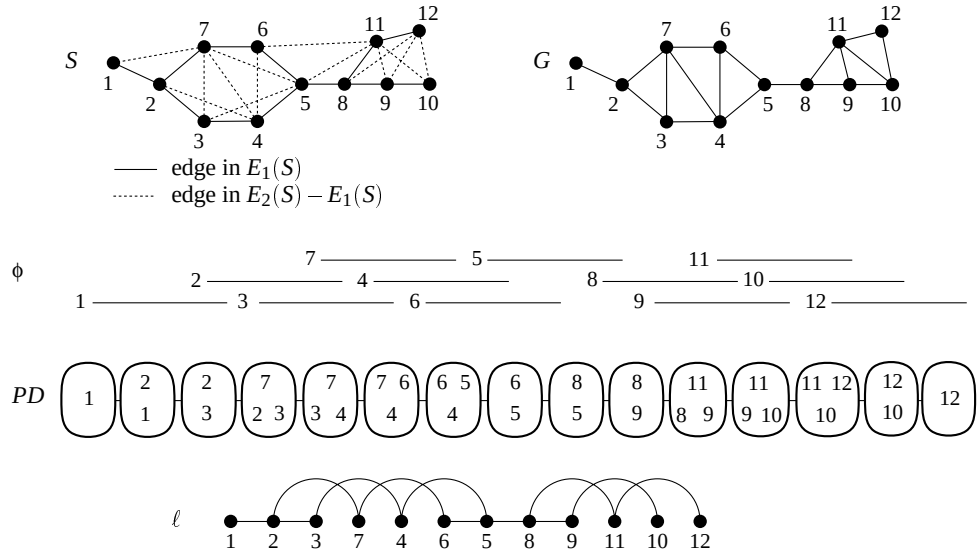


Figure 4.3. An example of Theorem 4.3.1.

Theorem 4.3.2. *2-UISG can be solved in $O(n)$ time.*

For 3-UISG, we use an algorithm which has the same structure as our algorithm for 3-ISG. A necessary condition for a sandwich graph S to admit a k-unit-intervalization is that the underlying graph $G_1(S)$ has pathwidth at most $k \Leftrightarrow 1$, since the proper pathwidth of a graph is at least its pathwidth. Therefore, we use the characterization of partial two-paths given in Chapter 3 for our algorithm: first the algorithm checks if $G_1(S)$ has pathwidth at most two, and if so, finds the structure as given in Chapter 3. This structure is then used to solve the problem.

In the remainder of this section, we first give an algorithm for solving 3-UISG on biconnected graphs. After that, we show how this algorithm can be improved for 3-UICG on biconnected graphs.

4.3.1 Three-Unit-Intervalizing Biconnected Sandwich Graphs

We start by considering the case that the underlying graph of the input sandwich graph is a cycle. After that, we extend the algorithm to sandwich blocks.

4.3.1.1 Cycles

First we study the structure of layouts of bandwidth two of a cycle.

Lemma 4.3.2. *Let $C = (V,E)$ be a cycle. The following holds.*

1. *In each layout ℓ of bandwidth two of C, the first two vertices in the ordering induced by ℓ are adjacent.*
2. *Let $\{u,w\} \in E$. There is exactly one layout ℓ of C of bandwidth two in which $\ell(u) = 1$ and $\ell(w) = 2$. Furthermore, for each i, $1 \leq i \leq n$, there is a vertex $u_i \in V$ with $\ell(u_i) = i$, and*

$$E = \{\{u_1,u_2\},\{u_{n-1},u_n\}\} \cup \{\{u_i,u_{i+2}\} \mid 1 \leq i \leq n \Leftrightarrow 2\}. \tag{4.1}$$

Proof.
1. Let ℓ be a layout of C of bandwidth two which starts at one, i.e. there is a vertex $u \in V$ for which $\ell(u) = 1$. Each vertex in C has degree two, and hence u has degree two. Let $v,w \in V$ be the neighbors of u. Then either $\ell(v) = 2$ and $\ell(w) = 3$, or vice versa.

2. Let ℓ be a layout of width two of C, and order the vertices in V corresponding to ℓ, i.e. let $V = \{u_1,\ldots,u_n\}$ such that for each $i < j$, $\ell(u_i) < \ell(u_j)$. We claim that equation (4.1) holds and that for each i, $\ell(u_i) = i$.

The proof is by induction on n. If $n = 3$, the claim clearly holds. Suppose $n > 3$. The vertices u_{n-1} and u_{n-2} are the neighbors of u_n. Hence the graph G which is obtained from C by removing vertex u_n and its incident edges, and adding an edge between vertices u_{n-2} and u_{n-1} is also a cycle, but with $n \Leftrightarrow 1$ vertices. Furthermore, ℓ restricted to $V(G)$ is a layout of G of bandwidth two. By the induction hypothesis,

$$E(G) = \{\{u_1,u_2\},\{u_{n-2},u_{n-1}\}\} \cup \{\{u_i,u_{i+2}\} \mid 1 \leq i \leq n \Leftrightarrow 3\},$$

and for each i, $1 \leq i \leq n \Leftrightarrow 1$, $\ell(u_i) = i$. It follows that Equation (4.1) holds. Furthermore, since $\{u_{n-2},u_n\} \in E(C)$, it must be the case that $\ell(u_n) \leq \ell(u_{n-2}) + 2$. Hence $\ell(u_n) = n$. □

Figure 4.4 shows the unique layout for a cycle with ten vertices as given in Lemma 4.3.2.

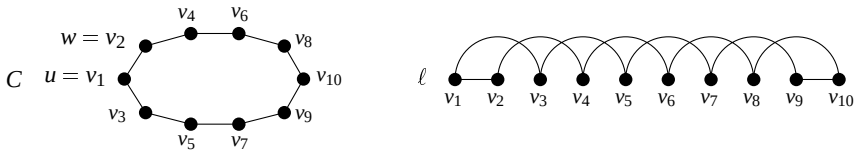


Figure 4.4. A cycle C with ten vertices, and the unique layout ℓ of bandwidth two of C with $\ell(u) = 1$ and $\ell(w) = 2$.

Lemma 4.3.2 implies that, for each cycle C with n vertices, there are exactly $2n$ layouts of bandwidth two of C which start at one: for each edge $\{v,w\} \in E(C)$, there is a layout with $\ell(v) = 1$ and $\ell(w) = 2$, and a layout with $\ell(w) = 1$ and $\ell(v) = 2$.

Suppose C is a cycle with n vertices with

$$V(C) = \{v_0,v_1,\ldots,v_{n-1}\} \text{ and } E(C) = \{\{v_i,v_{(i+1) \bmod n}\} \mid 0 \leq i < n\}.$$

For each i, $0 \leq i < n$, let ℓ^i_+ denote the layout of C of bandwidth two with $\ell^i_+(v_i) = 1$ and $\ell^i_+(v_{(i+1) \bmod n}) = 2$, and let ℓ^i_- denote the layout of C of bandwidth two with $\ell^i_-(v_i) = 1$ and $\ell^i_-(v_{(i-1) \bmod n}) = 2$. Furthermore, for each i, $0 \leq i < n$, let G^i_+ and G^i_- denote the interval completions of C for ℓ^i_+ and ℓ^i_-, respectively. Figures 4.5 and 4.6 show examples of ℓ^i_+, G^i_+, ℓ^i_- and G^i_- for the case that n is even and n is odd, respectively.

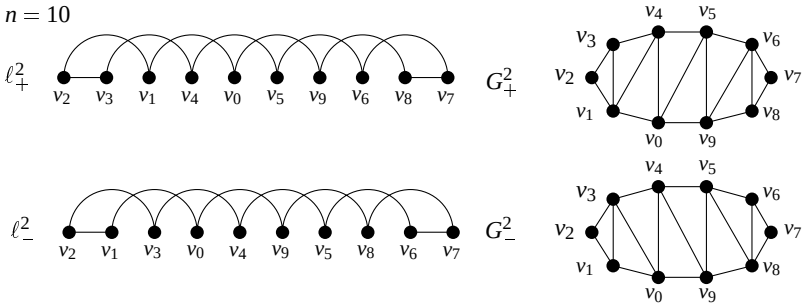


Figure 4.5. Examples of graphs G^2_+ and G^2_- for $n = 10$.

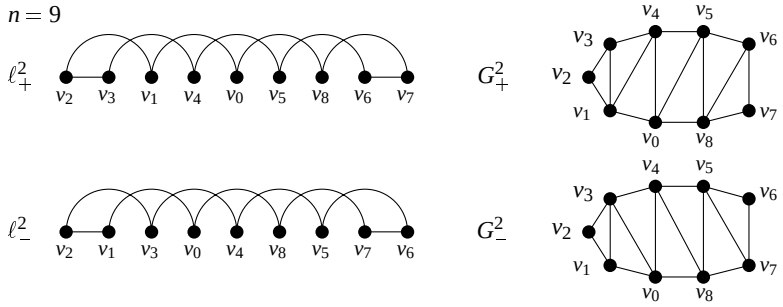


Figure 4.6. Examples of graphs G^2_+ and G^2_- for $n = 9$.

Suppose that n is even. Lemma 4.3.2 implies that for each i, $0 \leq i < n/2$, $G^i_+ = G^{i+n/2}_+$: for each $v \in V$, $\ell^i_+(v) = n+1 \Leftrightarrow \ell^{i+n/2}_+(v)$. Similarly, $G^i_- = G^{i+n/2}_-$. For example, in Figure 4.5, it can be seen that, if $n = 10$, then $G^2_+ = G^7_+$ and $G^2_- = G^7_-$. Furthermore, for each i, j, $0 \leq i, j < n/2$, $G^i_+ \neq G^j_-$, and if $i \neq j$, then $G^i_+ \neq G^j_+$ and $G^i_- \neq G^j_-$. This means that there are exactly n different interval completions of layouts of bandwidth two of C. Let $\{H_0, H_1, \ldots, H_{n-1}\}$ denote the set of all these graphs, and let $\{\ell_0, \ldots, \ell_{n-1}\}$ denote the set of layouts, such that for each i, $0 \leq i < n/2$, $H_i = G^i_+$ and $\ell_i = \ell^i_+$, and furthermore, $H_{i+n/2} = G^i_-$ and ℓ^i_-.

Consider the case that n is odd. For each i, $0 \leq i < n$, $G_+^i = G_-^{(i+\frac{n+1}{2}) \bmod n}$. For example, figure 4.6 shows that, if $n = 9$, then $G_+^2 = G_-^7$ and $G_+^6 = G_-^2$. Furthermore, for each $0 \leq i < j < n$, $G_+^i \neq G_+^j$ and $G_-^i \neq G_-^j$. This means again that there are exactly n different interval completions of layouts of bandwidth two of C. Let $\{H_0, H_1, \ldots, H_{n-1}\}$ again denote this set and let $\{\ell_0, \ldots, \ell_{n-1}\}$ denote a set of layouts, such that for each i, $0 \leq i < n$, $H_i = G_+^i$ and $\ell_i = \ell_+^i$.

For each $0 \leq i < j < n$ with $1 < j \Leftrightarrow i < n \Leftrightarrow 1$, we call $\{v_i, v_j\}$ a *diagonal* of C. The preceding discussion implies the following result.

Lemma 4.3.3. *Let $\{v_i, v_j\}$ be a diagonal of C. There are integers m_1 and m_2, $0 \leq m_1 < m_2 < n$, such that the edge $\{v_i, v_j\}$ is contained in H_{m_1} and H_{m_2} and in no other H_p, $p \neq m_1, m_2$. Furthermore, m_1 and m_2 can be computed in constant time from i, j and n.*

Proof. If $j \Leftrightarrow i$ is even, then $\{v_i, v_j\}$ is an edge in $G_+^{\frac{i+j}{2}}$ and in $G_-^{\frac{i+j}{2}}$ (and $G_+^{\frac{i+j}{2}} \neq G_-^{\frac{i+j}{2}}$). If $j \Leftrightarrow i$ is odd, then $\{v_i, v_j\}$ is an edge in $G_+^{\frac{i+j-1}{2}}$ and in $G_-^{\frac{i+j+1}{2}}$ (and $G_+^{\frac{i+j-1}{2}} \neq G_-^{\frac{i+j+1}{2}}$). This implies that m_1 and m_2 exist and can be easily computed from i, j and n. Also, it can be seen that there is no p, $0 \leq p < n$, with $p \neq m_1, m_2$ and $\{v_i, v_j\}$ is an edge in H_p. □

Suppose we have an input sandwich graph $S = (V, E_1, E_2)$, such that $G_1(S)$ is a cycle C with vertices $V = \{v_0, \ldots, v_{n-1}\}$ as before. Notice that a graph G is a three-unit-intervalization of S if and only if there is an i, $0 \leq i < n$, such that $G = H_i$. If there is such an i, then ℓ_i is a legal layout of S of bandwidth two. Furthermore, for each i, there are exactly $n \Leftrightarrow 3$ edges in $E(H_i)$ which are not in C. If H_i is a three-unit-intervalization of S, then $E(H_i) \subseteq E_2$. This implies the following algorithm for three-unit-intervalizing sandwich graphs of which the underlying graph is a cycle.

For each i, $0 \leq i < n$, we compute an integer $N(i)$ which denotes the number of edges of $E_2 \Leftrightarrow E_1$ which are also edges in H_i. By the preceding discussion, there is a three-unit-intervalization of S if and only if there is an i, $0 \leq i < n$, such that $N(i) = n \Leftrightarrow 3$. The algorithm for deciding 3-UISG on sandwich graphs S for which $G_1(S)$ is a cycle C with n vertices is now as follows.

Algorithm 3-UISG_Cycle(S)
Input: Sandwich graph S for which $G_1(S)$ is a cycle
Output: true if there is a three-unit-intervalization of S, false otherwise

1. Number the vertices of $G_1(S)$ by $v_0, \ldots, v_{n-1}$, such that
   for each j, $0 \leq j < n$, $\{v_j, v_{(j+1) \bmod n}\} \in E_1$.
2. **for** $i \leftarrow 0$ **to** $n \Leftrightarrow 1$
3. **do** $N(i) \leftarrow 0$
4. **for all** $\{v_i, v_j\} \in E_2$
5. **do if** $1 < |j \Leftrightarrow i| < n \Leftrightarrow 1$
6. **then** Compute the values of m_1 and m_2 as in Lemma 4.3.3 and increase $N(m_1)$ and $N(m_2)$ by one.
7. **for** $i \leftarrow 0$ **to** $n \Leftrightarrow 1$

8. **do if** $N(i) = n \Leftrightarrow 3$ **then return** true
9. **return** false

By the preceding discussion, this algorithm returns true if and only if there is a three-unit-intervalization of S. It runs in $O(|E_2|)$ time. We can easily modify the algorithm such that it returns a three-unit-intervalization if one exists: it is easy to construct the graph H_i for a given value of i.

Lemma 4.3.4. *Algorithm 3-UISG_Cycle solves 3-*UISG *in $O(|E_2(S)|)$ time and space for sandwich graphs S of which the underlying graph is a cycle.*

4.3.1.2 Blocks

In the following lemma we give a necessary condition for a biconnected partial two-path to have bandwidth two.

Lemma 4.3.5. *Let G be a biconnected partial two-path. If G has bandwidth two then G is a path of cycles in which each edge occurs in at most two chordless cycles.*

Proof. Suppose G has bandwidth two and let ℓ be a layout of G of bandwidth two. Let (C, E) be a cycle path for $\overline{G}$ with $C = (C_1, C_2, \ldots, C_p)$ and $E = (e_1, e_2, \ldots, e_{p-1})$. Let $W \subseteq V(G)$ be the set of vertices v for which there is an i, $1 < i < p$, such that $v \in V(C_i)$, $e_{i-1} = e_i$ and $v \notin e_i$. Consider the cycle C of $\overline{G}$ which is obtained from $\overline{G}$ by first removing all vertices from W and their incident edges, and then removing all edges e_i, $1 \leq i < p$, which occur more than once in E. Note that C is a subgraph of G.

The function ℓ' that is obtained by restricting ℓ to $V(C)$ is a layout of bandwidth two of C. Order the vertices of C by $u_1, \ldots, u_n$, such that for each $i < j$, $\ell(u_i) < \ell(u_j)$. By Lemma 4.3.2, $\ell(u_n) \Leftrightarrow \ell(u_1) = n \Leftrightarrow 1$. This means that for each vertex $w \in W$, either $\ell(w) < \ell(u_1)$ or $\ell(w) > \ell(u_n)$. If $\ell(w) < \ell(u_1)$, then w can only be adjacent to u_1 and u_2. But $\{u_1, u_2\} \in \underline{E}(C)$, contradiction. If $\ell(w) > \ell(u_n)$, we also get a contradiction. Hence $W = \phi$, and thus $\overline{G}$ is a path of cycles in which each edge occurs in at most one chordless cycle. This also implies that $\overline{G} = G$, which completes the proof. □

The next lemma easily follows from the preceding discussion.

Lemma 4.3.6. *Let $S = (V, E_1, E_2)$ be a sandwich graph, suppose $G_1(S)$ is a path of cycles in which each edge occurs in at most two chordless cycles. Let $F \subseteq E_1$ be the set of edges which occur in two chordless cycles, and let C be the cycle obtained from $G_1(S)$ by removing all edges in F, i.e. $C = (V, E_1 \Leftrightarrow F)$. A graph G is a three-unit-intervalization of S if and only if G is a three-unit-intervalization of C and $F \subseteq E(G) \subseteq E_2$.*

Let $S = (V, E_1, E_2)$ be a sandwich graph, suppose $G_1(S)$ is a path of cycles in which each edge occurs in at most two chordless cycles. Let $F \subseteq E_1$ denote the edges of $G_1(S)$ which occur in two chordless cycles, and let $C = (V, E_1 \Leftrightarrow F)$. Furthermore, let $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and $E = \{\{v_i, v_{(i+1) \bmod n}\} \mid 0 \leq i < n\}$. Lemma 4.3.5 shows that we can use an algorithm similar to the cycle algorithm to compute whether S has bandwidth two: together with the

array N we now compute an array B of positive integers. For each i, $0 \leq i < n$, $B(i)$ denotes the number of edges in F which occur in H_i. The array B can be computed in $O(|F|)$ time in the same way as we computed the array N. It can be seen that G_i, $0 \leq i < n$, is a three-unit-intervalization of S if and only if $N(i) = n \Leftrightarrow 3$ and $B(i) = |F|$. This implies the following algorithm for 3-UISG on biconnected sandwich graphs S.

Algorithm 3-UISG_BSG(S)
Input: Biconnected sandwich graph S
Output: true if there is a three-unit-intervalization of S, false otherwise
1. Check if $G_1(S)$ has pathwidth two, and if so find a cycle path (C,E) ($C = (C_1,\ldots,C_p)$, $E = (e_1,\ldots,e_{p-1})$) for $G_1(S)$. If not, **return** false.
2. **for** $i \leftarrow 1$ **to** $p \Leftrightarrow 1$
3. **do if** $e_i = e_{i+1}$ **then return** false
4. Let C be the cycle obtained from $G_1(S)$ by removing all edges of E.
5. Number the vertices of C by $v_0,\ldots,v_{n-1}$ such that for each i, $\{v_i, v_{(i+1) \bmod n}\} \in E(C)$.
6. Compute the arrays N and B.
7. **for** $i \leftarrow 0$ **to** $n \Leftrightarrow 1$
8. **do if** $N(i) = n \Leftrightarrow 3$ and $B(i) = p \Leftrightarrow 1$ **then return** true
9. **return** false

Step 1 is described in Section 3.2 and can be done in $O(n)$ time. The loop in lines 2 – 3, and steps 4 and 5 can easily be done in $O(n)$ time. Step 6 can be done in $O(|E_2|)$ time, as described above (note that $p = O(|E_2|)$), and the loop in lines 7 – 8 takes $O(n)$ time. Hence the total algorithm takes $O(|E_2|)$ time. Notice that we can easily make the algorithm constructive: in line 8, return the graph H_i if $N(i) = n \Leftrightarrow 3$ and $B(i) = p \Leftrightarrow 1$.

Theorem 4.3.3. *There exists an $O(|E_2(S)|)$ time algorithm that solves the constructive version* 3-UISG *for sandwich blocks S.*

4.3.2 Three-Unit-Intervalizing Biconnected Colored Graphs

Let $G = (V,E)$ be a graph and c a three-coloring of G. The sandwich graph $S = (V,E_1,E_2)$, with $E_1 = E$ and $E_2 = \{\{v,w\} \mid c(v) \neq c(w)\}$ is called the sandwich graph associated with G and c. For three-unit-intervalizing biconnected colored graphs we can use the algorithm from the previous section, by first computing the sandwich graph associated with input graph G and coloring c. Unfortunately, the number of edges in E_2 may be $\Omega(n^2)$, while the graph G must have $O(n)$ edges.

In this section we show that it is possible to obtain a faster algorithm for 3-UICG that does not use the associated sandwich graph. Instead of using the notions of bandwidth, (proper) pathwidth and unit-intervalizations for the sandwich graph S associated with graph G and coloring c, we use these notions for the pair G,c itself.

Consider the case in which the input graph is a cycle.

Lemma 4.3.7. *Let C be a cycle, suppose $n = |V(C)| \geq 4$. Let $c : V \rightarrow \{1,2,3\}$ be a coloring of C and suppose C,c has bandwidth two. There are exactly two vertices v and w in C which*

have two neighbors of the same color. In each legal layout ℓ of C,c of bandwidth two in which there is a vertex u such that $\ell(u) = 1$, either $\ell(v) = 2$ and $\ell(w) = n \Leftrightarrow 1$, or vice versa. Furthermore, if n even, then v and w have distance $n/2$ and u and w have distance $n/2 \Leftrightarrow 1$, and if n is odd, then v and w have distance $(n \Leftrightarrow 3)/2$, and u and w have distance $(n \Leftrightarrow 1)/2$.

Proof. Let ℓ be a legal layout for C,c which has bandwidth two such that $\ell(u) = 1$. Number the vertices from $u_1, \ldots, u_n$, such that for each i, $\ell(u_i) = i$. For each i, $1 \leq i \leq n \Leftrightarrow 2$, $\{u_i, u_{i+2}\} \in E(C)$ (Lemma 4.3.2), so $c(u_i) \neq c(u_{i+2})$, and furthermore, $c(u_i) \neq c(u_{i+1})$ and $c(u_{i+1}) \neq c(u_{i+2})$. Hence vertices u_1, u_2 and u_3 have different colors, and for each i, $1 \leq i \leq n$, if $i \bmod 3 = 1$), then $c(u_i) = c(u_1)$, if $i \bmod 3 = 2$, then $c(u_i) = c(u_2)$, and if $i \bmod 3 = 0$, then $c(u_i) = c(u_3)$.

Vertices u_1 and u_n have two neighbors of different colors. The neighbors of u_2 are u_1 and u_4, and these have the same color. The neighbors of u_{n-1} are u_n and u_{n-3}, which also have the same color. For each i, $3 \leq i \leq n \Leftrightarrow 2$, the neighbors of u_i are u_{i-2} and u_{i+2}, which have different colors.

Consider the two paths from u_2 to u_{n-1} in C. If n is odd, then these are the paths $P_1 = (u_2, u_4, u_6, \ldots, u_{n-1})$ and $P_2 = (u_2, u_1, u_3, \ldots, u_{n-2}, u_n, u_{n-1})$. Path P_1 has length $(n \Leftrightarrow 3)/2$, and P_2 has length $(n+3)/2$. Hence the distance between v and w is $(n \Leftrightarrow 3)/2$, and the distance between $u = u_1$ and w is $(n \Leftrightarrow 1)/2$.

If n is even then the paths from u_2 to u_{n-1} are $P_1 = (u_2, u_1, u_3, \ldots, u_{n-1})$ and $P_2 = (u_1, u_3, \ldots, u_{n-2}, u_n, u_{n-1})$, and they both have length $n/2$. Hence the distance between v and w is $n/2$, and the distance between u_1 and w is $n/2 \Leftrightarrow 1$. □

If we have a cycle C of length three with a three-coloring c, then C is a three-unit-intervalization of C,c.

Let C be a cycle and c a three-coloring c of C, and suppose $n = |V(C)| \geq 4$. Furthermore, suppose that C,c has bandwidth two and let v be a vertex in C which has two neighbors u_1 and u_2 of the same color. There are two legal layouts of C with bandwidth two in which v is mapped to the value 2, namely the layout ℓ_1 with $\ell_1(u_1) = 1$ and the layout ℓ_2 with $\ell_2(u_2) = 1$. If n is even, then it is easy to see that both ℓ_1 and ℓ_2 are legal layouts of C,c. Furthermore, the interval completions of C for ℓ_1 and ℓ_2 are the only three-unit-intervalizations of C,c.

If n is odd, then only one of the layouts is legal: let d_i denote the distance between u_i and w. Then either $d_1 = (n \Leftrightarrow 1)/2$ and $d_2 = (n \Leftrightarrow 5)/2$, or vice versa. In the first case, ℓ_1 is the only legal layout of C,c with $\ell_1(v) = 2$, and in the latter case, ℓ_2 it the only legal layout of C,c with $\ell_2(v) = 2$. Furthermore, there is a unique three-unit-intervalization of C,c: in the first case, this is the interval completion of C for ℓ_1. In the second case, it is the interval completion of C for ℓ_2.

So to check if there is a three-unit-intervalization of a cycle C and a three-coloring c of C, we can simply check if the conditions given in Lemma 4.3.7 hold. This can be done in $O(n)$ time. Also, the preceding discussion shows that we can, in $O(n)$ time, construct a three-unit-intervalization of C,c if one exists.

Lemma 4.3.8. *There is an $O(n)$ time algorithm which solves the constructive version of 3-UICG if the input graph is a cycle.*

Suppose we have a biconnected graph G and a three-coloring c of G, and suppose G has bandwidth two. Let F be the set of edges in G which occur in two chordless cycles, and let C be the cycle obtained from G by removing all edges in F. Clearly, each three-unit-intervalization of G,c is a three-unit-intervalization of C,c. Suppose there is a three-unit-intervalization G' of C,c. Then G' is a three-unit-intervalization of G,c if and only if $F \subseteq E(G')$.

Given a biconnected graph G with a three-coloring c, we can now check whether there is a three-unit-intervalization of G and c as follows.

Algorithm 3-UICG_BG(G,c)
Input: Biconnected graph G, three-coloring c for G
Output: true if there is a 3-unit-intervalization of G,c, false otherwise
1. Check if G is a path of cycles. If so, find a cycle path (C,E) for G, and check if E contains each edge only once. If not, **return** false.
2. Let C be the cycle obtained from G by removing all edges in E. If $|V(C)| = 3$, **return** true.
3. Find the set L of all (at most two) legal layouts of bandwidth two of C,c. If $L = \phi$, **return** false.
4. For each $\ell \in L$, check whether ℓ is a legal layout of bandwidth two of G,c. If so, **return** true.
5. **return** false

By the preceding discussion, this algorithm returns true if and only if there is a three-unit-intervalization. Steps 1, 2 and 3 use $O(n)$ time, as is shown before. In step 4, a layout $\ell \in L$ is a legal layout of bandwidth two of G,c if for each $\{u,v\} \in F$, $|\ell(u) \Leftrightarrow \ell(v)| = 1$. This can be checked in $O(n)$ time, since $|F| = O(n)$. Hence 3-UICG_BG uses $O(n)$ time. The algorithm can be made constructive as follows: if in step 4, a legal layout ℓ of G,c is found, then the interval completion of ℓ is returned. By Theorem 4.3.1, this interval completion is a three-unit-intervalization of G,c. This shows the following result.

Theorem 4.3.4. *There exists an $O(n)$ time algorithm that solves the constructive version* 3-UICG *for biconnected graphs.*

Chapter 5

Reduction Algorithms

In this chapter we give an introduction to *reduction algorithms* for decision and optimization problems on graphs. A reduction algorithm is based on a finite set of *reduction rules* and a finite set of graphs. Each reduction rule describes a way to modify a graph locally. The idea of a reduction algorithm is to solve a decision problem by repeatedly applying reduction rules on the input graph until no more rule can be applied. If the resulting graph is in the finite set of graphs, then the algorithm returns true, otherwise it returns false.

The idea of reduction algorithms originates from Duffin's [1965] characterization of series-parallel graphs (Definition 2.3.3): a multigraph is series-parallel if and only if it can be reduced to a single edge by applying a sequence of *series* and *parallel* reductions. A series reduction is the replacement of a vertex of degree two and its incident edges by an edge between its two neighbors, and a parallel reduction is the removal of one of two or more edges between the same vertices, i.e. the removal of an edge which has parallel edges (see also Figure 2.11). Valdes, Tarjan, and Lawler [1982] showed how a reduction algorithm based on this set of reduction rules can be implemented in linear time, and hence series-parallel graphs can be recognized in linear time.

Inspired by the characterization of Duffin and the similarity between graphs of treewidth two and series-parallel graphs, Arnborg and Proskurowski [1986] generalized this idea for recognizing simple graphs of treewidth at most three: they gave a set of reduction rules which characterizes graphs of treewidth at most three. They also showed that these reduction rules can be used to recognize partial three-trees in $O(n^3)$ time. Matoušek and Thomas [1991] gave a slightly different set of reduction rules, and showed that with this new set it is possible to recognize simple graphs of treewidth at most three in linear time. Additionally, they showed how to construct a tree decomposition of minimum width in linear time if the input graph has treewidth at most three.

A much more general approach is taken by Arnborg, Courcelle, Proskurowski, and Seese [1993]. They gave a set of conditions that must hold for a set of reduction rules to ensure that the reduction algorithm works correctly. They have also shown that for a large class of decision problems on simple graphs of bounded treewidth, there is a set of reduction rules for which these conditions hold, and that the algorithm based on such a set of reduction rules takes $O(n)$ time (but more than linear space). The results of Arnborg et al. are stated in a general, algebraic setting.

Bodlaender [1994] extended the notion of reduction algorithms to optimization problems:

he introduced a new notion of reduction rules for optimization problems, called *reduction-counter rules*, and gave a set of conditions which are necessary for a set of reduction-counter rules in order to make a reduction algorithm work correctly. For simple graphs of bounded treewidth, this results again in efficient linear time algorithms.

Bodlaender and Hagerup [1995] have shown that the sequential reduction algorithms of Arnborg et al. [1993] and Bodlaender [1994] can efficiently be parallelized, if some additional conditions hold for the set of reduction rules. Their reduction algorithm uses $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM. A sequential version of this algorithm gives a reduction algorithm which uses $O(n)$ time and space.

In this chapter, we give an overview of the results of Arnborg et al. [1993], Bodlaender [1994] and Bodlaender and Hagerup [1995] that are mentioned above. We combine the results and present them in a uniform setting, in order to get a comprehensible overview. The chapter also acts as an introduction for Chapters 6 and 7: in Chapter 6, we show how the reduction algorithms presented in this chapter can be extended to constructive algorithms, i.e. algorithms which solve constructive decision or optimization problems. In Chapter 7, we apply these results to a number of optimization problems.

This chapter is organized as follows. In Section 5.1 we discuss reduction algorithms for decision problems as introduced by Arnborg et al. [1993]. We do this in a more direct way, without making use of the algebraic theory. We also give a reduction algorithm which uses linear time and space, based on the ideas of Arnborg et al. [1993] and of Bodlaender and Hagerup [1995]. In Section 5.2, we describe reduction algorithms for optimization problems, as introduced by Bodlaender [1994]. In Section 5.3, we discuss the parallel reduction algorithms of Bodlaender and Hagerup [1995], and in Section 5.4, we mention some additional results.

5.1 Reduction Algorithms for Decision Problems

In this section, we start with definitions of reduction rules and reduction systems (Section 5.1.1). Then we give an efficient reduction algorithm based on a special type of reduction system (Section 5.1.2). Finally, we show that this reduction algorithm can be used to solve a large class of decision problems on graphs of bounded treewidth (Section 5.1.3).

5.1.1 Reduction Systems

The graphs we consider are simple unless stated otherwise. Recall that a graph property is a function P which maps each graph to the value true or false. We say a property is *effectively decidable* if an algorithm is *known* that decides the property.

For the definitions of terminal graphs, and the operation $\oplus$, see Definitions 2.2.3 and 2.2.4. Two terminal graphs $(V_1, E_1, \langle x_1, \cdots, x_k \rangle)$ and $(V_2, E_2, \langle y_1, \cdots, y_l \rangle)$ are said to be *isomorphic* if $k = l$ and there is an isomorphism from (V_1, E_1) to (V_2, E_2) which maps x_i to y_i for each i, $1 \leq i \leq k$.

Definition 5.1.1 (Reduction Rule). A *reduction rule* r is an ordered pair (H_1, H_2), where H_1 and H_2 are l-terminal graphs for some $l \geq 0$.

A *match* to reduction rule $r = (H_1, H_2)$ in graph G is a terminal graph G_1 which is isomorphic to H_1, such that there is a terminal graph G_2 with $G = G_1 \oplus G_2$.

If G contains a match to r, then an *application* of r to G is an operation that replaces G by a graph G', such that there are terminal graphs G_1, G_2 and G_3, with G_1 isomorphic to H_1, G_2 isomorphic to H_2, and $G = G_1 \oplus G_3$, $G' = G_2 \oplus G_3$. We also say that, in G, G_1 is replaced by G_2. An application of a reduction rule is also called a *reduction*.

Figure 5.1 shows an example of a reduction rule r, and an application of r to a graph G. We usually depict a reduction rule (H_1, H_2) by the two graphs H_1 and H_2 with an arrow from H_1 to H_2. Given a reduction rule $r = (H_1, H_2)$, we call H_1 the left-hand side of r, and H_2 the right-hand side of r.

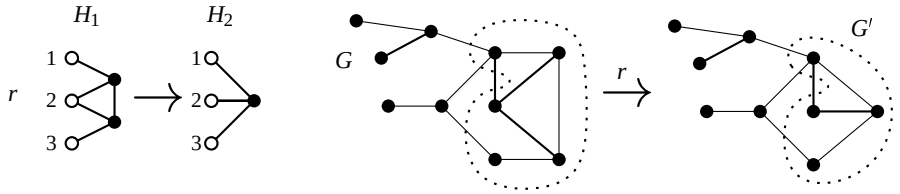


Figure 5.1. An example of a reduction rule $r = (H_1, H_2)$, and an application of r to a graph G, resulting in graph G'. The dotted lines in G and G' denote the parts of G and G' that are involved in the reduction.

Let G be a graph and $r = (H_1, H_2)$ a reduction rule. If G contains a match G_1 to r, then an application of r to G which replaces G_1 by a terminal graph isomorphic to H_2 is called a reduction corresponding to the match G_1.

Note that different applications of a reduction rule to a graph may result in different (i.e. non-isomorphic) graphs. If there is an application of rule r to graph G which results in a graph G', then we write $G \xrightarrow{r} G'$. Let R be a set of reduction rules. For two graphs G and G', we write $G \xrightarrow{R} G'$ if there exists an $r \in R$ with $G \xrightarrow{r} G'$. We say G contains a match G_1 if there is an $r \in R$ such that G_1 is a match to r in G. If G contains no match, then we say that G is *irreducible* (for R).

The following conditions are useful for a set of reduction rules in order to get a characterization of a graph property P.

Definition 5.1.2. Let P be a graph property and R a set of reduction rules.

- R is *safe* for P if, whenever $G \xrightarrow{R} G'$, then $P(G) \Leftrightarrow P(G')$.
- R is *complete* for P if the set I of irreducible graphs for which P holds is finite.
- R is *terminating* if there is no infinite sequence $G_1 \xrightarrow{R} G_2 \xrightarrow{R} G_3 \xrightarrow{R} \cdots$.

- R is *decreasing* if, whenever $G \xrightarrow{R} G'$, then G' contains fewer vertices than G.

Definition 5.1.3 (Reduction System). A *reduction system* for a graph property P is a pair (R, I), with R a finite set of reduction rules which is safe, complete and terminating for P, and I the set of irreducible graphs for which P holds.

A *decreasing reduction system* for P is a reduction system (R, I) for P in which R is decreasing.

A reduction system (R, I) for a property P gives a complete characterization of P: $P(G)$ holds for a graph G if and only if any sequence of reductions from R on G leads to a graph G' which belongs to I (i.e. is isomorphic to a graph in I).

As an example, consider the graph property of being a partial k-path for a fixed non-negative integer k. Arnborg and Proskurowski [1986] have given reduction systems $S_k = (R_k, I_k)$ for the properties that a graph is a partial k-path, for $0 \leq k \leq 3$ (see also Arnborg [1985]). For each k, I_k contains only the empty graph, denoted by G_{empty}. Furthermore, the set of rules R_k is a subset of the rules depicted in Figure 5.2: $R_0 = \{1\}$, $R_1 = \{1,2\}$, $R_2 = \{1,2,3\}$ and $R_3 = \{1,\ldots,6\}$.

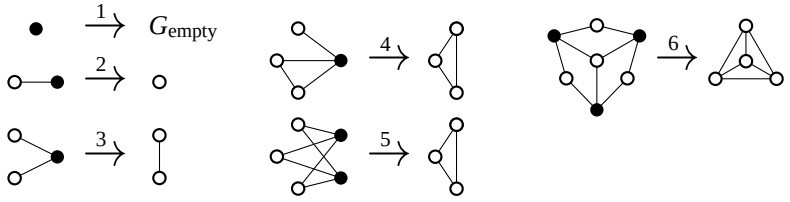


Figure 5.2. A set of reduction rules which is safe, complete and terminating for treewidth at most three.

5.1.2 An Efficient Reduction Algorithm

A decreasing reduction system (R, I) for a property P corresponds to a polynomial time algorithm that decides whether property P holds for a given graph G: repeat applying rules from R starting with the input graph, until no rule from R can be applied anymore. If the resulting graph belongs to the set I, then P holds for the input graph, otherwise, it does not. The number of reductions that has to be performed is at most n, since each reduction reduces the number of vertices by at least one. Furthermore, it takes $O(n^c)$ time to check whether G contains a match, where c is the maximum number of vertices in any left-hand side of a reduction rule.

In general, an algorithm as described above is not very efficient. However, there are several ways to make the algorithm more efficient. One way is to use a reduction system in which the reduction rules have a special structure, and to use this structure to efficiently determine whether a reduction rule can be applied. For example, Arnborg et al. [1993] define

such a special reduction system, and show that, with such a reduction system, a property can be decided in $O(n)$ time on an input graph with n vertices. However, their algorithm needs $\Theta(n^p)$ space, where p is the maximum number of terminals in any left-hand side of a reduction rule (although this result can be improved to $O(n^{1+\varepsilon})$ space for any $\varepsilon > 0$, with an increase in running time of only a constant factor [Hagerup, 1988]).

Another way to make a reduction algorithm more efficient is to design the reduction system (R, I) such that for any graph G for which the property holds, either G belongs to I, or G contains a match which can be found in an efficient way. As an example of this, we consider the method used by Bodlaender and Hagerup [1995], called the *bounded adjacency list search method*. (Bodlaender and Hagerup use their method to obtain an efficient parallel algorithm; we give an efficient sequential version of this parallel algorithm in this section.)

Definition 5.1.4. Let d be a positive integer. Let G be a graph given by some adjacency list representation and let G_1 be an l-terminal graph. We say G_1 is *d-discoverable* in G if

1. G_1 is open and connected, and the maximum degree of any vertex in G_1 is at most d,
2. there is an l-terminal graph G_2, such that $G = G_1 \oplus G_2$, and
3. G_1 contains an inner vertex v such that for all vertices $w \in V(G_1)$ there is a walk W in G_1 with $W = (u_1, u_2, \ldots, u_s)$, such that $v = u_1$, $w = u_s$, and for each i, $2 \leq i \leq s \Leftrightarrow 1$, in the adjacency list of u_i in G, the edges $\{u_{i-1}, u_i\}$ and $\{u_i, u_{i+1}\}$ have distance at most d.

Let G be a graph, d a positive integer, and let G_1 be a d-discoverable terminal graph in G. Let v be an inner vertex of G_1 satisfying condition 3 above, and let $W = (u_1, u_2, \ldots, u_s)$ be a walk in G_1 with $v = u_1$, such that for each i, $2 \leq i \leq s \Leftrightarrow 1$, the edges $\{u_{i-1}, u_i\}$ and $\{u_i, u_{i+1}\}$ have distance at most d in the adjacency list of u_{i+1} in G. We show that there is a walk W' from u_1 to u_s in G_1 satisfying the same condition in which each edge $e = \{x, y\}$ occurs at most twice in W'. If an edge $e = \{x, y\}$ occurs three times in W', then it is passed in the same direction at least twice, i.e. W contains either a subsequence of the form $S = x, y, \ldots, x, y$ or a subsequence of the form $S = y, x, \ldots, y, x$. In the first case, we remove the subsequence $y, \ldots, x$ of S, and in the latter case, we remove the subsequence $x, \ldots, y$ of S. The result is still a walk in G_1 satisfying the stated condition. This transformation can be repeated until W does not contain any edge more than twice.

In the same way we can show that if condition 3 holds, then there is a walk from any inner vertex w to any other inner vertex w' in G_1 in which two subsequent edges have distance at most d in the adjacency list of their common vertex, and each edge occurs at most twice. This, and the fact that each edge in an open terminal graph G_1 is incident with an inner vertex (which has degree at most d) implies the following result.

Lemma 5.1.1. *If a terminal graph G_1 is d-discoverable in a graph G for some $d \geq 1$, then for any inner vertex v of G_1, all vertices and edges of G_1 can be found from v in an amount of time that only depends on the integer d and the size of G_1, but not on the size of the graph G.*

Definition 5.1.5 (Special Reduction System). Let P be a graph property, and (R, I) a decreasing reduction system for P. Let n_{max} be the maximum number of vertices in any left-hand

side of a rule $r \in R$. (R, I) is a *special reduction system* for P if we know positive integers n_{min} and d, $n_{min} \leq n_{max} \leq d$, such that the following conditions hold.

1. For each reduction rule $(H_1, H_2) \in R$,
   (a) if H_1 has at least one terminal, then H_1 is connected and H_1 and H_2 are open, and
   (b) if H_1 is a zero-terminal graph, then $|V(H_2)| < n_{min}$.
2. For each graph G and each adjacency list representation of G, if $P(G)$ holds, then
   (a) each component of G with at least n_{min} vertices has a d-discoverable match, and
   (b) if all components of G have less than n_{min} vertices, then either $G \in I$ or G contains a match which is a zero-terminal graph.

Conditions 1a and 2a assure that each component H, with $|V(H)| \geq n_{min}$, of a graph G for which $P(G)$ holds, contains at least one d-discoverable match to a reduction rule, which can be applied without having to remove multiple edges. Conditions 1b and 2b are only needed for graph properties which do not imply that the input graph is connected; they assure that, if no other reduction rules are applicable, then matches to reduction rules with zero terminals of which the left-hand side is not connected, can be found efficiently.

As a simple example of a special reduction system, consider the graph property P, where $P(G)$ holds if and only if each component of G is a two-colorable cycle, and the number of components of G is odd. Let R be the set of reduction rules depicted in Figure 5.3 (G_{empty} denotes the empty graph), and let I be the set containing just the cycle on four vertices (see also Figure 5.3). It can easily be seen that (R, I) is a special reduction system for P ($d = n_{max} = 8$ and $n_{min} = 5$).

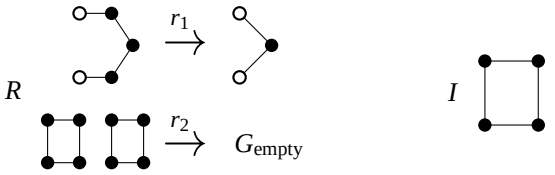


Figure 5.3. A reduction system for property P, which is the property that a graph is two-colorable, has an odd number of components, and each of its components is a cycle.

Theorem 5.1.1. *Let P be a graph property. If we have a special reduction system for P, then we have an algorithm which decides P in $O(n)$ time and $O(n)$ space.*

We prove Theorem 5.1.1 by giving the algorithm. The algorithm consists of two phases. In the first phase, the algorithm finds d-discoverable matches and executes the corresponding reductions, until there are no more d-discoverable matches. If the resulting graph is in I, then

P holds for the input graph, and true is returned. Otherwise, the algorithm proceeds with the second phase.

In the second phase, the algorithm checks whether G contains a component with at least n_{min} vertices. If so, false is returned, since P does not hold (by condition 2a of Definition 5.1.5). Otherwise, the algorithm repeats applying reduction rules of which the left-hand side has no terminals, until this is no longer possible. If the resulting graph G is in I, then P holds and true is returned. Otherwise, there is no further applicable rule: each component of G has less than n_{min} vertices, since it either was already a component of the graph after phase 1, or it is the result of a reduction in phase 2, which means that it is a component of a right-hand side of a reduction rule. Since the set of reduction rules is complete for P, this means that $P(G)$ does not hold, and hence false is returned.

We now give the complete algorithm, given the special reduction system (R, I) and the integers n_{min} and d. The algorithm is a simplified sequential simulation of the parallel algorithm given by Bodlaender and Hagerup [1995]. It resembles the algorithm of Arnborg et al. [1993], but uses $O(n)$ space, whereas the algorithm of Arnborg et al. uses $\Omega(n^p)$ space, where p equals the maximum number of terminal vertices in any reduction rule.

Algorithm Reduce(G)
Input: Graph G
Output: $P(G)$

```
1.  n_max ← max{|V(H)| | H is left-hand side of some r ∈ R }
2.  (* Phase 1 *)
3.  S ← {v ∈ V(G) | deg(v) ≤ d}
4.  while S ≠ φ
5.     do take v ∈ S
6.        if v is inner vertex of a d-discoverable match G_1 to a rule r ∈ R
7.          then apply r to G:
8.                  let G_2 be a new terminal graph isomorphic to H_2, such that G_1 and G_2
                    have the same set of terminals
9.                  (* Remove inner vertices and edges of G_1 *)
10.                 V(G) ← V(G) ⇔{v ∈ V(G_1) | v is inner vertex of G_1}
11.                 E(G) ← E(G) ⇔E(G_1)
12.                 S ← S ⇔{v ∈ V(G_1) | v is inner vertex of G_1}
13.                 (* Add inner vertices and edges of G_2 *)
14.                 V(G) ← V(G) + {v ∈ V(G_2) | v is inner vertex of G_1}
15.                 E(G) ← E(G) + E(G_2)
16.                 S ← S ∪ {v ∈ V(G_2) | deg(v) ≤ d}
17.                 for all terminals x of G_2
18.                    do let L denote adjacency list of x
19.                       for all {x,w} ∈ L for which L changed within distance d
20.                          do if deg(w) ≤ d then S ← S ∪ {w}
21.       S ← S ⇔{v}
22. if G ∈ I then return true
```

23. ($*$ Phase 2 $*$)
24. **if** G has a component with $\geq n_{min}$ vertices **then return** false
25. $R^0 \leftarrow \{r \in R \mid$ left-hand side of r is zero-terminal graph$\}$
26. **for all** $r = (H_1, H_2) \in R^0$
27. **do** make list L_r containing for each component C of H_1 a graph H isomorphic to C, an integer $i(H)$ denoting the number of components isomorphic to H in H_1, and an initially empty list $L(H)$ containing components of G which are isomorphic to H (L_r contains only one entry for isomorphic components of H_1)
28. $S' \leftarrow \{C \mid C$ is component of $G\}$
29. **while** $S' \neq \phi$
30. **do** take $C \in S'$
31. **for all** $r \in R^0$
32. **do if** L_r contains component H isomorphic to C
33. **then** add C to $L(H)$
34. **if** $\exists r \in R^0 : \forall H' \in L_r : L(H')$ contains $i(H')$ components
35. **then** apply r:
36. for each $H' \in L_r$, take $i(H')$ components of the list $L(H')$ and remove them from $L(H')$, from G, from S' and from all other lists of R^0
37. add graph G_1 isomorphic to right-hand side of r to G
38. add all components of G_1 to S'
39. $S' \leftarrow S' \Leftrightarrow \{C\}$
40. **if** $G \in I$ **then return** true **else return** false

We first show that phase 1 and phase 2 of the algorithm are correct. Let G be the graph that results after the main loop of phase 1 (lines 4 – 21) is finished. Phase 1 is correct if

- G does not contain any d-discoverable matches,
- $P(G)$ holds if and only if P holds for the input graph, and
- true is returned if and only if $G \in I$.

Phase 2 is correct if true is returned if P holds for the input graph, and false is returned otherwise.

Lemma 5.1.2. *Phase 1 of algorithm Reduce is correct.*

Proof. As the applied reduction rules are safe for P, it must be the case that P holds for the input graph if and only if $P(G)$ holds for the graph G resulting from the main loop. We prove that the main loop has the following invariant: for each d-discoverable match G_1 in G, there is a vertex $w \in S$ which is an inner vertex of G_1. Clearly, if this invariant holds when $S = \phi$, then G contains no d-discoverable matches. This implies that phase 1 is correct. We prove the invariant by induction on the number of iterations.

Initially (after the 0th iteration), the invariant holds. Now suppose it holds after the ith iteration ($i \geq 0$), and consider the $(i+1)$st iteration. If no reduction is applied in this iteration,

then no match is d-discoverable from v and the graph G does not change. Hence the invariant still holds after this iteration.

Suppose a reduction is applied in the $(i+1)$st iteration. Let G^i and G^{i+1} denote the graph G after the ith and the $(i+1)$st iteration, respectively. Let G_1, G_2 and G_3 be terminal graphs such that $G^i = G_1 \oplus G_3$, $G^{i+1} = G_2 \oplus G_3$, and G_1 is the match in G^i that corresponds to the applied reduction rule. Suppose that, after the $(i+1)$st iteration, there is a d-discoverable match H_1 in G^{i+1}. We show that there is an inner vertex of H_1 in S. If H_1 contains no vertices of G_1, then H_1 was already d-discoverable after the ith iteration, and hence S contains an inner vertex of H_1. Suppose $V(H_1) \cap V(G_2) \neq \phi$. If one of the inner vertices u of H_1 is a vertex of G_2, then u has degree at most d and hence u is added to S in line 16 in the $(i+1)$st iteration. If one of the inner vertices of G_2 is a terminal vertex x of H_1, then all neighbors of x are in G_2, and since x has a neighbor u which is an inner vertex of H_1, this means that $u \in S$.

The only remaining case is the case that the common vertices of G_2 and H_1 are terminal vertices of both G_2 and H_1. Then H_1 is also a match in the graph G^i. If H_1 was already a d-discoverable match in G^i, then S contains an inner vertex of H_1. Suppose H_1 was not a d-discoverable match in G^i. Hence there is an inner vertex v of H_1, and a vertex $w \in V(H_1)$ and a walk $(u_1, u_2, \ldots, u_s)$ in H_1 with $u_1 = v$ and $u_s = w$, in which there is a j, $2 \leq j \leq s \Leftrightarrow 1$, such that in the adjacency list of u_j in G^i, the edges $\{u_{j-1}, u_j\}$ and $\{u_j, u_{j+1}\}$ have distance more than d, but in the adjacency list of u_j in graph G^{i+1}, edges $\{u_{j-1}, u_j\}$ and $\{u_j, u_{j+1}\}$ have distance at most d. That implies that the adjacency list of u_j has changed during the reduction of G^i to G^{i+1}, and hence u_j is a terminal vertex of G_2, and also of H_1. Moreover, the change of the adjacency list of u_j was within distance d from both $\{u_{j-1}, u_j\}$ and $\{u_j, u_{j+1}\}$. Since H_1 is an open terminal graph, both u_{j-1} and u_{j+1} are inner vertices of H_1, and hence have degree at most d. This means that u_{j-1} and u_{j+1} have been added to S in line 20 of iteration $i+1$. □

Lemma 5.1.3. *Phase 2 of algorithm Reduce is correct.*

Proof. The main loop of phase 2 (lines 29 – 39) can be proved to have the following invariant: for each match G_1 in G to some rule $r = (H_1, H_2)$ in R^0, there is a component C of G_1 which is in S' and furthermore, for each component of C of G_1, either $C \in S'$, or C is in list $L(H)$ of L_r, where H is isomorphic to C and to a component of H_1. Clearly, if this invariant holds when $S' = \phi$, then this means that phase 2 is correct. The proof of this invariant is similar to the one for phase 1, and thus omitted. □

This proves the following result.

Lemma 5.1.4. *Algorithm Reduce correctly recognizes simple graphs for which a property P holds, given a special reduction system (R, I) for P.*

Consider the time complexity of the algorithm.

Lemma 5.1.5. *Algorithm Reduce uses $O(n)$ time and space.*

Proof. Consider phase 1 first. We first show that the main loop of this phase is iterated $O(n)$ times. We do this by showing that the number of times a vertex is added to S is $O(n)$.

Initially, in line 3, S contains $O(n)$ vertices. In the main loop, there are only vertices added to S if a reduction takes place. Since at most n reductions take place, and after each reduction, at most a constant number of vertices is added to S, this means that the total number of vertices added to S during the main loop is also $O(n)$. Since in each iteration of the main loop, at least one vertex is removed from S, this means that the main loop is executed $O(n)$ times.

Consider one iteration of the main loop. In line 6, a d-discoverable match in G that contains v as an inner vertex can be found in constant time, as we described. Furthermore, each reduction can be done in constant time. The loop in lines 17 – 20 can also be done in constant time: during the reduction, it is possible to store the places in the adjacency lists of the terminals where something changes, so that they can be easily found. Hence each iteration of the main loop takes $O(1)$ time. Hence phase 1 algorithm can be done in $O(n)$ time.

Consider phase 2 of the algorithm. Lines 25 – 27 can be done in constant time, and lines 24 and 28 in $O(n)$ time. We can show in a similar way as for phase 1 that the loop in lines 29 – 39 is iterated $O(n)$ times and that each iteration takes $O(1)$ time. Line 40 can be done in $O(1)$ time, hence phase 2 takes $O(n)$ time. It is easy to see that algorithm Reduce uses $O(n)$ space. □

This completes the proof of Theorem 5.1.1.

5.1.3 Decision Problems for Graphs of Bounded Treewidth

In this section, we show that algorithm Reduce can be used for a large class of graph properties on graphs of bounded treewidth.

Let P be a graph property and l a non-negative integer. Recall that for every two l-terminal graphs G_1 and G_2, the equivalence relation $\sim_{P,l}$ is defined as follows: $G_1 \sim_{P,l} G_2$ if and only if for all l-terminal graphs H, $P(G_1 \oplus H)$ holds if and only if $P(G_2 \oplus H)$ holds (Definition 2.2.5). We say graph property P is of finite index if for each non-negative integer l, $\sim_{P,l}$ has finitely many equivalence classes. As mentioned in Section 2.2.4, many important graph properties are of finite index. For instance, all MS-definable graph properties are of finite index [Courcelle, 1990].

Note that a set R of reduction rules for a property P is safe if and only if for each reduction rule $(H_1, H_2) \in R$, $H_1 \sim_{P,l} H_2$, for l the number of terminals of H_1 and H_2.

An equivalence relation $\sim'$ is a *refinement* of an equivalence relation $\sim$ if each equivalence class of $\sim'$ is a subset of an equivalence class of $\sim$. Clearly, if $\sim'$ is finite, then so is $\sim$.

Lemma 5.1.6. *Let P_1 and P_2 be graph properties of finite index. Let Q_1 and Q_2 be graph properties defined as follows. For each graph G, $Q_1(G) = P_1(G) \wedge P_2(G)$, and $Q_2(G) = P_1(G) \vee P_2(G)$. Then Q_1 and Q_2 are also of finite index.*

Proof. (Cf. Borie, Parker, and Tovey [1992], Fellows and Langston [1989].) For each $l \geq 0$ and every two l-terminal graphs G_1 and G_2, let $G_1 \sim_l G_2$ if and only if $G_1 \sim_{P_1,l} G_2$ and $G_1 \sim_{P_2,l} G_2$. Then $\sim_l$ is a refinement of both $\sim_{Q_1,l}$ and $\sim_{Q_2,l}$. Furthermore, $\sim_l$ is finite, since

each equivalence class of $\sim_l$ is the intersection of two equivalence classes of $\sim_{P_1,l}$ and $\sim_{P_2,l}$, and there are at most finitely many of these intersections. □

For each integer $k \geq 1$, let TW_k be the graph property defined as follows: for each graph G, $TW_k(G)$ holds if and only if $\mathrm{tw}(G) \leq k$. For each $k \geq 1$, TW_k is MS-definable (see e.g. Arnborg et al. [1991]). This immediately implies that TW_k is of finite index, for each $k \geq 1$. Unfortunately, for $k \geq 4$, no formulation of TW_k is known. However, Lagergren and Arnborg [1991] have given an effectively decidable equivalence relation $\sim_{k,l}$, which for each k and l, is a refinement of $\sim_{TW_k,l}$. Thus we have the following result.

Lemma 5.1.7 [Lagergren and Arnborg, 1991]. *For each fixed $k \geq 1$, TW_k is of finite index, and for each $l \geq 0$, there is a finite, effectively decidable refinement of $\sim_{TW_k,l}$.*

For a property P and an integer k, we define the property P_k as $P_k(G) = P(G) \wedge TW_k(G)$. It follows from Lemmas 5.1.6 and 5.1.7 that for each fixed $k \geq 1$, if P is of finite index, then so is P_k, and furthermore, if we have a refinement $\sim_l$ of $\sim_{P,l}$ which is effectively decidable, then we have a refinement $\sim'_l$ of $\sim_{P_k,l}$ which is effectively decidable.

As we have mentioned in Section 2.2.4, finite index corresponds to 'finite state': there exists a linear time algorithm that decides finite index properties on graphs, given their tree decomposition of bounded treewidth. Moreover, this algorithm is of a special, well-described structure [Courcelle, 1990; Borie et al., 1992; Abrahamson and Fellows, 1993]. The disadvantage of this algorithm is that a tree decomposition of the input graph is needed. Fortunately, for each fixed k, there is a linear time sequential algorithm which, given a graph G, checks if $\mathrm{tw}(G) \leq k$, and if so, computes a minimum width tree decomposition of G [Bodlaender, 1996a]. However, this algorithm is not very practical, due to the large constant factors involved. With reduction algorithms, this disadvantage can be overcome, as we will show in the remainder of this section.

Bodlaender and Hagerup [1995] have proved the following lemma. The proof is technical, so we do not include it here.

Lemma 5.1.8 [Bodlaender and Hagerup, 1995]. *Let k and n_{min} be positive integers. There are integers d and n_{max}, $2(n_{min} \Leftrightarrow 1) \leq n_{max} \leq d$, and a constant $c > 0$, such that in each connected graph G of treewidth at most k, if $n \geq n_{min}$, then G contains at least $\lceil cn \rceil$ d-discoverable open and connected l-terminal graphs H with $l \leq 2(k+1)$ and $n_{min} \leq |V(H)| \leq n_{max}$.*

The following theorem has originally been proved by Arnborg et al. [1993] for a slightly different kind of special reduction system. Bodlaender and Hagerup [1995] have adapted the proof for the special reduction system as defined here.

Theorem 5.1.2. *Let P a graph property, and suppose P is of finite index. For each integer $k \geq 1$, there exists a special reduction system (R, I) for P_k.*

If P is also effectively decidable, and there is an equivalence relation $\sim_l$ for each $l \geq 0$ which is a finite refinement of $\sim_{P,l}$ and is effectively decidable, then such a system (R, I) can effectively be constructed.

Proof. Let $k \geq 1$. We first construct all right-hand sides of reduction rules. For every $l \leq 2(k+1)$ and every equivalence class C of $\sim_{P_k,l}$, do the following. If C contains open l-terminal graphs with treewidth at most k, then choose a representing open l-terminal graph $H_C \in C$ with treewidth at most k. Let n_{min} be one more than the maximum number of vertices of all chosen graphs H_C. Let d, n_{max} and c be as in Lemma 5.1.8.

Let R denote the set of reduction rules to be built. First, for all zero-terminal graphs H with at least n_{min} and at most n_{max} vertices, if we have a representative for the class C which contains H, then add reduction rule (H, H_C) to R. Next, for all l with $1 \leq l \leq 2(k+1)$ and for all open connected l-terminal graphs H with at least n_{min} and at most n_{max} vertices, if we have a representative for the equivalence class C in which H is contained, then add the reduction rule (H, H_C) to R. Note that if we do not have such a representative, then H must have treewidth at least $k+1$, and hence there is no terminal graph G for which $P_k(H \oplus G)$ holds.

Let $I = \{G \mid G \text{ is irreducible } \wedge P_k(G)\}$.

It is easy to see that R is finite: there are finitely many l-terminal graphs with at most n_{max} vertices. Safeness of the resulting set R follows directly from the fact that each left- and right-hand side of a rule in R belong to the same equivalence class of the relation $\sim_{P_k,l}$.

Conditions 1a and 1b of a special reduction system (Definition 5.1.5) clearly hold. This also shows that R is decreasing.

We now show that R is complete, i.e. that $|I|$ is finite and that conditions 2a and 2b of Definition 5.1.5 hold. Let G be a graph for which $P_k(G)$ holds. Let C be a component of G. Note that $\text{tw}(C) \leq k$. If C has at least n_{min} vertices, then, by Lemma 5.1.8, C contains at least $\lceil c|V(C)| \rceil \geq 1$ d-discoverable open l-terminal graphs H with $l \leq 2(k+1)$ and $n_{min} \leq |V(H)| \leq n_{max}$. Hence, by construction of the reduction system, C contains a d-discoverable match, so condition 2a holds. If all components of G have less than n_{min} vertices, then by definition of R, G contains a match, and hence condition 2b holds.

We now show that $|I|$ is finite. Clearly, all connected graphs in I have less than n_{min} vertices. If we have a disconnected graph G which is irreducible and for which $P_k(G)$ holds, then all components of G have less than n_{min} vertices. We show that G has less than n_{min} vertices. Suppose not. Since $n_{max} \geq 2(n_{min} \Leftrightarrow 1)$, there is a subset S of the set of components of G which has at least n_{min} and at most n_{max} vertices. The graph induced by these components has treewidth at most k, and hence by the construction of the reduction rules with zero terminals in R, G contains a match. This completes the proof that R is complete, and hence that (R, I) is a special reduction system.

We now show how we can effectively construct such a reduction system. Note that the non-constructive parts in the proof until now are: computing I, finding a representative for each equivalence class which contains open terminal graphs with treewidth at most k, and testing in which equivalence class a graph is contained. For each l, let $\sim_l$ be an effectively decidable equivalence relation on l-terminal graphs that is a refinement of $\sim_{P,l}$ and has a finite number of equivalence classes.

Arnborg et al. [1993] gave a way to construct, for any given integer m, a representative of each equivalence class of $\sim_l$ $(0 \leq l \leq m+1)$ which contains a graph for which there exists a

tree decomposition of width m with all terminals in the same node.

Furthermore, Lagergren and Arnborg [1991] gave an effectively decidable equivalence relation $\sim'_{TW_k,l}$, which is finite for each k and l, and is a refinement of $\sim_{TW_k,l}$. This gives us enough ingredients to construct a special reduction system. First consider the construction of representatives.

For each l and k, let $\sim_{k,l}$ and $\sim'_{k,l}$ be equivalence relations on l-terminal graphs which are defined as follows.

$$\begin{aligned} G_1 \sim_{k,l} G_2 \quad &\Leftrightarrow \quad G_1 \sim_l G_2 \wedge G_1 \sim'_{TW_k,l} G_2 \\ G_1 \sim'_{k,l} G_2 \quad &\Leftrightarrow \quad G_1 \sim_{k,l} G_2 \wedge (G_1 \text{ is open } \Leftrightarrow G_2 \text{ is open}) \end{aligned}$$

By Lemma 5.1.6 and 5.1.7 it follows that both $\sim_{k,l}$ and $\sim'_{k,l}$ are effectively decidable, have a finite number of equivalence classes, and are a refinement of $\sim_{P_k,l}$. Furthermore, $\sim'_{k,l}$ is a refinement of $\sim_{k,l}$.

Let $G = (V, E, X)$ be an l-terminal graph with $l \leq 2(k+1)$, and suppose G has treewidth at most k. There is a tree decomposition of width at most $3k+2$ of G in which all terminals are in one node: take an arbitrary tree decomposition of width k of G, let $x \in X$. Add all vertices in $X \Leftrightarrow \{x\}$ to all nodes in T. Clearly, the new tree decomposition has a node containing all terminals, and has width at most $k + 2(k+1) = 3k+2$.

Use the result from Arnborg et al. [1993] to generate a representative for each equivalence class of $\sim'_{k,l}$ (for each $l \leq 3k+2$) which contains a graph for which there is a tree decomposition of width $3k+2$ with all terminals in one node. After the generation, throw away all representatives with more than $2(k+1)$ terminals or with treewidth $k+1$ or more. The resulting set contains a representative for each equivalence class of $\sim_{k,l}$, $0 \leq l \leq 2(k+1)$, which contains a graph of treewidth at most k. Let R denote this set.

Now delete all graphs from R which are not open. The resulting set contains a representative for each equivalence class of $\sim_{k,l}$ which contains open l-terminal graphs of treewidth at most k, and hence this is the set we need.

Now it is easy to construct a special reduction system. Let n_{min} be one more than the maximum number of vertices of any graph in R. Let d and n_{max} be as found in Lemma 5.1.8. For all $l \leq 2(k+1)$, for all open and, if $l \geq 1$, connected l-terminal graphs H with at least n_{min} and at most n_{max} vertices, find an $H' \in R$ for which $H \sim_{k,l} H'$ (using the algorithm for deciding $\sim_{k,l}$). If an H' is found, then add the reduction rule (H, H') to an initially empty set of reduction rules $\mathcal{R}$.

The computation of $\mathcal{I}$ can be done as follows. As we have showed, all graphs in $\mathcal{I}$ have less than n_{min} vertices. Since P is effectively decidable, so is P_k, and hence we can compute $\mathcal{I}$ by computing $P_k(G)$ for all graphs G with less than n_{min} vertices, and putting the graphs for which $P_k(G)$ holds in $\mathcal{I}$. □

From the proof of Theorem 5.1.2, we can also conclude the following.

Corollary 5.1.1. *Let P be a graph property, and for each $l \geq 0$, let $\sim_l$ be a refinement of $\sim_{P,l}$. Let $k \geq 1$. If $\sim_l$ is finite for each $l \geq 0$, then there is a special reduction system $(\mathcal{R}, \mathcal{I})$ for*

P_k, such that for each $(H,H') \in R$, $H \sim_l H'$. Moreover, if $\sim_l$ and P are effectively decidable, then such a system can effectively be constructed.

As each MS-definable graph property is of finite index (Section 2.2.4), Theorem 5.1.2 immediately implies the following result.

Corollary 5.1.2. *Let P be a graph property which is MS-definable. For each integer $k \geq 1$, there is a linear time algorithm which decides P_k without using a tree decomposition of the input graph. Moreover, such an algorithm can be automatically constructed from an MSOL predicate for P.*

5.2 Reduction Algorithms for Optimization Problems

In this section we show how the idea of reduction algorithms can be extended to optimization problems. This idea was introduced by Bodlaender [1994].

5.2.1 Reduction-Counter Systems and Algorithms

Let Φ be a function which maps each graph to a value in $\mathbb{Z} \cup \{\mathsf{false}\}$ (we assume that isomorphic graphs are mapped to the same value). Typically, Φ will be an optimization problem like MAX INDEPENDENT SET. We will call Φ a *graph optimization problem*. The value false is used to denote that a certain condition does not hold, i.e. that there is no optimum for a graph. Let Z denote the set $\mathbb{Z} \cup \{\mathsf{false}\}$. Define addition on Z as follows: if $i, j \in \mathbb{Z}$, then we take for $i+j$ the usual sum, and for all $i \in Z$, $i + \mathsf{false} = \mathsf{false} + i = \mathsf{false}$.

Instead of reduction rules, we use *reduction-counter* rules for graph optimization problems.

Definition 5.2.1 (Reduction-Counter Rule). A *reduction-counter rule* is a pair (r,i), where r is a reduction rule, and i an integer.

A match to a reduction-counter rule (r,i) in a graph G is a match to r in G.

If G contains a match to a reduction-counter rule $r' = (r,i)$, then an application of r' to a graph G and an integer counter *cnt* is an operation which applies r to G and replaces *cnt* by $cnt + i$. An application of a reduction-counter rule is also called a reduction.

Let G and G' be two graphs. If there is a reduction-counter rule r such that applying r to G and some counter *cnt* can result in G', then we write $G \xrightarrow{r'} G'$. If we have a set R of reduction-counter rules, we write $G \xrightarrow{R} G'$ if there exists an $r \in R$ with $G \xrightarrow{r} G'$. If a graph G has no match in R, then we say that G is irreducible (w.r.t. R).

We extend the notions of safeness, completeness, termination and decrease to reduction-counter rules.

Definition 5.2.2. Let Φ be a graph optimization problem and let R be a set of reduction-counter rules.

- R is *safe* for Φ if, whenever $G \xrightarrow{r} G'$ for some $r = (r',i) \in R$, then $\Phi(G) = \Phi(G') + i$.

- R is *complete* for Φ if the set I of irreducible graphs G for which $\Phi(G) \neq \mathsf{false}$ is finite.
- R is *terminating* if there is no infinite sequence $G_1 \xrightarrow{R} G_2 \xrightarrow{R} G_3 \xrightarrow{R} \cdots$.
- R is *decreasing* if whenever $G \xrightarrow{R} G'$, then G' contains fewer vertices than G.

Definition 5.2.3 (Reduction-Counter System). A *reduction-counter system* for a graph optimization problem Φ is a triple (R, I, ϕ), where R is a finite set of reduction-counter rules which is safe, complete and terminating for Φ, I is the set of graphs G which are irreducible and for which $\Phi(G) \neq \mathsf{false}$, and ϕ is a function mapping each graph $G \in I$ to the value $\Phi(G)$.

A decreasing reduction-counter system for Φ is a reduction-counter system which is decreasing.

As a simple example we give a reduction-counter system is the optimization problem MAX INDEPENDENT SET on cycles: for each graph G, if G is a cycle then $\Phi(G)$ is the size of a maximum independent set in G, otherwise $\Phi(G) = \mathsf{false}$. Let $R = \{(r,1)\}$, where r is the reduction rule depicted in Figure 5.4, let $I = \{C_3, C_4\}$, where C_3 and C_4 are the cycles on three and four vertices (see Figure 5.4), and let $\phi(C_3) = 1$, $\phi(C_4) = 2$. It can easily be seen that (R, I, ϕ) is a reduction-counter system for Φ.

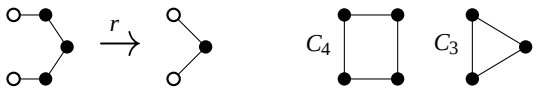


Figure 5.4. A reduction rule and a set of irreducible graphs for MAX INDEPENDENT SET on cycles.

Let Φ be a graph optimization problem. Let P be the graph property with for each graph G, $P(G) = \mathsf{true}$ if $\Phi(G) \in \mathbb{Z}$, and $P(G) = \mathsf{false}$ if $\Phi(G) = \mathsf{false}$. We call P the *derived* graph property (of Φ). From a reduction-counter system (R, I, ϕ) for Φ, we can derive a reduction system for P: let $R' = \{r \mid (r,i) \in R \text{ for some } i \in \mathbb{Z}\}$. Then (R', I) is a reduction system for P. We call this system the *derived reduction system* (from (R, I, ϕ)).

If we are given a decreasing reduction-counter system $S = (R, I, \phi)$ for a graph optimization problem Φ, we can again use a reduction algorithm to solve Φ in polynomial time. Let S' denote the derived reduction system (which is also decreasing). A reduction algorithm based on S is a modification of a reduction algorithm for the derived graph property based on S': instead of repeatedly applying reduction rules from S' on the input graph G, repeatedly apply reduction-counter rules from S on the graph G and a counter *cnt*. Initially, *cnt* is set to zero.

Let G_j denote the graph after the jth reduction is done, and let cnt_j denote the value of the counter at this moment (hence G_0 denotes the input graph, and $cnt_0 = 0$). It is important to note that the sum $\Phi(G_j) + cnt_j$ is invariant during the reduction process, because of the safeness property. Thus, at each moment in the reduction algorithm, $\Phi(G_0) = \Phi(G_j) + cnt_j$. Hence, when the reduction process stops after t iterations, because G_t is irreducible, then

$\Phi(G_0) \in \mathbb{Z}$ if and only if $G_t \in I$ (or, more precisely, G is isomorphic to a graph $H \in I$). Hence if $G_t \in I$, then $\Phi(G_0) = \phi(G_t) + cnt_t$, otherwise, $\Phi(G_0) = \mathsf{false}$.

Definition 5.2.4 (Special Reduction-Counter System). A special reduction-counter system special is a reduction-counter system of which the derived reduction system is special (Definition 5.1.5).

Note that the reduction-counter system for MAX INDEPENDENT SET on cycles that we have given above is also a special reduction-counter system for this problem.

Clearly, if we have a special reduction-counter system for a graph optimization problem Φ, then we can apply the modifications described above to algorithm Reduce in order to get a linear time algorithm for solving Φ.

Theorem 5.2.1. *Let Φ be a graph optimization problem. If we have a special reduction-counter system for Φ, then we have an algorithm which, for each graph G, computes $\Phi(G)$ in $O(n)$ time with $O(n)$ space.*

5.2.2 Optimization Problems for Graphs of Bounded Treewidth

In this section, we derive a similar result as Theorem 5.1.2 for reduction-counter systems.

In analogy to $\sim_{P,l}$ for graph properties P, we define an equivalence relation $\sim_{\Phi,l}$ for graph optimization problems Φ.

Definition 5.2.5. For a graph optimization problem Φ the equivalence relation $\sim_{\Phi,l}$ on l-terminal graphs is defined as follows. Let G_1 and G_2 be two l-terminal graphs.

$$G_1 \sim_{\Phi,l} G_2 \quad \Leftrightarrow \quad \text{there is an } i \in \mathbb{Z} \text{ such that for all } l\text{-terminals graphs } H\text{: } \Phi(G_1 \oplus H) = \Phi(G_2 \oplus H) + i.$$

Optimization problem Φ is of *finite integer index* if $\sim_{\Phi,l}$ is finite for each fixed l.

Note that a if reduction-counter rule $((H,H'),i)$ is safe for a graph optimization problem Φ, then $H \sim_{\Phi,l} H'$. Furthermore, if $H \sim_{\Phi,l} H'$ for two l-terminal graphs H and H', then there is an $i \in \mathbb{Z}$ for which the reduction-counter rule $((H,H'),i)$ is safe for Φ. Note furthermore that, for each $l \geq 0$, $\sim_{\Phi,l}$ is a refinement of $\sim_{P,l}$, where P is the derived graph property of Φ. Hence if Φ is of finite integer index, then the derived property P is of finite index.

For any graph optimization problem Φ and any integer $k \geq 1$, Φ_k is the graph optimization problem with for each graph G,

$$\Phi_k(G) = \begin{cases} \mathsf{false} & \text{if } \mathrm{tw}(G) > k \\ \Phi(G) & \text{otherwise.} \end{cases}$$

From Lemma 5.1.6 and Lemma 5.1.7, it follows that, if Φ is of finite integer index, then for each $k \geq 1$, Φ_k is of finite integer index.

The following theorem is the analog of Theorem 5.1.2 for finite integer index problems.

Theorem 5.2.2. *Let Φ is a graph optimization problem of finite integer index. For each integer $k \geq 1$ there exists a special reduction-counter system for Φ_k.*

If Φ is also effectively computable and there is an equivalence relation $\sim_l$, for each $l \geq 0$, which is a finite refinement of $\sim_{\Phi,l}$ and is effectively decidable, then such a special reduction-counter system S can effectively be constructed. Moreover, for each reduction-counter rule $((H,H'),i)$ in S, $H \sim_l H'$.

Proof. Let $k \geq 1$. Let P be the derived graph property of Φ. Since for each $l \geq 0$, $\sim_{\Phi_k,l}$ is a refinement of $\sim_{P_k,l}$, Corollary 5.1.1 implies that there is a special reduction system $S = (R, I)$ for P, such that for each $(H,H') \in R$, $H \sim_{\Phi_k,l} H'$. We show that we can construct a special reduction-counter system for Φ for which S is the derived reduction system. For each reduction rule (H,H'), make a reduction-counter rule $((H,H'),i)$, where $i = 0$ if for all G, $\Phi(H \oplus G) = \mathsf{false}$ (and hence $\Phi(H' \oplus G) = \mathsf{false}$), and $i = \Phi(H \oplus G) \Leftrightarrow \Phi(H' \oplus G)$ for some G such that $\Phi(H \oplus G) \in \mathbb{Z}$ otherwise. Let R' denote the set of all these reduction-counter rules. Let $\phi : I \to \mathbb{Z}$ be the function mapping each graph $G \in I$ to its value $\Phi(G)$. Then (R', I, ϕ) is a special reduction-counter system for Φ.

If Φ is effectively computable and we have a refinement $\sim_l$ of $\sim_{\Phi,l}$, for each $l \geq 0$, then Φ_k is effectively computable and P and P_k are effectively decidable. Hence we can effectively construct a special reduction system (R, I) for P_k, such that for each rule (H,H'), $H \sim_l H'$. Furthermore, we can turn this reduction system in a special reduction-counter system (R', I, ϕ) for Φ in the following way. The function ϕ can be computed by simply computing $\Phi(G)$ for each $G \in I$.

For each reduction rule $r = (H,H') \in R$, we compute an integer i such that (r,i) is a safe reduction-counter rule in R. Suppose H and H' are l-terminal graphs. Let G be a finite class of l-terminal graphs containing at least one terminal graph of each equivalence class of $\sim_{\Phi,l}$. Such a set G can be effectively computed, as is described in the proof of Theorem 5.1.2. Now if there is a $G \in G$ for which $\Phi(H \oplus G) \in \mathbb{Z}$, then take any such G and let $i = \Phi(H \oplus G) \Leftrightarrow \Phi(H' \oplus G)$. Note that, since $H \sim_{\Phi,l} H'$, for each $G \in G$ with $\Phi(G \oplus H) \in \mathbb{Z}$, $\Phi(G \oplus H) \Leftrightarrow \Phi(G \oplus H')$ has the same value, hence this gives a proper value. If G contains no graph G for which $\Phi(H \oplus G) \in \mathbb{Z}$, then let $i = 0$. Note that in this case, for every l-terminal graph G, $\Phi(H \oplus G) = \Phi(H' \oplus G) = \mathsf{false}$, and hence $\Phi(H \oplus G) = \mathsf{false} = \mathsf{false} + 0 = \Phi(H' \oplus G) + i$. Let R' be the set of all reduction-counter rules that are found this way. □

Unfortunately, we can not apply Theorem 5.2.2 to all MS-definable graph optimization problems (as defined in Section 2.2.4). Hence the analog of Corollary 5.1.2 does not hold for optimization problems. However, there are a number of problems for which we can prove that they are of finite integer index. We give them in the next theorem. A precise definition of these problems is given in Appendix A. In Chapter 7, we prove that these problems are of finite integer index (Theorem 7.1.2). These proofs make use of techniques introduced in Chapter 6.

Theorem 5.2.3. *The following problems are of finite integer index:* MAX INDUCED d-DEGREE SUBGRAPH *for all $d \geq 1$,* MAX INDEPENDENT SET, MIN VERTEX COVER, MIN

p-DOMINATING SET *for all* $p \geq 1$, MAX CUT *on graphs with bounded degree*, MIN PARTITION INTO CLIQUES, MIN HAMILTONIAN PATH COMPLETION, *and* MAX LEAF SPANNING TREE.

5.3 Parallel Reduction Algorithms

Bodlaender and Hagerup [1995] have given an efficient parallel variant of algorithm Reduce, based on a variant of the special reduction system. In this section, we describe this variant of the special reduction system, and we describe the structure of the algorithm. Because the algorithm is quite complicated, we do not give all details. We start with reduction algorithms for decision problems.

5.3.1 Decision Problems

The basic idea of the parallel reduction algorithm is that, if there are two or more possible applications of reduction rules at a certain time, and these applications do not interfere, then they can be applied concurrently.

Definition 5.3.1 (Non-Interfering Matches). Let R be a set of reduction rules and let G be a graph with a fixed adjacency list representation. Two matches G_1 and G_2 in G are said to be *non-interfering* if

- no inner vertex of G_i ($i = 1,2$) is a vertex of G_{3-i},
- the sets of edges of G_1 and G_2 are disjoint, and
- if G_1 and G_2 have a common terminal x, then in the adjacency list of x, there are no two consecutive edges e_1 and e_2 such that $e_1 \in E(G_1)$ and $e_2 \in E(G_2)$. A set of matches in G is non-interfering if all matches in the set are pairwise non-interfering.

Let R be a set of reduction rules and let G be a graph with a fixed adjacency list representation. If we have a set of non-interfering matches in G, then the reductions corresponding to these matches can be executed in parallel, which gives the same result as if the reductions were executed subsequently, in an arbitrary order. In order to make a parallel reduction algorithm for a given graph property P efficient, we must have a special reduction system which gives sufficiently many matches in any graph G for which P holds. Therefore, we introduce a special *parallel* reduction system.

Definition 5.3.2 (Special Parallel Reduction System). Let P be a graph property, and (R, I) a decreasing reduction system for P. Let n_{max} be the maximum number of vertices in any left-hand side of a rule $r \in R$. (R, I) is called a *special parallel reduction system* for P if we know positive integers n_{min} and d, $n_{min} \leq n_{max} \leq d$, and a constant $c > 0$, such that the following conditions hold.

1. For each reduction rule $(H_1, H_2) \in R$,
   (a) if H_1 has $l > 0$ terminals, then H_1 is connected and H_1 and H_2 are open, and

(b) if H_1 is a zero-terminal graph, then $|V(H_2)| < n_{min}$.

2. For each graph G and each adjacency list representation of G, if $P(G)$ holds, then

   (a) each component C of G with at least n_{min} vertices has at least $c \cdot |V(C)|$ d-discoverable matches, and

   (b) if all components of G have less than n_{min} vertices, then any subgraph G' of G induced by a set of components of G with $n_{min} \leq |V(G')| \leq n_{max}$ is a match.

Note that, since for each integer $n > 1$ and each constant c, if $c > 0$ then $cn > 0$, a special parallel reduction system is also a special reduction system.

Consider the graph property which holds if a graph is two-colorable, has an odd number of components, and all its components are cycles. The reduction system that we have given for this property on page 100 is an example of a special parallel reduction system (take $d = n_{max} = 8$, $n_{min} = 5$ and $c = 1/5$).

Let P be a graph property and $S = (R, I)$ a special parallel reduction system for P. Let n_{min}, n_{max}, d and c be as in Definition 5.3.2. The parallel reduction algorithm introduced by Bodlaender and Hagerup [1995] based on S works as follows. Like algorithm Reduce, the algorithm consists of two phases (see also page 100 – 102). In the first phase, the algorithm finds d-discoverable matches and executes the corresponding reductions, until there are no more d-discoverable matches. In the second phase, the algorithm tries to finish the reductions by applying reduction rules with zero terminals. We first describe the first phase in more detail.

Phase 1. Suppose we are given an input graph G with n vertices. The first phase consists of a number of reduction rounds, which are subsequently executed. In each reduction round, $\Omega(m)$ reductions are applied to the current graph, which has m vertices, if possible. Furthermore, each component of less than n_{min} vertices is removed from the graph, and is added to a graph H (which is initially the empty graph). This is done in four steps.

1. In the first step, for each vertex v which has degree at most d it is checked whether v is contained in a component of less than n_{min} vertices, and if so, then this component is removed from G and added to H.
2. In the second step, the algorithm tries to discover a match from each vertex v which has degree at most d, and if this succeeds, the corresponding reduction rule r is looked up. Let A denote the set of all matches that are found. Note that A is not necessarily non-interfering. If $|A| < cm$, where m is the number of vertices of the current graph, then P does not hold for the input graph (if $m \neq 0$, then it must be the case that $m \geq n_{min}$, since otherwise, in step 1 of this reduction round, all components of G are removed), and false is returned. Otherwise, go on with the next step.
3. In the third step, the algorithm computes a subset A' of A with size $\Omega(|A|)$, which is a set of non-interfering matches.
4. In the last step, all reductions corresponding to the matches in A' are applied.

The first, second and fourth step can be done in constant time on m processors, without concurrent reading or writing: in step 1 and 2, take one processor for each vertex of degree at most d. In step 4, for each match in A', let the processor which discovered the match in step 2 apply its corresponding reduction. The third step is more complicated. It is basically done as follows. First, a *conflict graph* of all matches in A is built. This graph contains a vertex for each match in A, and an edge between two vertices if and only if the corresponding matches are interfering. Now an independent set in the conflict graph corresponds to a set of non-interfering matches. It can be seen that the conflict graph has bounded degree. This means that there is an independent set A' of size $\Omega(|A|)$ which can be found efficiently in parallel on an EREW PRAM (for more details, see Bodlaender and Hagerup [1995]).

The reduction rounds are finished as soon as the graph is empty. The actual result of the reductions is the graph H, which consists of components with less than n_{min} vertices. The algorithm proceeds with phase 2, with the graph H.

Phase 2. In the second phase, a graph H is given which consists of a number of components, each having less than n_{min} vertices. Let C be a collection of subgraphs of H. Initially, let C be the collection of connected components of H. Until C contains only one element, a number of *join-reduce* rounds is performed with C and H.

In each join-reduce round, a set $S = \{(G_1, G_1'), (G_2, G_2'), \ldots, (G_t, G_t')\}$ of pairs of graphs in C is computed, such that each graph of C occurs at most once in S, and the set S contains as many pairs as possible, i.e. $|S| = \lfloor |C|/2 \rfloor$ For each such pair (G_i, G_i') in S it is checked whether $G_i \cup G_i'$ is a match to some rule $r = (H_1, H_2)$. If so, the corresponding reduction rule is applied: in H, G_i and G_i' are replaced by a graph G' which is isomorphic to H_2, and in C, we replace G_1 and G_2 by G_3. If G_i and G_i' do not form a match, then either $|V(G_i) \cup V(G_i')| < n_{min}$ or $|V(G_i) \cup V(G_i)| \geq n_{min}$. In the first case, G_i and G_i' are replaced by $G_i \cup G_i'$ in the collection C. In the latter case, false is returned, since P does not hold (condition 2b of a special parallel reduction system).

The algorithm performs the join-reduce rounds until $|C| = 1$. If $H \in I$ then true is returned, otherwise, false is returned.

By the definition of a special parallel reduction system it can be seen that phase 2 returns true if and only if P holds for the graph H as it was passed to this phase. This means that the algorithm described above is correct.

Consider the amount of resources used by the algorithms. In each reduction round of the first phase, the number of vertices of the current graph is decreased by at least a constant fraction. This means that there are $O(\log n)$ reduction rounds. The only part in a reduction round of phase 1 which takes more than constant time is step 3. By a careful analysis, it can be seen that phase 1 of the algorithm can be made to run in $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM. For a CRCW PRAM, the algorithm can be slightly improved: it runs in $O(\log n)$ time with $O(n)$ operations and space.

In phase 2, the collection C initially contains at most $|V(H)|$ components and $|V(H)| \leq n$. Hence phase 2 consists of $O(\log n)$ join-reduce rounds. Furthermore, by the definition of a special parallel reduction system, C contains only graphs with less than n_{min} vertices, which

implies that phase 2 can be done in $O(\log n)$ time with $O(n)$ operations and space on an EREW PRAM. Hence we have the following result.

Theorem 5.3.1. *Let P be a graph property. If we have a special parallel reduction system for P, then we have an algorithm which decides P in $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and in $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM.*

The definition of a special parallel reduction system, Lemma 5.1.8 and (the proof of) Theorem 5.1.2 immediately imply the following result.

Theorem 5.3.2. *Let P a graph property, and suppose P is of finite index. For each integer $k \geq 1$, there is a special parallel reduction system for P_k.*

If P is also effectively decidable, and there is an equivalence relation $\sim_l$ for each $l \geq 0$, which is a finite refinement of $\sim_{P,l}$ and is effectively decidable, then such a system (R, I) can effectively be constructed. Moreover, for each rule $(H, H') \in R$, $H \sim_l H'$

The analog of Corollary 5.1.2 also holds for the parallel case.

In the parallel case, there exist algorithms that decide finite index properties in $O(\log n)$ time with $O(n)$ operations and space, given a tree decomposition of bounded width of the graph (see also Section 2.2.4). However, the best known parallel algorithm for finding a tree decomposition of the input graph takes $O(\log^2 n)$ time with $O(n)$ operations on an EREW or CRCW PRAM [Bodlaender and Hagerup, 1995]. Hence the reduction algorithms presented in this section are more efficient.

5.3.2 Optimization Problems

It is again easy to adapt the parallel reduction algorithm for optimization problems. Therefore, we define a special parallel reduction-counter system to be a reduction-counter system of which the derived reduction system is a special parallel reduction system.

For instance, the reduction-counter system for MAX INDEPENDENT SET on cycles that we defined on page 109 is a special parallel reduction-counter system for this problem.

Let Φ be a graph optimization problem, and $S = (R, I, \phi)$ a special parallel reduction-counter system for Φ. A parallel reduction algorithm based on S is a combination of the parallel reduction algorithm based on the derived reduction system, and the sequential reduction algorithm described in Section 5.2. Each processor has a counter, which is initially set to zero. If a processor applies a reduction-counter rule in either phase 1 or phase 2 of the algorithm, then it uses its own counter. After the last round of phase 2 is finished, the counters of all processors are added up. Let *cnt* denote the resulting counter, let G denote the input graph and H the reduced graph. Now, if $H \in I$, then $\Phi(G) = cnt + \phi(H)$, otherwise, $\Phi(G) = \Phi(H) = \mathsf{false}$. The sum of all the counters can be computed in $O(\log n)$ time with $O(n)$ operations and space on an EREW PRAM, which means that the total algorithm runs in $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, or in $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM.

Theorem 5.3.3. *Let Φ be a graph optimization problem. If we have a special parallel reduction-counter system for Φ, then we have an algorithm which, for each graph G with n vertices, computes $\Phi(G)$ in $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and in $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM.*

By Lemma 5.1.8 and Theorem 5.2.2, we also have the following result.

Theorem 5.3.4. *Let Φ be a graph optimization problem which is of finite integer index. For each integer $k \geq 1$, there exists a special parallel reduction system S for Φ_k.*

If, in addition, Φ is effectively computable, and there is an equivalence relation $\sim_l$, for each $l \geq 0$, which is a finite refinement of $\sim_{\Phi,l}$ and is effectively decidable, then such a system S can effectively be constructed, and for each reduction-counter rule $((H,H'),i)$ in S, $H \sim_l H'$.

Theorem 5.3.4 implies that there are special parallel reduction-counter systems for the following problems on graphs of bounded treewidth (see also Theorem 5.2.3): MAX INDUCED d-DEGREE SUBGRAPH for all $d \geq 0$, MIN p-DOMINATING SET for all $p \geq 1$, MIN VERTEX COVER, MAX CUT on graphs with bounded degree, MIN PARTITION INTO CLIQUES, MIN HAMILTONIAN PATH COMPLETION, and MAX LEAF SPANNING TREE.

5.4 Additional Results

It is possible to generalize the results in this chapter to directed, mixed and/or labeled graphs. In the case of labeled graphs, we can allow the input graph to have a labeling of the vertices and/or edges, where the labels are taken from a set of constant size. These labels could also act as weights for finite integer index problems, e.g., we can deal with MAX WEIGHTED INDEPENDENT SET, with each vertex a weight from $\{1,2,\ldots,c\}$ for some fixed c, in the same way as we dealt with MAX INDEPENDENT SET. Each of these generalizations can be handled in a very similar way as the results that are given in this chapter.

It is possible to generalize the results on reduction algorithms based on special reduction systems to multigraphs. As we will use this generalized result in Chapters 8 and 9, we give a brief description of how this is done. Instead of ordinary terminal graphs (Definition 2.2.3), we use terminal multigraphs, which are terminal graphs that may have multiple edges. The operation $\oplus$ on terminal multigraphs is similar to the operation $\oplus$ on ordinary terminal graphs (Definition 2.2.4), except that it does not remove multiple edges. Reduction rules, matches and applications of reduction rules are the same as for ordinary graphs (Definition 5.1.1), except that they are based on the definitions of terminal multigraphs and the new operation $\oplus$ as given above.

Wherever we mentioned the number of vertices of a (terminal) graph, we replace this by the number of vertices plus the number of edges of the (terminal) multigraph. For example, a set of reduction rules R is decreasing if for each $(H_1,H_2) \in R$, $|V(H_2)| + |E(H_2)| < |V(H_1)| + |E(H_1)|$. Note that this implies that on each graph $G = (V,E)$, a sequence of at most $|V| + |E|$ reduction rules can be applied, and hence this influences the running time of the reduction algorithms.

We also slightly modify the definition of discoverability. We do not require that the left-hand and right-hand side of a reduction rule are open. As a consequence, not only all vertices of a match must be d-discoverable, but also all edges.

Definition 5.4.1. Let d be a positive integer, let G be a multigraph, given by some adjacency list representation, and let G_1 be an l-terminal multigraph. We say G_1 is d-discoverable in G if

1. G_1 is connected, the maximum degree (i.e. the maximum number of incident edges) of any vertex in G_1 is at most d,
2. there is an l-terminal multigraph G_2, such that $G = G_1 \oplus G_2$, and
3. G_1 contains an edge e, such that for all edges $e' \in E(G_1)$, there is a walk W in G_1 with $W = (u_0, e_0, u_1, e_1, \ldots, e_s, u_{s+1})$, where $e_0 = e$, $e_s = e'$, and for each i, $1 \leq i \leq s$, in the adjacency list of u_i in G, the edges e_{i-1} and e_i have distance at most d.

Note that if condition 3 of Definition 5.4.1 holds for a terminal graph G_1 in G, then for every two edges e and e' of G_1 there is a walk $W = (u_0, e_0, \ldots, e_s, u_{s+1})$ in G_1 with $e = e_0$ and $e' = e_s$, such that for each i, $0 \leq i < s$, e_i and e_{i+1} have distance at most d in the adjacency list of u_i, and furthermore, each edge of G_1 occurs at most twice in W (see also page 99).

We redefine a special reduction system for graph properties on multigraphs.

Definition 5.4.2 (Special Reduction System for Multigraphs). Let P be a graph property, and $(\mathcal{R}, \mathcal{I})$ a decreasing reduction system for P. Let n_{max} be the maximum number of vertices plus edges in any left-hand side of a rule $r \in \mathcal{R}$. $(\mathcal{R}, \mathcal{I})$ is called a *special reduction system* for P if we know positive integers n_{min} and d, $n_{min} \leq n_{max} \leq d$, such that the following conditions hold.

1. For each reduction rule $(H_1, H_2) \in \mathcal{R}$,
   (a) if H_1 has $l > 0$ terminals, then H_1 is connected, and
   (b) if H_1 is a zero-terminal graph, then $|V(H_2)| + |E(H_2)| < n_{min}$.
2. For each graph G and each adjacency list representation of G, if $P(G)$ holds, then
   (a) each component of G with at least n_{min} vertices plus edges has a d-discoverable match, and
   (b) if all components of G have less than n_{min} vertices plus edges, then either $G \in \mathcal{I}$ or G contains a match which is a zero-terminal graph.

A special parallel reduction system for graph properties on multigraphs is defined as follows.

Definition 5.4.3 (Special Parallel Reduction System for Multigraph). Let P be a graph property, and $(\mathcal{R}, \mathcal{I})$ a decreasing reduction system for P. Let n_{max} be the maximum number of vertices plus edges in any left-hand side of a rule $r \in \mathcal{R}$. $(\mathcal{R}, \mathcal{I})$ is called a *special parallel reduction system* for P if we know positive integers n_{min} and d, $n_{min} \leq n_{max} \leq d$, and a constant $c > 0$, such that the following conditions hold.

1. conditions 1a and 1b of Definition 5.4.2 hold.
2. For each graph G and each adjacency list representation of G, if $P(G)$ holds, then
   (a) each component C of G with at least n_{min} vertices plus edges has at least $c \cdot (|V(C)| + |E(C)|)$ d-discoverable matches, and
   (b) if all components of G have less than n_{min} vertices plus edges, then any subgraph G' of G induced by a set of components of G with $n_{min} \leq |V(G')| + |E(G')| \leq n_{max}$ is a match.

A special (parallel) reduction-counter system is again a (parallel) reduction-counter system of which the derived (parallel) reduction system is special.

The algorithms can now by adapted as follows. First consider the sequential reduction algorithm which is given on page 101. We modify it as follows. Instead of searching for d-discoverable matches from vertices, we search from edges. Therefore, we first fill the set S (line 3) with all edges in $E(G)$. In each iteration of the main loop of phase 1, we take one edge from S and we look for a d-discoverable match G_1 to some reduction rule $r = (H_1, H_2)$. If one can be found, we apply it by replacing G_1 by a graph G_2 isomorphic to H_2. We remove all edges of G_1 which are not in G_2 from the set S. After that, we add all edges of G_2 which were not already in G_1 to the set S, and furthermore, we add all edges e of the graph G for which there is a terminal x of G_1 such that $x \in e$, and in the adjacency list of x, something changed within a distance d from e (where d denotes the constant for discoverability). Phase 2 of the algorithm does not have to be changed. It can be seen that the algorithm is still correct, and that it runs in $O(n+m)$ time using $O(n+m)$ space. For the sequential reduction-counter system we can use the same modifications and get the same time and space bounds.

Theorem 5.4.1. *Let P be a graph property and Φ a graph optimization problem, both for multigraphs.*

- *If we have a special reduction system for P, then we have an algorithm which solves P in $O(n+m)$ time and space.*
- *If we have a special reduction-counter system for Φ, then we have an algorithm which solves Φ in $O(n+m)$ time and space.*

Consider the parallel algorithm (page 113). We use the same type of modification, i.e. we search from edges instead of vertices of degree at most d. In the first step of phase 1, the algorithm checks for each edge whether it is contained in a component of less than n_{min} vertices plus edges, and in the second step of phase 1, the algorithm tries to find a d-discoverable match from each edge. It is easy to check that this gives the following results.

Theorem 5.4.2. *Let P be a graph property and Φ a graph optimization problem, both for multigraphs.*

- *If we have a special parallel reduction system for P, then we have an algorithm which solves P in $O(\log(n+m)\log^*(n+m))$ time with $O(n+m)$ operations and space on an EREW PRAM, and in $O(\log(n+m))$ time with $O(n+m)$ operations and space on a CRCW PRAM.*

- *If we have a special parallel reduction-counter system for Φ, then we have an algorithm which solves Φ in $O(\log(n+m)\log^*(n+m))$ time with $O(n+m)$ operations and space on an EREW PRAM, and in $O(\log(n+m))$ time with $O(n+m)$ operations and space on a CRCW PRAM.*

Unfortunately, the results on the existence of special (parallel) reduction(-counter) systems for many problems on graphs of bounded treewidth (Theorems 5.1.2, 5.2.2, 5.3.2, and 5.3.4) can not directly be generalized to multigraphs of bounded treewidth.

Chapter 6

Constructive Reduction Algorithms

In Chapter 5 we have shown that for many decision and optimization problems on graphs of bounded treewidth, efficient reduction algorithms can be designed that decide these problems. Many decision and optimization problems however, have a constructive version, in which we are not only interested in whether a certain property holds for a given graph, but we are also interested in a *solution*, if the property holds. For example, in the constructive version of k-COLORABILITY we want to find a k-coloring of a given graph, if one exists. In the constructive version of MAX INDEPENDENT SET, we want to find an independent set of maximum size in a given graph, and we are not only interested in the maximum size itself.

The reduction algorithms that we described in Chapter 5 do not provide a possibility to construct solutions. In this chapter, we show how reduction algorithms can be adapted in such a way that solutions can be constructed, and we show that these algorithms run within the same time and resource bounds as the basic reduction algorithms (both sequentially and in parallel). We also show that for a number of graph problems on graphs of bounded treewidth, these algorithms can be used. For example, they can be used for all MS-definable construction problems whose structure of a solution satisfies certain conditions.

The basic idea of a constructive reduction algorithm is the following. The algorithm consists of two parts. In the first part, an ordinary reduction algorithm is applied. The reduced graph is then passed to the second part. In this part, a solution is constructed for the reduced graph, if it exists (in the case of an optimization problem, this solution is guaranteed to be optimal). After that, the reductions that are applied in part 1 are undone one by one, in reversed order, and each time a reduction is undone, the (optimal) solution of the graph is adapted to an (optimal) solution of the new graph. This results in a solution of the input graph, which is an optimal solution in the case of a graph optimization problem.

In order to keep the running time and amount of resources for the second part within the same bounds as for first part, we must be able to efficiently construct an (optimal) solution for the new graph from an (optimal) solution of the old graph, after an undo-action is applied. Therefore, we require that the new solution can efficiently be constructed from the old solution.

In this chapter, we define a type of reduction system for which this can be done efficiently, both for decision and for optimization problems. We also determine for what kind of problems such a reduction system exists. In Section 6.1, we develop the theory for sequential reduction algorithms for decision problems. In Section 6.2, we extend this to optimization

problems, and in Section 6.3, we show that this method can also be applied in parallel. Finally, in Section 6.4, we discuss some additional results. In all sections, except Section 6.4, the graphs we consider are simple.

6.1 Decision Problems

We start with a definition of a constructive reduction system and an extension of the efficient reduction algorithm presented in Section 5.1.2 to construction problems. After that, we show how this algorithm can be applied to solve a large class of construction problems on graphs of bounded treewidth.

6.1.1 Constructive Reduction Systems and Algorithms

Recall that a graph property is a function P which maps each graph to one of the values true and false. Many graph properties are of the form

$$P(G) = \text{'there is an } S \in D(G) \text{ for which } Q(G,S) \text{ holds'},$$

where $D(G)$ is a *solution domain* (or shortly domain), which is some set depending on G, and Q is an extended graph property of G and S, i.e. $Q(G,S) \in \{\text{true}, \text{false}\}$ for all graphs G and all $S \in D(G)$. An $S \in D(G)$ for which $Q(G,S)$ holds is called a *solution* for G. For example, for the perfect matching problem on a graph G, $D(G)$ can be $\mathcal{P}(E)$, the power set of E, and for $S \in D(G)$, $Q(G,S)$ holds if and only if every vertex in G is end point of exactly one edge in S. Hence S is a solution for G if S is a perfect matching of G.

If a graph property is of the form $P(G) =$ 'there is an $S \in D(G)$ for which $Q(G,S)$ holds', then we call P a *construction property* defined by the pair (D,Q).

In this section, we consider constructive reduction algorithms which, for a construction property P defined by (D,Q), do not only decide P, but if P holds for an input graph G, also construct an $S \in D(G)$ for which $Q(G,S)$ holds.

Definition 6.1.1 (Constructive Reduction System). Let P be a construction property defined by (D,Q). A *constructive reduction system* for P is a quadruple $(\mathcal{R}, \mathcal{I}, A_R, A_I)$, where

- $(\mathcal{R}, \mathcal{I})$ is a reduction system for P,
- A_R is an algorithm which, given
  - a reduction rule $r = (H_1, H_2) \in \mathcal{R}$,
  - two terminal graphs G_1 and G_2, such that G_1 is a isomorphic to H_1 and G_2 is isomorphic to H_2,
  - a graph G with $G = G_2 \oplus H$ for some H, and
  - an $S \in D(G)$, such that $Q(G,S)$ holds,

  computes an $S' \in G_1 \oplus H$ such that $Q(G_1 \oplus H, S')$ holds,
- A_I is an algorithm which, given a graph G which is isomorphic to some $H \in \mathcal{I}$, computes an $S \in D(G)$ for which $Q(G,S)$ holds.

Algorithm A_I in a constructive reduction system (R, I, A_R, A_I) is used to construct an initial solution of the reduced graph G, if $G \in I$. Algorithm A_R is used to reconstruct a solution, each time a reduction is undone on the graph.

As an example, consider the constructive version of the graph property P which holds for graphs G of which each component is a two-colorable cycle, and the number of components is odd (see Chapter 5, page 100): we are looking for a two-coloring of the graph, if the graph is two-colorable, all its components are cycles, and the number of its components is odd. For each graph G, let $D(G)$ be the set of partitions (V_1, V_2) of $V(G)$, and for each $S \in D(G)$, let $Q(G,S)$ be true if and only if S is a two-coloring of G, each component of G is a cycle, and the number of components of G is odd.

We extend the reduction system for P given on page 100 (Figure 5.3) to a constructive reduction system for P. Algorithm A_R uses a table: for each reduction rule $(H_1, H_2) \in R$, and each possible two-coloring of the terminal graph H_2, it gives a two-coloring of the terminal graph H_1 which is the same on the set of terminals. The contents of this table are depicted in part I of Figure 6.1 (symmetric cases are considered only once, hence for both rules, there is only one two-coloring). Given as input a reduction rule r, two terminal graphs G_2 and G_1, a graph $G = G_2 \oplus H$, and a two-coloring of G, algorithm A_R can easily compute a two-coloring of $G_1 \oplus H$ using the given table: the algorithm looks which vertices of G_2 have which color, and looks up the corresponding coloring of G_1 in the table. Then it removes the inner vertices of G_2 from the solution, and adds the inner vertices of G_1 in the correct way.

Algorithm A_I also uses a table: for the only element $H \in I$, this table contains a two-coloring of H. See part II of Figure 6.1. Hence (R, I, A_R, A_I) is a constructive reduction system for P defined by (D, Q). Note that both algorithms can be made to run in $O(1)$ time if we use a convenient data structure.

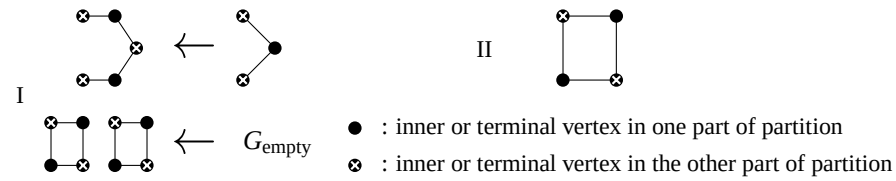


Figure 6.1. Example of tables used by A_R and A_I for constructive reduction system for two-colorability on graphs of which each component is a cycle, and the number of components is odd.

In order to make an efficient constructive reduction algorithm based on a constructive reduction system (R, I, A_R, A_I), we want that algorithms A_R and A_I work efficiently. This is required in a special constructive reduction system.

Definition 6.1.2 (Special Constructive Reduction System). Let P be a construction property defined by (D, Q). A constructive reduction system (R, I, A_R, A_I) for P is a *special constructive reduction system* for P if

1. (R, I) is a special reduction system for P (Definition 5.1.5), and
2. algorithms A_R and A_I run in $O(1)$ time.

Note that the constructive reduction system we gave for two-colorability of graphs of which each component is a cycle, is a special constructive reduction system, since algorithms A_I and A_R as described above take constant time, and we have shown on page 100 that the reduction system depicted in Figure 5.3 is a special reduction system for the problem.

One way to obtain an algorithm A_R in a constructive reduction system which runs in $O(1)$ time is to ensure that A_R only has to change a solution locally, i.e. that the solution to construct only differs from the input solution in the part of the graph that was involved in the reduction. We use this technique in most of our algorithms.

Let P be a construction property defined by (D,Q) and let (R, I, A_R, A_I) be a special constructive reduction system for P. The following algorithm computes for a given graph G a solution of G if one exists.

Algorithm Reduce-Construct(G)
Input: Graph G
Output: $S \in D(G)$ for which $Q(G,S)$ holds if $P(G)$ holds, false otherwise

1. ($*$ Part 1 $*$)
2. Apply as many reductions as possible on G in the way of algorithm Reduce. Store the resulting sequence $(G_1^1, G_2^1), (G_1^2, G_2^2), \ldots, (G_1^t, G_i^t)$, where t denotes the number of reductions, and for each i, $1 \leq i \leq t$, in the ith reduction, G_1^i is replaced by G_2^i. Let G be the reduced graph.
3. ($*$ Part 2 $*$)
4. **if** $G \notin I$ **then return** false
5. ($*$ Construct initial solution $*$)
6. $S \leftarrow A_I(G)$
7. **for** $i \leftarrow t$ **downto** 1
8. **do** let $r = (H_1, H_2) \in R$ such that H_1 and G_1^i are isomorphic and H_2 and G_2^i are isomorphic.
9. ($*$ reconstruct solution $*$)
10. $S \leftarrow A_R(r, G_1^i, G_2^i, G, S)$
11. ($*$ undo ith reduction $*$)
12. $G \leftarrow H \oplus G_1$ (H denotes the terminal graph for which $G = G_2^i \oplus H$)
13. **return** S

It is clear from Lemma 5.1.4 and the definition of a constructive reduction system that algorithm Reduce-Construct is correct. Consider the running time of the algorithm. Part 1 takes $O(n)$ time, by Lemma 5.1.5. In part 2, the initial solution can be constructed in constant time, since algorithm A_I takes $O(1)$ time. Every undo-action also takes constant time: undoing a reduction can be done in the same way as applying it, which takes $O(1)$ time, and algorithm A_R uses $O(1)$ time (note that the terminal graph H as described in line 12 is not explicitly computed). Hence the complete algorithm takes $O(n)$ time. This proves the following theorem.

Theorem 6.1.1. *Let P be a construction property defined by the pair (D,Q). If we have a special constructive reduction system for P, then we have an algorithm which, given a graph G, returns a solution $S \in D(G)$ for which $Q(G,S)$ holds, if $P(G)$ holds, and* false *otherwise. The algorithm runs in $O(n)$ time and uses $O(n)$ space.*

6.1.2 Construction Problems for Graphs of Bounded Treewidth

In this section we show that algorithm Reduce-Construct can be used for a large class of construction properties on graphs of bounded treewidth.

Let D be some solution domain, i.e. for each graph G, $D(G)$ denotes a 'set of all possible solutions' in G. Let G and H be l-terminal graphs and let $S \in D(G \oplus H)$. We want to be able to restrict S to the terminal graphs G and H. For these restrictions, we use the notation $S[G]$ and $S[H]$. We can define such a restriction in several ways, but we require that $S[G]$ does not contain any vertices or edges which are not in G. Suppose we have given a definition of $[\,]$ for a solution domain D. We say $[\,]$ is *properly defined* if, for any terminal graphs G and H, and any $S \in D(G \oplus H)$, $S[G]$ contains no vertices or edges which are not in G. An obvious proper definition of $[\,]$ is thus to obtain $S[G]$ from S by removing all vertices and edges which are not in G from S. For example, if for each graph G, $D(G)$ is the set of all partitions of $V(G)$ in three sets, then a proper definition of $[\,]$ would be that for each l-terminal graphs G and H and each $S = (V_1,V_2,V_3) \in D(G \oplus H)$, $S[G] = (V_1 \cap V(G), V_2 \cap V(G), V_3 \cap V(G))$.

Definition 6.1.3. Let D be a solution domain, let a proper definition of $[\,]$ be given. For each $l \geq 0$, and each l-terminal graph G, define

$$D_{[]}(G) = \{S[G] \mid S \in D(G \oplus H) \text{ for some } l\text{-terminal graph } H\}.$$

Each $S \in D_{[]}(G)$ is called a *partial solution* of G, and $D_{[]}$ is called the partial solution domain for D.

Definition 6.1.4. Let D be a solution domain and let a proper definition of $[\,]$ be given. D is *inducible* for $[\,]$ if each two l-terminal graphs G and H ($l \geq 0$) and $S_G \in D_{[]}(G)$ and $S_H \in D_{[]}(H)$, there is at most one $S \in D(G \oplus H)$ such that $S[G] = S_G$ and $S[H] = S_H$.

Let G and H be l-terminal graphs, let $S_G \in D_{[]}(G)$ and $S_H \in D_{[]}(H)$. If there is exactly one $S \in D(G \oplus H)$ such that $S[G] = S_G$ and $S[H] = S_H$, then (G,S_G) and (H,S_H) are called $\oplus$*-compatible*, and we write $S_G \oplus S_H = S$.

As an example of Definition 6.1.4, consider the solution domain D with for each graph G, $D(G) = P(V)$. Let $[\,]$ be defined as follows. For every two l-terminal graphs H_1 and H_2 and each $S \in D(H_1 \oplus H_2)$, let $S[H_1] = S \cap V(H_1)$ (hence $D_{[]}(H_1) = P(V(H_1))$). Then $[\,]$ is properly defined and D is inducible for $[\,]$. If $H_1 = (V_1, E_1, \langle x_1, \ldots, x_l \rangle)$ and $H_2 = (V_2, E_2, \langle y_1, \ldots, y_l \rangle)$ are l-terminal graphs, and $S_1 \in D_{[]}(H_1)$ and $S_2 \in D_{[]}(H_2)$, then (H_1,S_1) and (H_2,S_2) are $\oplus$-compatible if and only if S_1 and S_2 contain the same set of terminals, i.e.

$$\{i \mid 1 \leq i \leq l \wedge x_i \in S_1\} = \{i \mid 1 \leq i \leq l \wedge y_i \in S_2\}.$$

In that case, $S_1 \oplus S_2$ is simply the union of S_1 and S_2 in $H_1 \oplus H_2$.

As another example of Definition 6.1.4, consider the solution domain D with for each G,

$$D(G) = \{V \subseteq P(V) \mid \forall_{V \in V} |V| = 3\},$$

i.e. each $S \in D(G)$ is a set of subsets of cardinality three of $V(G)$. The obvious proper definition of $[]$ is as follows. For each two terminal graphs G and H, each $S \in D(G \oplus H)$, $S[G] = \{W \cap V(G) \mid W \in S\}$. However, D is not inducible for this definition. Consider for example two one-terminal graphs G and H as depicted in part I of Figure 6.2, and consider the partial solutions $S_G \in D_{[]}(G)$ and $S_H \in D_{[]}(H)$ denoted by the dotted lines. Part II of Figure 6.2 shows an $S \in D(G \oplus H)$ for which $S[G] = S_G$ and $S[H] = S_H$. However, part III shows an $S' \in D(G \oplus H)$ with $S' \neq S$, but $S'[G] = S_G$ and $S'[H] = S_H$. Hence with this definition of $[]$, we are not able to combine two partial solutions of two terminal graphs G and H into a solution of the graph $G \oplus H$.

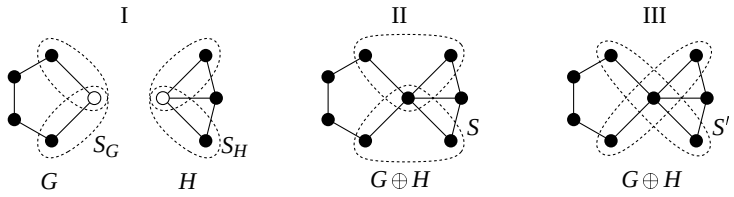


Figure 6.2. Example of two one-terminal graphs G and H with partial solutions S_G and S_H, and the graph $G \oplus H$ with $S, S' \in D(G \oplus H)$, such that $S[G] = S'[G] = S_G$ and $S[H] = S'[H] = S_H$.

Let P be a construction property defined by (D, Q) such that D is inducible for $[]$. Let G and H be terminals graphs, and let $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$. The value of $Q(G \oplus H, S \oplus S')$ is only defined if

- G and H are both l-terminal graphs for some $l \geq 0$, and
- (G, S) and (H, S') are $\oplus$-compatible.

For shorter notation, we define $Q(G \oplus H, S \oplus S')$ to be false if G and H are not both l-terminal graphs for some $l \geq 0$, or if (G, S) and (H, S') are not $\oplus$-compatible.

Definition 6.1.5 (Compatibility). Let D be a solution domain which is inducible for some definition of $[]$. Let G_1 and G_2 be l-terminal graphs for some $l \geq 0$, and let $S_1 \in D_{[]}(G_1)$ and $S_2 \in D_{[]}(G_2)$. (G_1, S_1) and (G_2, S_2) are *compatible* if for each l-terminal graph H and each $S \in D_{[]}(H)$, (G_1, S_1) is $\oplus$-compatible with (H, S) if and only if (G_2, S_2) is $\oplus$-compatible with (H, S).

Note that compatibility is an equivalence relation. The set of all equivalence classes of this relation is denoted by $C_{\text{cmp},l}$, for each l, and the equivalence classes are also called compatibility classes. For two equivalence classes C and C' of some equivalence relation which is

a refinement of compatibility, we say that C and C' are $\oplus$-compatible if, for each $(G,S) \in C$ and $(H,S') \in C'$, (G,S) and (H,S') are $\oplus$-compatible.

Let P be a construction property defined by (D,Q), where D is inducible for $[]$.

Definition 6.1.6. For each $l \geq 0$, $\sim_{Q,l}$ is an equivalence relation on pairs of l-terminal graphs and partial solutions, which is defined as follows. Let G_1, G_2 be l-terminal graphs, and $S_1 \in D_{[]}(G_1)$ and $S_2 \in D_{[]}(G_2)$.

$$(G_1,S_1) \quad \sim_{Q,l} \quad (G_2,S_2) \qquad \Leftrightarrow \qquad \begin{array}{l}(G_1,S_1) \text{ and } (G_2,S_2) \text{ are compatible and} \\ \text{for all } l\text{-terminal graphs } H \text{ and all } S \in D_{[]}(H): \\ \quad Q(G_1 \oplus H, S_1 \oplus S) = Q(G_2 \oplus H, S_2 \oplus S)\end{array}$$

The set of equivalence classes of $\sim_{Q,l}$ is denoted by $C_{Q,l}$, and for each l-terminal graph G and $S \in D_{[]}(G)$, the equivalence class of $C_{Q,l}$ that contains (G,S) is denoted by $\mathrm{ec}_{Q,l}(G,S)$.

By $\sim_{rQ,l}$, we usually denote an equivalence relation which is a refinement of $\sim_{Q,l}$. By $C_{rQ,l}$ we denote the set of equivalence classes of $\sim_{rQ,l}$, and for each l-terminal graph G and each $S \in D_{[]}(G)$, $\mathrm{ec}_{rQ,l}(G,S) = C$ if (G,S) is in equivalence class $C \in C_{rQ,l}$.

Definition 6.1.7. Let $\sim_{rQ,l}$ be a refinement of $\sim_{Q,l}$ for each $l \geq 0$. By $\approx_{rQ,l}$ we denote the equivalence relation on l-terminal graphs, which is defined as follows. For every two l-terminal graphs G_1 and G_2,

$$G_1 \approx_{rQ,l} G_2 \quad \Leftrightarrow \quad \{\, \mathrm{ec}_{rQ,l}(G_1,S_1) \mid S_1 \in D_{[]}(G_1) \,\} = \{\, \mathrm{ec}_{rQ,l}(G_2,S_2) \mid S_2 \in D_{[]}(G_2) \,\}.$$

Recall that for a graph property P and an integer $k \geq 1$, P_k denotes the property with for each graph G, $P_k(G)$ holds if and only if $P(G) \wedge TW_k(G)$ holds. Suppose P is a construction property defined by (D,Q). For each $k \geq 1$, let Q_k denote the property with for each graph G, each $S \in D(G)$, $Q_k(G,S)$ holds if and only if $Q(G,S) \wedge TW_k(G)$ holds. Note that P_k is the construction property defined by (D,Q_k).

For each $k \geq 1$, let $\sim_{rQk,l}$ be the refinement of $\sim_{rQ,l}$ which is defined as follows. For every two l-terminal graphs G_1 and G_2 and each $S_1 \in D_{[]}(G_1)$ and $S_2 \in D_{[]}(G_2)$,

$$(G_1,S_1) \sim_{rQk,l} (G_2,S_2) \quad \Leftrightarrow \quad (G_1,S_1) \sim_{rQ,l} (G_2,S_2) \wedge G_1 \sim_{TW_k,l} G_2.$$

Lemma 6.1.1. *Let $\sim_{rQ,l}$ be a refinement of $\sim_{Q,l}$, and let $k \geq 1$.*

1. *For each $l \geq 0$, $\approx_{rQ,l}$ is a refinement of $\approx_{Q,l}$.*
2. *For each $l \geq 0$, $\approx_{Q,l}$ is a refinement of $\sim_{P,l}$.*
3. *For each $l \geq 0$, if $\sim_{rQ,l}$ is finite, then $\approx_{rQ,l}$ is finite.*
4. *For each $l \geq 0$, if $\sim_{rQ,l}$ is finite, then $\approx_{rQk,l}$ is finite.*

Proof.

1. Follows directly from the definition of $\approx_{rQ,l}$.

2. Follows from the fact that for every two l-terminal graphs G_1 and G_2, if $G_1 \approx_{Q,l} G_2$, then for each $S_1 \in D_{[]}(G_1)$ there is an $S_2 \in D_{[]}(G_2)$ such that $(G_1,S_1) \sim_{Q,l} (G_2,S_2)$.

3. The number of equivalence classes of $\approx_{rQ,l}$ is at most $2^{|C_{rQ,l}|}$.

4. Follows from Lemmas 5.1.7 and 5.1.6. □

Note that for $|C_{rQ,l}|$ to be finite, $|C_{Q,l}|$ must be finite, and hence also $|C_{\mathrm{cmp},l}|$ must be finite.

The next theorem is the analog of Theorem 5.1.2 for construction properties: we give a set of conditions for a construction property P, and we show that these conditions are sufficient for proving the existence of a special constructive reduction system for P_k for any $k \geq 1$.

Theorem 6.1.2. *Let P a construction property defined by (D,Q), suppose that a proper definition of $[]$ is given and that the following conditions hold.*

1. *Domain D is inducible for $[]$, Q is decidable, a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ is decidable and $|C_{rQ,l}|$ is finite, for fixed $l \geq 0$.*
2. *There is a representation of (partial) solutions for which the following holds.*
   (a) *There is a function s which assigns to each terminal graph G a positive integer, such that for each $S \in D_{[]}(G)$, the number of bits needed to represent S is at most $s(G)$.*
   (b) *For each two fixed l-terminal graphs H and H', the following holds. For each l-terminal graph G, if $S \in D(H \oplus G)$, then $S[H]$ can be computed from S and H in constant time, and for each $S' \in D_{[]}(H')$, if $(H,S[H]) \sim_{rQ,l} (H',S')$, then $S' \oplus S[G]$ can be computed in constant time from S, S', H and H'.*

Then for each $k \geq 1$, there exists a special constructive reduction system (R,I,A_R,A_I) for P_k defined by (D,Q_k), such that for each $(H_1,H_2) \in R$, $H_1 \approx_{rQ,l} H_2$.

If, in addition, (i) Q and $\sim_{rQ,l}$ are effectively decidable, (ii) s is effectively computable, and (iii) in condition 2b $S[H]$ and $S' \oplus S[G]$ are effectively computable from S, S', H and H', then such a special constructive reduction system can effectively be constructed.

Proof. Let $k \geq 1$. Since $|C_{rQ,l}|$ is finite, $\approx_{rQk,l}$ has a finite number of equivalence classes, and it is a refinement of $\sim_{P_k,l}$. Let (R,I) be a special reduction system for P_k, such that for each rule $(H_1,H_2) \in R$, $H_1 \approx_{rQk,l} H_2$. By Corollary 5.1.1, such a system exists, and it can be constructed if s is effectively computable and $\sim_{rQ,l}$ is effectively decidable, since in that case, P and $\approx_{rQk,l}$ are effectively decidable as well. We now show how to construct algorithms A_R and A_I such that (R,I,A_R,A_I) is a special constructive reduction system for P_k.

Both algorithm A_R and A_I use a table (see also the example for two-colorability on page 123). For algorithm A_R, we make a table which contains for each rule $r = (H_1,H_2) \in R$ and each $S_2 \in D_{[]}(H_2)$ an $S_1 \in D_{[]}(H_1)$ such that $(H_1,S_1) \sim_{rQ,l} (H_2,S_2)$. This table is computed as follows. For each reduction rule (H_1,H_2) in R, we construct all $S_1 \in D_{[]}(H_1)$ and all $S_2 \in D_{[]}(H_2)$. Then, for each $S_2 \in D_{[]}(H_2)$, we pick one $S_1 \in D_{[]}(H_1)$ for which

$(H_1,S_1) \sim_{rQ,l} (H_2,S_2)$. By condition 2a, this table has finite size. Condition 2b assures that this table can be used to make algorithm A_R run in constant time.

For algorithm A_I, we make a table which contains for each $H \in I$ a solution S of H. This is done as follows. For each $H \in I$, we construct all $S \in D(H)$, and we pick one such S for which $Q(H,S)$ holds. It is easy to see that algorithm A_I can be made to run in $O(1)$ time using this table.

Note that if $\sim_{rQ,l}$ is effectively decidable and s is effectively computable, then we can effectively construct the tables for algorithms A_R and A_I. Condition 2 assures that the system (R, I, A_R, A_I) is a special constructive reduction system. □

As an important special case, we now consider the MS-definable construction properties. Let D be a solution domain of the following form: there is a $t \geq 1$, such that for all graphs G, all elements of $D(G)$ are t-tuples $(S_1,S_2,\ldots,S_t)$, where for each i, $1 \leq i \leq t$, S_i is an element of $V(G)$, of $E(G)$, of $P(V(G))$ or of $P(E(G))$. If D is of this form, we say that D is a *t-vertex-edge-tuple*. An example of a domain which is a t-vertex-edge-tuple is the domain D for which for each graph G, $D(G)$ contains all ordered t-partitions of $V(G)$, i.e. for each $S \in D(G)$, $S = (V_1,\ldots,V_t)$, where $V_1,\ldots,V_t$ partition $V(G)$. The construction properties defined by (D,Q), where D is a t-vertex-edge tuple and Q is an MS-definable extended graph property correspond exactly to the MS-definable construction problems as defined in Section 2.2.4.

Theorem 6.1.3. *Let P be a construction property defined by (D,Q), where D is a t-vertex-edge tuple and Q is MS-definable. For each $k \geq 1$ there is a special constructive reduction system for P_k, which can be effectively constructed.*

Proof. For each two l-terminal graphs G_1 and G_2, $G = G_1 \oplus G_2$, each $S = (S_1,\ldots,S_t) \in D(G)$, let $S[G_1] = (S_1[G_1],\ldots,S_t[G_1])$, where for each i, $S_i[G_1]$ is defined as follows.

$$S_i[G_1] = \begin{cases} S_i \cap V(G_1) & \text{if } S_i \in P(V(G)) \\ S_i \cap E(G_1) & \text{if } S_i \in P(E(G)) \\ S_i & \text{if } (S_i \in V(G) \wedge S_i \in V(G_1)) \vee (S_i \in E(G) \wedge S_i \in E(G_1)) \\ \varepsilon & \text{if } (S_i \in V(G) \wedge S_i \notin V(G_1)) \vee (S_i \in E(G) \wedge S_i \notin E(G_1)) \end{cases}$$

With this definition of $[\,]$, D is inducible, and $|C_{\mathrm{cmp},l}|$ is finite, for each $l \geq 0$.

Borie, Parker, and Tovey [1992] have shown that for each $k \geq 1$, there is a homomorphism h, mapping each pair (G,S), where either G is an ordinary graph and $S \in D(G)$ or G is an l-terminal graph, $l \leq k$, and $S \in D_{[]}(G)$, to an element of a finite set A_k, such that the following conditions hold.

1. For every two graphs G_1 and G_2, and each $S_1 \in D(G_1)$ and $S_2 \in D(G_2)$, if $h(G_1,S_1) = h(G_2,S_2)$, then $Q(G_1,S_1) = Q(G_2,S_2)$.
2. There is a function $f_\oplus : A_k \times A_k \to A_k$, such that for each $l \leq k$, every two l-terminal graphs G and H, and each $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$, if (G,S) and (H,S') are $\oplus$-compatible, then

$$h(G \oplus H, S \oplus S') = f_\oplus(h(G,S),h(H,S')).$$

This homomorphism can be computed from an MSOL predicate for Q.

For each $l \geq 0$, each l-terminal graph G and $S \in D_{[]}(G)$, let $\mathrm{ec}_l(G,S) = (h(G,S),C)$, where $C \in C_{\mathrm{cmp},l}$ is such that (G,S) belongs to compatibility class C. Furthermore, let $C_l = A_k \times C_{\mathrm{cmp},l}$, and let $(G_1,S_1) \sim_l (G_2,S_2)$ if and only if $\mathrm{ec}_l(G_1,S_1) = \mathrm{ec}_l(G_2,S_2)$. Since $|A_k|$ and $|C_{\mathrm{cmp},l}|$ are both finite, $|C_l|$ is also finite. We now show that $\sim_l$ is a refinement of $\sim_{Q,l}$.

Let $l \geq 0$, let G_1 and G_2 be l-terminal graphs, and let $S_1 \in D_{[]}(G_1)$ and $S_2 \in D_{[]}(G_2)$, such that $(G_1,S_1) \sim_l (G_2,S_2)$. We have to show that for all l-terminal graphs H and all $S \in D_{[]}(H)$, $Q(G_1 \oplus H, S_1 \oplus S) = Q(G_2 \oplus H, S_2 \oplus S)$. Let H be an l-terminal graph, and let $S \in D_{[]}(H)$ such that (G_1,S_1) and (H,S) are $\oplus$-compatible. Then, since $h(G_1,S_1) = h(G_2,S_2)$,

$$\begin{aligned} h(G_1 \oplus H, S_1 \oplus S) &= f_{\oplus}(h(G_1,S_1),h(H,S)) \\ &= f_{\oplus}(h(G_2,S_2),h(H,S)) \\ &= h(G_2 \oplus H, S_2 \oplus S). \end{aligned}$$

Hence $Q(G_1 \oplus H, S_1 \oplus S) = Q(G_2 \oplus H, S_2 \oplus S)$. This shows that condition 1 of Theorem 6.1.2 holds.

Now consider condition 2. We use a data structure for storing tuples $S = (S_1,\ldots,S_t) \in D_{[]}(G)$ which consists of an array of t data structures, one for each S_i. If S_i is a set of vertices or edges, then these vertices or edges are put in a (doubly linked) list. If S_i is a vertex or edge, or ε, then this vertex or edge or ε is stored. Furthermore, we keep a pointer from each vertex and edge in the graph to each place in the data structure where this vertex or edge occurs. There are at most t of these pointers for each vertex and each edge. This shows that condition 2a of Theorem 6.1.2 is satisfied.

Consider condition 2b of Theorem 6.1.2. For every two fixed l-terminal graphs H and H' and each l-terminal graph G, if we have an $S \in D(G \oplus H)$ stored in the way described above, then we can compute $S[H]$ as follows. Make a new data structure for $S[H]$ with $S_i[H]$ empty for each i. For each vertex v in H, follow the pointers from v to the places in which it occurs in S, and check in which part S_i of S it occurs. Then add v to $S_i[H]$. Do the same for all edges. Then for each i, check if S_i is a set of vertices or edges, but there is no vertex or edge in the data structure at the location of $S_i[H]$, and if so, add ε to $S_i[H]$. This can all be done in constant time, since H has constant size, and each vertex or edge occurs at most once in each S_i, so at most t times in S.

Let $S' = (S'_1,\ldots,S'_t) \in D_{[]}(H')$ such that $(H,S) \sim_{rQ,l} (H',S')$. $S' \oplus S[G]$ can be computed as follows. For each vertex v of H which is not a terminal, follow the pointers from v to all places in S where it occurs, and delete it there. Do the same for all edges in H for which at least one end point is not a terminal. For each vertex v of H' which is a terminal, follow the pointers from v to all pointers in S' where it occurs, and delete v at that place. Do the same for all edges in H' of which both end points are terminals. Next, for each i, $1 \leq i \leq t$, append the list S'_i to the list S_i. The resulting data structure represents $S' \oplus S[G]$. Hence condition 2b of Theorem 6.1.2 holds. □

Theorem 6.1.3 implies that for each MS-definable construction property, there is a linear time algorithm which solves P on graphs of bounded treewidth, without making use of a

tree decomposition of the input graph. For instance, this gives linear time algorithms for the constructive versions of e.g. HAMILTONIAN CIRCUIT and k-COLORABILITY for fixed k, all on graphs of bounded treewidth.

As a corollary, we also have the following result, which may be easier to use than Theorem 6.1.2.

Corollary 6.1.1. *Let P be a construction property defined by (D,Q). If D is a t-vertex-edge-tuple for some $t \geq 1$, and furthermore Q is decidable, and a finite refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ is decidable, then for each $k \geq 1$, there is a special constructive reduction system for P_k.*

If, in addition, Q and $\sim_{rQ,l}$ are effectively decidable, then such a system can be effectively constructed.

6.2 Optimization Problems

In this section we show how the idea of constructive reduction algorithms can be extended to constructive optimization problems. We start with a definition of a constructive reduction-counter system and an efficient reduction algorithm for constructive optimization problems. After that, we show that this algorithm can be used to solve a large class of constructive optimization problems on graphs of bounded treewidth.

6.2.1 Constructive Reduction-Counter Systems and Algorithms

Many graph optimization problems are of the form

$$\Phi(G) = \text{opt}\{z(S) \mid S \in D(G) \wedge Q(G,S)\},$$

where D is a solution domain, for each $S \in D(G)$, z is a function from $D(G)$ to $\mathbb{Z}$, and either opt $=$ max or opt $=$ min. (If there is no $S \in D(G)$ for which $Q(G,S)$ holds, then we define $\Phi(G)$ to be false.) If Φ is of this form, then we say Φ is a constructive optimization problem defined by the quadruple (D,Q,z,opt). MAX INDEPENDENT SET is an example of such an optimization problem: for this problem, we can choose opt $=$ max, $D(G) = P(V(G))$, $Q(G,S)$ holds if and only if for each $v,w \in S$, $\{v,w\} \notin E(G)$, and $z(S) = |S|$.

In this section, we consider reduction algorithms for constructive optimization problems Φ which return the value of $\Phi(G)$ for an input graph G, and also construct an optimal solution for G, i.e. a solution $S \in D(G)$ for which $Q(G,S)$ holds and $z(S) = \Phi(G)$ (if $\Phi(G) \neq$ false). We first define the constructive version of a reduction-counter system.

Definition 6.2.1 (Constructive Reduction-Counter System). Let Φ be a constructive optimization problem defined by (D,Q,z,opt). A *constructive reduction-counter system* for Φ is a quadruple (R,I,ϕ,A_R,A_I), where

- (R,I,ϕ) is a reduction-counter system for Φ (Definition 5.2.3),
- A_R is an algorithm which, given
  - a reduction rule $r = (H_1,H_2) \in R$,

- two terminal graphs G_1 and G_2, such that G_1 is a isomorphic to H_1 and G_2 is isomorphic to H_2,
- a graph G with $G = G_2 \oplus H$ for some H, and an $S \in D(G)$ such that $Q(G,S)$ holds and $z(S) = \Phi(G)$,

computes an $S' \in G_1 \oplus H$ such that $Q(G_1 \oplus H, S')$ holds and $z(S') = \Phi(G_1 \oplus H)$,

- A_I is an algorithm which, given a graph G which is isomorphic to some $H \in I$, computes an $S \in D(G)$ for which $Q(G,S)$ holds and $z(S) = \Phi(G)$.

As an example, consider the optimization problem Φ defined as follows. For each graph G, $\Phi(G)$ is the maximum size of an independent set if G is a cycle, $\Phi(G) = \mathsf{false}$ otherwise (see Chapter 5 on page 109). Consider the constructive version of Φ defined by $(D,Q,z,\max)$, where D, Q and z are defined as follows. For each graph G, $D(G) = P(V(G))$, and for each $S \in D(G)$, $Q(G,S)$ holds if and only if G is a cycle and S is an independent set of G, and $z(S) = |S|$.

We extend the reduction-counter system for Φ depicted in Figure 5.4 to a constructive reduction-counter system for Φ. Therefore, we again use the table method. For algorithm A_R, we make a table which contains the following information. For the only reduction rule $r = (H_1,H_2) \in R$ and each independent set S_2 of H_2 for which there is a maximum independent set S in some graph $H_2 \oplus H$ with $S_2 = S \cap V(H_2)$, the table contains an independent set S_1 of H_1 such that S_1 and S_2 contain the same terminals and $|S_1| = |S_2| + 1$. All these cases are depicted in part I of Figure 6.3 (symmetric cases are given only once). Note that algorithm A_R can be made to run in $O(1)$ time with this table, since it only has to remove inner vertices of H_2 from the independent set of the old graph and add some inner vertices of H_1 to the independent set of the new graph.

For algorithm A_I, we make a table which contains for each $H \in I$ a maximum independent set of H (see part II of Figure 6.3). Algorithm A_I also uses $O(1)$ time. It can be seen that (R, I, ϕ, A_R, A_I) is a constructive reduction-counter system for Φ defined by $(D,Q,z,\max)$.

Let Φ be a constructive optimization problem defined by (D,Q,z,opt). Let P be the construction property defined by (D,Q). We call P the *derived construction property*. From a constructive reduction-counter system (R, I, ϕ, A_R, A_I) for Φ we can derive a constructive reduction system S for P: let $R' = \{ r \mid (r,i) \in R \}$, and let $S = (R', I, A_R, A_I)$. We call S the *derived constructive reduction system*.

Definition 6.2.2 (Special Constructive Reduction-Counter System). A special constructive reduction-counter system is a constructive reduction-counter system whose derived constructive reduction system is special.

Note that the constructive reduction-counter system that we gave for MAX INDEPENDENT SET on cycles is special.

Let Φ be a constructive optimization problem defined by (D,Q,z,opt), such that D is inducible for a given definition of $[]$. Let $S = (R, I, \phi, A_R, A_I)$ be a special constructive reduction-counter system for Φ. We can modify algorithm Reduce-Construct (page 124) to

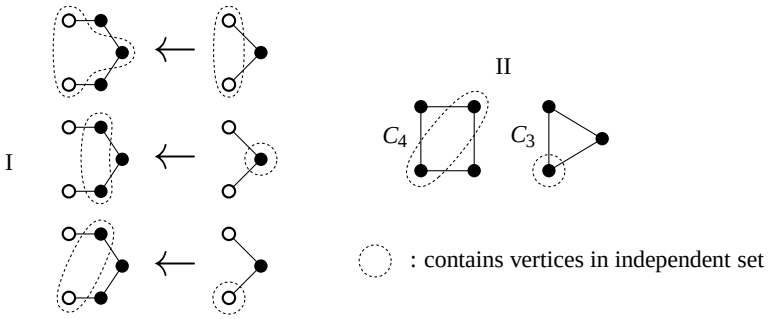


Figure 6.3. Example of tables for algorithms A_R and A_I for constructive reduction-counter system for MAX INDEPENDENT SET on cycles.

obtain a constructive reduction algorithm for Φ based on S: in part 1, use the reduction-counter algorithm as described in Section 5.2 (page 109) instead of algorithm Reduce. In Part 2, line 6 of algorithm Reduce-Construct, store the value $\phi(G)$ in some variable *opt*. In line 13, return with S the value *opt*.

Hence we have the following result.

Theorem 6.2.1. *Let Φ be a constructive optimization problem defined by (D,Q,z,opt). If we have a special constructive reduction-counter system for Φ then we have an algorithm which, given any graph G, computes $\Phi(G)$ and, if $\Phi(G) \neq \text{false}$, computes an $S \in D(G)$ such that $Q(G,S)$ holds and $z(S) = \Phi(G)$. The algorithm uses $O(n)$ time and space.*

6.2.2 Constructive Optimization Problems for Graphs of Bounded Treewidth

In this section we show that for a large class of constructive optimization problems on graphs of bounded treewidth, there exists a special constructive reduction-counter system.

Let (D,Q,z,opt) define a constructive optimization problem Φ and suppose D is inducible for $[]$. For each $l \geq 0$, let $\sim_{rQ,l}$ be a refinement of $\sim_{Q,l}$.

Let G be a terminal graph. We want to be able to compare the quality of two partial solutions S and S' for which $(G,S) \sim_{rQ,l} (G,S')$, i.e. we want that one of them is always at least as good as the other one. More formally, we want that either

- for each terminal graph H and each $S_H \in D_{[]}(H)$ for which $Q(G \oplus H, S \oplus S_H)$ holds, $z(S \oplus S_H) \geq z(S' \oplus S_H)$, or
- for each terminal graph H and each $S_H \in D_{[]}(H)$ for which $Q(G \oplus H, S \oplus S_H)$ holds, $z(S' \oplus S_H) \geq z(S \oplus S_H)$.

Therefore, we define an extension of the function z to the domain of terminal graphs.

Definition 6.2.3. Let $\overline{z}$ be a function which, for each terminal graph G and each $S \in D_{[]}(G)$, maps S to a value in $\mathbb{Z}$. Function $\overline{z}$ is an *extension* of z with respect to $\{\sim_{rQ,l} | \; l \geq 0\}$ if, for each $l \geq 0$, and each $C, C' \in C_{rQ,l}$ for which C and C' are $\oplus$-compatible, there is a constant $d_l(C,C') \in \mathbb{Z}$ such that the following holds. For every two l-terminal graphs G and H and all $S_G \in D_{[]}(G)$ and $S_H \in D_{[]}(H)$ such that $\mathrm{ec}_{rQ,l}(G,S_G) = C$ and $\mathrm{ec}_{rQ,l}(H,S_H) = C'$,

$$Q(G \oplus H, S_G \oplus S_H) \quad \Rightarrow \quad z(S_G \oplus S_H) = \overline{z}(S_G) + \overline{z}(S_H) \Leftrightarrow d_l(C,C')$$

The constants $d_l(C,C')$ are called the *extension constants* for $\overline{z}$.

Note that, if there is a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ for each $l \geq 0$ and there is an extension $\overline{z}$ of z with respect to $\{\sim_{rQ,l} | \; l \geq 0\}$, then it is not necessarily the case that $\overline{z}$ is an extension of z with respect to $\{\sim_{Q,l} | \; l \geq 0\}$. However, $\overline{z}$ is an extension for z with respect to any refinement of $\sim_{rQ,l}$.

Lemma 6.2.1. *Suppose $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} | \; l \geq 0\}$. Let G be an l-terminal graph ($l \geq 0$). Let $S, S' \in D_{[]}(G)$ such that $(G,S) \sim_{rQ,l} (G,S')$. If $\overline{z}(S) \geq \overline{z}(S')$ (or $\overline{z}(S) > \overline{z}(S')$), then for each terminal graph H and each $S_H \in D_{[]}(H)$, if $Q(G \oplus H, S \oplus S_H)$ holds, then $z(S \oplus S_H) \geq z(S' \oplus S_H)$ (or $z(S \oplus S_H) > z(S' \oplus S_H)$, respectively).*

Proof. Let $C = \mathrm{ec}_{rQ,l}$ and let d_l denote the extension constants for $\overline{z}$. Suppose $\overline{z}(S) \geq \overline{z}(S')$ and let H be a terminal graph, $S_H \in D_{[]}(H)$ such that $Q(G \oplus H, S \oplus S_H)$ holds. Let $C' = \mathrm{ec}_{rQ,l}(H, S_H)$. Then $Q(G \oplus H, S' \oplus S_H)$ also holds. Furthermore, $z(S \oplus S_H) = \overline{z}(S) + \overline{z}(S_H) \Leftrightarrow d_l(C,C') \geq \overline{z}(S') + \overline{z}(S_H) \Leftrightarrow d_l(C,C') = z(S' \oplus S_H)$. For $>$ the proof is similar. □

In other words, Lemma 6.2.1 shows that if $\overline{z}(S) > \overline{z}(S')$, then S is better than S' if $\mathrm{opt} = \max$, and S' is better than S if $\mathrm{opt} = \min$.

Let G be an l-terminal graph, and $C \in C_{rQ,l}$. Let

$$\mathrm{opt}(G,C) \quad = \quad \mathrm{opt}\{\overline{z}(S) \mid S \in D_{[]}(G) \wedge \mathrm{ec}_{rQ,l}(G,S) = C \wedge \overline{z}(S) \in \mathbb{Z}\}$$

(hence $\mathrm{opt}(G,C) = \mathsf{false}$ if there is no $S \in D_{[]}(G)$ for which $\mathrm{ec}_{rQ,l}(G,S) = C$). If $\mathrm{opt}(G,C) \in \mathbb{Z}$, then let $\mathrm{optS}(G,C)$ denote an $S \in D_{[]}(G)$ for which $\overline{z}(S) = \mathrm{opt}(G,C)$. Informally speaking, $\mathrm{opt}(G,C)$ represents 'the value of the best partial solution of G in equivalence class C', and $\mathrm{optS}(G,C)$ gives such a partial solution (if existing).

Let $S \in D_{[]}(G)$, let $C = \mathrm{ec}_{rQ,l}(G,S)$ and suppose S may lead to an optimal solution, i.e. there is a terminal graph H and an $S_H \in D_{[]}(H)$ such that $Q(G \oplus H, S \oplus S_H)$ holds and $z(S \oplus S_H) = \Phi(G \oplus H)$. Lemma 6.2.1 shows that $\overline{z}(S) = \mathrm{opt}(G,C)$. Hence only partial solutions S for which $\overline{z}(S) = \mathrm{opt}(G, \mathrm{ec}_{rQ,l}(G,S))$ may lead to optimal solutions.

Theorem 6.2.2. *Let Φ be a constructive optimization problem defined by (D,Q,z,opt). Suppose D is inducible for $[]$ and there is a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ for which the following conditions hold.*

1. *Q is decidable, for each $l \geq 0$, $\sim_{rQ,l}$ is decidable and $|C_{rQ,l}|$ is finite.*

2. *There is an extension $\overline{z}$ of z with respect to $\{\sim_{rQ,l} | \; l \geq 0\}$ and for each $l \geq 0$, there is a constant $K_l \in \mathbb{N}$, such that for each l-terminal graph G and every $S, S' \in D_{[]}(G)$, if both S and S' can lead to optimal solutions, then $|\overline{z}(S) \Leftrightarrow \overline{z}(S')| \leq K_l$.*

3. *There is a representation of (partial) solutions for which the following holds.*

   (a) *There is a function s, which assigns to each terminal graph G a positive integer, such that for each $S \in D_{[]}(G)$, the number of bits needed to represent S is at most $s(G)$.*

   (b) *For each two fixed l-terminal graphs H and H', the following holds. For each l-terminal graph G, if $S \in D(G \oplus H)$, then $S[H]$ can be computed from S and H in constant time, and for each $S' \in D_{[]}(H')$, if $(H, S[H]) \sim_{rQ,l} (H', S')$, then $S' \oplus S[G]$ can be computed in constant time from S, S', H, and H'.*

Then for each $k \geq 1$, there exists a special constructive reduction-counter system S for Φ_k defined by (D, Q_k, z, opt), and for each reduction-counter rule $((H_1, H_2), i)$ in S, $H_1 \approx_{rQ,l} H_2$.

If, in addition, (i) Q and $\sim_{rQ,l}$ are effectively decidable, (ii) z is effectively computable, (iii) in condition 2, $\overline{z}$ and K_l are effectively computable, and (iv) in condition 3, s is effectively computable, and $S[H]$ and $S' \oplus S[G]$ are effectively computable from S, S', H and H', then such a special constructive reduction-counter system can be effectively constructed.

Proof. Suppose conditions 1, 2, and 3 hold for Φ. Let $\overline{z}$ be the extension of condition 2 and let $d_l(C, C')$ denote the corresponding extension constants for all $C, C' \in C_{rQ,l}$. For each $l \geq 0$, let $K_l \in \mathbb{N}$ be as in condition 2. Let s be as in condition 3. Let P be the construction property derived from Φ (i.e. P is defined by (D, Q)).

We first construct a refinement $\sim_l$ of $\sim_{rQ,l}$ such that for each pair (G_1, G_2) of l-terminal graphs, if $|V(G_2)| < |V(G_1)|$ and $G_1 \approx_l G_2$, then there is an $i \in \mathbb{Z}$ for which the following holds.

1. $((G_1, G_2), i)$ is a safe reduction-counter rule for Φ, and
2. for each $S_2 \in D_{[]}(G_2)$ which can lead to an optimal solution, there is an $S_1 \in D_{[]}(G_1)$ such that $(G_1, S_1) \sim_l (G_2, S_2)$ and for each l-terminal graph H and each $S \in D_{[]}(H)$, if $Q(G_2 \oplus H, S_2 \oplus S)$ holds and $z(S_2 \oplus S) = \Phi(G_2 \oplus H)$, then $Q(G_1 \oplus H, S_1 \oplus S)$ holds, and $z(S_1 \oplus S) = \Phi(G_1 \oplus H)$.

We also show that $\sim_l$ is finite. After that, we show how to use $\sim_l$ to build a special constructive reduction-counter system for Φ_k ($k \geq 1$).

For each $l \geq 0$, each l-terminal graph G, do the following. If there is a partial solution in G which can lead to an optimal solution, then let $\tilde{S}_G \in D_{[]}(G)$ such that $\tilde{S}_G$ can lead to an optimal solution. Let $i_G = \overline{z}(\tilde{S}_G)$ (note that $i_G \in \mathbb{Z}$). Otherwise, $\tilde{S}_G$ is not defined and $i_G = 0$. Let $h_G : C_{rQ,l} \to \{\Leftrightarrow K_l, \ldots, K_l\} \cup \{\mathsf{false}\}$ be a function with for each $C \in C_{rQ,l}$,

$$h_G(C) = \begin{cases} \text{opt}(G, C) \Leftrightarrow i_G & \text{if } |\text{opt}(G, C) \Leftrightarrow i_G| \leq K_l \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

For each $l \geq 0$, each pair G_1, G_2 of l-terminal graphs and each $S_1 \in D_{[]}(G_1)$ and $S_2 \in D_{[]}(G_2)$, let

$$(G_1,S_1) \sim_l (G_2,S_2) \;\Leftrightarrow\; (G_1,S_1) \sim_{rQ,l} (G_2,S_2) \wedge h_{G_1}(ec_{rQ,l}(G_1,S_1)) = h_{G_2}(ec_{rQ,l}(G_2,S_2)).$$

Note that $\sim_l$ is a refinement of $\sim_{rQ,l}$ and hence of $\sim_{Q,l}$. For each $l \geq 0$, the range of h_G for any l-terminal graph G has finite cardinality, and $\sim_{rQ,l}$ is finite, which means that $\sim_l$ is also finite.

Consider the equivalence relation $\approx_l$ on l-terminal graphs as defined in Definition 6.1.7. Let $l \geq 0$, let G_1 and G_2 be l-terminal graphs, such that $|V(G_2)| < |V(G_1)|$ and $G_1 \approx_l G_2$. By definition of $\sim_l$ and $\approx_l$, $h_{G_1} = h_{G_2}$. Let $i = i_{G_1} \Leftrightarrow i_{G_2}$, and let $h = h_{G_1} = h_{G_2}$. We show that G_1, G_2 and i satisfy conditions 1 and 2 given above.

Note that, if there is an $S \in D_{[]}(G_1)$ which can lead to a solution, then there is an $S' \in D_{[]}(G_2)$ which can lead to a solution, and vice versa.

Claim. *Suppose there is a partial solution in G_1 which can lead to a solution. Let $C \in C_{rQ,l}$ such that* $\mathrm{opt}(G_1,C) \in \mathbb{Z}$. *Let H be an l-terminal graph. Let $S_1 = \mathrm{optS}(G_1,C)$, $S_2 = \mathrm{optS}(G_2,C)$ and $S_H \in D_{[]}(H)$, and suppose $Q(G_1 \oplus H, S_1 \oplus S_H)$ holds. Then $z(S_1 \oplus S_H) = z(S_2 \oplus S_H) + i$.*

Proof. As there is a partial solution in G_1 which can lead to a solution, $\tilde{S}_{G_1}$ is defined and $\overline{z}(\tilde{S}_{G_1}) = i_{G_1}$. This also means that $\tilde{S}_{G_2}$ is defined and $\overline{z}(\tilde{S}_{G_2}) = i_{G_2}$. Hence, by condition 2 of the theorem, $|\overline{z}(S_1) \Leftrightarrow i_{G_1}| \leq K_l$, so $\overline{z}(S_1) = i_{G_1} + h(C)$, and similarly, $\overline{z}(S_2) = i_{G_2} + h(C)$. Furthermore,

$$\begin{aligned} z(S_1 \oplus S_H) &= \overline{z}(S_1) + \overline{z}(S_H) \Leftrightarrow d_l(C,C') \\ &= h(C) + i_{G_1} + \overline{z}(S_H) \Leftrightarrow d_l(C,C') \\ &= h(C) + i_{G_2} \Leftrightarrow i_{G_2} + i_{G_1} + \overline{z}(S_H) \Leftrightarrow d_l(C,C') \\ &= \overline{z}(S_2) + \overline{z}(S_H) \Leftrightarrow d_l(C,C') \Leftrightarrow i_{G_2} + i_{G_1} \\ &= z(S_2 \oplus S_H) \Leftrightarrow i_{G_2} + i_{G_1} \\ &= z(S_2 \oplus S_H) + i. \end{aligned}$$

□

Claim. $((G_1,G_2),i)$ *is safe for* Φ.

Proof. Let H be an l-terminal graph. We have to show that $\Phi(G_1 \oplus H) = \Phi(G_2 \oplus H) + i$. Since $G_1 \approx_l G_2$, and $\approx_l$ is a refinement of $\approx_{Q,l}$, which in turn is a refinement of $\sim_{P,l}$, $\Phi(G_1 \oplus H)$ is false if and only if $\Phi(G_2 \oplus H)$ is false. Hence if $\Phi(G_1 \oplus H) = \mathsf{false}$, then $\Phi(G_1 \oplus H) = \Phi(G_2 \oplus H) + i$.

Now suppose $\Phi(G_1 \oplus H) \in \mathbb{Z}$, and let $S \in D(G_1 \oplus H)$ such that $z(S) = \Phi(G_1 \oplus H)$. let $S_1 = S[G_1]$ and $S_H = S[H]$. Let $S_2 = \mathrm{optS}(G_2, \mathrm{ec}_{rQ,l}(G_1,S_1))$. By the previous claim,

$z(S_1 \oplus S_H) = z(S_2 \oplus S_H) + i$, and hence if opt = max, then $\Phi(G_1 \oplus H) \leq \Phi(G_2 \oplus H) + i$, and if opt = min, then $\Phi(G_1 \oplus H) \geq \Phi(G_2 \oplus H) + i$. By symmetry, we can also show that if opt = max, then $\Phi(G_2 \oplus H) \leq \Phi(G_1 \oplus H) \Leftrightarrow i$ and if opt = min then $\Phi(G_2 \oplus H) \geq \Phi(G_1 \oplus H) \Leftrightarrow i$, and hence $\Phi(G_1 \oplus H) = \Phi(G_2 \oplus H) + i$. □

Claim. *For each $S_2 \in D_{[]}(G_2)$ which can lead to an optimal solution, there is an $S_1 \in D_{[]}(G_1)$ such that $(G_1, S_1) \sim_l (G_2, S_2)$ and for each l-terminal graph H and each $S \in D_{[]}(H)$, if $Q(G_2 \oplus H, S_2 \oplus S)$ holds and $z(S_2 \oplus S) = \Phi(G_2 \oplus H)$, then $Q(G_1 \oplus H, S_1 \oplus S)$ holds, and $z(S_1 \oplus S) = \Phi(G_1 \oplus H)$.*

Proof. Let $S_2 \in D_{[]}(G_2)$ such that S_2 can lead to an optimal solution, let $C = \mathrm{ec}_{rQ,l}(G_2, S_2)$. Note that $\mathrm{opt}(G_2, C) = \overline{z}(S_2) \neq \mathsf{false}$ (and hence $\mathrm{opt}(G_1, C) \neq \mathsf{false}$). Let $S_1 = \mathrm{opt}(G_1, C)$. Let H be an l-terminal graph, let $S_H \in D_{[]}(H)$ and let $C' = \mathrm{ec}_{rQ,l}(H)$. Suppose $Q(G_2 \oplus H, S_2 \oplus S_H)$ holds and $z(S_2 \oplus S_H) = \Phi(G_2 \oplus H)$. By a previous claim, $z(S_1 \oplus S_H) = z(S_2 \oplus S_H) + i$. Since $\Phi(G_1 \oplus H) = \Phi(G_2 \oplus H) + i$ and $\Phi(G_2 \oplus H) = z(S_2 \oplus H)$, this implies that $z(S_1 \oplus H) = \Phi(G_1 \oplus H)$. □

The claims show that conditions 1 and 2 hold.

Let $k \geq 1$. We show that there is a special constructive reduction-counter system for Φ_k. Theorem 6.1.2 shows that there is a special constructive reduction system $S = (R, I, A_R, A_I)$ for P_k such that for each $(H_1, H_2) \in R$, $H_1 \approx_l H_2$. We show how to transform S into a special constructive reduction-counter system $S' = (R', I', \phi, A'_R, A'_I)$ for Φ_k. First, we make a set R' of reduction-counter rules from R: for each $r = (H_1, H_2) \in R$, make a reduction-counter rule (r, i) in R' with $i = i_{H_1} \Leftrightarrow i_{H_2}$. As is shown before, R' is safe for Φ_k.

Next, let $I' = I$, and for each $G \in I'$, let $\phi(G) = \Phi(G)$. We let the algorithms A'_R and A'_I be the same as A_R and A_I, but with different tables. For A'_I, we make a table which gives for each $G \in I'$ an $S \in D(G)$ such that $\Phi(G) = z(S)$. For A'_R, we make a table which, for each reduction-counter rule $r = ((H_1, H_2), i) \in R'$, and each $S_2 \in H_2$ for which $\overline{z}(S_2) = \mathrm{opt}(H_2, \mathrm{ec}_l(H_2, S_2))$, contains $\mathrm{optS}(H_1, S_1)$. Now, $(R', I', \phi, A'_R, A'_I)$ is a special constructive reduction-counter system for Φ_k. The effectiveness result easily follows. □

Note that, if only conditions 1 and 2 hold for Φ, then Φ is of finite integer index, and hence for each $k \geq 1$, there is a special reduction-counter system for Φ_k.

As a corollary, we also have the following result.

Corollary 6.2.1. *Let Φ be a constructive optimization problem defined by (D, Q, z, opt), where D is a t-vertex-edge-tuple for some $t \geq 1$. Suppose there is a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ for which the following conditions hold.*

1. *Q is decidable and for each $l \geq 0$, $\sim_{rQ,l}$ is decidable and $|C_{rQ,l}|$ is finite.*
2. *There is an extension $\overline{z}$ of z with respect to $\{\sim_{rQ,l} | \ l \geq 0\}$ and for each $l \geq 0$, there is a constant $K_l \in \mathbb{N}$, such that for each l-terminal graph G, each $S, S' \in D_{[]}(G)$, if both S and S' can lead to optimal solutions, then $|\overline{z}(S) \Leftrightarrow \overline{z}(S')| \leq K_l$.*

Then for each $k \geq 1$, there exists a special constructive reduction-counter system for Φ_k defined by (D, Q_k, z, opt).

If, in addition, (i) Q and $\sim_{rQ,l}$ are effectively decidable, (ii) z is effectively computable, and (iii) in condition 2, $\overline{z}$, K_l and i_G are effectively computable, then such a special constructive reduction-counter system can be effectively constructed.

With these results, we can prove that there are efficient constructive reduction algorithms for the following problems: MAX INDUCED d-DEGREE SUBGRAPH, MIN VERTEX COVER, MIN p-DOMINATING SET for all $p \geq 1$, MAX CUT on graphs with bounded degree, MIN PARTITION INTO CLIQUES, CHROMATIC NUMBER, MIN HAMILTONIAN PATH COMPLETION, and MAX LEAF SPANNING TREE. We prove this in Chapter 7, Theorem 7.1.2.

6.3 Parallel Constructive Reduction Algorithms

We now show how the results of Sections 6.1 and 6.2 can be extended to parallel reduction algorithms. We first consider decision problems, and then optimization problems.

6.3.1 Construction Problems

We start with adapting the definition of a special constructive reduction system.

Definition 6.3.1. Let P be a construction property defined by (D, Q) and let (R, I, A_R, A_I) be a constructive reduction system for P. Algorithm A_R is *non-interfering* if for each graph G, each $S \in D(G)$, every two terminal graphs G_1 and G_2, and every two reduction rules $r_1, r_2 \in R$, if r_1, G_1, G, S and r_2, G_2, G, S are correct inputs of A_R, and no inner vertex of G_1 occurs in G_2 and vice versa, then running A_R simultaneously on these two inputs (using the same versions of G and S) gives the same result as running A_R successively on these two inputs.

Definition 6.3.2 (Special Parallel Constructive Reduction System). Let P be a construction property defined by (D, Q). A constructive reduction system $S = (R, I, A_R, A_I)$ for P is a *special parallel constructive reduction system* for P if

- (R, I) is a special parallel reduction system for P,
- algorithms A_R and A_I use $O(1)$ time, and
- algorithm A_R is non-interfering.

Note that the constructive reduction system that we have defined for two-colorability of graphs of which the number of components is odd, and each component is a cycle (page 123) is a special parallel constructive reduction system: we represent each two-coloring as a labeling of the graph, i.e. each vertex is labeled with an integer denoting its color. We can implement algorithm A_R such that it is non-interfering, and it runs in $O(1)$ time (use the tables as given on page 123). Algorithm A_I also takes $O(1)$ time.

If we have a special parallel constructive reduction system for a given construction property P defined by (D, Q), then we can use a parallel variant of algorithm Reduce-Construct to

construct a solution for an input graph G, if one exists. The parallel algorithm consists of two parts. In part one, reductions are applied as often as possible, using the parallel algorithm described in Section 5.3.1. Recall that this algorithm consists of $O(\log n)$ reduction rounds, and in each round $\Omega(m)$ reductions are applied in parallel, where m denotes the number of vertices of the current graph. In each round, the reductions that are applied are non-interfering.

Part two of the algorithm starts with constructing an initial solution for the reduced graph, if P holds. This is done by one processor in constant time, by using algorithm A_I. After that, the reduction rounds of part one are undone in reversed order. In each undo-action of a reduction round, all reductions of that round are undone, and the solution is adapted. Each undo-action of a reduction is executed by the same processor that applied the rule in the first part of the algorithm. This processor also applies algorithm A_R. Since A_R is non-interfering, this results in the correct output.

Part one of the algorithm takes $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM. Part two can be done in $O(\log n)$ time with $O(n)$ operations and space on an EREW PRAM: each undo action of a reduction can be done in $O(1)$ time on one processor, and the local adaptation of the solution can also be done in $O(1)$ time by the same processor, since algorithm A_R takes constant time. This implies the following result.

Theorem 6.3.1. *Let P be a construction property defined by (D,Q). If we have a special parallel constructive reduction system for P, then we have an algorithm which, given a graph G, checks if $P(G)$ holds and if so, constructs an $S \in D(G)$ for which $Q(G,S)$ holds. The algorithm takes $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM.*

We next show that for a large class of construction properties on graphs of bounded treewidth, there is a special constructive reduction system. For simplicity, we only consider construction properties P defined by (D,Q), where D is a t-vertex-edge-tuple for some $t \geq 1$.

Theorem 6.3.2. *Let P be a construction property defined by (D,Q). If D is a t-vertex-edge-tuple for some $t \geq 1$, Q is decidable, and a finite refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ is decidable, then for each $k \geq 1$, there is a special parallel constructive reduction system for P_k.*

If, in addition, Q and $\sim_{rQ,l}$ are effectively decidable, then such a system can be effectively constructed.

Proof. Let $k \geq 1$. Let $S = (R, I, A_R, A_I)$ be a special constructive reduction system for P_k as defined in the proof of Theorem 6.1.2. We show that A_R and A_I can be made such that S is a special parallel reduction system for P_k.

We use the following data structure for storing (partial) solutions. Suppose G is the current graph and $S = (S_1, S_2, \ldots, S_t)$ is the current solution for G. With each vertex v, we store booleans $b_1, \ldots, b_t$: for each i, $1 \leq i \leq t$, b_i is true if and only if $D_i(G) = V(G)$ and $v = S_i$, or $D_i(G) = P(V(G))$ and $v \in S_i$. Similarly, with each edge e, we store booleans $b_1, \ldots, b_t$: for each i, $1 \leq i \leq t$, b_i is true if and only if $D_i(G) = E(G)$ and $e = S_i$, or $D_i(G) = P(E(G))$ and $e \in S_i$. It is easy to see that with this data structure, we can make A_R such that it is non-interfering and runs in $O(1)$ time. Furthermore, A_I also runs in $O(1)$ time. □

Note that, with the data structure for t-vertex-edge-tuples as described in the proof of Theorem 6.3.2, a returned solution for a given input graph is represented as a labeling of the vertices and edges of the graph. However, we can transform this representation into the representation as described on page 130: for each i, $1 \leq i \leq t$, use a parallel prefix algorithm (see e.g. JáJá [1992]) to make a list of all vertices or edges for which b_i is true. Since t is fixed, this takes $O(\log n)$ time with $O(n)$ operations on an EREW PRAM, and hence does not increase the total running time.

In particular, Theorem 6.3.2 shows that many well-known graph problems, when restricted to graphs of bounded treewidth, can be solved constructively within the stated resource bounds. These include all MS-definable construction properties for which the domain is a t-vertex-edge tuple.

6.3.2 Constructive Optimization Problems

A similar approach can be taken for constructive optimization problems. Let Φ be a constructive optimization problem defined by (D, Q, z, opt). Let S be a special constructive reduction-counter system for P. Then S is a *special parallel constructive reduction-counter system* if the derived constructive reduction system is a special parallel constructive reduction system.

Note that the constructive reduction-counter system that we defined for MAX INDEPENDENT SET on cycles (page 132) is a special parallel constructive reduction-counter system, if we represent an independent set as a labeling of the vertices of the graph: each vertex is labeled with a boolean which is true if and only if the vertex is in the independent set.

In the same way as described above we can transform the parallel algorithm for optimization problems as given in Section 5.3.2 into a parallel algorithm for constructive optimization problems, based on a special parallel constructive reduction-counter system.

Theorem 6.3.3. *Let Φ be a constructive optimization problem defined by (D, Q, z, opt). If we have a special parallel constructive reduction-counter system for Φ, then we have an algorithm which, given a graph G, checks if $\Phi(G) \in \mathbb{Z}$, and if so, constructs an $S \in D(G)$ for which $Q(G,S)$ holds and $z(S) = \Phi(G)$. The algorithm takes $O(\log n \log^* n)$ time with $O(n)$ operations and space on an EREW PRAM, and $O(\log n)$ time with $O(n)$ operations and space on a CRCW PRAM.*

From Theorem 6.3.2 and Corollary 6.2.1, we also derive the following result.

Theorem 6.3.4. *Let Φ be a constructive optimization problem defined by (D, Q, z, opt), where D is a t-vertex-edge-tuple for some $t \geq 1$. Suppose there is a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ for which the following conditions hold.*

1. *Q is decidable and for each $l \geq 0$, $\sim_{rQ,l}$ is decidable and $|C_{rQ,l}|$ is finite.*
2. *There is an extension $\overline{z}$ with respect to $\{\sim_{rQ,l} | \, l \geq 0\}$ and for each $l \geq 0$, there is a constant $K_l \in \mathbb{N}$, such that for each for each l-terminal graph G, each $S, S' \in D_{[]}(G)$, if both S and S' can lead to optimal solutions, then $|\overline{z}(S) \Leftrightarrow \overline{z}(S')| \leq K_l$.*

Then for each $k \geq 1$, there exists a special parallel constructive reduction-counter system for Φ_k defined by $(D, Q_k, z, \mathrm{opt})$.

If, in addition, (i) Q and $\sim_{rQ,l}$ are effectively decidable, (ii) z is effectively computable, and (iii) in condition 2, $\overline{z}$, K_l and i_G are effectively computable, then such a reduction-counter system can be effectively constructed.

This implies the existence of parallel algorithms with the stated resource bounds for the constructive versions of MAX INDUCED d-DEGREE SUBGRAPH for all $d \geq 0$, MIN p-DOMINATING SET for all $p \geq 1$, MIN VERTEX COVER, MAX CUT on graphs with bounded degree, and MAX LEAF SPANNING TREE when restricted to graphs of bounded treewidth. For a proof, see Theorem 7.1.2.

6.4 Additional Results

It is possible to generalize the results in this chapter to directed, mixed and/or labeled graphs, in the same way as is described in Section 5.4. The results of this chapter can also be used to give algorithms that generate all solutions for a construction property P, or all optimal solutions for a constructive optimization problem Φ.

In the same way as described in Section 5.4, we can also generalize the results of this chapter to multigraphs.

Chapter 7

Applications of Reduction Algorithms

In this chapter we apply the results of Chapter 6 to a number of constructive optimization problems on simple graphs. In Section 7.1 we prove a weaker version of Theorem 6.2.2 which is easier to use. We use this result and Theorem 6.2.2 to prove that a number of constructive optimization problems can be solved efficiently on graphs of bounded treewidth. In Section 7.2 we show that for a number of MS-definable constructive optimization problems, we can not apply the results of Chapters 5 and 6. The fact that these problems are MS-definable shows that there are efficient algorithms that solve them if a tree decomposition of bounded width of the input graph is given (Section 2.2.4).

7.1 Positive Results

While Theorem 6.2.2 may seem complex to use, it is in most cases not hard to find an equivalence relation $\sim_{rQ,l}$ which satisfies condition 1 and 3. Only condition 2 is often not easy to prove. Therefore, we give an alternative for condition 2, which is slightly weaker but easier to use, as will be demonstrated later in this section.

Theorem 7.1.1. *Let Φ be a constructive optimization problem defined by (D,Q,z,opt). Suppose that D is inducible for $[]$, that there is a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ for which Q is decidable, for each $l \geq 0$, $\sim_{rQ,l}$ is decidable and $|C_{rQ,l}|$ is finite, and furthermore, that the following condition holds.*

4. *There is an extension $\overline{z}$ of z with respect to $\{\sim_{rQ,l} | \ l \geq 0\}$, for each $l \geq 0$, there is a constant $K'_l \in \mathbb{N}$, and with each l-terminal graph G, we can associate an equivalence class $C_G \in C_{rQ,l}$, such that the following holds.*
   (a) *For all l-terminal graphs G and H, and $S \in D_{[]}(G)$, $S' \in D_{[]}(H)$, if $\text{ec}_{rQ,l}(G,S) = C_G$ and $\text{ec}_{rQ,l}(H,S') = C_H$, then (G,S) and (H,S') are $\oplus$-compatible, and $Q(G \oplus H, S \oplus S')$ holds.*
   (b) *If $\text{opt} = \max$, then for all l-terminals graphs G and all $S \in D_{[]}(G)$, if S can lead to a solution (i.e. there is an (H,S') such that $Q(G \oplus H, S \oplus S')$ holds), then $\overline{z}(S) \Leftrightarrow \text{opt}(G,C_G) \leq K'_l$.*
   (c) *If $\text{opt} = \min$, then for all l-terminals graphs G, all $S \in D_{[]}(G)$, if S can lead to a solution, then $\text{opt}(G,C_G) \Leftrightarrow \overline{z}(S) \leq K'_l$.*

Then condition 2 of Theorem 6.2.2 also holds.

Proof. Let $d_l(C,C')$ be the extension constants for $\overline{z}$. For each $l \geq 0$, let

$$K_l = 2K'_l + 4 \cdot \max\{|d_l(C,C')| \mid C,C' \in C_{rQ,l} \wedge C \text{ and } C' \text{ are } \oplus\text{-compatible}\}.$$

We show that with these definitions of $\overline{z}$ and K_l, condition 2 of Theorem 6.2.2 holds. We only consider the case that opt = max. The case that opt = min can be proved similarly.

Let G be an l-terminal graph. We show that for each $S \in D_{[]}(G)$, if S can lead to an optimal solution, then $|\overline{z}(S) \Leftrightarrow \text{opt}(G,C_G)| \leq K_l/2$. This implies that condition 2 holds. Let $S \in D_{[]}(G)$, and suppose S can lead to an optimal solution. By condition 4b, $\overline{z}(S) \Leftrightarrow \text{opt}(G,C_G) \leq K'_l \leq K_l/2$. Hence we only have to show that $\text{opt}(G,C_G) \Leftrightarrow \overline{z}(S) \leq K_l/2$, i.e. that $\overline{z}(S) \geq \text{opt}(G,C_G) \Leftrightarrow K_l/2$.

Let H be an l-terminal graph and let $S_H \in D_{[]}(H)$ such that $Q(G \oplus H, S \oplus S_H)$ holds and $z(S \oplus S_H) = \Phi(G \oplus H)$. By condition 4b, $\overline{z}(S_H) \Leftrightarrow \text{opt}(H,C_H) \leq K'_l \leq K_l/2$. Note that $\text{opt}(G,C_G) \in \mathbb{Z}$ and $\text{opt}(H,C_H) \in \mathbb{Z}$. Let $C = \text{ec}_{rQ,l}(G,S)$ and let $C' = \text{ec}_{rQ,l}(H,S_H)$. Then

$$\begin{aligned}
\overline{z}(S) &= \Phi(G \oplus H) \Leftrightarrow \overline{z}(S_H) + d_l(C,C') \\
&\geq \Phi(G \oplus H) \Leftrightarrow (\text{opt}(H,C_H) + K'_l) + d_l(C,C') \\
&\geq z(\text{optS}(G,C_G) \oplus \text{optS}(H,C_H)) \Leftrightarrow \text{opt}(H,C_H) \Leftrightarrow K'_l + d_l(C,C') \\
&= \overline{z}(\text{optS}(G,C_G)) + \overline{z}(\text{optS}(H,C_H)) \Leftrightarrow d_l(C_G,C_H) \Leftrightarrow \text{opt}(H,C_H) \Leftrightarrow K'_l + d_l(C,C') \\
&= \text{opt}(G,C_G) \Leftrightarrow (K'_l + d_l(C_G,C_H) \Leftrightarrow d_l(C,C')) \\
&\geq \text{opt}(G,C_G) \Leftrightarrow (K'_l + 2\max\{|d_l(C,C')| \mid C,C' \in C_{rQ,l} \wedge C \text{ and } C' \text{ are } \oplus\text{-compatible}\}) \\
&= \text{opt}(G,C_G) \Leftrightarrow K_l/2
\end{aligned}$$

Hence $\text{opt}(G,C_G) \Leftrightarrow \overline{z}(S) \leq K_l/2$. This completes the proof. □

Informally, condition 4 states that each l-terminal G graph has a basic equivalence class C_G such that (4a) for all l-terminal graphs H, a partial solution in C_G and a partial solution in C_H together form a solution of $G \oplus H$, and (4b and 4c) all partial solutions $S \in D_{[]}(G)$ which can lead to a solution are at most a constant term better than the best solution in C_G.

In the following theorem, we show for a number of constructive optimization problems that they are efficiently solvable, using the methods of Chapters 5 and 6 and of Theorem 7.1.1. For definitions of these problems, see Appendix A.

Theorem 7.1.2. *Each of the following constructive optimization problems can be solved in $O(n)$ time and space on graphs of bounded treewidth without making a tree decomposition of the input graph.*

1. MAX INDUCED d-DEGREE SUBGRAPH *for all $d \geq 0$,*
2. MIN VERTEX COVER,
3. MIN p-DOMINATING SET *for all $p \geq 1$,*
4. MAX CUT *on graphs with bounded degree,*
5. MIN PARTITION INTO CLIQUES,

6. CHROMATIC NUMBER,
7. MIN HAMILTONIAN PATH COMPLETION,
8. MIN HAMILTONIAN CIRCUIT COMPLETION, *and*
9. MAX LEAF SPANNING TREE.

Each of these problems can be solved with $O(n)$ operations and space, and in $O(\log n \log^* n)$ time on an EREW PRAM or in $O(\log n)$ time on a CRCW PRAM. However for problems 5, 7 and 8 the parallel algorithm only gives the solution as a labeling of the graph within these bounds.

Proof. For each $l \geq 0$, let $I_l = \{1, \ldots, l\}$, and $F_l = \{\{i, j\} \mid 1 \leq i < j \leq l\}$. Furthermore, for each l-terminal graph $G = (V, E, \langle x_1, \ldots, x_l \rangle)$, let

$$F(G) = \{\{i, j\} \mid \{x_i, x_j\} \in E\},$$

and for each $W \subseteq V(G)$ let

$$I(W) = \{i \in I_l \mid x_i \in W\}.$$

We consider the problems one by one.

1 MAX INDUCED d-DEGREE SUBGRAPH. Let $d \geq 0$ be fixed. Let Φ be defined by $(D, Q, z, \max)$, where D, Q and z are defined as follows. For each graph G, let $D(G) = P(V)$, and for each $S \in D(G)$, let

$$Q(G, S) = \text{`for all } v \in S\text{: } |N_{G,S}(v)| \leq d\text{'},$$

where $N_{G,S}(v) = \{w \in S \mid \{v, w\} \in E(G)\}$. Furthermore, let $z(S) = |S|$. We show that for each $k \geq 1$, there is a special constructive reduction-counter system for Φ_k, by using Theorem 6.2.2 and Theorem 7.1.1. For two l-terminal graphs G and H, and $S \in D(G \oplus H)$, let $S[G] = S \cap V(G)$. Hence $D_{[]}(G) = D(G)$, and two solutions $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$ are compatible and $\oplus$-compatible if they contain the same terminals.

We define a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ by giving the sets $C_{rQ,l}$ and the functions $\mathrm{ec}_{rQ,l}$. For each $l \geq 0$, let

$$\begin{aligned} C_{rQ,l} = {} & \{(I, \mathsf{false}) \mid I \subseteq I_l\} \cup \\ & \{(F, I, N) \mid F \subseteq F_l \wedge I \subseteq I_l \wedge N \subseteq \{(i, n) \mid i \in I_l \wedge n \in \{1, \ldots, d\}\}. \end{aligned}$$

$|C_{rQ,l}|$ is bounded, because d is fixed. For each l-terminal graph $G = (V, E, \langle x_1, \ldots, x_l \rangle)$, each $S \in D_{[]}(G)$, let $\mathrm{ec}_{rQ,l}(G, S) \in C_{rQ,l}$ be defined as follows. If there is a $v \in S$ such that $|N_{G,S}(v)| > d$, then $\mathrm{ec}_{rQ,l}(G, S) = (I(S), \mathsf{false})$ (S can not lead to a solution), otherwise, $\mathrm{ec}_{rQ,l}(G, S) = (F(G), I(S), N)$, where

$$N = \{(i, |N_{G,S}(x_i)|) \mid i \in I(S)\}.$$

We first show that $\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$ for all l. Suppose $(G_1,S_1) \sim_{rQ,l} (G_2,S_2)$. Clearly, (G_1,S_1) and (G_2,S_2) are compatible. Let H be an l-terminal graph, let $S_H \in D_{[]}(H)$ such that (G_1,S_1) and (H,S_H) are $\oplus$-compatible. We have to show that $Q(G_1 \oplus H, S_1 \oplus S_H)$ holds if and only if $Q(G_2 \oplus H, S_2 \oplus H)$ holds. If

$$\mathrm{ec}_{rQ,l}(G_1,S_1) = \mathrm{ec}_{rQ,l}(G_2,S_2) = (I(S_1), \mathsf{false}),$$

then $Q(G_1 \oplus H, S_1 \oplus S_H) = \mathsf{false} = Q(G_2 \oplus H, S_2 \oplus S_H)$. Suppose

$$\mathrm{ec}_{rQ,l}(G_1,S_1) = \mathrm{ec}_{rQ,l}(G_2,S_2) = (F,I,N),$$

where $N = \{(i,n_i) \mid i \in I\}$. Let $X = \langle x_1,\ldots,x_l\rangle$, $Y = \langle y_1,\ldots,y_l\rangle$, and $Z = \langle z_1,\ldots,z_l\rangle$ denote the terminal sets of G_1, G_2 and H, respectively.

$$\begin{aligned}
&Q(G_1 \oplus H, S_1 \oplus S_H)\\
&\quad = (\forall_{v \in S_1 \oplus S_H} |N_{G_1 \oplus H, S_1 \oplus S_H}(v)| \le d)\\
&\quad = (\forall_{i \in I} |N_{H,S_H}(z_i)| + |N_{G_1,S_1}(x_i)| \Leftrightarrow |\{j \in I \mid x_j \in N_{G_1,S_1}(x_i) \wedge z_j \in N_{H,S_H}(z_i)\}| \le d)\\
&\qquad \wedge (\forall_{v \in S_1 - X} |N_{G_1,S_1}(v)| \le d) \wedge (\forall_{v \in S_H - Z} |N_{H,S_H}(v)| \le d)\\
&\quad = (\forall_{i \in I} |N_{H,S_H}(z_i)| + |n_i| \Leftrightarrow |\{j \in I \mid \{i,j\} \in F \wedge \{z_i,z_j\} \in E(H)\}| \le d)\\
&\qquad \wedge (\forall_{v \in S_1 - X} |N_{G_1,S_1}(v)| \le d) \wedge (\forall_{v \in S_H - Z} |N_{H,S_H}(v)| \le d)\\
&\quad = (\forall_{i \in I} |N_{H,S_H}(z_i)| + |N_{G,S_2}(y_i)| \Leftrightarrow |\{j \in I \mid y_j \in N_{G_2,S_2}(y_i) \wedge z_j \in N_{H,S_H}(z_i)\}| \le d)\\
&\qquad \wedge (\forall_{v \in S_2 - Y} |N_{G_2,S_2}(v)| \le d) \wedge (\forall_{v \in S_H - Z} |N_{H,S_H}(v)| \le d)\\
&\quad = Q(G_2 \oplus H, S_2 \oplus S_H)
\end{aligned}$$

Hence $\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$. This proves condition 1 of Theorem 6.2.2.

Consider condition 4 of Theorem 7.1.1. For each terminal graph G, each $S \in D_{[]}(G)$, let $\overline{z}(S) = |S|$. We show that $\overline{z}$ is an extension of z. Let $C,C' \in C_{rQ,l}$, such that C and C' are compatible. Let $I \subseteq I_l$ such that $C = (I,\mathsf{false})$ or $C = (F,I,N)$ for some F and N, and $C' = (I,\mathsf{false})$ or $C' = (F',I,N')$ for some F' and N'. Let G and H be l-terminal graphs, let $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$ such that $\mathrm{ec}_{rQ,l}(G,S) = C$ and $\mathrm{ec}_{rQ,l}(H,S') = C'$. Then $z(S \oplus S') = |S \oplus S'| = |S \cup S'| = |S| + |S'| \Leftrightarrow |I| = \overline{z}(S) + \overline{z}(S') \Leftrightarrow |I|$, hence $d_l(C,C') = |I|$, which shows that $\overline{z}$ is an extension of z.

We next define K_l for all $l \ge 0$, and C_G for all l-terminal graphs G, and show that condition 4 holds of Theorem 7.1.1 holds with these definitions. For each $l \ge 0$, let $K_l = l$, and for each l-terminal graph G, let $C_G = (F(G),\phi,\phi)$. Clearly, for all l-terminal graphs G and H, each $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$, if $\mathrm{ec}_{rQ,l}(G,S) = C_G$ and $\mathrm{ec}_{rQ,l}(H,S') = C_H$, then (G,S) and (H,S') are $\oplus$-compatible, and $Q(G \oplus H, S \oplus S')$ holds. Furthermore, for each l-terminal graph $G = (V,E,X)$, and each $S \in D_{[]}(G)$ that can lead to a solution (i.e. $\mathrm{ec}_{rQ,l}(G,S) \neq (F(G),\mathsf{false})$), $\mathrm{ec}_{rQ,l}(G,S \Leftrightarrow X) = C_G$ and $|S| \Leftrightarrow \mathrm{opt}(G,C_G) \le |S| \Leftrightarrow |S \Leftrightarrow X| \le l = K_l$.

Condition 3 of Theorem 6.2.2 also holds, as D is a one-vertex-edge-tuple. Hence for each $k \ge 1$, there is a special (parallel) constructive reduction-counter system for Φ_k.

2 MIN VERTEX COVER. There are two ways to prove that MIN VERTEX COVER can be solved efficiently on graphs of bounded treewidth. Firstly, it is well-known that for each graph G, if S is a maximum independent set in G, then $V(G) \Leftrightarrow S$ is a minimum vertex cover in G. Hence we can solve MIN VERTEX COVER by first computing a maximum independent set of G (using the result for MAX INDUCED d-DEGREE SUBGRAPH), and then taking the complement of this set.

Alternatively, one can prove in a direct way that there is a special (parallel) constructive reduction-counter system for Min Vertex Cover on graphs of bounded treewidth. This proof is similar to the proof for MAX INDUCED d-DEGREE SUBGRAPH, and we do not give it here.

3 MIN p-DOMINATING SET. Let $p \geq 1$ be fixed. Let Φ be defined by $(D, Q, z, \min)$, where D, Q and z are defined as follows. For each graph G, $D(G) = P(V)$, and for each $S \in D(G)$,

$$Q(G,S) = \text{`for all } v \in V \Leftrightarrow S\text{: } |N_{G,S}(v)| \geq p\text{'},$$

and $z(S) = |S|$. $[]$ is defined in the same way as for MAX INDUCED d-DEGREE SUBGRAPH, and so are $\oplus$ and ($\oplus$-)compatibility. We define $\sim_{rQ,l}$ by giving $C_{rQ,l}$ and $\sim_{rQ,l}$. For each $l \geq 0$, let

$$\begin{aligned} C_{rQ,l} = {} & \{(I, \mathsf{false}) \mid I \subseteq I_l\} \cup \\ & \{(F,I,N) \mid F \subseteq F_l \wedge I \subseteq I_l \wedge N \subseteq \{(i,n) \mid i \in I_l \Leftrightarrow I \wedge n \in \{1,\ldots,p\}\}. \end{aligned}$$

For each l-terminal graph $G = (V,E,X)$ with $X = \langle x_1, \ldots, x_l \rangle$), and for each $S \in D_{[]}(G)$, let $\mathrm{ec}_{rQ,l}(G,S) \in C_{rQ,l}$ be defined as follows. If there is a $v \in V \Leftrightarrow X$ such that $|N_{G,S}(v)| < p$, then $\mathrm{ec}_{rQ,l}(G,S) = (I(S), \mathsf{false})$ (S can not lead to a solution). Otherwise, $\mathrm{ec}_{rQ,l}(G,S) = (F(G), I(S), N)$, where

$$N = \{(i,n) \mid i \in I(S) \Leftrightarrow I_l \wedge ((n = |N_{G,S}(x_i)| \wedge n \leq p) \vee (n = p \wedge |N_{G,S}(x_i)| \geq p))\}.$$

In the same way as for MAX INDUCED d-DEGREE SUBGRAPH, it can be shown that $\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$, and hence condition 1 of Theorem 6.2.2 holds.

We next show that condition 4 of Theorem 7.1.1 holds. For each terminal graph G, each $S \in D_{[]}(G)$, let $\overline{z}(S) = |S|$. It can be seen that $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} \mid l \geq 0\}$. For each l, let $K_l = l$ and for each terminal graph G, let $C_G = (F(G), I_l, \emptyset)$. Clearly, for all l-terminal graphs G and H and $S \in D_{[]}(G)$, $S' \in D_{[]}(H)$ such that $\mathrm{ec}_{rQ,l}(G,S) = C_G$ and $\mathrm{ec}_{rQ,l}(H,S') = C_H$, $Q(G \oplus H, S \oplus S')$ holds. Furthermore, for each l-terminal graph $G = (V,E,X)$, each $S \in D_{[]}(G)$, if S can lead to a solution, then $\mathrm{ec}_{rQ,l}(G,S) \neq (F(G), \mathsf{false})$ and $\mathrm{ec}_{rQ,l}(G, S \cup X) = C_G$, and hence $\mathrm{opt}(G, C_G) \Leftrightarrow |S| \leq |S \cup X| \Leftrightarrow |S| \leq l = K_l$. This proves condition 4 of Theorem 7.1.1. Condition 3 of Theorem 6.2.2 is also satisfied, as D is again a one-vertex-edge-tuple. This implies that for each $k \geq 1$, there is a special (parallel) constructive reduction-counter system for Φ_k.

4 MAX CUT on graphs with bounded degree. Let $d \geq 0$ be a bound on the maximum degree of the input graph. Let Φ be defined by $(D, Q, z, \max)$, where D, Q and z are as follows. For

each graph G, let $D(G)$ be the set of all partitions (V_1,V_2) of $V(G)$, and for each $S \in D(G)$, let $Q(G,S) = \text{true}$. For each graph G and each $S = (V_1,V_2) \in D(G)$, let

$$z(S) = |\{\{u,v\} \in E(G) \mid u \in V_1 \wedge v \in V_2\}|.$$

We show that for each $k \geq 1$, there is a special parallel constructive reduction-counter system for Φ_k.

Let $[]$ be defined in the obvious way, i.e. for each two l-terminal graphs G and H, and $S = (V_1,V_2) \in D(G \oplus H)$, let $S[G] = (V_1 \cap V(G), V_2 \cap V(G))$. Note that $D_{[]}(G)$ is the set of all partitions (V_1,V_2) of $V(G)$, and that D is inducible for $[]$. Two pairs (G,S) and (H,S') are ($\oplus$-)compatible if S and S' are the equal on the set of terminals.

For each $l \geq 0$, let

$$C_{rQ,l} = \{(F,I) \mid F \subseteq F_l \wedge I \subseteq I_l\},$$

and for each l-terminal graph $G = (V,E,X = \langle x_1,\ldots,x_l\rangle)$ and each $S = (V_1,V_2) \in D_{[]}(G)$, let $\text{ec}_{rQ,l}(G,S) = (F(G), I(V_1))$.

Let G_1 and G_2 be l-terminal graphs, $S_1 \in D_{[]}(G_1)$, $S_2 \in D_{[]}(G_2)$. If $(G_1,S_1) \sim_{rQ,l} (G_2,S_2)$, then (G_1,S_1) and (G_2,S_2) are compatible, and hence $\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$. This means that condition 1 of Theorem 6.2.2 holds.

Consider condition 2 of Theorem 6.2.2. For each terminal graph G, each $S = (V_1,V_2) \in D_{[]}(G)$, let $\overline{z}(S) = |\{\{u,v\} \in E(G) \mid u \in V_1 \wedge v \in V_2\}|$. We show that $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} \mid l \geq 0\}$.

Let G and H be l-terminal graphs, let $S = (V_1,V_2) \in D_{[]}(G)$ and $S' = (W_1,W_2) \in D_{[]}(H)$, such that (G,S) and (H,S') are $\oplus$-compatible. Let $\text{ec}_{rQ,l}(G,S) = (F,I)$, and let $\text{ec}_{rQ,l}(H,S) = (F',I)$. Then

$$\begin{aligned} z(S \oplus S') &= |\{\{u,v\} \in E(G \oplus H) \mid u \in V_1 \cup W_1 \wedge v \in V_2 \cup W_2\}| \\ &= |\{\{u,v\} \in E(G) \mid u \in V_1 \wedge v \in V_2\}| \\ &\quad + |\{\{u,v\} \in E(H) \mid u \in W_1 \wedge v \in W_2\}| \\ &\quad \Leftrightarrow |\{\{u,v\} \in E(G) \cap E(H) \mid u \in W_1 \wedge v \in W_2\}| \\ &= \overline{z}(S) + \overline{z}(S') \Leftrightarrow |\{\{i,j\} \in F \cap F' \mid i \in I \wedge j \in I_l \Leftrightarrow I\}|. \end{aligned}$$

Hence $d_l((F,I),(F',I)) = |\{\{i,j\} \in F \cap F' \mid i \in I \wedge j \in I_l \Leftrightarrow I\}|$.

For each $l \geq 0$, let $K_l = 2 \cdot l \cdot d$. Let $G = (V,E,X)$ be an l-terminal graph, let $S,S' \in D_{[]}(G)$ such that S and S' can lead to optimal solutions. We have to show that $|\overline{z}(S_1) \Leftrightarrow \overline{z}(S_2)| \leq K_l$. Let $S = (V_1,V_2)$ and $S' = (V_1',V_2')$. Let $\overline{S} = (W_1,W_2) \in D_{[]}(G)$, where

$$W_1 = (V_1 \Leftrightarrow X) + (V_1' \cap X) \quad \text{and} \quad W_2 = (V_2 \Leftrightarrow X) + (V_2' \cap X).$$

Then $(G,S') \sim_{rQ,l} (G,\overline{S})$, and hence $\overline{z}(\overline{S}) \leq \overline{z}(S')$. Note that S and $\overline{S}$ only differ on the set of terminals. Furthermore,

$$\begin{aligned}
&\overline{z}(S) \Leftrightarrow \overline{z}(S') \\
&\quad \leq \overline{z}(S) \Leftrightarrow \overline{z}(\overline{S}) \\
&\quad = \overline{z}(S) \Leftrightarrow (\,\overline{z}(S) \Leftrightarrow |\{\{u,v\} \in E(G) \mid u \in V_1 \wedge v \in V_2 \wedge (u,v \in W_1 \vee u,v \in W_2)\}| \\
&\qquad\qquad + |\{\{u,v\} \in E(G) \mid u \in W_1 \wedge v \in W_2 \wedge (u,v \in V_1 \vee u,v \in V_2)\}|\,) \\
&\quad \leq |\{\{u,v\} \in E(G) \mid u \in V_1 \wedge v \in V_2 \wedge (u,v \in W_1 \vee u,v \in W_2)\}| \\
&\quad = |\{\{u,v\} \in E(G) \mid (u \in V_1 \cap W_1 \wedge v \in V_2 \cap W_1) \vee (u \in V_1 \cap W_2 \wedge v \in V_2 \cap W_2)\}| \\
&\quad \leq |\{\{u,v\} \mid (v \in V_2 \cap W_1) \vee (u \in V_1 \cap W_2)\}| \\
&\quad \leq |\{\{u,v\} \mid u \in X \vee v \in X\}| \\
&\quad \leq 2 \cdot l \cdot d = K_l.
\end{aligned}$$

Because of symmetry, this means that $|\overline{z}(S) \Leftrightarrow \overline{z}(S')| \leq K_l$. Hence condition 2 of Theorem 6.2.2 holds.

Condition 3 of Theorem 6.2.2 also holds, since D is a two-vertex-edge-tuple. Hence for each $k \geq 1$, there is a special (parallel) constructive reduction-counter system for Φ_k on graphs with bounded degree.

5 MIN PARTITION INTO CLIQUES. Let Φ be defined by $(D,Q,z,\min)$, where D, Q and z are defined as follows. For each graph G, let $D(G)$ be the set of all (unordered) partitions $S = \{V_1,\ldots,V_s\}$ of $V(G)$ for some $s \geq 1$, for which each $V_i \in S$ induces a connected subgraph of G. For each $S \in D(G)$, let

$$Q(G,S) = \text{‘for all } W \in S\text{: } G[W] \text{ is a complete graph’},$$

and let $z(S) = |S|$. We show that for each $k \geq 1$, there is a special constructive reduction-counter system for Φ_k.

For each two l-terminal graphs $G = (V,E,X)$ and $H = (V',E',Y)$, each $S \in D(G \oplus H)$, let

$$S[G] = \{W \cap V(G) \mid W \in S \wedge W \cap V(G) \neq \emptyset\}.$$

Hence $D_{[]}(G)$ is the set of all partitions S of $V(G)$ in which for each $W \in S$, all connected components of $G[W]$ contain at least one terminal vertex.

Note that D is inducible for $[]$, since, for an $S \in D(G \oplus H)$, there is no $W \in S$ which contains vertices of both G and H while it does not contain terminals of G. Two pairs (G,S) and (H,S') are ($\oplus$-)compatible if the terminals of G and H are partitioned in the same way in S and S', i.e.

$$\{I \subseteq I_l \mid I \neq \emptyset \wedge \exists_{W \in S} I = I(W)\} = \{I \subseteq I_l \mid I \neq \emptyset \wedge \exists_{W \in S'} I = I(W)\}.$$

For each $l \geq 0$, let

$$\begin{aligned} C_{rQ,l} = \; & \{(F, \mathsf{false}) \mid F \subseteq F_l\} \cup \\ & \{(F, \{(J_1, b_1), \ldots, (J_t, b_t)\}) \mid F \subseteq F_l \wedge \\ & \quad t \geq 1 \wedge \{J_1, \ldots, J_t\} \text{ partitions } I_l \wedge (\forall_i \, J_i \neq \phi \wedge b_i \in \{\mathsf{true}, \mathsf{false}\})\} \} \end{aligned}$$

For each l-terminal graph $G = (V, E, \langle x_1, \ldots, x_l \rangle)$, each $S \in D_{[]}(G)$, let $\mathrm{ec}_{rQ,l}(G,S) \in C_{rQ,l}$ be defined as follows. If there is a $W \in S$ which contains a pair $v, w \in W$ for which $\{v,w\} \notin E(G)$ and $\{v,w\} \not\subseteq X$, then $\mathrm{ec}_{rQ,l}(G,S) = (F(G), \mathsf{false})$ (S can not lead to a solution). Otherwise, $\mathrm{ec}_{rQ,l}(G,S) = (F(G), J)$, where

$$J \; = \; \{(J,b) \mid (\exists_{W \in S} \, J = I(W) \wedge J \neq \phi \wedge (b \Leftrightarrow (W \subseteq X))\}.$$

It is fairly easy to check that if $\mathrm{ec}_{rQ,l}(G_1,S_1) = \mathrm{ec}_{rQ,l}(G_2,S_2)$, then $(G_1,S_1) \sim_{Q,l} (G_2,S_2)$. This shows that condition 1 of Theorem 6.2.2 holds.

Consider condition 4 of Theorem 7.1.1. For each terminal graph G, each $S \in D_{[]}(G)$, let $\overline{z}(S) = |S|$. We show that $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} \mid l \geq 0\}$. Let G and H be l-terminal graphs, let $S \in D_{[]}(G)$, $S' \in D_{[]}(H)$, such that (G,S) and (H,S') are $\oplus$-compatible. Let $\mathrm{ec}_{rQ,l}(G,S) = (F,J)$, and let $\mathrm{ec}_{rQ,l}(H,S) = (F',J')$. Then

$$\begin{aligned} |S| + |S'| \Leftrightarrow |S \oplus S'| &= |\{W \in S \oplus S' \mid W \cap X \neq \phi\}| \\ &= |J|. \end{aligned}$$

Hence $d_l((F,J),(F',J')) = |J|$.

For each $l \geq 0$, let $K_l = l$, and for each l-terminal graph G, let

$$C_G = (F(G), \{(\{i\}, \mathsf{true}) \mid 1 \leq i \leq l\}).$$

Clearly, condition 4a of Theorem 7.1.1 holds. Let G be an l-terminal graph, let $S \in D_{[]}(G)$, such that S may lead to a solution, i.e. $\mathrm{ec}_{rQ,l}(G,S) \neq (F(G), \mathsf{false})$. Furthermore, let

$$S' = \{\{v\} \mid v \in X\} \cup \{W \Leftrightarrow X \mid W \in S \wedge W \not\subseteq X\}.$$

Then $S' \in D_{[]}(G)$ and $\mathrm{ec}_{rQ,l}(G,S') = C_G$, and hence $\mathrm{opt}(G, C_G) \Leftrightarrow |S| \leq |S'| \Leftrightarrow |S| \leq l = K_l$. This shows that condition 4 of Theorem 7.1.1 holds.

Consider condition 3 of Theorem 6.2.2. We represent (partial) solutions S as follows. We construct a list of all elements $W \in S$. For each $W \in S$, we construct a list of all vertices in W, and for each $v \in W$, we make a pointer to W and to vertex v in W. It is easy to check that condition 3 of Theorem 6.2.2 holds for this representation. This completes the proof that for each $k \geq 1$, there is a special constructive reduction-counter system for Φ_k.

For the parallel algorithm, we use a different representation of (partial) solutions. For each (partial) solution S, we label the vertices in G in such a way that two vertices v and w have the same label if and only if they are in the same clique, i.e. there is a $W \in S$ such that

$v, w \in W$. It can be seen that with this representation one can use the parallel algorithm as described in Section 6.3.2 to solve MIN PARTITION INTO CLIQUES in $O(\log n \log^* n)$ time with $O(n)$ operations on an EREW PRAM, and in $O(\log n)$ time with $O(n)$ operations on a CRCW PRAM. However, we have no method to obtain a list of all cliques from the vertex labeling within the same resource bounds.

6 CHROMATIC NUMBER. We can not prove that there is a constructive reduction-counter system for CHROMATIC NUMBER on graphs of bounded treewidth. However, it is well-known that for each $k \geq 1$, each graph of treewidth at most k has a $(k+1)$-coloring. Furthermore, for each $m \geq 1$, the m-COLORABILITY problem is MS-definable, and the solutions are m-vertex-edge-tuples. Therefore, given an integer $k \geq 1$, we can solve CHROMATIC NUMBER on a graph G of treewidth at most k as follows. For each m, $1 \leq m \leq k+1$, try to find an m-coloring of G. Take the smallest m for which this is possible, and return an m-coloring. Sequentially, this can be done in $O(n)$ time and space. In parallel, this can be done with $O(n)$ operations and space in $O(\log n \log^* n)$ time on an EREW PRAM, and in $O(\log n)$ time on a CRCW PRAM.

7 MIN HAMILTONIAN PATH COMPLETION. Let Φ be defined by $(D, Q, z, \min)$, where D, Q and z are defined as follows. For each graph G, let each element S in $D(G)$ be a set of non-empty paths in G (i.e. paths containing at least one vertex), such that the set $\{V(P) \mid P \in S\}$ partitions G. For each $S \in D(G)$, let $Q(G,S) = \mathsf{true}$, and let $z(S) = |S| \Leftrightarrow 1$. Note that this correctly describes the problem HAMILTONIAN PATH COMPLETION NUMBER. We show that for each $k \geq 1$, there is a special constructive reduction-counter system for Φ_k.

Let G and H be l-terminal graphs, $S \in D(G \oplus H)$. Let

$$S[G] = \bigcup_{P \in S} \{P' \mid P' \text{ is a connected component of } P[V(G)]\},$$

i.e. $S[G]$ is the set of paths in G which is obtained from S by deleting all vertices and edges which are not in G, and deleting empty paths. Domain D is inducible for this definition of $[]$: let G and H be l-terminal graphs, $S \in D(G \oplus H)$. Then S is the set of components of the subgraph G' of $G \oplus H$ with

$$V(G') = V(G \oplus H)$$
$$E(G') = \{e \in E(G \oplus H) \mid \exists_{P \in S[G]} e \in E(P) \vee \exists_{P \in S[H]} e \in E(P)\}.$$

Let $G = (V, E, X = \langle x_1, \ldots, x_l \rangle)$ be an l-terminal graph G, let $P = (v_1, \ldots, v_s)$ be a path in G. Suppose $V(P) \cap X = \{x_{i_1}, \ldots, x_{i_q}\}$, $q \geq 1$, and for each $1 \leq j < m \leq q$, x_{i_j} occurs on the left side of x_{i_m} in P (i.e. by walking from v_1 to v_s in P, we meet x_{i_j} earlier than x_{i_m}). Let vx and novx be dummy vertices. Then $\mathrm{Ind}(P)$ is defined as follows.

$$\mathrm{Ind}(P) = (d_0, i_1, d_1, i_2, d_2, \ldots, d_{q-1}, i_q, d_q),$$

where for each i, $0 \leq i \leq q$, $d_i \in \{\mathsf{vx}, \mathsf{novx}\}$ as follows. If $x_{i_1} = v_1$, then no vertex precedes x_{i_1} in P and hence $d_0 = \mathsf{novx}$, otherwise $d_0 = \mathsf{vx}$. If $x_{i_q} = v_s$, then no vertex follows x_{i_1} and hence

$d_q = \mathsf{novx}$, otherwise $d_q = \mathsf{vx}$. For each m, $0 < m < q$, if there is a j, $1 \leq j < s$, such that $x_{i_m} = v_j$ and $x_{i_{m+1}} = v_{j+1}$, then there is no vertex between x_{i_m} and $x_{i_{m+1}}$, and hence $d_m = \mathsf{novx}$, otherwise, $d_m = \mathsf{vx}$.

For each l-terminal graph G and each $S \in D_{[]}(G)$, let $\mathrm{ec}_{rQ,l}(G,S)$ be defined as follows:

$$\mathrm{ec}_{rQ,l}(G,S) = (F(G), \{\mathrm{Ind}(P) \mid P \in S \wedge V(P) \cap X \neq \emptyset\}).$$

For each $l \geq 0$, let $C_{rQ,l}$ contain all possible values of $\mathrm{ec}_{rQ,l}(G,S)$ and let $\sim_{rQ,l}$ be defined as usual. It can be seen that if $(G_1,S_1) \sim_{rQ,l} (G_2,S_2)$, then (G_1,S_1) and (G_2,S_2) are compatible, and hence $(G_1,S_1) \sim_{Q,l} (G_2,S_2)$. This proves condition 1 of Theorem 6.2.2.

Consider condition 4 of Theorem 7.1.1. For each terminal graph G, each $S \in D_{[]}(G)$, let $\overline{z}(S) = |S|$. We show that $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} | \, l \geq 0\}$. Let $G = (V,E,X)$ and $H = (V',E',Y)$ be l-terminal graphs, $S \in D_{[]}(G)$ and $S' \in D_{[]}(H)$. If (G,S) and (H,S') are $\oplus$-compatible, then

$$\begin{aligned} z(S) + z(S') \Leftrightarrow z(S \oplus S') &= |\{P \in S \mid V(P) \cap X \neq \emptyset\}| + |\{P \in S' \mid V(P) \cap Y \neq \emptyset\}| \\ &\Leftrightarrow |\{P \in S \oplus S' \mid V(P) \cap X \neq \emptyset\}|. \end{aligned}$$

This value can be computed from $\mathrm{ec}_{rQ,l}(G,S)$ and $\mathrm{ec}_{rQ,l}(H,S)$, hence $\overline{z}$ is an extension of z.

For each $l \geq 0$, let $K_l = 2l$, and for each l-terminal graph G, let

$$C_G = (F(G), \{(\mathsf{novx}, i, \mathsf{novx}) \mid 1 \leq i \leq l\}).$$

If $\mathrm{ec}_{rQ,l}(G,S) = C_G$ and $\mathrm{ec}_{rQ,l}(H,S') = C_H$, then (G,S) and (H,S') are $\oplus$-compatible.

Let G be an l-terminal graph, $S \in D_{[]}(G)$. Let G' be the graph consisting of all paths in S (note that $V(G') = V(G)$). Obtain G'' from G' by removing all edges $\{v,w\} \in E(G')$ for which $v \in X$. Let $S' = \{P \mid P \text{ is a component of } G''\}$. Then $S' \in D_{[]}(G)$, and $\mathrm{ec}_{rQ,l}(G,S') = C_G$. Furthermore, $|S'| \leq |S| + 2l$, and hence $\mathrm{opt}(G,C_G) \Leftrightarrow \overline{z}(S) \leq |S'| \Leftrightarrow |S| \leq 2l = K_l$. This completes the proof of condition 4 of Theorem 7.1.1.

Consider condition 3 of Theorem 6.2.2. We represent each (partial) solution S as follows. We construct a list of all paths $P \in S$. For each $P \in S$, we construct a list of all vertices in the order in which they occur in the path. We keep pointers from vertices in the graph to the corresponding vertices in the path and vice versa, and from each vertex in the graph to the path in which it occurs. With this representation, condition 3 can be proved. Hence for each $k \geq 1$, there is a special constructive reduction-counter system for Φ_k. The algorithm returns a set of paths which partition the vertices of the input graph, such that the number of these paths is minimum. If we are interested in a minimum set of edges which have to be added to the graph in order to get a graph which contains a Hamiltonian path, then we can compute such a set from the minimum set of paths: take any ordering of these paths, and let the edge set contain all edges from an end point of one path to the starting point of the next path in the ordering. This set can be computed in $O(n)$ time.

An efficient parallel algorithm which solves Φ_k for any $k \geq 1$ can only be obtained if we represent (partial) solutions as a labeling of the graph: given a (partial) solutions S, label each

vertex and edge in the graph such that vertices and edges in the same path have the same label. With this representation it can be shown that the algorithm described in Section 6.3.2 can be used.

8 MIN HAMILTONIAN CIRCUIT COMPLETION. The MIN HAMILTONIAN CIRCUIT COMPLETION can easily be solved with use of MIN HAMILTONIAN PATH COMPLETION and the constructive version of HAMILTONIAN CIRCUIT: suppose we have an input graph G. Let m_p and m_c denote the minimum number of edges that have to be added to G such that it contains a Hamiltonian path, and a Hamiltonian circuit, respectively. If $m_p \geq 1$, then $m_c = m_p + 1$. If $m_p = 0$, then there are two cases. Either G contains a Hamiltonian circuit, in which case $m_c = 0$, or G does not contain a Hamiltonian circuit, in which case $m_c = 1$. Hence to solve MIN HAMILTONIAN CIRCUIT COMPLETION, we first solve MIN HAMILTONIAN PATH COMPLETION as described above. If this gives a number $m_p \geq 1$ of paths, then it is a solution for MIN HAMILTONIAN CIRCUIT COMPLETION. If $m_p = 0$, then we solve the constructive version of HAMILTONIAN CIRCUIT. If this algorithm returns a Hamiltonian circuit, then we take this as a solution for MIN HAMILTONIAN CIRCUIT COMPLETION, otherwise, we take the solution of MIN HAMILTONIAN PATH COMPLETION. As HAMILTONIAN CIRCUIT is MS-definable and its solution domain can be seen as a two-vertex-edge-tuple, we have an efficient algorithm for MIN HAMILTONIAN CIRCUIT COMPLETION on graphs of bounded treewidth.

9 MAX LEAF SPANNING TREE. Let Φ be defined by $(D, Q, z, \max)$, where D, Q and z are defined as follows. For each graph G, let $D(G)$ be the set of all spanning trees of G. For each $S \in D(G)$, let $Q(G,S) = \text{true}$, and let $z(S)$ be the number of vertices of degree one in S. We show that there is a special parallel constructive reduction-counter system for Φ_k ($k \geq 1$).

For every two l-terminal graphs G and H and each $S \in D(G \oplus H)$, let $S[G]$ be the forest in G obtained by deleting all vertices and edges from S which are not in G, i.e. $S[G] = (V(S) \cap V(G), E(S) \cap E(G))$. Hence $D_{[]}(G)$ is the set of all spanning forests F of G for which each connected component of F contains at least one terminal of the graph. Note that D is inducible for $[]$.

For each l-terminal graph $G = (V, E, \langle x_1, \ldots, x_l \rangle)$, each $S \in D_{[]}(G)$, if S contains more than one connected component, and one of these components does not contain a terminal, then let $\text{ec}_{rQ,l} = (F(G), \text{false})$ (there are no H and $S_H \in D_{[]}(H)$ such that $S \oplus S_H$ is a spanning tree of $G \oplus H$), otherwise, let $\text{ec}_{rQ,l}(G,S) = (F(G), J, A)$ where

$$J = \{(J,F) \mid \exists_{V' \subseteq V}\ S[V'] \text{ is a connected component of } S \wedge J = I(V') \wedge F = \{\{i,j\} \mid i,j \in J \wedge \{x_i, x_j\} \in E(S)\}\},$$
$$A = \{(i, s_i) \mid 1 \leq i \leq l \wedge s_i = |N_{S,V}(x_i)| \text{ if } |N_{S,V}(x_i)| \leq 2, \text{ otherwise } s_i = 2\}.$$

More informally, for each component T of S, J contains the subgraph of T induced by the terminals in T (note that there are at most l such components). Furthermore, A denotes for each terminal i whether it has zero neighbors, one neighbor, or more than one neighbor in S.

Let $C_{rQ,l}$ be the set of all possible values of $\sim_{rQ,l}$. Note that $|C_{rQ,l}|$ is finite. For each $l \geq 0$, let $\sim_{rQ,l}$ be defined as usual.

One can check that if $(G_1,S_1) \sim_{rQ,l} (G_2,S_2)$, then (G_1,S_1) and (G_2,S_2) are compatible, i.e. that for each l-terminal graph H and each $S_H \in D_{[]}(H)$, the graph $(V(S_1) \cup V(S_H), E(S_1) \cup E(S_H)$ is a tree if and only if the graph $(V(S_1) \cup V(S_H), E(S_1) \cup E(S_H))$ is a tree. This implies that $(G_1,S_1) \sim_{Q,l} (G_2,S_2)$. Hence condition 1 of Theorem 6.2.2 holds.

Consider condition 2 of Theorem 6.2.2. For each terminal graph G and each $S \in D_{[]}(G)$, let $\overline{z}(S) =$ the number of vertices of degree one in S. We show that $\overline{z}$ is an extension of z with respect to $\{\sim_{rQ,l} | \ l \geq 0\}$.

Let G and H be l-terminal graphs, let $S \in D_{[]}(G)$, $S' \in D_{[]}(H)$, such that (G,S) and (H,S') are $\oplus$-compatible. Let $\mathrm{ec}_{rQ,l}(G,S) = (F,J,A)$, and let $\mathrm{ec}_{rQ,l}(H,S) = (F',J',A')$, where $A = \{(i,s_i) \mid 1 \leq i \leq l\}$ and $A' = \{(i,s'_i) \mid 1 \leq i \leq l\}$. Then

$$z(S) + z(S') \Leftrightarrow z(S \oplus S') = |\{i \in I_l \mid |N_{S \oplus S', V(G) \cup V(H)}(x_i)| \geq 2 \wedge s_i = 1\}| + |\{i \in I_l \mid |N_{S \oplus S', V(G) \oplus V(H)}(x_i)| \geq 2 \wedge s'_i = 1\}|.$$

It can be seen that this value can be computed from (F,J,A) and (F',J',A'), and hence $\overline{z}$ is a proper extension.

For each $l \geq 0$, let $K_l = 4l$. For each terminal graph G, let F_G be a maximal spanning forest of G such that $\overline{z}(F_G)$ is maximum. A spanning forest F of G is called maximal if each component of F spans a component of G. We show that for each l-terminal graph G and each $S \in D_{[]}(G)$, if S can lead to an optimal solution, then $|\overline{z}(S) \Leftrightarrow \overline{z}(F_G)| \leq K_l/2$. This implies condition 2.

Let G be an l-terminal graph, let $S \in D_{[]}(G)$ and suppose S can lead to an optimal solution. Note that S contains at most l components. We first show that $\overline{z}(S) \leq \overline{z}(F_G) + K_l/2$. Let S' be a maximal spanning forest of G such that S is a subgraph of S'. Note S' can be obtained from S by adding at most $l \Leftrightarrow 1$ edges. Hence $\overline{z}(S') \geq \overline{z}(S) \Leftrightarrow 2(l \Leftrightarrow 1)$. Since $\overline{z}(S') \leq \overline{z}(F_G)$, this implies that $\overline{z}(S) \leq \overline{z}(F_G) + K_l/2$.

We next show that $\overline{z}(S) \geq \overline{z}(F_G) \Leftrightarrow K_l/2$. Let H be an l-terminal graph, and let $S_H \in D_{[]}(H)$ such that $Q(G \oplus H, S \oplus S_H)$ holds and $z(S \oplus S_H) = \Phi(G \oplus H)$. Let G' be the subgraph of $G \oplus H$ with $V(G') = V(G \oplus H)$, and $E(G') = E(F_G) \cup E(S_H)$. The number of vertices of degree one in G' is at least $\overline{z}(F_G) + \overline{z}(S_H) \Leftrightarrow l$. Furthermore, we can obtain from G' a spanning tree T of G by removing a number of edges of G'. This does not decrease the number of vertices of degree one, since if a vertex has one incident edge, then this edge can not be removed. Hence $z(T) \geq \overline{z}(F_G) + \overline{z}(S_H) \Leftrightarrow l$. Since $z(T) \leq \Phi(G \oplus H) = z(S \oplus S_H)$, we can derive the following.

$$\begin{aligned}
\overline{z}(S) &\geq z(S \oplus S_H) \Leftrightarrow \overline{z}(S_H) \\
&\geq z(T) \Leftrightarrow \overline{z}(S_H) \\
&\geq \overline{z}(F_G) + \overline{z}(S_H) \Leftrightarrow l \Leftrightarrow \overline{z}(S_H) \\
&= \overline{z}(F_G) \Leftrightarrow l \\
&\geq \overline{z}(F_G) \Leftrightarrow K_l/2
\end{aligned}$$

This proves condition 2 of Theorem 6.2.2.

As D is a two-vertex-edge-tuple, condition 3 of Theorem 6.2.2 is also satisfied, and hence for each $k \geq 1$, there is a special (parallel) constructive reduction-counter system for MAX LEAF SPANNING TREE on graphs of bounded treewidth. □

7.2 Negative Results

There is a number of (constructive) optimization problems for which we do not succeed in proving conditions 1, 2 (and 3) of Theorem 6.2.2, although the problems are MS-definable, and thus standard methods can be used to solve these problems in $O(n)$ time sequentially or $O(\log n)$ time with $O(n)$ operations in parallel on an EREW PRAM if a tree decomposition of the input graph is given (Section 2.2.4). We show that it is not possible to prove conditions 1 and 2 for these problems, by showing that the problems are not of finite integer index. Indeed, we show that $\sim_{\Phi,l}$ has infinitely many equivalence classes for some $l \geq 0$. We do this by giving an infinite class of graphs and showing that the elements of this class are pairwise not equivalent.

We next show for a number of constructive optimization problems that they are not of finite integer index.

Theorem 7.2.1. *The following problems are not of finite integer index.*

1. MAX CUT
2. MIN COVERING BY CLIQUES
3. LONGEST PATH
4. LONGEST CYCLE

Proof. Let Φ denote the respective optimization problem.

1 MAX CUT. We give an infinite set $\mathcal{G}$ of two-terminal graphs such that for each G and G' in this set, if $G \neq G'$, then $G \not\sim_{\Phi,2} G'$. For each $n \geq 2$, let G_n be a two-terminal graph which is defined as follows (see also Figure 7.1).

$$V(G_n) = X \cup A \cup B_n \cup C_n,$$

where all sets are disjoint, $X = \langle x_1, x_2 \rangle$ is the set of terminals, $A = \{a_1, a_2\}$, and B_n and C_n each contain n vertices, and furthermore,

$$\begin{aligned} E(G_n) = \{\{x_1, a_1\}, \{x_2, a_2\}\} \cup \{\{a_i, v\} \mid 1 \leq i \leq 2 \wedge v \in B_n \cup C_n\} \\ \cup \{\{x_2, b\} \mid b \in B_n\} \cup \{\{x_1, c\} \mid c \in C_n\}. \end{aligned}$$

Let $\mathcal{G} = \{G_n \mid n \geq 2 \wedge n \text{ even}\}$.

Claim. *Let $n \geq 1$, let H be a two-terminal graph. Let (W_1, W_2) be a maximum cut of $G_n \oplus H$ (i.e. (W_1, W_2) partitions $V(G \oplus H)$ and the number of edges with one end point in W_1 and one end point in W_2 is maximum). Either $((W_1 \cap V(H)) + A, (W_2 \cap V(H)) + B_n + C_n)$ or $((W_1 \cap V(H)) + B_n + C_n, (W_2 \cap V(H)) + A)$ is a maximum cut of $G \oplus H$.*

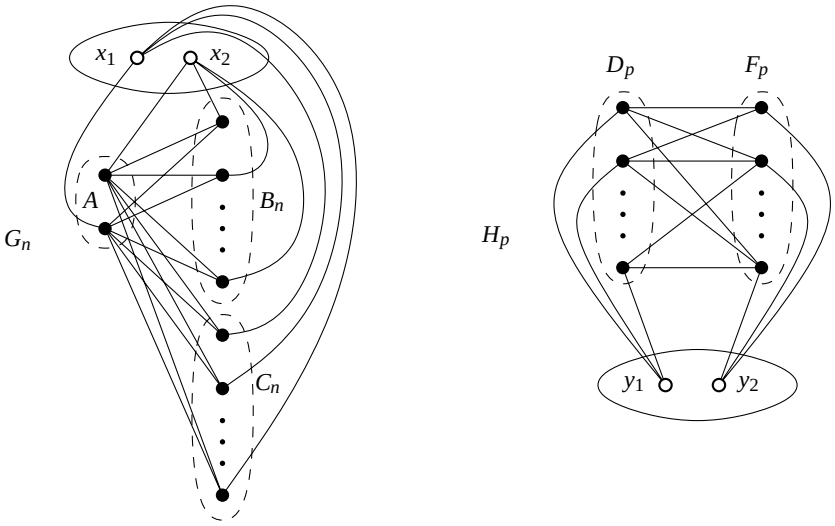


Figure 7.1. The graphs G_n ($n \geq 2$) and H_p ($p \geq 0$) for MAX CUT.

Proof. Let $M^* = z(W_1, W_2)$, i.e. M^* denotes the number of edges in $G_n \oplus H$ with one end point in W_1 and one in W_2. Let

$$\begin{aligned} A_1 &= W_1 \cap A, \qquad & BC_1 &= W_1 \cap (B_n \cup C_n), \\ A_2 &= W_2 \cap A, \qquad & BC_2 &= W_1 \cap (B_n \cup C_n). \end{aligned}$$

Furthermore, let

$$\begin{aligned} (V_1, V_2) &= (W_1 \Leftrightarrow BC_1 + A_2, W_2 \Leftrightarrow A_2 + BC_1), \text{ and} \\ (V_1', V_2') &= (W_1 \Leftrightarrow A_1 + BC_2, W_2 \Leftrightarrow BC_2 + A_1). \end{aligned}$$

Note that (V_1, V_2) and (V_1', V_2') are the cuts mentioned in the claim. We show that either (V_1, V_2) or (V_1', V_2') is a maximum cut. Let $M = z(V_1, V_2)$ and let $M' = z(V_1', V_2')$. We consider two cases, namely

1. $|A_2| = 0 \vee |BC_1| = 0$, and
2. $0 < |A_2| \leq |A|$ and $0 < |BC_1| \leq |BC_1 \cup BC_2|$.

In case 1,

$$\begin{aligned} M &\geq M^* + |A_1| \cdot |BC_1| + |A_2| \cdot |BC_2| \Leftrightarrow |A_2| \Leftrightarrow |BC_1| \\ &= M^* + |A_2|(|BC_2| \Leftrightarrow 1) + |BC_1|(|A_1| \Leftrightarrow 1) \\ &\geq M^*. \end{aligned}$$

In case 2,

$$\begin{aligned} M' &\geq M^* + |A_1| \cdot |BC_1| + |A_2| \cdot |BC_2| \Leftrightarrow |A_1| \Leftrightarrow |BC_2| \\ &= M^* + |A_1|(|BC_1| \Leftrightarrow 1) + |BC_2|(|A_2| \Leftrightarrow 1) \\ &\geq M^*. \end{aligned}$$

This proves the claim. □

For each $p \geq 0$, let H_p be the graph defined as follows (see also Figure 7.1).

$$V(H_p) = Y \cup D_p \cup F_p,$$

where all sets are disjoint, $Y = \langle y_1, y_2 \rangle$ is the set of terminals, D_p and F_p each contain p vertices, and

$$E(H_p) = \{\{d, f\} \mid d \in D_p \wedge f \in F_p\} \cup \{y_1, d\} \mid d \in D_p\} \cup \{\{y_2, f\} \mid f \in F_p\}.$$

Claim. *Let $p \geq 0$, let G be a two-terminal graph, and let (W_1, W_2) be a maximum cut of $G \oplus H_p$. Either $((W_1 \cap V(G)) + D_p, (W_2 \cap V(G)) + F_p)$ or $((W_1 \cap V(G)) + F_p, (W_2 \cap V(G)) + D_p)$ is a maximum cut of $G \oplus H_p$.*

Proof. Similar to the proof of Claim 7.2.1. □

We now show that for each $G_n, G_m \in G$, if $n \neq m$, then $G_n \not\sim_{\Phi,2} G_m$.

For $i \geq 2$, each $p \geq 0$, consider the graph $G_i \oplus H_p$. Claim 7.2.1 and Claim 7.2.2 show that there are eight candidates for maximum cuts in $G_i \oplus H_p$. In the following table, all these cuts are given, together with their values.

nr.	cut , value	
1	$(A \cup D_p \cup X\ ,\ B_i \cup C_i \cup F_p)$	$4i + p^2 + 2i + p$
2	$(A \cup F_p \cup X\ ,\ B_i \cup C_i \cup D_p)$	$4i + p^2 + 2i + p$
3	$(A \cup D_p \cup \{x_1\}\ ,\ B_i \cup C_i \cup F_p \cup \{x_2\})$	$4i + p^2 + i + 1$
4	$(A \cup F_p \cup \{x_1\}\ ,\ B_i \cup C_i \cup D_p \cup \{x_2\})$	$4i + p^2 + i + 1 + 2p$
5	$(A \cup D_p \cup \{x_2\}\ ,\ B_i \cup C_i \cup F_p \cup \{x_1\})$	$4i + p^2 + i + 1 + 2p$
6	$(A \cup F_p \cup \{x_2\}\ ,\ B_i \cup C_i \cup D_p \cup \{x_1\})$	$4i + p^2 + i + 1$
7	$(A \cup D_p\ ,\ B_i \cup C_i \cup F_p \cup X)$	$4i + p^2 + 2 + p$
8	$(A \cup F_p\ ,\ B_i \cup C_i \cup D_p \cup X)$	$4i + p^2 + 2 + p$

Note that either cuts 1 and 2 or cuts 4 and 5 are maximum, since $i \geq 2$, and $p \geq 0$.

Let $n > m > 1$, n, m even. If $p = 0$, then cuts 1 and 2 are maximum for both $G_n \oplus H_0$ and $G_m \oplus H_0$. Hence $\Phi(G_n \oplus H_0) = 6n$ and $\Phi(G_m \oplus H_0) = 6m$, so $\Phi(G_n \oplus H_0) \Leftrightarrow \Phi(G_m \oplus H_0) = 6(n \Leftrightarrow m)$.

Let $p = \frac{1}{2}(n+m) \Leftrightarrow 1$. Then

$$\begin{aligned}\Phi(G_n \oplus H_p) &= 4n + p^2 + \max\{2n + \frac{1}{2}(n+m) \Leftrightarrow 1, n+1+(n+m) \Leftrightarrow 2\} \\ &= 4n + p^2 + \max\{\frac{5}{2}n + \frac{1}{2}m \Leftrightarrow 1, 2n+m \Leftrightarrow 1\} \\ &= 4n + p^2 + \frac{5}{2}n + \frac{1}{2}m \Leftrightarrow 1 \\ &= \frac{13}{2}n + \frac{1}{2}m + p^2 \Leftrightarrow 1,\end{aligned}$$

and

$$\begin{aligned}\Phi(G_m \oplus H_p) &= 4m + p^2 + \max\{2m + \frac{1}{2}(n+m) \Leftrightarrow 1, m+1+(n+m) \Leftrightarrow 2\} \\ &= 4m + p^2 + \max\{\frac{5}{2}m + \frac{1}{2}n \Leftrightarrow 1, 2m+n \Leftrightarrow 1\} \\ &= 4m + p^2 + 2m + n \Leftrightarrow 1 \\ &= 6m + n + p^2 \Leftrightarrow 1\end{aligned}$$

Hence

$$\begin{aligned}\Phi(G_n \oplus H_p) \Leftrightarrow \Phi(G_m \oplus H_p) &= (\frac{13}{2}n + \frac{1}{2}m + p^2 \Leftrightarrow 1) \Leftrightarrow (6m + n + p^2 \Leftrightarrow 1) \\ &= \frac{11}{2}(n \Leftrightarrow m)\end{aligned}$$

However, $\frac{11}{2}(n \Leftrightarrow m) \neq 6(n \Leftrightarrow m) = \Phi(G_n \oplus H_0) \Leftrightarrow \Phi(G_m \oplus H_0)$, since $n \neq m$. So $G_n \not\sim_{\Phi,2} G_m$. As each G_n, $n > 1$, n even, belongs to a different equivalence class of $\sim_{\Phi,2}$, the MAXIMUM CUT problem is not of finite integer index.

2 MIN COVERING BY CLIQUES. For each $n \geq 1$, let G_n be the two-terminal graph with (see also Figure 7.2)

$$\begin{aligned}V(G_n) &= \{x_1, x_2\} \cup \{a_1, \ldots, a_n\}, \text{ and} \\ E(G_n) &= \{\{x_i, a_j\} \mid 1 \leq i \leq 2 \wedge 1 \leq j \leq n\}.\end{aligned}$$

Vertices x_1 and x_2 are the first and the second terminal, respectively.

Let $G = \{G_n \mid n \geq 1\}$. We show that for each $G_n, G_m \in G$, if $n \neq m$, then $G_n \not\sim_{\Phi,2} G_m$.

Let H be the two-terminal graph consisting of terminals y_1 and y_2 and no edges, and let H' be the two-terminal graph consisting of terminals y_1 and y_2 and edge $\{y_1, y_2\}$ (see Figure 7.2).

For each i, $i \geq 1$, $\Phi(G_i \oplus H) = |E(G_i)| = 2i$, since $G_i \oplus H$ contains no cliques of more than two vertices. Furthermore, $\Phi(G_i \oplus H') = |\{\{x_1, x_2, a_j\} \mid 1 \leq j \leq n\}| = i$. This means that for all n and m, $n \neq m$,

$$\Phi(G_n \oplus H) \Leftrightarrow \Phi(G_m \oplus H) = 2n \Leftrightarrow 2m \neq n \Leftrightarrow m = \Phi(G_n \oplus H') \Leftrightarrow \Phi(G_m \oplus H'),$$

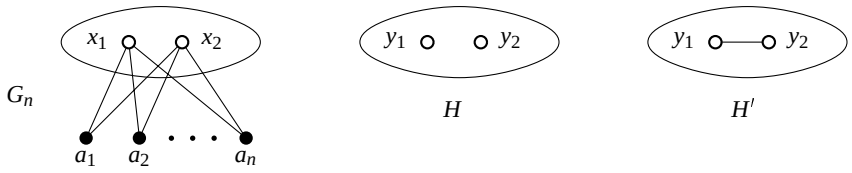


Figure 7.2. The graphs G_n ($n \geq 2$), H and H' for MIN COVERING BY CLIQUES.

and hence $G_n \not\sim_{\Phi,l} G_m$.

3 LONGEST PATH. For each $n \geq 1$, let G_n be the two-terminal graph defined by (see also Figure 7.3)

$$V(G_n) = \{x_1, x_2\} \cup \{a_1, \ldots, a_n\}, \text{ and}$$
$$E(G_n) = \{\{x_1, a_1\}\} \cup \{\{a_i, a_{i+1}\} \mid 1 \leq i < n\}$$

(x_1 and x_2 are the first and the second terminal, respectively). Let $G = \{G_n \mid n \geq 1 \wedge n \text{ even}\}$.

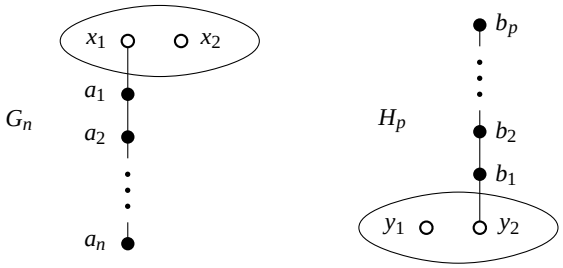


Figure 7.3. The graphs G_n ($n \geq 1$), and H_p ($p \geq 1$) for LONGEST PATH.

Furthermore, for each $p \geq 1$, let H_p be the two-terminal graph with vertex set

$$V(H_p) = \{y_1, y_2\} \cup \{b_1, \ldots, b_p\}, \text{ and}$$
$$E(H_p) = \{\{y_2, b_1\}\} \cup \{\{b_i, b_{i+1}\} \mid 1 \leq i < p\},$$

(y_1 and y_2 are the first and the second terminal, respectively). For each $i \geq 1$, $j \geq 1$, $\Phi(G_i \oplus H_j) = \max\{i, j\}$.

Let $1 \leq n < m$, such that n and m are even. Then $\Phi(G_n \oplus H_{n+1}) \Leftrightarrow \Phi(G_m \oplus H_{n+1}) = n+1 \Leftrightarrow m = n \Leftrightarrow m+1 < 0$. Furthermore, $\Phi(G_n \oplus H_m) \Leftrightarrow \Phi(G_m \oplus H_m) = m \Leftrightarrow m = 0$. Hence $G_n \not\sim_{\Phi,l} G_m$.

4 LONGEST CYCLE. The proof is similar to the proof for LONGEST PATH, but with graphs G_n and H_p as depicted in Figure 7.4. □

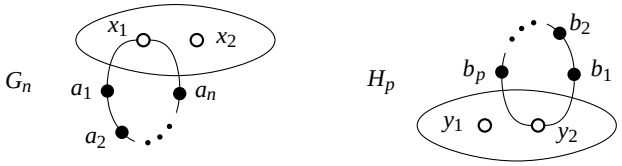


Figure 7.4. The graphs G_n ($n \geq 1$), and H_p ($p \geq 1$) for LONGEST CYCLE.

Chapter 8

Parallel Algorithms for Series-Parallel Graphs

This chapter is concerned with parallel algorithms for recognizing series-parallel graphs and finding sp-trees of series-parallel graphs. We consider four variants of this problem: the input is either a source-sink labeled multigraph which is directed or undirected, or an ordinary multigraph which is directed or undirected. (Definitions and some known results can be found in Section 2.3.3.) The best known sequential algorithms for these problems are constructive reduction algorithms which use $O(m)$ time [Valdes et al., 1982] (m denotes the number of edges of the input graph). We apply the theory of parallel constructive reduction systems as introduced in Section 6.3.1 to obtain efficient parallel algorithms for the problems.

The precise definitions of the problems under considerations are as follows.

SOURCE-SINK LABELED SERIES-PARALLEL GRAPH
Instance: A source-sink labeled multigraph (G,s,t).
Find: An sp-tree of (G,s,t), if (G,s,t) is series-parallel.

For undirected input graphs the problem is denoted by LSPG, and for directed input graphs by DLSPG.

SERIES-PARALLEL GRAPH
Instance: A multigraph G.
Find: An sp-tree of G, if G is series-parallel.

For undirected input graphs the problem is denoted by SPG, and for undirected input graphs by DSPG.

He and Yesha [1987] gave a parallel algorithm for DLSPG and DSPG that uses $O(\log^2 n + \log m)$ time, and $O(n+m)$ processors on an EREW PRAM, and hence $O((n+m)(\log^2 n + \log m))$ operations. The sp-tree that is returned by the algorithm is a binary sp-tree. He [1991] showed that this algorithm can be extended for LSPG and SPG. The resulting algorithms also use $O(\log^2 n + \log m)$ time with $O(n+m)$ processors on an EREW PRAM.

Eppstein [1992] improved this result for simple graphs: his algorithms run in $O(\log n)$ time on a CRCW PRAM with $O(m \cdot \alpha(m,n))$ operations ($\alpha(m,n)$ is the inverse of Ackermann's function, which is at most four for all practical purposes). As any algorithm on a CRCW PRAM can be simulated on an EREW PRAM with a loss of $O(\log n)$ time, this

implies an algorithm with $O(\log^2 n)$ time and $O(m \log n \cdot \alpha(m,n))$ operations on an EREW PRAM.

We improve upon these results, both for the EREW PRAM model and the CRCW PRAM model. We give algorithms which solve LSPG, SPG, DLSPG and DSPG in $O(\log m \log^* m)$ time with $O(m)$ operations on an EREW PRAM, and in $O(\log m)$ time with $O(m)$ operations on a CRCW PRAM. The algorithms make heavy use of the results on constructive reduction algorithms presented in Chapter 6. For LSPG, we apply Theorem 6.3.1: we give a special parallel constructive reduction system for the problem. This system can be used in the parallel algorithm given in Section 6.3.1. This results in algorithms for the EREW and CRCW PRAM model with the stated resource bounds. The algorithms for SPG, for DLSPG and for DSPG are based on these algorithms.

If the input graph is simple, then we can make our algorithms to run in $O(\log n \log^* n)$ on an EREW PRAM and $O(\log n)$ on a CRCW PRAM, and the number of operations is $O(n)$.

As series-parallel graphs have treewidth at most two, we can solve many problems in $O(\log m)$ time with $O(m)$ operations if a tree decomposition of small width of the graph is given, including all all finite state problems (Section 2.2.4). If no tree decomposition is given, then we can first use the results of this chapter to construct a binary sp-tree of the graph, and then transform the sp-tree into a tree decomposition of width two of the graph. We show that this transformation can be done in $O(1)$ time with $O(m)$ operations on an EREW PRAM. Hence we can solve a large class of problems on series-parallel graphs in $O(\log m \log^* m)$ time with $O(m)$ operations on an EREW PRAM and in $O(\log m)$ time with $O(m)$ operations on a CRCW PRAM.

This chapter is organized as follows. Section 8.1 contains some preliminary results. In Section 8.2, we give a special parallel constructive reduction system for the problem LSPG. In Section 8.3, we show that each of the problems LSPG, SPG, DLSPG and DSPG can be solved within the stated resource bounds.

8.1 Preliminary Results

The graphs we consider in this chapter are multigraphs, which means that we use the modified definitions as given in Section 5.4 for reduction rules (Definition 5.1.1) and constructive reduction systems (Definition 6.1.1). Definitions of series-parallel graphs and sp-trees, and some preliminaries can be found in Section 2.3.3.

We give a number of simple or well-known lemmas on series-parallel graphs.

Lemma 8.1.1. *Let G be a series-parallel graph and let T be an sp-tree of G. If α and β are nodes of T, α is an ancestor of β, and the labels of α and β both contain a vertex v, then all nodes on the path between α and β in T contain v in their label.*

Lemma 8.1.2. *If (G,s,t) is a series-parallel graph, then $(G+\{s,t\},s,t)$ is a series-parallel graph, where $G+\{s,t\}$ is the graph obtained by adding an (extra) edge between s and t to G.*

Proof. This follows from the parallel composition of G with a one-edge series-parallel graph. □

Lemma 8.1.3. *If (G,s,t) is a series-parallel graph with sp-tree T_G, and there is a node α in T_G labeled with (u,v), then $(G+\{u,v\},s,t)$ is a series-parallel graph.*

Proof. Suppose G_α is the series-parallel graph associated with node α. Add between α and its parent a p-node β which has two children, namely node α and a leaf node representing the added edge $\{u,v\}$. The new tree is an sp-tree of $(G+\{u,v\},s,t)$. □

Lemma 8.1.4. *Let G be a series-parallel graph, T an sp-tree of G, and $u,v \in V(G)$. The nodes in T which are labeled with (u,v) induce a (possibly empty) subtree of T.*

As shown in Lemma 2.3.5, any series-parallel graph has treewidth at most two. From the construction in the proof of Lemma 2.3.5 is easy to see that any binary sp-tree of G can be transformed into a tree decomposition of width at most two of G in $O(1)$ time with $O(m)$ operations on an EREW PRAM.

In the following lemmas, we frequently use the fact that a series-parallel graph can not have K_4 as a minor (which follows from Lemma 2.3.5 and Lemma 2.2.8).

Lemma 8.1.5. *Let (G,s,t) be a series-parallel graph.*

1. *If there is a node α with label (x,y) in an sp-tree of G, then there is a path P in G with $P=(s,\ldots,x,\ldots,y,\ldots,t)$.*
2. *If there is a node with label (x,y) in an sp-tree of G that is an ancestor of a node with label (v,w), then there is a path $(s,\ldots,x,\ldots,v,\ldots,w,\ldots,y,\ldots,t)$ in G.*
3. *For every edge $e=\{x,y\}\in E(G)$, there is a path $(s,\ldots,x,y,\ldots,t)$, or there is a path $(s,\ldots,y,x,\ldots,t)$ in G.*

Proof.

1. We prove that for any node β with label (v,w) on the path from α to the root of the sp-tree of G, there is a path $(v,\ldots,x,\ldots,y,\ldots,w)$ in the graph G_β associated with node β. We use induction on the length of the path from α to β in the sp-tree. (Using this result with β the root of the sp-tree gives the desired result.)

First, suppose $\alpha=\beta$. As any series-parallel graph is connected, there is a path from v to w in the series-parallel graph associated with node α.

Next, suppose β is an ancestor of α, and has label (v,w). Let γ be the child of β on the path from α to β. If β is a p-node, then the label of γ is also (v,w). By the induction hypothesis, there is a path $(v,\ldots,x,\ldots,y,\ldots,w)$ in the graph associated with γ, and the result follows for β. Suppose β is an s-node with children $\delta_1,\ldots,\delta_r$, and δ_i has label (v_i,v_{i+1}) for each i, $1\le i\le r$. Let j, $1\le j\le r$, be such that $\delta_j=\gamma$. For any i, $1\le i\le r$, there is a path P_i from v_i to v_{i+1} in G_{δ_i} (the graph associated with δ_i). By the induction hypothesis, there is a path $P_j=(v_j,\ldots,x,\ldots,y,\ldots,v_{j+1})$ in G_{δ_j}. Concatenating $P_1,P_2,\ldots,P_r$ gives the required path of the form $(v,\ldots,x,\ldots,y,\ldots,w)$ in G_β.

2. Similar.

3. Note that there is a node with label (x,y) or a node with label (y,x). Now we can use part 1 of this lemma. □

Lemma 8.1.6. *Let (G,s,t) be series-parallel and suppose there is a path $(s,\ldots,x,y,\ldots,t)$ in G. The following holds.*

1. *There is no path from s to y that avoids x or there is no path from x to t that avoids y.*
2. *No node in any sp-tree of G is labeled with the pair (y,x).*

Proof.
1. Suppose not. Then $(G+\{s,t\},s,t)$ contains K_4 as a minor, which is a contradiction.

2. This follows from part 1 of this lemma and Lemma 8.1.5. □

Lemma 8.1.7. *Suppose (G,s,t) is a series-parallel graph with $G=(V,E)$, and let $\{x,y\}\in E$. Suppose there is a path $(s,\ldots,x,y,\ldots,t)$ in G. Let W be the set*

$$W=\{v\in V-\{x,y\} \mid \textit{there is a path } (s,\ldots,x,\ldots,v,\ldots,y,\ldots,t) \textit{ in } G\}.$$

Then the following holds.

1. *For all $\{v,w\}\in E$, $v\in W$ implies that $w\in W\cup\{x,y\}$.*
2. *For every sp-tree of G, if a node is labeled with (v,w) or (w,v), and $v\in W$, then $w\in W\cup\{x,y\}$.*
3. *Let T be an sp-tree of G, let α be the highest node with label (x,y). The series-parallel graph G_α associated with α is exactly the graph $G[W\cup\{x,y\}]$. Furthermore, if $|W|\geq 1$, then α is a parallel node.*

Proof.
1. Suppose $\{v,w\}\in E$, $v\in W$, $w\notin\{x,y\}$. By Lemma 8.1.5, there is a path $(s,\ldots,v,w,\ldots,t)$ or there is a path $(s,\ldots,w,v,\ldots,t)$.

Suppose there is a path $(s,\ldots,v,w,\ldots,t)$. If the subpath from s to v avoids x and y, then $G+\{s,t\}$ contains K_4 as a minor, contradiction. Hence either x or y belongs to the path from s to v. Similarly, x or y belongs to the part of the path from w to t. If y appears on the first part, and x appears on the last part, then we have a contradiction with Lemma 8.1.6. Hence, we have a path of the form $(s,\ldots,x,\ldots,v,w,\ldots,y,\ldots,t)$. This implies that $w\in W$.

The case in which there is a path $(s,\ldots,w,v,\ldots,t)$ is similar.

2. Note that if a node in the sp-tree of G is labeled with (v,w), then $G+\{v,w\}$ is also a series-parallel graph (Lemma 8.1.3). Hence, the result follows from part 1 of the lemma.

3. We first show that G_α is a subgraph of $G[W\cup\{x,y\}]$. Let $v\in V(G_\alpha)$. There is a descendant β of α which contains v in its label. According to Lemma 8.1.5, there is a path $(s,\ldots,x,\ldots,v,\ldots,y,\ldots,t)$, so $v\in W$.

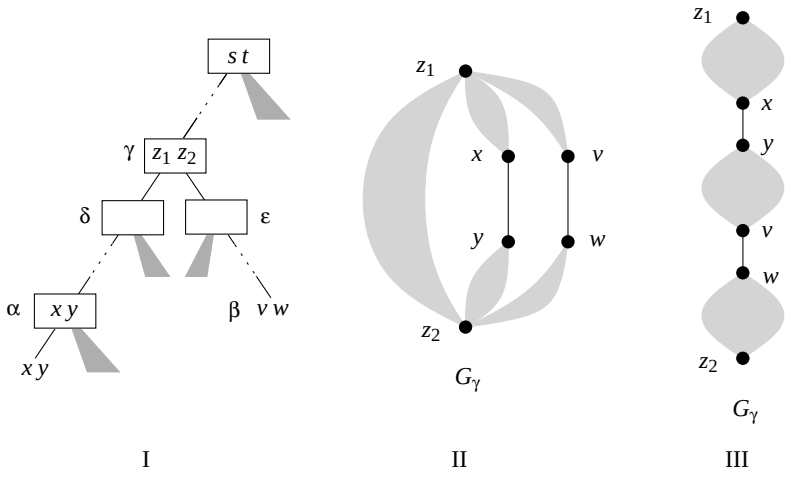


Figure 8.1. The sp-tree and possible graphs for the proof of Lemma 8.1.7

Next we show that $G[W \cup \{x,y\}]$ is a subgraph of G_α. Let $e = \{v,w\} \in E(G[W \cup \{x,y\}])$, let β be the leaf node of e, and suppose w.l.o.g. that β has label (v,w). We show that β is a descendant of α. If $e = \{x,y\}$, this clearly holds.

Suppose $e \neq \{x,y\}$ and β is not a descendant of α. Then we have a node γ, with label $(z_1,z_2) \neq (x,y)$, with children δ and ε, such that α is equal to or a descendant of δ, and β is equal to or a descendant of ε (see Figure 8.1, part I).

If $z_1 \in W$, then G contains a path from s to x that avoids z_1, and G contains a path from z_1 to y that avoids x. Also, G contains a path $(s,\ldots,z_1,\ldots,x,y)$, hence $G + \{s,t\}$ contains a K_4 minor, contradiction. So, we may assume that $z_1 \notin W$, and similarly, that $z_2 \notin W$.

First suppose that γ is a p-node. Figure 8.1, part II shows the structure of the series-parallel graph G_γ associated with node γ. The graph G_ε associated with ε contains a path $(z_1,\ldots,x,y,\ldots,z_2)$, because of Lemma 8.1.5, part 2. Similarly, the graph G_δ associated with node δ contains a path $(z_1,\ldots,v,w,\ldots,z_2)$. Since the only common vertices of G_ε and G_δ are z_1 and z_2, there is a path $(x,\ldots,z_1,\ldots,v,\ldots,z_2,\ldots,y)$ in G. Since $(x,y) \neq (z_1,z_2)$ and $z_1,z_2 \notin W$, this means that this path contains an edge between a vertex in W and a vertex in $V - W - \{x,y\}$, which is in contradiction with part 1 of this lemma.

Suppose γ is an s-node, and suppose that node δ is on the left side of node ε. Part III of Figure 8.1 shows the structure of the series-parallel graph G_γ. There is no path $(z_1,\ldots,v,\ldots,y)$ in G_γ, which means that any path in G which goes from x to y and contains v must look like $(x,\ldots,z_1,\ldots,z_2,\ldots,v,\ldots,y)$. This again means that there is an edge between a vertex in W and a vertex in $V - W - \{x,y\}$, contradiction. If δ is on the right side of ε, then in the same way, we have a path $(x,\ldots,v,\ldots,z_1,\ldots,z_2,\ldots,y)$. This is again a contradiction. Hence β is a descendant of α. This proves that $G_\alpha = G[W \cup \{x,w\}]$.

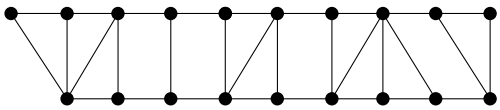


Figure 8.2. A graph with only two matches to the series and the parallel reduction rule.

If α is an s-node, then it is the only node with label (x,y). This is impossible, because there is a leaf node with label (x,y). If α is a leaf node, then G_α consists only of the edge $\{x,y\}$. Hence if $|W| \geq 1$, then α is a p-node. This completes the proof of part 3. □

8.2 A Special Parallel Constructive Reduction System

In this section, we give a special parallel constructive reduction system for LSPG(see Definition 6.3.2), called $S_{\text{sp}} = (R_{\text{sp}}, I_{\text{sp}}, A_R^{\text{sp}}, A_I^{\text{sp}})$. Algorithms A_R^{sp} and A_I^{sp} will be made in such a way that the constructed sp-tree will be a minimal sp-tree of the graph. In Section 8.2.1, we give a set R_{sp} of reduction rules and a set I_{sp} of graphs, and we show that the set R_{sp} is safe for LSPG. In Section 8.2.2, we give the construction algorithms A_R^{sp} and A_I^{sp} and show that they use $O(1)$ time and that algorithm A_R^{sp} is non-interfering. After that, in Section 8.2.3, we show that in each series-parallel graph (G,s,t) with at least two edges, there are $\Omega(|E(G)|)$ matches to rules in R_{sp}. In Section 8.2.4, we extend this result to discoverable matches. Together, these results show that $(R_{\text{sp}}, I_{\text{sp}}, A_R^{\text{sp}}, A_I^{\text{sp}})$ is a special parallel constructive reduction system for LSPG.

8.2.1 A Safe Set of Reduction Rules

Duffin [1965] has shown that the system $S = (R, I)$, where R contains the series and the parallel rule (see Figure 2.11), and I contains the base series-parallel graph (which consists of one edge), is a reduction system for series-parallel graphs. Valdes et al. [1982] have given a constructive reduction algorithm for series-parallel graphs, based on this reduction system: they have shown how S can be used to recognize series-parallel graphs in $O(m)$ time, and to build an sp-tree of the input graph within the same time bounds.

For an efficient parallel algorithm, the series and the parallel rule are not sufficient: for example, the graph shown in Figure 8.2 is series-parallel, but it contains at most two matches to the series and parallel rule. Moreover, if we apply a sequence of series and parallel reductions on this graph, then at each point in the sequence, the current graph contains at most four matches. We can make an arbitrarily large graph of this type, and it takes $\Omega(m)$ reduction rounds to reduce this graph to a single edge if we only have the series and the parallel rule. Therefore, we introduce a larger set of reduction rules. Let R_{sp} be the set of 18 reduction rules depicted in Figure 8.3. Rules 1 and 2 are the series and the parallel rule, rules 3 – 18 are for the graphs of the type depicted in Figure 8.2. Note that each of the rules 3 – 18 can be applied by contracting one or two edges. These edges are marked gray in Figure 8.3.

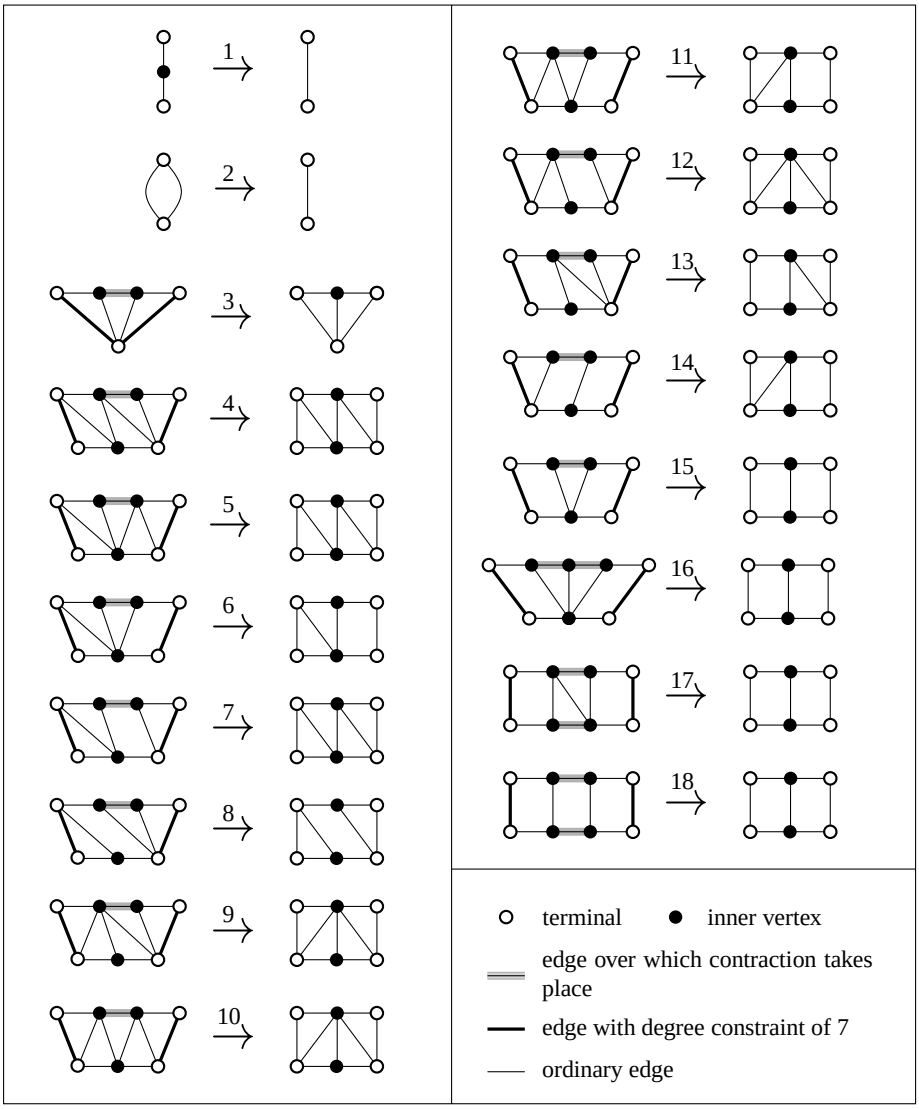


Figure 8.3. Reduction rules for series-parallel graphs

In rules 3 – 18, we pose *degree constraints* on the edges between terminals: if we apply one of the rules 3 – 18 to a graph G, then in the match H that is involved in the reduction, for each edge between two terminals H, at least one of the end points of this edge has degree at most seven in G . (Note that all inner vertices of left-hand sides of rules 3 – 18 also have degree at most seven). In Figure 8.3, the fat edges denote the edges with a degree constraint of seven. The degree constraints are useful for proving that sufficiently many applications of the reduction rules can be found.

The reduction rules will be applied on source-sink labeled graphs. To assure that the graph will remain a source-sink labeled graph during a sequence of applications of reduction rules, we require that a reduction is only performed if the source and sink of a the graph are not inner vertices of the match corresponding to the reduction. With these two extra requirements, we get a new definition of a match.

Definition 8.2.1 (Match). Let $r = (H_1, H_2)$ be a reduction rule in R_{sp}. Let (G, s, t) be a source-sink labeled graph. A *match* to r in (G, s, t) is a terminal graph G_1 which is isomorphic to H_1, such that

- there is a terminal graph G_2 with $G = G_1 \oplus G_2$,
- s and t are not inner vertices of G_1, and
- if r is one of the rules 3 – 18, then for each edge $e = \{u, v\} \in E(G_1)$ for which u and v are terminals of G_1, u or v has degree at most seven in G.

Let I_{sp} contain the series-parallel graph consisting of one edge between source s and sink t. The above discussion shows that, if R_{sp} is safe, then R_{sp} is complete and that I_{sp} contains all irreducible series-parallel graphs. It can also be seen that R_{sp} is decreasing.

In the following four lemmas, we show that R_{sp} is safe for series-parallel graphs, which completes the proof that $(R_{\text{sp}}, I_{\text{sp}})$ is a reduction system for series-parallel graphs. The proofs of the lemmas are given in such a way that they can be used for the design of algorithm A_R^{sp}.

Lemma 8.2.1. *If (G', s, t) is obtained from (G, s, t) by applying rule 1, then (G, s, t) is a series-parallel graph if and only if (G', s, t) is a series-parallel graph.*

Proof. Suppose G' is obtained by removing vertex c of degree two, and adding an edge between its two neighboring vertices a and b. Suppose we have a minimal sp-tree for G'. There must be a leaf α with label (a, b) or (b, a). Suppose w.l.o.g. that α has label (a, b). If α's parent is a p-node (see Figure 8.4, right-hand side of case i), then α is replaced by an s-node with two children, successively labeled (a, c) and (c, b) (left-hand side of case i). (The light-gray parts in the figure denote the parts of the sp-tree that are involved in the modification.) The resulting tree is a minimal sp-tree of (G, s, t). If α's parent is an s-node (see Figure 8.4, right-hand side of case ii), then replace α by two leaf nodes, successively labeled (a, c) and (c, b) (left-hand side of case ii).

Suppose we have a minimal sp-tree for G. As c is not a terminal, and not s or t, there must be a series composition that composed $\{b, c\}$ and $\{c, a\}$. This means that we have a subtree as depicted in the left-hand side of case i or case ii of Figure 8.4. Hence the modification above can be reversed. □

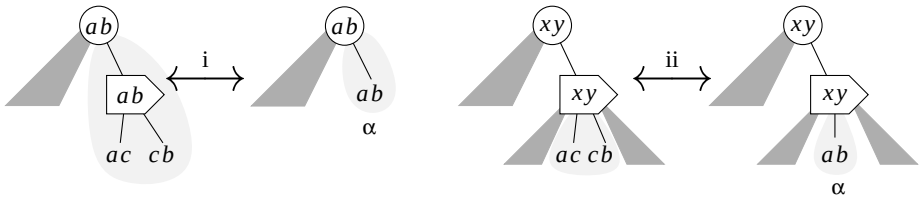


Figure 8.4. Transformation of sp-tree for rule 1.

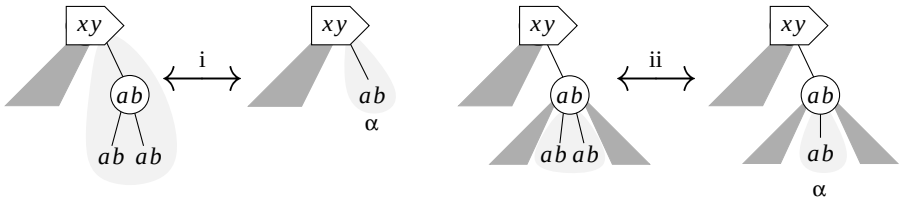


Figure 8.5. Transformation of sp-tree for rule 2.

Lemma 8.2.2. *If (G',s,t) is obtained from (G,s,t) by applying rule 2, then (G,s,t) is a series-parallel graph if and only if (G',s,t) is a series-parallel graph.*

Proof. Suppose (G',s,t) is obtained by removing edge e_2 from G, where e_2 is parallel to edge e_1. If we have an sp-tree for G', then this tree has a leaf node α which corresponds to e_2 (and hence the end points of e_1 are in its label). Suppose α is labeled (a,b). If α's parent is an s-node (see Figure 8.5, right-hand side of case i), then replace α by a p-node with two leaf children, both labeled (a,b) (left-hand side of case i). The resulting tree is a minimal sp-tree for (G,s,t). If the parent of α is a p-node (right-hand side of case ii in Figure 8.5), then attach an additional leaf below this parent, with label (a,b) (left-hand side of case ii).

Suppose we have an sp-tree for (G,s,t). This tree contains a leaf node α corresponding to edge e_2. Hence we have a subtree as depicted in the left-hand side of case i or case ii of Figure 8.5. Let β denote α's parent. Remove α from the tree. If β has only one child left (case i), then β is removed and its child is directly attached to the parent of β. □

Note that in a match to one of the rules 3 – 18, edges matching an edge between two terminals in the left-hand side of the rule can have parallel edges in the graph, but edges matching an edge in the left-hand side of the rule with at least one end point an inner vertex can not have a parallel edge.

Lemma 8.2.3. *Suppose (G',s,t) is obtained from (G,s,t) by one application of rule 3. Then (G,s,t) is a series-parallel graph if and only if (G',s,t) is a series-parallel graph.*

Proof. Suppose (G,s,t) is a series-parallel graph, and let T be the minimal sp-tree of (G,s,t). Let H be the match to rule 3, as depicted in the left-hand side of Figure 8.6. Suppose H is

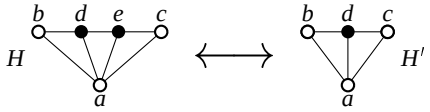


Figure 8.6. Matches to left-hand and right-hand sides of rule 3.

replaced by H', which is depicted in the right-hand side of Figure 8.6. Consider a path P from s to t that uses the edge $\{a,b\}$. We distinguish between two cases, namely the case that P visits a before b, and the case that P visits b before a.

Case 1. Suppose that the path P visits a before b. We distinguish between two further cases, namely the case that P avoids e and the case that P visits e.

Case 1.1. Suppose that P avoids vertex e. Let

$$W = \{v \in V \mid \text{there is a path } (s,\dots,a,\dots,v,\dots,b,\dots,t), \text{ and } v \text{ belongs to the same component as } e \text{ in } G[V-\{a,b\}]\}.$$

Note that $c,d,e \in W$, and hence (by part 1 of Lemma 8.1.7), all vertices in the component of $G[V-\{a,b\}]$ which contains e are in W. There must be a parallel node α in T with label (a,b), with the subgraph containing the nodes in W 'below it' (see part 3 of Lemma 8.1.7). Let G_α be the graph associated with α. Each vertex $v \neq a,b$ of G_α can occur in at most one graph associated with one of the children of α.

Let β be the s-node that is a child of α such that the series-parallel graph G_β associated with β contains e. We claim that G_β is the graph obtained from $G[W \cup \{a,b\}]$ by deleting all edges between a and b. If a vertex $w \in W$ is not in G_β, then all paths from w to e use a or b, which means that w is not in the component of $G[V-\{a,b\}]$ which contains e. Hence $w \in V(G_\beta)$. Hence each vertex of W occurs only in G_β, which means that all edges between vertices in W and in $W \cup \{a,b\}$ are in G_β.

On the other hand, if there is a vertex $x \in V(G_\beta)$, $x \notin \{a,b\}$, then there is a path $P = (a,\dots,x,\dots,b)$ in G_β (Lemma 8.1.5). If P contains no vertex from W, then β is not a series node. Hence P contains a vertex from W. Together with part 1 of Lemma 8.1.7, this means that all vertices on P are in $W \cup \{a,b\}$, so $x \in W$. The graph G_β can not contain an edge between a and b, since then β is not an s-node. This proves the claim.

Suppose β has children with labels $(a,x_1),(x_1,x_2),\dots,(x_t,b)$, respectively. We show that $t = 1$ and $x_1 = x_t = c$. Suppose not. First suppose that $x_t \neq c$. Add an edge between x_t and b; this again gives a series-parallel graph. Now, by contracting all nodes in W except c to d, we get a K_4 minor, contradiction. Hence $x_t = c$. Now suppose that $t > 1$. There is a leaf node with label (a,c) or label (c,a) which is a descendant of β, since there is an edge $\{a,c\}$. But vertex a occurs only in the labels of the subtree of the child of β with label (a,x_1). Furthermore, vertex c occurs only in the labels of the subtrees of the children of β with labels (c,b) and (x_{t-1},c). Since $x_1 \neq c$ and $x_{t-1} \neq a$, this means that there can be no leaf node with label (a,c)

or (c,a), which gives a contradiction. So $t = 1$, the children of β have labels (a,c) and (c,b), respectively. It can be seen that the child with label (c,b) is a leaf node, corresponding to edge $\{b,c\}$. By straightforward deduction, it follows that the sp-tree of G has the tree from the left-hand side of Figure 8.7, case i as a subtree. We can replace the light-gray part of this subtree by the light-gray part of the subtree shown in the right-hand side of this case and get an sp-tree of G'.

Case 1.2. Suppose the path P from s to t that uses the edge $\{a,b\}$, also uses node e. There are a two different cases, namely the case that P visits e before a, and the case that P visits e after b. In the first case, $G+\{s,t\}$ is series parallel, but contains K_4 as a minor, contradiction. In the second case, we have a path $(s,\ldots,a,e,\ldots,t)$, that does not use b. This case can be analyzed in exactly the same way as the cases above, leading to a subtree transformation, as shown in Figure 8.7, case iii.

Case 2. Suppose that the path P visits b before a. This case can be dealt with in the same way as Case 1, only with directions reversed. See Figure 8.7, cases ii and iv.

This ends 'only if' part of the proof. The 'if' part is very similar. In this case, the same transformations as above are done, but in opposite direction. □

Lemma 8.2.4. *Suppose (G',s,t) is obtained from (G,s,t) by one application of one of the rules 4 – 18. Then (G,s,t) is a series-parallel graph if and only if (G',s,t) is a series-parallel graph.*

Proof. The proof is similar to the proof of Lemma 8.2.3. Suppose (G,s,t) is a series-parallel graph, and let T be a minimal sp-tree of (G,s,t). Let H be the match to one of the rules 4 – 18 and let the terminals of H be named a, b, c and d, as shown in Figure 8.8 for the case that H is a match to rule 4 (i.e. the terminals are named a, b, c, and d, such that a and b are adjacent in H, and c and d are adjacent in H, and furthermore, if we 'walk around' H clockwise, starting at terminal a, then we visit the terminals in the order a,b,d,c).

Consider a path P from s to t in G that uses the edge $\{a,b\}$. First suppose P visits a before b. We distinguish four cases.

Case 1. P does not use vertices c and d. We can show that (G',s,t) is series-parallel in the same way as in Case 1.1 in the proof of Lemma 8.2.3 (define W to be the vertices of the component of $G[V-\{a,b\}]$ which contains c and d).

Case 2. P uses c but not d. Then either c is on the subpath $(s,\ldots,a)$ of P or c is on the subpath $(b,\ldots,t)$ of P. In both cases, $G+\{s,t\}$ contains a K_4 minor, which gives a contradiction.

Case 3. P uses d but not c. This case is similar to Case 2, and hence gives a contradiction.

Case 4. P uses both c and d. If c and d both occur on the subpath $(s,\ldots,a)$ of P, or on the subpath $(b,\ldots,t)$ of P, then $G+\{s,t\}$ contains a K_4 minor.

If $P=(s,\ldots,d,\ldots,a,b,\ldots,c,\ldots,t)$, then $G+\{s,t\}$ also contains a K_4 minor.

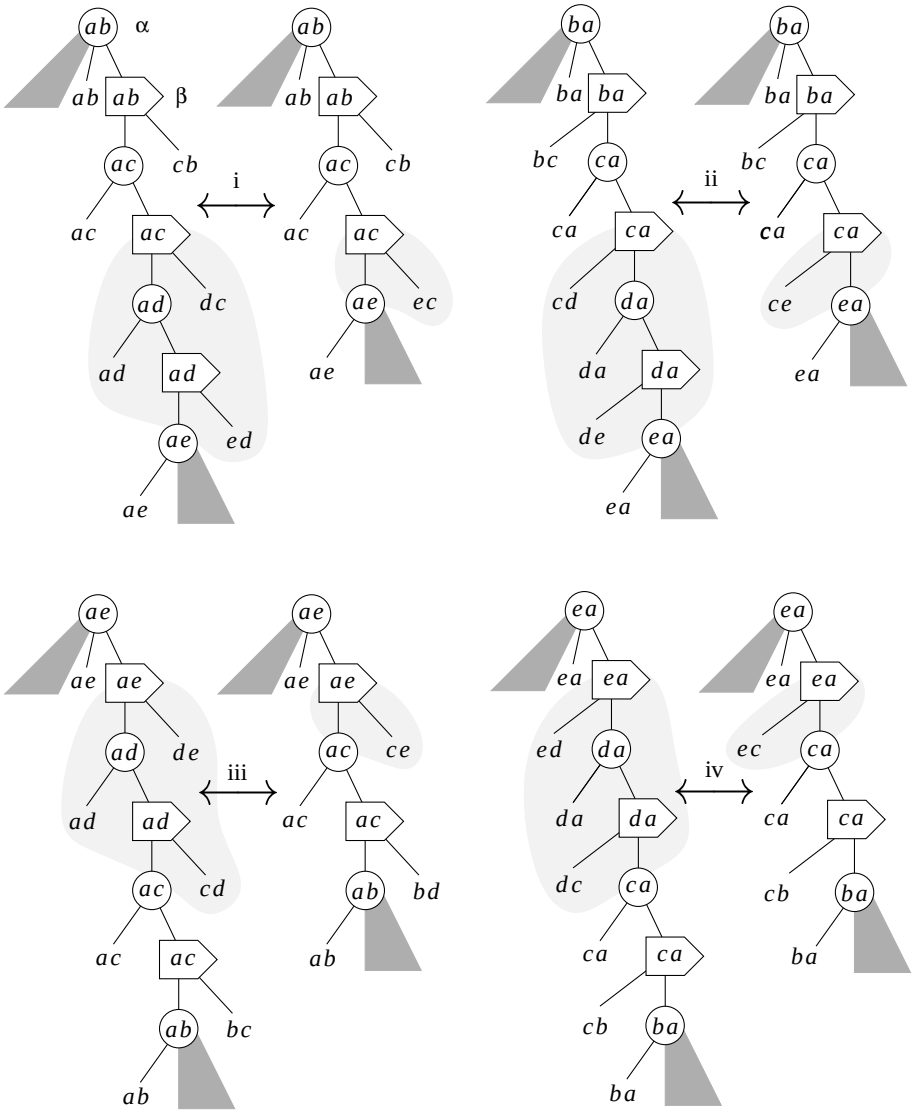


Figure 8.7. Transformations of subtrees for rule 3.

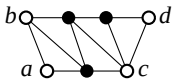


Figure 8.8. Match H to rule 4.

If $P = (s,\ldots,c,\ldots,a,b,\ldots,d,\ldots,t)$ then there is a path from s to t that uses the edge $\{c,d\}$, and does not use a and b. This case is similar to Case 1.

The case that P visits vertex b before a can be solved in the same way. This ends the 'only if' part of the proof. The 'if' part can be handled in the same way.

For the proof of rules 5 – 18, we can apply exactly the same technique. □

We conclude the following result.

Corollary 8.2.1. $(R_{\text{sp}}, I_{\text{sp}})$ *is a reduction system for* LSPG.

8.2.2 The Construction Algorithms

We now give algorithms A_R^{sp} and A_I^{sp}. We first describe the data structure that we use to store sp-trees. We make a list of all nodes in the sp-tree. Each node is marked with its label and its type (s-node, p-node or leaf node), each node has a pointer to its left-most and its right-most child, to its parent, and to its neighboring siblings on the left-hand and the right-hand side (if one of these nodes does not exist, the pointer is nil). Furthermore, each leaf node is marked with the type of its parent, and we keep a pointer from each edge in the graph to the corresponding leaf node in the sp-tree.

As we want to prove that S_{sp} is a special parallel constructive reduction system (Definition 6.3.2), we have to show that A_I^{sp} and A_R^{sp} use $O(1)$ time, and that A_R^{sp} is non-interfering, i.e. if A_R^{sp} is executed simultaneously for two or more different non-interfering reductions on the same graph, then the two executions do not read or write concurrently in the same memory position, and the resulting sp-tree is the same as when the executions were applied consecutively.

Algorithm A_I^{sp} is easy: given a source-sink labeled graph (G,s,t) consisting of one edge, it constructs the sp-tree of (G,s,t) consisting of one leaf-node, labeled with (s,t).

For algorithm A_R^{sp}, we use the constructions from the proofs of Lemmas 8.2.1 – 8.2.4 (see also Figures 8.4 – 8.7). We show that algorithm A_R^{sp} can do its construction in $O(1)$ time without interference. Given a reduction rule $r = (H_1,H_2) \in R_{\text{sp}}$, terminal graphs G_1 and G_2 such that G_1 and H_1 are isomorphic and G_2 and H_2 are isomorphic, a graph G such that $G = G_2 \oplus H$, and a minimal sp-tree T of G, algorithm A_R^{sp} does the following.

First the algorithm finds the local structure of the sp-tree, i.e. it finds the structure of the part of the sp-tree that contains edges in G_2. For rules 1 and 2, the different forms are the right-hand sides of cases i and ii in Figures 8.4 and 8.5, respectively. For rule 3, the different forms are the right-hand sides of cases i, ii, iii and iv in Figure 8.7. The parts of the sp-tree

that are marked light-gray are the parts that must be modified. The local structure is found as follows. Take an edge e of G_2 which is not an edge between two terminals in the case of rules 3 – 18 (for rules 1 and 2, the only possibility is the edge $\{a,b\}$, for rule 3, edge $\{c,e\}$ is the best edge to take, as this edge will be removed). Look at the corresponding leaf node in T. For rules 1 and 2, check the type of its parent node, and for rule 3 – 18, search the 'neighborhood' of this leaf node in T which is involved in the modification (for rule 3, this is the light-gray part in the right-hand side of cases i, ii, iii and iv in Figure 8.7). The leaf node can be found in constant time without interfering with any other constructions. For rules 1 and 2, it is clear that we can check the type of its parent in constant time without interfering with other constructions performed at the same time, as each leaf node is marked with the type of its parent. For rule 3, we can see from Figure 8.7 that the structure of the neighborhood can be determined in $O(1)$ time without interfering with other constructions, as no other construction involves any of the nodes of the light-gray part of the sp-tree. For rules 4 – 18, the cases are similar to the cases of rule 3, and the structure can also be found in $O(1)$ time without interference.

After the local structure of the sp-tree is found, this part of the sp-tree is replaced by a new part. The structure of this new part depends on the structure of the old part. For rules 1, 2 and 3, these new parts are the parts in left-hand sides of the cases in Figures 8.4, 8.5 and 8.7 that are marked light-gray. For rules 4 – 18, a similar approach as for rule 3 can be taken. For rules 3 – 18, it is easy to see that the modification can be done in $O(1)$ time without interference. For rules 1 and 2, case i is also easy (see Figures 8.4 and 8.5: node α gets a different type, and gets two leaf children). In case ii, the modification needs more care, as the neighboring siblings of node α may be leaf nodes that are involved in another rule 1 or 2 reduction at the same time. Hence we have to ensure that the corresponding executions do not read from or write to the same memory location at the same time. This is done in the following way.

A new leaf-node β is added as the neighboring sibling on the right-hand side of the leaf node α. For rule 1, nodes α and β get labels (a,c) and (c,b), respectively, and for rule 2 they both get label (a,b). Clearly, the construction takes $O(1)$ time. As each leaf node that is a sibling of node α and is involved in a reduction of rule 1 or 2 adds a new sibling on its right-hand side, we can make sure that no two of these constructions concurrently read from or write to the same memory location, and that the result is correct. Hence the algorithm is non-interfering.

This completes the description of algorithms A_I^{sp} and A_R^{sp}, and the proof that they use $O(1)$ time and that A_R^{sp} is non-interfering. Together with the fact that $(R_{\text{sp}}, I_{\text{sp}})$ is a reduction system for LSPG, this also implies the following result.

Lemma 8.2.5. *$S_{\text{sp}} = (R_{\text{sp}}, I_{\text{sp}}, A_R^{\text{sp}}, A_I^{\text{sp}})$ is a constructive reduction system for* LSPG.

8.2.3 A Lower Bound on the Number of Matches

In this section we show that each series-parallel graph (G,s,t) with at least two edges contains at least $\Omega(|E(G)|)$ matches to rules in R_{sp}.

Lemma 8.2.6. *Let (G,s,t) be a series-parallel graph with $|E(G)| \geq 2$. (G,s,t) contains at least $|E(G)|/139$ matches to rules 1 – 18.*

Proof. Consider the minimal sp-tree T of G. The number of leaves of T equals $|E(G)|$. We argue that the number of leaves of T is at most equal to 139 times the number of matches. To obtain this, we distinguish the following 'classes' of leaves.

A leaf node α in T is *good* if it is a child of a parallel node and has at least one sibling which is a leaf (i.e. α is child of a parallel node which has at least two leaf children), or it is a child of a series node and one of α's neighboring siblings also is a leaf node (i.e. α is child of a series node which has at least two successive leaf children of which α is one). Note that the edges that correspond to good leaf nodes occur in matches to rule 1 or 2.

An internal node in T is *green* if it has at least one good leaf child.

A node in T is *branching* if it is an internal node, and has at least two internal nodes as its children.

A leaf is *bad* if it is not good, and its parent is branching or green. Most edges that correspond to bad leaves can not occur in any match.

Note that the leaf children of a branching node which is not green are all bad, the leaf children of a green p-node are all good, and the leaf children of a green s-node are either bad or good.

Now consider the other nodes in T. An internal node is *blue* if it is not branching or green, but it has a descendant that is branching or green at distance at most 33.

An internal node is *yellow* if it is not branching, green or blue.

The total number of leaves in T equals the number of good leaves plus the number of bad leaves plus the number of leaf children of blue nodes plus the number of leaf children of yellow nodes. We now derive an upper bound for the number of leaves in each of these classes, in terms of the number of matches.

Good leaves. If a green p-node has m good leaves, then the edges corresponding to its good leaves correspond to at least $m(m-1)/2$ matches to rule 2. If a green s-node has m good leaves, then the edges corresponding to these leaves correspond to at least $m/2$ matches to rule 1. Hence the number of good leaves is at most twice the number of applications of reduction rules 1 and 2.

Bad leaves.

Claim 8.2.1. *The number of bad leaves is at most three times the number of branching nodes plus twice the number of green nodes.*

Proof. Let α be a bad leaf. If α's parent is a p-node, then account α to its parent (which has at most one bad leaf). If α's parent is an s-node, then account α to its neighboring sibling on the right if it has one, or to its parent otherwise. In this way, each branching node has at most two leaves accounted to it: at most one of its children and possibly its neighboring sibling on the left. Each green node has at most one bad leaf accounted to it: a green p-node has no bad leaf children, hence can only have a neighboring bad sibling on the left accounted to

it; a green s-node has at most one bad leaf accounted to it, and it has a p-node as a parent, which means that it has no bad siblings accounted to it. Each yellow or blue node which has a yellow or blue parent does not have any bad leaves accounted to it. Each yellow or blue node which has a branching or green parent has at most one bad leaf accounted to it, namely its neighboring sibling on the left.

Let β be a yellow or blue node which has a bad leaf accounted to it. It must be the case that β has a branching or green parent. Let γ be the highest descendant of β which is green or branching. Note that there exists such a node γ. All nodes on the path from β to γ, except γ, are yellow or blue. Hence no node on this path, except β and γ, has a bad leaf accounted to it, as none of these nodes has a branching or green parent. Account the bad leaf that is accounted to β, to γ instead. This way, each branching or green node has at most one extra bad leaf accounted to it, and hence each branching node has at most three leaves accounted to it, and each green node has at most two leaves accounted to it. □

In each green node, there is a match to rule 1 or 2 in two of the edges corresponding to its good leaves. Hence the number of green nodes is at most equal to the number of matches to rules 1 and 2. We now bound the number of branching and blue nodes by the number of green nodes in order to bound the number of bad leaves.

Claim 8.2.2. *The number of branching nodes is at most the number of green nodes.*

Proof. Construct a tree T' from T by removing all nodes that are not green and not branching, while preserving successor-relationships. Note that, in T, every internal node that has only leaves as child is green, hence every branching node still has at least two children in T'. Moreover, every leaf of T' is green. Since, in any tree, the number of internal nodes with two or more children is at most the number of leaves, the number of branching nodes is at most the number of green nodes in T', and hence in T. □

Claims 8.2.1 and 8.2.2 show that the number of bad leaves is at most equal to $3+2=5$ times the number of green nodes, which is at most 5 times the number of matches to rules 1 and 2.

Leaves of blue nodes. The number of blue nodes is at most 33 times the number of branching and green nodes: account each blue node to the closest descendant which is branching or green. Since the number of branching nodes is at most the number of green nodes, this means that the number of blue nodes is at most $2 \cdot 33 = 66$ times the number of green nodes. Each blue node has at most two leaf children, which means that the number of leaves of blue nodes is at most $2 \cdot 66 = 132$ times the number of matches to rules 1 and 2.

Leaves of yellow nodes. Consider a path in T which consists of 33 successive yellow and blue nodes, such that the highest node in this path is a parallel node. Each node in this path either is a p-node with as its children one leaf node and one s-node, or it is an s-node with as its children one p-node and one or two non-neighboring leaf nodes.

The edges associated to the leaves that are a child of the nodes in this path form a subgraph of G of a special form: they form a sequence of 16 cycles of length three or four, each sharing

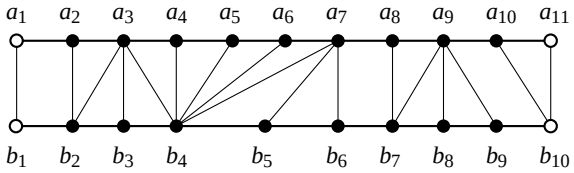


Figure 8.9. Subgraph of G corresponding to a path of 33 yellow or blue nodes in the sp-tree, of which the highest one is a p-node with label (a_1, b_1), and the lowest one is a p-node with label (a_{11}, b_{10}). Only a_1, b_1, a_{11} and b_{10} may be incident with edges outside the subgraph.

one edge with the previous cycle, and one edge with the next (except of course for the first and last cycle in the sequence); three successive cycles do not share a common edge. As no series node on the path has two successive leaf nodes, we have that the shared edges of a cycle of length four do not have a vertex in common. We call such a subgraph a *cycle-sequence*. See Figure 8.9 for an example.

Consider a sequence $\alpha_1, \alpha_2, \ldots, \alpha_n$ of n successive yellow and blue nodes starting and ending with a p-node, and its corresponding cycle-sequence (α_1 is the node that is closest to the root). For each i, let (x_i, y_i) denote the label of α_i.

Note that for each $i < j$, if $x_i = x_j$, then for each l, $i < l < j$, $x_i = x_l$. Furthermore, if $x_i \neq x_{i+1}$, then α_i must be an s-node, and α_{i+1} has at least one sibling on its left side. As α_{i+1} is a yellow or blue node, it has exactly one sibling on the left, which is a leaf node with label (x_i, x_{i+1}), hence there is an edge $\{x_i, x_{i+1}\} \in E(G)$. This shows that the sequence $x_1, x_2, \ldots, x_n$ and the sequence $y_1, y_2, \ldots, y_n$ both form a path in G. We call them *bounding paths* of the cycle-sequence. In Figure 8.9 for example, we have a cycle-sequence consisting of 16 cycles, with bounding paths $(a_1, a_2, \ldots, a_{11})$ and $(b_1, b_2, \ldots, b_{10})$. The length of a bounding path is the number of edges on this path.

Note that the left-hand and right-hand sides of rules 3 – 18 are cycle-sequences. Before proving that a cycle-sequence with 16 cycles contains a match to one of the rules 3 – 18, we first prove the following.

Claim 8.2.3. *Any cycle-sequence with one bounding path of length at least two and one bounding path of length at least three contains as a subsequence the left-hand side of one of the rules 3 – 18.*

Proof. Let C be a cycle-sequence with one bounding path of length two or more and one of length three or more. The left-hand sides of rules 4 – 15 represent exactly the cycle-sequences with one bounding path of length two and one of length three which do not contain the left-hand side of rule 3 as a subsequence. The left-hand sides of rules 17 and 18 represent exactly the cycle-sequences with two bounding paths of length three which do not contain the left-hand side of one of the rules 3 – 15. Hence if C contains a subsequence with one bounding

path of length three and one of length two or three, then it contains the left-hand side of one of the rules 3 – 15, 17 or 18.

Now suppose C does not contain such a subsequence. We show that it contains the left-hand side of rule 16. The shortest of the two bounding paths has length at least two and the longest one has length at least three. Remove one of the outermost cycles of the cycle-sequence until one of these conditions would be violated by removing another outermost cycle. Let P_1 be the shortest bounding path and P_2 the longest bounding path of the obtained cycle-sequence.

If P_1 has length four or more, then we can remove another outer-cycle, as this decreases the length of P_1 and P_2 by at most one. Hence P_1 has length at most three. If P_1 has length three, then P_2 must have length three, otherwise we can remove another outer-cycle. But that means that it contains the left-hand side of one of the rules 4 – 15, 17 and 18 as a subsequence. Hence P_1 has length two. If P_2 has length three, then the sequence contains the left-hand side of one of the rules 4 – 15, hence P_2 has length four or more. Note that the first and the last vertex of P_1 are adjacent to only one vertex of P_2, otherwise we can remove another outer cycle. If P_2 has length five or more, then the middle vertex of P_1 has at least four neighbors in P_2, and hence C contains the left-hand side of rule 3. This means that P_2 has length four. Then the outermost cycles must be squares, otherwise the middle vertex of P_1 still has at least four neighbors in P_2. But that means that the cycle-sequence is equal to the left-hand side of rule 16. This proves the claim. □

We can now prove the following claim.

Claim 8.2.4. *In a cycle-sequence of G that consists of 16 cycles, there is a match to one of the rules 3 – 18.*

Proof. Let C be a cycle-sequence in G consisting of 16 cycles, and let P_1 and P_2 denote the bounding paths. If C contains a cycle-sequence of five successive triangles with one vertex in common as its subsequence, as in Figure 8.10, then it contains a match to rule 3 (formed by the middle three triangles). Suppose such a subsequence does not exist. It follows that the edge between the fifth and sixth three- or four-cycle in the sequence does not have an end point that is also end point of an edge not in the subgraph; similarly for the edge between the 11th and 12th cycle. Consider the cycle-sequence C' formed by the sixth cycle up to and including the 11th cycle in C (C' consists of six cycles). As each vertex of C is contained in at most six three- or four-cycles of C, and none of the vertices in C is incident with an edge outside C in G, all vertices of C' have degree at most seven in G. This means that if C' contains as a subsequence one of the left-hand sides of rules 4 – 18, then it contains a match to the corresponding rule. Let P_1' and P_2' be the bounding paths of C' and suppose P_1' has length m and P_2' has length n. We now show that the cycle-sequence contains a match to one of the rules 4 – 18.

Suppose w.l.o.g. that $m \leq n$. We first show that $m \geq 2$ and $n \geq 3$. If $m = 0$, then the only vertex of P_1' occurs in six triangles, and we have a match to rule 3. If $m = 1$, then one of the vertices of P_1' occurs in three triangles, and we have again a match to rule 3. Suppose $m \geq 2$.

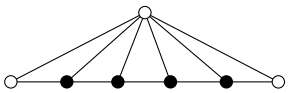


Figure 8.10. Five successive triangles with one vertex in common

If $m = n = 2$, then the cycle-sequence consists of at most four cycles. Hence $n \geq 3$ and $m \geq 2$. Claim 8.2.3 shows that the cycle-sequence contains a left-hand side of one of the rules 3 – 18 as a subsequence, and hence it contains a match to one of these rules. □

In a sequence of 34 successive yellow and blue nodes in T, we can find one path of 33 successive yellow and blue nodes, such that the highest node in this path is a p-node. We can find a number of disjoint paths of 34 successive yellow and blue nodes, such that each yellow node is in exactly one such path. This means that the largest number of disjoint paths of successive yellow and blue nodes of length 34 that we can find in T is at least $1/34$ times the number of yellow nodes. Hence the number of matches to rules 3 – 18 is at least $1/34$ times the number of yellow nodes. Since each yellow node has at most two leaf children, we have that the number of leaf children of yellow nodes is at most $2 \cdot 34 = 68$ times the number of matches to rules 3 – 18.

The total number of leaves in T is now at most $2 + 5 + 132 = 139$ times the number of matches to rules 1 and 2 plus 68 times the number of matches to rules 3 – 18. Hence the number of leaves in T is at most 139 times the number of matches in R_{sp}. This completes the proof. □

8.2.4 A Lower Bound on the Number of Discoverable Matches

In this section, we complete the proof that S_{sp} is a special parallel constructive reduction system. As we have already shown that A_I^{sp} and A_R^{sp} run in $O(1)$ time and that A_R^{sp} is non-interfering, we only have to show that $(R_{\text{sp}}, I_{\text{sp}})$ is a special parallel reduction system for series-parallel graphs (see Definition 5.4.3). We have to show that there are integers n_{min} and d, $n_{min} \leq 19 \leq d$, and there is a constant $c > 0$, such that each series-parallel graph (G,s,t) with $|V(G)| + |E(G)| \geq n_{min}$, given by some adjacency list representation, has at least $c \cdot (|V(G)| + |E(G)|)$ d-discoverable matches (Definition 5.4.1) in R_{sp} (with respect to d). All other conditions of a special parallel reduction system are satisfied.

Let $n_{min} = 4$ and $d = 20$. As each series-parallel graph is connected and hence $n = O(m)$, it suffices to show that each series-parallel graph with at least two edges contains $\Omega(m)$ d-discoverable matches. As we have already shown that each series-parallel graph with at least two edges contains $\Omega(m)$ matches, we only have to show that sufficiently many of these matches are d-discoverable.

Note that a match to one of the rules 1 or 3 – 18 is always a d-discoverable match. A match to rule 2 is not always d-discoverable. Let (G,s,t) be a source-sink labeled graph given by some adjacency list, and suppose H is a match to rule 2 in (G,s,t) with $V(H) = \{u,v\}$ and

$E(H) = \{e_1, e_2\}$. Then H is a d-discoverable match if and only if in the adjacency list of u or v, edges e_1 and e_2 have distance at most 20.

Let G be a graph given by some adjacency list representation. Let $e \in E(G)$ and suppose e has end points u and v. Edge e is called *bad* if it has a parallel edge, but all parallel edges have distance at least 21 to e in the adjacency lists of u and of v. Note that an edge that has parallel edges is bad if and only if it is not contained in a match to rule 2 that is d-discoverable.

Lemma 8.2.7. *Let G be a multigraph of treewidth at most two given by some adjacency list representation. There are at most $|E(G)|/5$ bad edges in G.*

Proof. Consider a tree decomposition (T, X) of G of width at most two with $T = (I, F)$ and $X = \{X_i \mid i \in I\}$, and choose an arbitrary node $i \in I$ as root of T. For a $v \in V$, let r_v be the highest node in T with $v \in X_{r_v}$. Let $e \in E$ with end points v and w. There is a node containing v and w, hence either $r_v = r_w$, or r_v is an ancestor of r_w, or r_w is an ancestor of r_v.

For every bad edge between v and w, associate the edge with v if $r_v = r_w$, or r_w is an ancestor of r_v; otherwise, associate the edge with w. Suppose bad edge e between v and w is associated with v. Then X_{r_v} must contain both v and w. It follows that there are at most $|X_{r_v}| - 1 \leq 2$ different vertices u for which bad edges between v and u can be associated with v (namely, the vertices in $X_{r_v} - \{v\}$). For each such u, each 20 successive positions in the (cyclic) adjacency list of v can contain at most one bad edge between u and v, hence there are at most $\deg(v)/20$ bad edges between v and u that are associated with v, and hence in total, at most $\deg(v)/10$ bad edges are associated with v. The stated bound is derived by taking the sum over all vertices. □

As each series-parallel graph has treewidth at most two, it follows that each series-parallel graph (G, s, t) has at most $|E(G)|/5$ bad edges.

Lemma 8.2.8. *There is a constant $c > 0$ for which each series-parallel graph (G, s, t) with at least two edges contains at least $c|E(G)|$ d-discoverable matches in G.*

Proof. Let $n = |V(G)|$ and $m = |E(G)|$. We distinguish two cases, namely that case that $m \geq 4n$ and the case that $m < 4n$. If $m \geq 4n$, then there are at least $m - 2n$ edges that are parallel to another edge, of which at most $m/5$ are bad. Hence, there are at least $4/5m - 2n \geq 4/5m - 1/2m = 3m/10$ edges e for which there is a parallel edge which has distance at most 20 to e in the adjacency list of one of the end points of e. This implies that there are at least $3m/20$ d-discoverable matches to rule 2 in (G, s, t).

Suppose $m < 4n$. We now apply Lemma 8.2.6 above on the simple graph underlying G. Let G' be obtained from G by removing all second and further occurrences of parallel edges. Note that (G', s, t) is a series-parallel graph, and G' has at least $n - 1$ edges. If G' has one edge, then G consists of two vertices with $m \leq 8$ parallel edges, and hence G contains at least one d-discoverable match to rule 2. This means that if G' has one edge, then G has at least $m/8$ d-discoverable matches.

Suppose G' has at least two edges. By Lemma 8.2.6 there are at least $(n-1)/139 \geq n/278$ matches to rules 1 and 3 – 18 in (G', s, t). As each of the matches to rules 1 and 3 – 18 is

d-discoverable in G', this implies that (G', s, t) has at least $n/278$ d-discoverable matches. For each match in this set, there are two possibilities: either it is also a d-discoverable match in G, or it is disturbed by the addition of one or more parallel edges. We will call a match of the first type a *non-disturbed* match, and a match of the last type a *disturbed* match. We now show that the number of disturbed matches is at most k times the number of matches to rule 2 in G, for some positive integer k.

Consider a disturbed match H. There are two cases: either an inner vertex v of H is incident with parallel edges, or a terminal vertex v which has degree at most seven in G' has degree more than seven in G (and hence is incident with parallel edges). In both cases, the vertex v has degree at most seven, and hence in G, there is a d-discoverable match to rule 2 which contains vertex v: any sublist of length 20 of the adjacency list contains at least two edges with the same end points, as there at most seven different sets of end points possible.

Account each disturbed match in G' to a d-discoverable match to rule 2 in G which contains a vertex of degree at most seven of the disturbed match. By the discussion above, this is possible. We show that each d-discoverable match to rule 2 has at most a constant number of disturbed matches accounted to it. Let v be a vertex of G which has degree at most seven. Each match containing v contains only vertices and edges which are reachable by a path from v in G' of length at most seven (the maximum number of vertices in any left-hand side of a rule, minus one) through vertices of degree at most seven (except possibly the last vertex of the path). There are at most a constant number of vertices which can be reached from v by such a path, and thus there are at most a constant k number of matches containing v. Consider a d-discoverable match H to rule 2 in G, with $V(H) = \{u, v\}$. Each disturbed match which is accounted to H contains either vertex u, in which case u has degree at most seven in G', or vertex v, in which case v has degree at most seven in G'. Hence at most $2k$ disturbed matches are accounted to H.

Consider the number of d-discoverable matches in G. This number is at least equal to the number of non-disturbed matches plus the number of d-discoverable matches to rule 2 in G, which is at least the number of non-disturbed matches plus $1/(2k)$ times the number of disturbed matches. Hence the number of d-discoverable matches in G is at least $1/(2k)$ times the number of d-discoverable matches in G'. This latter number is at least $n/278$, and hence there are at least $n/(556k)$ d-discoverable matches. As $m < 4n$, this means that there are at least $m/(2224k)$ d-discoverable matches in G. □

Note that the constant c in Lemma 8.2.8 is quite bad. However, the bound we have derived can probably be tightened by using more detailed estimates.

We have proved the following result.

Theorem 8.2.1. *S_{sp} is a special parallel constructive reduction system for* LSPG.

8.3 Algorithms

In this section, we show that the problems LSPG, SPG, DLSPG and DSPG can be solved efficiently in parallel. We also show that a large number of problems on series-parallel graphs can be solved efficiently in parallel.

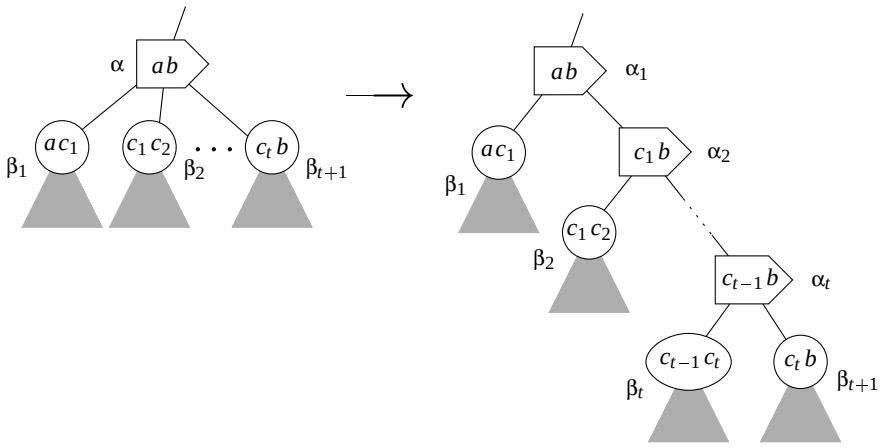


Figure 8.11. Transformation of s-node α in a minimal sp-tree to several s-nodes in a binary sp-tree.

Theorem 8.2.1 and Theorem 6.3.1 show that we have an algorithm which, given a source-sink labeled graph (G,s,t), finds a minimal sp-tree of (G,s,t), if one exists. The algorithm uses $O(\log m \log^* m)$ time on an EREW PRAM and $O(\log m)$ time on a CRCW PRAM, both with $O(m)$ operations and space. If we want a binary sp-tree instead of a minimal sp-tree, then we can slightly modify algorithm A_R^{sp} of the reduction system S_{sp} such that the constructed sp-tree is binary. The proofs of Lemmas 8.2.1 – 8.2.4 can easily be modified such that the modified algorithm A_R^{sp} is still non-interfering and runs in $O(1)$ time.

Another way to compute a binary sp-tree is to first compute a minimal sp-tree, and then transform this tree into a binary sp-tree. This transformation can be done as follows. Each s- or p-node α with children $\beta_1, \beta_2, \ldots, \beta_{t+1}$ is split into nodes $\alpha_1, \alpha_2, \ldots, \alpha_t$ of the same type. Figure 8.11 shows how this transformation is done for the case that α is an s-node. It can be seen that if we do this transformation for all nodes in the sp-tree, then the resulting tree is a binary sp-tree of the graph. We can not do this transformation for all nodes in parallel: it gives a problem if a node is transformed at the same time as its parent or one of its children. To this end, we first compute for each node its distance to the root node. This can be done in $O(\log m)$ time with $O(m)$ operations on an EREW PRAM (see e.g. JáJá [1992]). After this, first all nodes with even distance are transformed, and then all nodes with odd distance are transformed. In this way, a node is not transformed at the same time as its parent or one of its children. Both transformations can be done in $O(1)$ time with $O(m)$ operations. Hence the complete transformation takes $O(\log m)$ time with $O(m)$ operations.

Theorem 8.3.1. *The following problems can be solved with $O(m)$ operations and space in $O(\log m \log^* m)$ time on an EREW PRAM and in $O(\log m)$ time on a CRCW PRAM: given a source-sink labeled graph (G,s,t), determine whether (G,s,t) is series-parallel, and if so,*

find a minimal or binary sp-tree of (G,s,t).

In the remainder of this section, we show that the algorithm can also be used to solve the problem for directed series-parallel graphs, and for series-parallel graphs without specified source and sink. Also, it can be used as a first step to solve many other problems on series-parallel graphs.

First, suppose we are given a graph G, and want to determine whether G is series-parallel with a proper choice of the source and sink. We solve this problem by first computing a source and a sink and then solving the problem with this source and sink. He [1991] and Eppstein [1992] have shown (using results from Duffin [1965]) that this problem reduces in a direct way to the problem with specified vertices, as the following result holds.

Lemma 8.3.1 [He, 1991; Eppstein, 1992]. *Let $G=(V,E)$ be a graph. If G is series-parallel then the following holds.*

1. *If G is not biconnected, then the blocks of G form a path: each cut vertex of G is in exactly two blocks, all blocks have at most two cut vertices, and there are exactly two blocks which contain one cut vertex.*
2. *The graph (G,s,t) is series-parallel if s and t are vertices of G chosen as follows.*
   (a) *If G is biconnected, then s and t are adjacent.*
   (b) *If G is not biconnected, then let B_1 and B_2 be the blocks of G which contain one cut vertex, and let c_1 and c_2 denote these cut vertices. Source s is a vertex of B_1 which is adjacent to c_1, and sink t is a vertex of B_2 which is adjacent to c_2.*

We next show how s and t can be found such that they satisfy conditions 2a and 2b of Lemma 8.3.1. Therefore, we apply Theorem 6.3.2: we give a special parallel constructive reduction system for the problem. However, Theorem 6.3.2 does not apply for multigraphs, and our input graph is a multigraph. Therefore, we make a new, simple graph $G'=(V',E')$ from the multigraph G as follows.

$$V' = V(G) + E(G)$$
$$E' = \{\{v,e\} \mid v \in V(G) \wedge e \in E(G) \wedge v \text{ is incident with } e\}$$

We make a labeling of the vertices in G': each vertex originating from $V(G)$ is labeled vertex, and each vertex originating from $E(G)$ is labeled edge. It is easy to see that the resulting graph is a simple graph and has $n+m$ vertices and $2m$ edges, and furthermore, if G is series-parallel, then G' is series-parallel. The transformation can be performed in $O(1)$ time with $O(n+m)$ operations.

We define a construction property for the new type of graph. Note that for a multigraph G and the simple graph G' obtained from G as described above, each non-trivial block of G corresponds to a block of G', and each block of G consisting of one edge e corresponds to two trivial blocks of G' which are connected to each other by the cut vertex e. Hence part 2 of Lemma 8.3.1 is equivalent to the following statement.

Lemma 8.3.2. *If $G = (V,E)$ is a series-parallel graph, then (G,s,t) is series-parallel if s and t are vertices with label* vertex *in G' chosen as follows.*

1. *If G' is biconnected, then s and t have distance two.*
2. *If G' is not biconnected, then let B_1 and B_2 be the blocks of G with one cut vertex, and let c_1 and c_2 denote these vertices. Then s is in B_1, and if B_1 is non-trivial, s has distance two to c_1, otherwise, s is adjacent to c_1. Furthermore, t is in B_2, and if B_2 is non-trivial then t has distance two to c_2, otherwise, t is adjacent to c_2.*

Let P be the construction property defined by (D,Q), where D and Q are defined as follows. For each graph G, $D(G)$ is the set of all pairs of vertices which are both labeled vertex, and for each $(s,t) \in D(G)$, $Q(G,(s,t))$ holds if and only if s and t satisfy conditions 1 and 2 of Lemma 8.3.2. It can be seen that there is an MSOL predicate for Q (using techniques from e.g. Borie et al. [1992]), and that D is a two-vertex-edge-tuple. As all MS-definable properties are of finite index, we can apply Theorem 6.3.2 to the problem, with a bound of two on the treewidth. This results in a parallel algorithm that uses $O(\log m \log^* m)$ time with $O(m)$ operations and space on an EREW PRAM, and $O(\log m)$ time with $O(m)$ operations and space on a CRCW PRAM. While the resulting algorithm will probably not be efficient, this result does not rely on non-constructive arguing. (We expect that it is also possible to find s and t in the following way. First reduce the graph using the set R_{sp} of reduction rules, without taking care of source and sink. Then in the reduced graph, which consists of one edge if the graph is series-parallel, make s and t the end points of this edge. After that, undo the reductions in reversed order and reconstruct s and t in a proper way.)

If the input graph is a source-sink labeled directed graph (G,s,t), then one can use the modification, described by Eppstein [1992]: solve the problem on the underlying undirected graph, then orient the edges with help of the minimal sp-tree (there is at most one possible orientation for which the directed graph is series-parallel), and check if this orientation corresponds to the original graph.

If the input graph is directed, and no source and sink are specified, then there must be exactly one vertex with indegree zero and one with outdegree zero, otherwise, the graph is not series-parallel. Let the source be this first vertex, and the sink the latter vertex, and solve the problem for the graph with this source and sink. Note that these vertices can be found in $O(\log m)$ time with $O(m)$ operations on an EREW PRAM.

Theorem 8.3.2. *Each of the following problems can be solved with $O(m)$ operations, in $O(\log m \log^* m)$ time on an EREW PRAM, and $O(\log m)$ time on a CRCW PRAM.*

1. *Given a graph G, determine if there are $s,t \in V(G)$ for which (G,s,t) is series-parallel, and if so, find an sp-tree of G.*
2. *Given a directed source-sink labeled graph (G,s,t), determine whether (G,s,t) is series-parallel, and if so, find an sp-tree of (G,s,t).*
3. *Given a directed graph G, determine if there are $s,t \in V(G)$ for which (G,s,t) is series-parallel, and if so, find an sp-tree of G.*

If the input graph is simple, then we can make the algorithms to run in $O(\log n \log^* n)$ time on an EREW PRAM and $O(\log n)$ time on a CRCW PRAM, both with $O(n)$ operations and space. This can be done by doing the following preprocessing step. Note that if $|E| > 2|V|$ for some simple graph $G = (V, E)$, then G has treewidth more than two (Lemma 2.2.6), and hence G is not series-parallel. If $|E| \leq 2|V|$, then the number of edges can be counted in $O(\log n)$ time with $O(n)$ operations and space on an EREW PRAM. Therefore, we start counting the number of edges of the graph, but we do at most $O(\log n)$ steps of this counting, with $O(n)$ operations. If, after these steps, the edges are counted and $|E| \leq 2|V|$, then we go on with the rest of the algorithm. Otherwise, we can conclude that $|E| > 2|V|$, and hence the input graph does not have treewidth at most two and is not series-parallel. In this case, we return false.

As we have mentioned before (Section 2.2.4), many problems can be solved in $O(\log p)$ time, and $O(p)$ operations and space, when the input graph is given together with a tree decomposition of bounded treewidth consisting of p nodes. These include all (constructive) decision problems and (constructive) optimization problems that are MS-definable. Since series-parallel graphs have treewidth at most two, we can solve these problems efficiently on series-parallel graphs, if a tree decomposition of small width is given. A binary sp-tree of a series-parallel graph can be transformed into a tree decomposition of width at most two in constant time, by using the construction of Lemma 2.3.5. Hence we have the following result.

Corollary 8.3.1. *The following problem can be solved in $O(\log m \log^* m)$ time, $O(m)$ operations, and $O(m)$ space on an EREW PRAM, and in $O(\log m)$ time, $O(m)$ operations and $O(m)$ space on a CRCW PRAM: given a series-parallel graph G, find a tree decomposition of width at most two of G.*

The resulting tree decomposition has $O(m)$ nodes. Hence we can solve the problems described above in $O(\log m)$ time with $O(m)$ operations given this tree decomposition.

Chapter 9

Parallel Algorithms for Treewidth Two

This chapter is concerned with parallel algorithms for the problem of finding a tree decomposition of width at most two of a graph, if one exists, or in other words, for the constructive version of 2-TREEWIDTH. We consider both simple graphs and multigraphs. The best known parallel algorithm for recognizing simple graphs of treewidth at most k is due to Bodlaender and Hagerup [1995]. It uses $O(n)$ operations and $O(\log n)$ time on a CRCW PRAM or $O(\log n \log^* n)$ time on an EREW PRAM. Bodlaender and Hagerup also give a parallel algorithm for building a tree decomposition of width at most k, if one exists. This algorithm uses $O(n)$ operations and $O(\log^2 n)$ time both on an EREW PRAM and on a CRCW PRAM. Related, earlier results can be found e.g. in Granot and Skorin-Kapov [1991] and Lagergren [1996].

For 1-TREEWIDTH there is a more efficient algorithm than the one of Bodlaender and Hagerup [1995]. A connected simple graph of treewidth one is a tree, and a tree can be recognized by using a tree contraction algorithm. This takes $O(\log n)$ time with $O(n)$ operations on an EREW PRAM [Abrahamson, Dadoun, Kirkpatrick, and Przytycka, 1989]. One can easily construct a tree decomposition of a tree in $O(1)$ time with $O(n)$ operations on an EREW PRAM. The algorithm can be modified such that it can be used on input graphs which are not necessarily connected (see also Section 9.3).

In this chapter, we improve the result of Bodlaender and Hagerup [1995] for treewidth two. We give an algorithm that constructs a tree decomposition of width at most two of a given multigraph, if one exists. This algorithm runs in $O(\log m)$ time on a CRCW PRAM and $O(\log m \log^* m)$ time on an EREW PRAM, in both cases with $O(m)$ operations and space. We modify the algorithm for simple graphs, such that the m can be replaced by n in the time and operations bound. The algorithm makes use of the relation between graphs of treewidth at most two and series-parallel graphs: we extend the constructive reduction system that is given in Chapter 8 for series-parallel graphs, such that it can be used for graphs of treewidth at most two.

This chapter is organized as follows. In Section 9.1 we give some preliminary results and definitions that will be used in the remainder of the chapter. In Section 9.2 we give a constructive reduction system for 2-TREEWIDTH, assuming a special class of input graphs, namely connected multigraphs of which some edges are labeled. We also show that the system is a special parallel constructive reduction system for this problem, thus implying that the constructive version of 2-TREEWIDTH can be solved with $O(m)$ operations in $O(\log m)$ time

on a CRCW PRAM or in $O(\log m \log^* m)$ time on an EREW PRAM. In Section 9.3, we show how this reduction system can be used to solve the constructive version of 2-TREEWIDTH on simple graphs or multigraphs which are not necessarily connected.

9.1 Preliminary Results

The following result on trees is used in Section 9.2.2.

Lemma 9.1.1. *Let H be a tree. Let $l(H)$ denote the number of leaves of H, and let $nr(H)$ denote the sum of the degrees of all vertices of degree at least three. Then $nr(H) \leq 3l(H)$.*

Proof. We prove this by induction on the number n of vertices of H. If $n \leq 2$, then clearly $nr(H) \leq 3l(H)$.

Suppose $n > 2$. Let v be a leaf of H, and let w be the only neighbor of v. Let d denote the degree of w in H and note that $d \geq 2$. Furthermore, let $H' = H[V - \{v\}]$. By the induction hypothesis, $nr(H') \leq 3l(H')$. If $d = 2$, then $l(H) = l(H')$ and $nr(H) = nr(H')$, so $nr(H) \leq 3l(H)$. If $d = 3$, then $l(H) = l(H') + 1$ and $nr(H) = nr(H') + 3$, and thus $nr(H) \leq 3l(H)$. If $d \geq 4$, then $l(H) = l(H') + 1$ and $nr(H) = nr(H') + 1$, and hence also $nr(H) \leq 3l(H)$. □

In Section 9.2, we extend the reduction system for series-parallel graphs as it is given in Chapter 8 for graphs of treewidth at most two. However, series-parallel graphs are connected multigraphs, and we are especially interested in simple graphs which are not necessarily connected. Therefore, we first make a reduction system for connected multigraphs of treewidth at most two instead of simple graphs. In Section 9.3 we show how the constructive reduction algorithm based on this system can be adapted such that it can be used for simple graphs which are not necessarily connected.

Lemma 9.1.2. *A multigraph G has treewidth at most two if and only if each block of G is series-parallel.*

Proof. Suppose G has treewidth at most two. Let G' be a block of G (G' has treewidth at most two). We show by induction on $|V(G')| + |E(G')|$ that G' is series-parallel.

If $|V(G')| \leq 3$, then it clearly holds. Suppose $|V(G')| > 3$, note that $|E(G')| \geq |V(G')|$. If G' contains parallel edges, then apply the parallel reduction rule once on G' (rule 2 of Figure 8.3). The graph obtained this way has treewidth two and is biconnected. By the induction hypothesis, it is series-parallel. Since the parallel reduction rule is safe for series-parallel graphs, this implies that G' is series-parallel.

Suppose G' does not contain any parallel edges. Let $TD = (T, X)$ be a tree decomposition of width two of G' with $T = (I, F)$ and $X = \{X_i \mid i \in I\}$. Modify TD by repeating the following as often as possible. For each $i \in I$, if i has exactly one neighbor $j \in I$, and $X_i \subseteq X_j$, then remove X_i. Note that TD is a tree decomposition of width two of G', and it has at least two nodes. Let $i \in I$ such that i has exactly one neighbor $j \in I$ in T. There is a $v \in X_i$ such that $v \notin X_j$.

Let $v \in X_i$ such that $v \notin X_j$. Vertex v must have degree two in G', and both v's neighbors are contained in X_i. Apply the series reduction rule on v and its neighbors u and w (rule 1

of Figure 8.3). This gives the graph $G'' = (V(G') - \{v\}, E(G') + \{u,w\})$. Graph G'' has treewidth two, since the tree decomposition obtained from (T,X) by removing vertex v from node X_i is a tree decomposition of width two of G''. Furthermore, G'' is biconnected. By the induction hypothesis, G'' is a series-parallel graph. As the series rule is safe for series-parallel graphs, this means that G' is also series-parallel.

Now suppose each block of G is series-parallel. By Lemma 2.3.5, each block of G has treewidth at most two. By Lemma 2.2.1, the treewidth of G is at most two. □

Let $G = (V,E)$ be a multigraph. Recall that a bridge of G is an edge $e \in E$ for which the multigraph $(V, E - \{e\})$ has more connected components than G. In order to make the set of reduction rules conveniently small, we put a labeling on the edges of a multigraph: each edge in a multigraph is either labeled with label B, or it is not labeled (the label B stands for 'bridge'). We call such a multigraph a B-*labeled multigraph*. We extend the notion of treewidth at most two for multigraphs to treewidth at most two for B-labeled multigraphs.

Definition 9.1.1. Let $G = (V,E)$ be a connected B-labeled multigraph. Let G' be the underlying unlabeled multigraph. The graph G has *treewidth at most two* if and only if G' has treewidth at most two and for each edge $e \in E$, if e has label B, then e is a bridge of G.

A *tree decomposition of width at most two* of G is a tree decomposition $TD = (T,X)$ of width at most two of G' with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$, such that for each edge e with label B and end points u and v, there is a node $i \in I$ with $X_i = \{u,v\}$ such that there is no component in $T[I - \{i\}]$ which contains vertices of two components of $(V, E - \{e\})$.

We can easily prove by induction that a B-labeled multigraph G has treewidth at most two if and only if there is a tree decomposition of width at most two of G.

Note that an edge in a multigraph is a bridge if and only if the edge is a (trivial) block. Hence we can derive the following from Lemma 9.1.2.

Corollary 9.1.1. *Let G be a connected* B*-labeled multigraph. G has treewidth at most two if and only if each non-trivial block of G has no labeled edges and is series-parallel.*

We use B-labeled terminal multigraphs instead of unlabeled ones: a B-labeled terminal multigraph is a terminal multigraph of which some edges have label B. Two B-labeled terminal multigraphs G_1 and G_2 are isomorphic if there is an isomorphism from the underlying unlabeled terminal multigraph of G_1 to the underlying unlabeled terminal multigraph of G_2, such that labeled edges in G_1 are mapped to labeled edges in G_2 and unlabeled edges in G_1 are mapped to unlabeled edges in G_2.

Reduction rules consist of pairs of B-labeled terminal multigraphs instead of ordinary terminal multigraphs.

9.2 A Special Parallel Constructive Reduction System

In this section we define a constructive reduction system $S_{\text{tw}} = (R_{\text{tw}}, I_{\text{tw}}, A_R^{\text{tw}}, A_I^{\text{tw}})$ for the following problem.

TREEWIDTH AT MOST TWO (TW2)
Instance: A connected B-labeled multigraph G.
Find: A tree decomposition of width at most two of G, if the treewidth of G is at most two.

We also show that S_{tw} is a special parallel constructive reduction system. The set R_{tw} of reduction rules is based on the set R_{sp} of reduction rules for series-parallel graphs that is defined in Chapter 8 (see Figure 8.3).

In Section 9.2.1 we give the extension of R_{tw} with respect to R_{sp}, we give I_{tw}, and we show that the set R_{tw} is safe for TW2. After that, we show in Section 9.2.2 that each connected B-labeled multigraph G contains $\Omega(|E(G)|)$ matches to rules in R_{tw}, if $|E(G)| \geq 1$. In Section 9.2.3 we show that sufficiently many of these matches are discoverable. Finally, in Section 9.2.4 we give the construction algorithms A_R^{tw} and A_I^{tw}.

9.2.1 A Safe Set of Reduction Rules

Figure 9.1 shows the extension of R_{tw} with respect to R_{sp}. The set R_{tw} contains all rules of R_{sp} (Figure 8.3), with a small change of rule 1, plus five new rules, called rules 19 – 23. Rule 1 of R_{sp} is split into rules 1a and 1b: in rule 1a, there are no edges labeled B, in rule 1b, one or two of the edges in the left-hand side are labeled, and the edge in the right-hand side is also labeled. Rules 1a and 1b together are called rule 1. Rule 19 also consists of two parts: 19a and 19b, the first with no labeled edges, the second with one or more labeled edges. In rule 20, we pose a degree constraint of eight on the terminal vertex of the left-hand side. This degree constraint requires that if rule 20 is applied, then the vertex in the graph that matches the terminal of the left-hand side of the rule, has degree at most eight in the graph.

We give a new definition of a match to one of the rules 1 – 23 in a B-labeled multigraph.

Definition 9.2.1 (Match). Let $r = (H_1, H_2)$ be a reduction rule in R_{tw} and let G be a B-labeled multigraph. A *match* to r in G is a B-labeled terminal multigraph G_1 which is isomorphic to H_1, such that

- there is a B-labeled terminal multigraph G_2 with $G = G_1 \oplus G_2$,
- if r is rule 20, then the terminal vertex u in G_1 has degree at most eight in G.
- if r is one of the rules 3 – 18, then for each edge $e = \{u, v\} \in E(G_1)$ for which u and v are terminals, u or v has degree at most seven in G.

The set I_{tw} consists of only one graph, namely the graph consisting of one isolated vertex.

Lemma 9.2.1. *The set R_{tw} of reduction rules is safe for* TW2.

Proof. Let G be a connected B-labeled multigraph, let $r \in R_{\text{tw}}$, and suppose G contains a match H to r. Let G' be the graph obtained from G by applying the reduction corresponding to match H. We show that G has treewidth at most two if and only if G' has treewidth at most two. Note that a B-labeled multigraph has treewidth at most two if and only if all its blocks have treewidth at most two.

First suppose r is one of the rules 2 – 18. Then H is contained in one of the blocks of G. Let B denote this block (note that B is a non-trivial block and H is also a match in B), and let

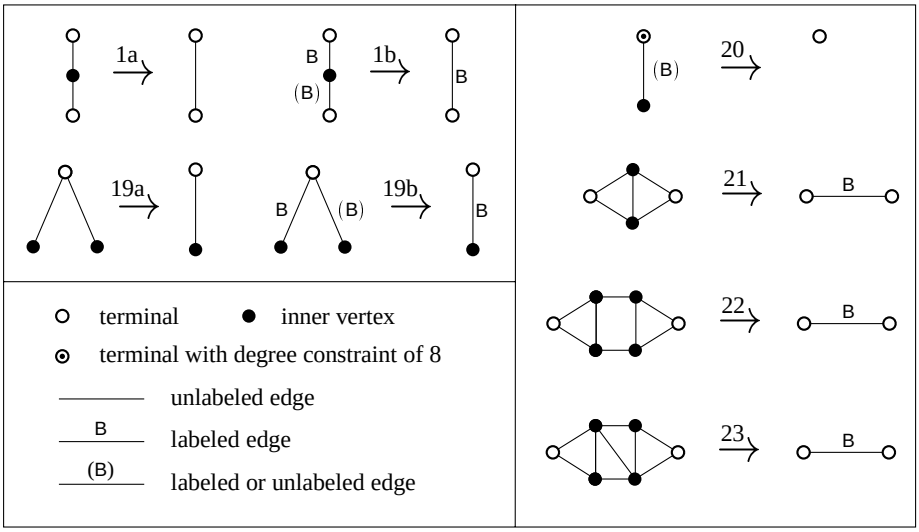


Figure 9.1. The modifications made to R_{sp} in order to get R_{tw}.

B' be the graph obtained from B by applying the rule. Then B' is a block of G'. Therefore, it suffices to show that B has treewidth at most two if and only if B' has treewidth at most two.

Let $s,t \in V(B)$ such that s and t are adjacent and are not inner vertices of H. Note that this is possible. Furthermore, B has treewidth at most two if and only if B has no labeled edges and (B,s,t) is series-parallel (by Corollary 9.1.2 and Lemma 8.3.1). The graph H is also a match in (B,s,t), as s and t are not inner vertices of H. Since rules 2 – 18 are safe for series-parallel graphs, this means that (B,s,t) is series-parallel if and only if (B',s,t) is series-parallel. Furthermore, since B' is biconnected, B' has treewidth at most two if and only if (B',s,t) is series-parallel. This proves that B has treewidth at most two if and only if B' has treewidth at most two.

Suppose r is rule 1. If all vertices of H are contained in one block B, then this is a non-trivial block. In the same way as for rules 2 – 18, we can show that G has treewidth at most two if and only if G' has treewidth at most two.

Suppose the vertices of H are not in one block. Then the two edges of H are separate blocks, and they are both bridges (hence they both have treewidth at most two). This implies that the new edge is a block in G', and it is also a bridge in G' (hence it also has treewidth at most two). This shows that G has treewidth at most two if and only if G' has treewidth at most two.

It is easy to see that rules 19 and 20 are safe for TW2: if r is rule 19 or 20, then the blocks of G have treewidth at most two if and only if the blocks of G' have treewidth at most two.

Suppose r is one of the rules 21, 22 and 23. Let x and y be the terminals of H. Suppose G

has treewidth at most two. If G contains a path between the terminals of H which avoids the inner vertices of H, then G contains a K_4 minor, hence this is not the case. This means that x and y are cut vertices of G, and hence H is a block of G. This implies that in G', the edge between x and y is a bridge of G', and hence it is a block of G' which has treewidth at most two. Hence G' has treewidth at most two.

If G' has treewidth at most two, then the edge between x and y is a bridge, and hence is a block with treewidth at most two. This implies that H is a block in G. As H has treewidth at most two, we have that G has treewidth at most two. □

9.2.2 A Lower Bound on the Number of Matches

In this section, we show that each connected multigraph G with at least one edge which has treewidth at most two has at least $\Omega(|E(G)|)$ matches. We do this by bounding the number of edges of G by an integer constant times the number of matches in G. We first prove the following lemma.

Lemma 9.2.2. *Let G be a connected* B*-labeled multigraph and let $v \in V(G)$ such that v has degree at most eight. Then the number of matches to rules 1 – 23 in G which contain v is at most some integer constant k.*

Proof. We give a very rude bound which is probably far too large, but easy to prove. Note that all inner vertices of left-hand sides of rules 1 – 23 have degree at most eight. Let G_1 be a match in G which contains v. It can be seen that all vertices and edges in G_1 are reachable by a path P from v to this vertex or edge, such that all vertices on the path except possibly the first and the last one are inner vertices of G_1, or are terminals of G_1 with degree at most eight. Hence each vertex on such a path, except the last one, has degree at most eight. Furthermore, the path has length at most seven, as each left-hand side of a reduction rule has at most eight vertices. Therefore, the number of vertices and edges in G which are reachable from v by such a path is at most 8^7. This implies that there is at most a constant number of matches containing v. □

Let $G = (V, E)$ be a connected B-labeled multigraph, suppose the treewidth of G is at most two, and let $|E| \geq 1$.

A *dangling edge* in G is an edge $e = \{u, v\}$ for which either u or v has degree one. If u has degree one, then e is called a dangling edge of v. A *star* is a graph consisting of one vertex with dangling edges. A *pseudo block* is a graph which is a star, or consists of one block with dangling edges, i.e. G consists of a block of which some vertices have dangling edges.

We divide G into pseudo blocks as follows. If G is a star, then G itself is the only pseudo block. Otherwise, let B denote the set of all blocks of G, and let $B' \subseteq B$ be the set of all blocks which are non-trivial, or which have two or more cut vertices. Note that B' contains exactly all blocks which are not dangling edges, and each dangling edge has an end point in one of the blocks in B'. Assign each dangling edge to a block in B' which contains one of its end points. A pseudo block of G consists of a block in B' with the dangling edges assigned

to them. Let PB be the set of all the pseudo blocks. For each pseudo block $PB \in PB$, we call the block of *PB* which is in B' the *underlying* block of *PB*.

The vertices that are contained in two or more pseudo blocks are called the *strong cut vertices* of *G*, and we denote the set of all strong cut vertices by S. If *v* is a strong cut vertex, then *v* is a cut vertex of *G* and *v* is contained in the underlying block of each pseudo block it is contained in.

Note that the pseudo blocks of *G* partition the set of edges $E(G)$. We divide the edges of *G* into different classes, which correspond to the type of pseudo block that they are contained in. After that, we give for each class an upper bound on the number of edges in this class with respect to the number of matches in *G*. Therefore, we first construct a *pseudo block tree* $T = (N,F)$ as follows.

$$N = PB \cup S$$
$$F = \{\{v, PB\} \mid v \in S \wedge PB \in PB \wedge v \in V(PB)\}$$

Hence T contains as its vertices the pseudo blocks and strong cut vertices of *G*, and there is an edge between two vertices in T if and only if one of them is a cut vertex *v*, the other one is a pseudo block *PB*, and *v* is contained in *PB*. Note that the degree of a strong cut vertex in T equals the number of pseudo blocks it is contained in, and the degree of a pseudo block in T equals the number of strong cut vertices it contains. We call a pseudo block a *degree d pseudo block* if its corresponding node in *N* has degree *d* in T. A degree one pseudo block is also called a leaf pseudo block. Note that each leaf pseudo block has at least two edges (if it had only one edge, then it would be a dangling edge of one of the blocks it shares a vertex with).

We partition the set PB of pseudo blocks into four sets: PB_0, PB_1, PB_2 and $PB_{\geq 3}$. For $i = 0,1,2$, PB_i is the set of degree *i* pseudo blocks. The set $PB_{\geq 3}$ is the set of all degree *d* pseudo blocks with $d \geq 3$. For $i = 0,1,2$, let E_i denote the set of all edges of pseudo blocks in PB_i, and let $E_{\geq 3}$ denote the set of edges of all pseudo blocks in $PB_{\geq 3}$. Note that, as *G* is connected, either *G* itself is its only pseudo block and has degree zero, or *G* contains two or more pseudo blocks which all have degree one or more. Note furthermore that only degree zero pseudo blocks can be stars.

Let *M* denote the set of all matches in *G*.

Lemma 9.2.3. *$|E_0| \leq k_0|M|$ for some integer constant k_0.*

Proof. If $PB_0 = \phi$, then it clearly holds. Suppose $PB_0 = \{G\}$, and let $m = |E|$. If $m = 1$, then *G* contains a match to rule 20. Suppose $m \geq 2$. If *G* is a star, then *G* has $m(m-1)/2$ matches to rule 19, and hence $m \leq 2|M|$. If *G* consists of an edge with one or more dangling edges at each end point, then either $m = 3$ and *G* has a match to rule 1 or 19, or $m > 3$ and *G* has at least $(m-1)(m-3)/8$ matches to rule 19 (at least $(m-1)/2$ edges are dangling edges of the same end point). Hence $m \leq 9|M|$.

Suppose *G* consists of a non-trivial block *B* with dangling edges *D*. Note that *B* has no edges labeled B. Let D_1 denote the dangling edges which are dangling edges of some vertex

of B that has one dangling edge, and let $D_{\geq 2}$ denote the other dangling edges. Note that G has at least $|D_{\geq 2}|/2$ matches to rule 19, and hence $|D_{\geq 2}| \leq 2|M|$.

Consider block B. As B is series-parallel and has at least two edges, it contains at least $|E(B)|/139$ matches to rules $1-18$ (Lemma 8.2.6). Consider the set M_{sp} of all these matches. Let $H \in M_{\text{sp}}$. Either H is a match in G or not. If H is not a match in G, we call H a disturbed match.

If H is disturbed, then either an inner vertex v of H has one or more dangling edges, or a terminal vertex v of H which has degree at most seven in B has one or more dangling edges. In both cases, v has degree at most seven in B. Furthermore, if v has one dangling edge, then it has degree at most eight in G and hence e is a match to rule 20. If v has two or more dangling edges, then two of these edges form a match to rule 19 in G. By Lemma 9.2.2, the number of matches in G which contains v is at most k. Hence the number of disturbed matches is at most k times the number of matches to rules 19 and 20 in G. This means that we can derive the following upper bound for $|M_{\text{sp}}|$.

$$\begin{aligned} |M_{\text{sp}}| &= |\{\text{non-disturbed matches}\}| + |\{\text{disturbed matches}\}| \\ &\leq |\{\text{matches to rules } 1-18\}| + k \cdot |\{\text{matches to rules 19 and 20}\}| \\ &\leq k|M| \end{aligned}$$

Furthermore, $|D_{\geq 2}| \leq 2|M|$ and $|D_1| \leq |V(B)| \leq |E(B)|$. Hence

$$\begin{aligned} m &= |E(B)| + |D_1| + |D_{\geq 2}| \\ &\leq 2|E(B)| + 2|M| \\ &\leq 278|M_{\text{sp}}| + 2|M| \\ &\leq 278(k+1)|M|, \end{aligned}$$

hence the lemma holds with $k_0 = 278(k+1)$. □

In the following discussion, we denote for each pseudo block PB the set of matches in PB by M_{PB}. A match in M_{PB} is either a match in G, in which case it is called a *non-disturbed* match, or it is not a match in G, in which case it is called a *disturbed* match. The set of non-disturbed matches in M_{PB} is denoted by M_{PB}^{nd}, and the set of disturbed matches in M_{PB} is denoted by M_{PB}^{d}. Note that $M_{PB}^{\text{nd}} \subseteq M$. Lemma 9.2.3 implies the following result.

Corollary 9.2.1. *For each pseudo block PB,* $|E(PB)| \leq k_0|M_{PB}|$.

Consider a disturbed match H in M_{PB}^{d}. Then there is a strong cut vertex v in PB for which either

- v is an inner vertex of H,
- H is a match to rule 20 and v is a terminal of H, v has degree at most eight in PB and v has degree more than eight in G, or
- H is a match to one of the rules $3-18$, v is a terminal of H, v has degree at most seven in PB, and v has degree more than seven in G.

If one of these cases holds for a strong cut vertex v and a disturbed match H, we say that v *disturbs* H.

Lemma 9.2.4. *Each strong cut vertex disturbs at most k matches in each pseudo block it is contained in.*

Proof. Let PB be a pseudo block and let v be a strong cut vertex in PB. Let H be a match that is disturbed by v. Note that v has degree at most eight in PB. Hence, by Lemma 9.2.2, there are at most k matches in PB which contain v. This means that v disturbs at most k matches. □

Lemma 9.2.5. $|E_1| \leq k_1|M|$ *for some integer constant* k_1*, and each leaf pseudo block contains at least one match in G.*

Proof. Let PB be a leaf pseudo block, let x denote the strong cut vertex in PB. By Lemma 9.2.4, $|M^{\mathrm{d}}_{PB}| \leq k$. If $|M^{\mathrm{nd}}_{PB}| \geq 1$, then $|M^{\mathrm{d}}_{PB}| \leq k|M^{\mathrm{nd}}_{PB}|$, and hence

$$|E(PB)| \leq k_0|M_{PB}| \leq k_0(|M^{\mathrm{nd}}_{PB}| + |M^{\mathrm{d}}_{PB}|) \leq k_0(1+k)|M^{\mathrm{nd}}_{PB}| \leq k_0(1+k)|M|.$$

This will show that $|E_1| \leq k_1|M|$ with $k_1 = k_0(1+k)$. Furthermore, if $|M^{\mathrm{nd}}_{PB}| \geq 1$, then each leaf pseudo block contains at least one match.

We next show that $|M^{\mathrm{nd}}_{PB}| \geq 1$. If the underlying block of PB is an edge e, then the end point of e which is not x has at least one dangling edge, and hence there is at least one match to rule 19 or 20. This match is not disturbed by x, so $|M^{\mathrm{nd}}_{PB}| \geq 1$.

Suppose the underlying block B of PB is non-trivial. Note that $x \in V(B)$. Let y be a neighbor of x in B. By Lemma 8.3.1, (B,x,y) is series-parallel and hence it has at least one match to rules 1 or 2 which does not have x or y as inner vertex. This means that PB has at least one match to rule 1, 2, 19 or 20 which does not have x as an inner vertex, and hence is not disturbed in G. Hence $|M^{\mathrm{nd}}_{PB}| \geq 1$. □

Lemma 9.2.6. $|E_{\geq 3}| \leq k_3|M|$ *for some integer constant* k_3.

Proof. Let $d \geq 3$ and let PB be a degree d pseudo block. Note that the underlying block B of PB is non-trivial. By Lemma 9.2.4, there are at most $d \cdot k$ disturbed matches in PB. Lemma 9.1.1 shows that the sum over all degrees of the pseudo blocks in $PB_{\geq 3}$ is at most three times the number of leaf pseudo blocks. By Lemma 9.2.5, each leaf pseudo block contains at least one match in M. Hence we can derive the following.

$$\begin{aligned}
|E_{\geq 3}| &= \sum_{PB \in PB_{\geq 3}} |E(PB)| \\
&\leq \sum_{PB \in PB_{\geq 3}} k_0|M_{PB}| \\
&= k_0 \sum_{PB \in PB_{\geq 3}} (|M^{\mathrm{nd}}_{PB}| + |M^{\mathrm{d}}_{PB}|) \\
&\leq k_0|M| + \sum_{PB \in PB_{\geq 3}} \deg(PB) \cdot k
\end{aligned}$$

$$\begin{aligned}
&\leq k_0|M| + 3k|PB_1| \\
&\leq k_0|M| + 3k|\{\text{matches in leaf pseudo blocks}\}| \\
&\leq |M|(k_0 + 3k)
\end{aligned}$$

Hence the lemma holds with $k_3 = k_0 + 3k$. □

Consider the set PB_2. We split this set in two parts PB_2^{t} and PB_2^{nt}. The first set contains all degree two pseudo blocks of which the underlying block is an edge, and the second set contains all other degree two pseudo blocks (i.e. the degree two pseudo blocks of which the underlying block is non-trivial). let E_2^{t} denote the set of edges in pseudo blocks of PB_2^{t} and let E_2^{nt} denote the set of edges in pseudo blocks of PB_2^{nt}.

Lemma 9.2.7. *For all degree two pseudo blocks $PB \in PB_2^{\mathrm{nt}}$, $|M_{PB}^{\mathrm{nd}}| \geq 1$.*

Proof. Let PB be a degree two pseudo block of PB_2^{nt}, and let B denote the non-trivial block of PB. Let x and y denote the strong cut vertices of PB. Suppose all matches in M_{PB} are disturbed, i.e. $M_{PB}^{\mathrm{nd}} = \phi$. As matches to rules 2 and 19 can not be disturbed, PB has no parallel edges, and there is no vertex in B with two or more dangling edges. Furthermore, no vertex of $V(B) - \{x,y\}$ with degree at most seven in B has a single dangling edge, otherwise, this edge would be a match to rule 20 which is not disturbed. This implies that vertices in $V(B) - \{x,y\}$ with degree $d \leq 7$ in B also have degree d in PB, and hence if B contains a match to one of the rules $1-23$, then PB also contains this match. As $M_{PB}^{\mathrm{nd}} = \phi$, this match is disturbed by x or y.

Note that $x,y \in V(B)$, and let $a \in V(B)$ such that $\{x,a\} \in E(B)$ and $a \neq y$. By Corollary 9.1.1 and Lemma 8.3.1, B has no labeled edges and the graph (B,x,a) is series-parallel. As each series-parallel graph with two or more edges contains a match to the series or the parallel rule (rule 1 or 2), and B does not contain a match to the parallel rule, it contains a vertex v of degree two with $v \notin \{x,a\}$. Then in PB, v also has degree two, and hence PB contains a match to rule 1. This match must be disturbed in G, and thus $v = y$. Hence y has degree two in B. By symmetry, vertex x also has degree two, and there can be no other vertices of degree two. Let b be the second neighbor of x, and let c and d denote the neighbors of y. Let T' be the minimal sp-tree of (B,x,a), and let α denote the root node of T', which has label (x,a). Node α must be a p-node, as there is a leaf node with label (x,a).

Color and name the internal nodes of T' in the same way as in the proof of Lemma 8.2.6 (an internal node is green if it is a p-node with two leaf children, or an s-node with two neighboring leaf children; an internal node is branching if at least two of its children are internal nodes; an internal node is yellow or blue if it is not green or branching). Tree T' contains at least one s- or p-node which has only leaf nodes, and hence this node is green. Let β be a green node in T'. Then β can not be a p-node, since then there would be a match to rule 2. Hence $\beta \neq \alpha$. If β is an s-node, then there is a match to rule 1 in B which consists of the edges corresponding to the leaf children of β. This must be the match consisting of y, c and d, and hence β has two children, both leaves, labeled with (c,y) and (y,d), or (d,y) and (y,c). Suppose w.l.o.g. that the first case holds. Then β has label (c,d) (possibly with $c = x$

and/or $d = a$). Furthermore, β is the only green node, as vertex y does not occur in the label of any other s-node.

Tree T' can not contain any branching nodes, otherwise T' would have at least two green nodes. It follows that all nodes of T', except β, are either leaf nodes, yellow nodes or blue nodes. Each yellow or blue node has one yellow, blue or green child, and one or two non-adjacent leaf children. Hence the s- and p-nodes of T' form a sequence of yellow and blue nodes, followed by the green node β. See part I of Figure 9.2 for an example.

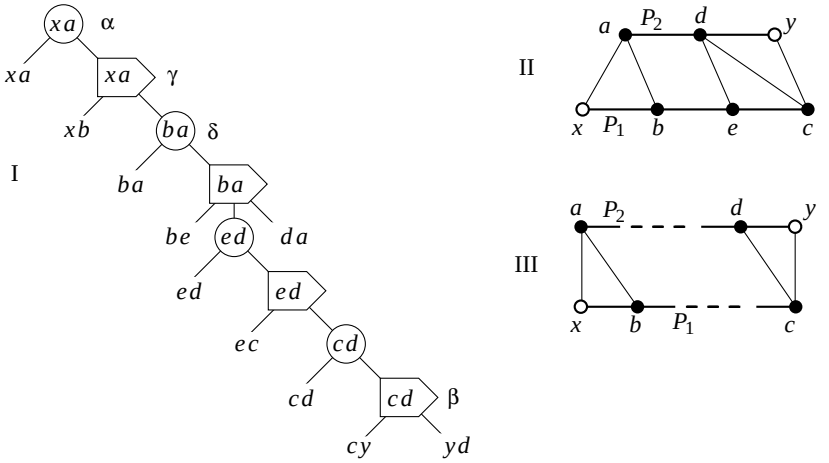


Figure 9.2. An example of the sp-tree T' of B (part I), its corresponding cycle-sequence (part II), and the general structure of the cycle-sequence B (part III).

Consider the p-node γ which is a child of α. Node γ has label (x,a). If $\gamma = \beta$, then $x = c$ and $a = d$, which means that B is a four cycle. But that it not possible, since then a has degree two. Suppose $\gamma \neq \beta$. Then γ has a child which is a p-node δ. Note that the leftmost child of γ is a leaf child labeled (x,b), as x has degree two. Hence δ has label (b,e) for some vertex $e \in V(B)$. If $e \neq a$, then γ also has a leaf child labeled (e,a). But that means that a has degree two, which is not possible. Hence δ is the rightmost child of α, and has label (b,a). As δ is a p-node, and has at least one leaf child, it follows that there is an edge between a and b. Furthermore, there is an edge between c and d, as β's parent is a p-node with label (c,d), and has at least one leaf child.

It can be seen that B is a cycle-sequence with bounding paths $P_1 = (x,b,\ldots,c)$ and $P_2 = (a,\ldots,d,y)$ (see page 177 for a definition). The first and the last cycle have three vertices, the first cycle contains x, a and b, and x has degree two in B, and the last cycle contains y, c and d, and y has degree two in B. Part II of Figure 9.2 shows the cycle sequence corresponding to the sp-tree of part I of the figure. Part III shows the general structure of the cycle-sequence B. The bounding paths in part II and III are denoted by the fat edges. Note that, as β is not a

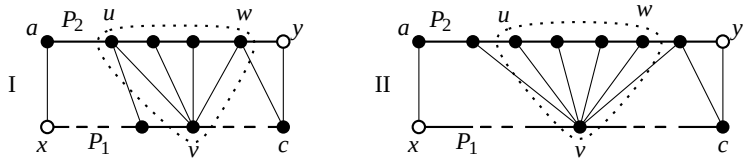


Figure 9.3. An example of a match to rule 3 if P_1 contains a vertex v with four neighbors in P_2 (part I) or with six neighbors in P_2 (part II).

child of α in T', $x \neq c$ and $a \neq d$, and hence P_1 and P_2 have length one or more (P_1 has length one if $b = c$ and P_2 has length one if $a = d$).

If P_1 and P_2 both have length one, then B, and hence PB, contains a match to rule 21 with terminals x and y. As this match is not disturbed, this can not be the case. Hence either P_1 or P_2 has length more than one.

If P_1 has length one and P_2 has length two, then B forms a match to rule 3 with terminals x, y and b (note that b has degree four), which is also a match in PB, and is not disturbed. Hence this is not possible. Similarly if P_2 has length one and P_1 has length two.

Suppose P_1 contains a vertex v which has at least four neighbors in P_2. Note that $v \neq x$ and $v \neq y$. If v has four or five neighbors in P_2, then it has degree at most seven in B, and hence it forms a match to rule 3 with four of its neighbors in P_2. See e.g. part I of Figure 9.3. In this picture, v has four neighbors in P_2, and the part of the graph surrounded by the dotted lines forms a match to rule 3 with terminals u, w and v. Vertex v has degree at most seven in B, but u and w may have degree more than seven. The match to rule 3 is also a match in PB, and it is not disturbed, as v has degree at most seven in G. Hence this gives a contradiction.

If v has six or more neighbors in P_2, then at most two of these neighbors have degree seven or more in B, namely the outermost two. The remaining at least four neighbors all have degree three. We can take four of these vertices which, together with v, form a match to rule 3. See e.g. part II of Figure 9.3 for the case that v has six neighbors in P_2. Let u and w be the terminals of the match to rule 3, together with v. Note that u and w are not equal to x or y. Hence this match is also a match in PB and in G, which gives a contradiction. This means that P_1 does not contain vertices with four or more neighbors in P_2, and by symmetry, the same holds for the vertices of P_2.

Suppose P_1 has length one and P_2 has length three or more. Then all vertices on P_2 are neighbors of vertex b, which is impossible by the discussion above. Hence P_1 can not have length one, and by symmetry, P_2 can not have length one.

Until now, we have shown that it is impossible that P_1 or P_2 has length one, and that no vertex in P_1 has four or more neighbors in P_2, or vice versa. Suppose both P_1 and P_2 have length two or more, and suppose all vertices in P_1 have at most three neighbors in P_2, and vice versa. Suppose w.l.o.g. that P_2 is at least as long as P_1. If P_1 and P_2 both have length two, then B forms a match to rule 22 or 23, and hence PB contains this match, and it is not disturbed, which gives a contradiction.

Finally, consider the case that P_1 has length two or more and P_2 has length three or more. Note that all vertices in B have degree at most six, and hence all vertices except x and y have degree at most six in G. Furthermore, x and y are not adjacent, and hence if the cycle-sequence has one of the left-hand sides of rules 3 – 18 as a subsequence, then it has a match to this rule which is also a match in PB and is not disturbed. It follows from Claim 8.2.3 that B contains such a subsequence, and hence we have a contradiction. This proves the lemma. □

Lemma 9.2.8. *$|E_2^{\mathrm{nt}}| \leq k_2|M|$ for some integer constant k_2, and each pseudo block in PB_2^{nt} contains at least one match in G.*

Proof. Let $PB \in PB_2^{\mathrm{nt}}$. Let x and y denote the strong cut vertices of PB. Note that $|M_{PB}^{\mathrm{d}}| \leq 2k$, by Lemma 9.2.4. By Lemma 9.2.7, $|M_{PB}^{\mathrm{nd}}| \geq 1$. Hence $|M_{PB}^{\mathrm{d}}| \leq 2k|M_{PB}^{\mathrm{nd}}|$, and it follows that $|E(PB)| \leq k_0|M_{PB}| = k_0(|M_{PB}^{\mathrm{nd}}| + M_{PB}^{\mathrm{d}}) \leq k_0(2k+1)|M_{PB}^{\mathrm{nd}}| \leq k_0(2k+1)|M|$. This proves the lemma with $k_2 = k_0(2k+1)$. □

Lemma 9.2.9. *$|E_2^{\mathrm{t}}| \leq k_4|M|$ for some integer constant k_4.*

Proof. We partition the set PB_2^{t} into two sets A and B:

- A is the set of pseudo blocks in PB_2^{t} of which the underlying block is an edge and has at least two dangling edges on one of its end points,
- B is the set of pseudo blocks in PB_2^{t} of which the underlying block is an edge and has at most one dangling edge on each end point.

Let E_A denote the edges of all pseudo blocks in A and E_B the set of edges of all pseudo blocks in B.

Claim. *$|E_A| \leq 9|M|$, and each pseudo block $PB \in A$ contains at least one match in G.*

Proof. Let $PB \in A$, let e be the underlying block of PB. Note that the end points x and y of e are the strong cut vertices of PB. Let $m = |E(PB)|$. If $m = 4$, then PB has one match to rule 19, which is not disturbed. Hence in this case $m \leq 4|M_{PB}^{\mathrm{nd}}|$. If $m > 4$, then there are at least $(m-1)(m-3)/8$ matches to rule 19 in PB, and these are also matches in G. Hence in this case, $m \leq 9|M_{PB}^{\mathrm{nd}}|$. This implies that for each pseudo block $PB \in A$, $|E(PB)| \leq 9|M_{PB}^{\mathrm{nd}}|$, and hence PB contains at least one match. Summing over all pseudo blocks in A shows that $|E_A| \leq 9|M|$. □

Note that each pseudo block in B has at most three edges. We bound $|B|$ from above in terms of the number of matches in G. Thereto, we partition B into two sets C and D:

- C contains all pseudo blocks of B for which one of the strong cut vertices has degree at least three in the pseudo block tree, and
- D contains all pseudo blocks of B of which both strong cut vertices have degree two in the pseudo block tree.

Note that there are no cut vertices of degree one, and hence C and D partition B.

Claim. $|C| \leq 3|M|$.

Proof. The number of pseudo blocks in C is at most the sum over the degrees of all nodes in the pseudo block tree which have degree at least three. Hence, by Lemma 9.1.1, $|C|$ is at most three times the number of leaf pseudo blocks, and as each leaf pseudo block has at least one match (Lemma 9.2.5), $|C| \leq 3|M|$. □

Consider the pseudo blocks of D. We partition this set in four sets D_1, D_{2a}, D_{2b} and D_3:

- D_1 contains all pseudo blocks in D which have a strong cut vertex x that is also contained in a pseudo block from PB_1.
- D_{2a} contains all pseudo blocks in D which are not in D_1 and have a strong cut vertex x that is contained in two pseudo blocks from D,
- D_{2b} contains all pseudo blocks in D which are not in D_1 or D_{2a}, and have a strong cut vertex x that is also contained in a pseudo block from $PB_2 - D$, and
- D_3 contains all pseudo blocks in D which are not in D_1 or D_2.

Note that the pseudo blocks in D_3 are the pseudo blocks which have two strong cut vertices that are contained in a pseudo block from $PB_{\geq 3}$.

Claim. $|D_1| \leq |M|$.

Proof. Consider a pseudo block PB in D_1. Let x be a strong cut vertex such that the other pseudo block PB' containing x is in PB_1. Pseudo block PB' contains at least one match. Account PB to such a match. Each match in a leaf pseudo block has at most one pseudo block of D_1 accounted to it. This proves the claim. □

Claim. $|D_{2a}| \leq 2|M|$.

Proof. Consider a pseudo block PB in D_{2a}. Let x be a strong cut vertex such that the other pseudo block PB' containing x is also in D. Note that in G, x has at most two dangling edges. If x has one or two dangling edges, then one of these dangling edges forms a match to rule 20, otherwise, x has degree two and it forms a match to rule 1, together with its neighbors. We account each pseudo block in D_{2a} to such a match to rule 1 or 20. In this way, each match to rule 1 or 20 has at most two pseudo blocks accounted to it, and hence $|D_{2a}| \leq 2|M|$. □

Claim. $|D_{2b}| \leq 10|M|$.

Proof. Consider a pseudo block PB in D_{2b}. Let x be a strong cut vertex of PB which is contained in a pseudo block PB' of $PB_2^{\text{nt}} \cup A \cup C$. If $PB' \in PB_2^{\text{nt}}$, then PB' contains at least one match, by lemma 9.2.8, and hence $|PB_2^{\text{nt}}| \leq |M|$. If $PB' \in A$, then PB' also contains a match, as is shown above, and hence $|A| \leq |M|$. Otherwise, $PB' \in C$, and we have shown that $|C| \leq 3|M|$. Account PB to PB'. Each pseudo block in $PB_2^{\text{nt}} \cup A \cup C$ has at most two pseudo blocks of D_{2b} accounted to it, and hence

$$\begin{aligned} |D_{2b}| &\leq 2(\ |PB_2^{\text{nt}}| + |A| + |D|\ ) \\ &\leq 2(\ |M| + |M| + 3|M|\ ) \\ &= 10|M|. \end{aligned}$$

□

Claim. $|D_3| \leq 3|M|$.

Proof. Consider a pseudo block PB in D_3. Let x be a strong cut vertex of PB which is contained in a pseudo block PB' of $PB_{\geq 3}$. Account PB to the edge $\{x, PB'\}$ in the pseudo block tree T. Each such edge has at most one pseudo block of D_3 accounted to it. Lemma 9.1.1 implies that the number of these edges is at most three times the number of leaves of the pseudo block tree T. As each leaf in T is a leaf pseudo block, and each leaf pseudo block contains at least one match in G, this implies that $|D_3| \leq 3|M|$. □

We now get the following.

$$\begin{aligned}
|E_2^{\mathrm{t}}| &= |E_A| + |E_B| \\
&\leq |E_A| + 3(|C| + |D|) \\
&= |E_A| + 3(|C| + |D_1| + |D_{2a}| + |D_{2b} + |D_3|) \\
&\leq 9|M| + 3|M|(3 + 1 + 2 + 10 + 3) \\
&= 66|M|,
\end{aligned}$$

which proves Lemma 9.2.9 with $k_4 = 66$. □

Note that $|E| = |E_0| + |E_1| + |E_2^{\mathrm{nt}}| + |E_2^{\mathrm{t}}| + |E_{\geq 3}|$ and hence $|E| \leq (k_0 + k_1 + k_2 + k_4 + k_3)|M|$. This means that we have proved the following result.

Lemma 9.2.10. *There is a constant $c > 0$, such that each connected* B*-labeled multigraph $G = (V, E)$ with treewidth at most two and $|E| \geq 1$ contains at least $c|E|$ matches.*

Lemma 9.2.10 also shows that R_{tw} is complete for TW2: each connected B-labeled multigraph with at least one edge has a match to one of the rules in R_{tw}. Furthermore, the graph consisting of one vertex has treewidth at most two, and does not contain a match. Hence it is the only irreducible graph of treewidth at most two. It is easy to see that R_{tw} is decreasing, hence we have proved the following.

Corollary 9.2.2. $(R_{\mathrm{tw}}, I_{\mathrm{tw}})$ *is a reduction system for* TW2.

9.2.3 A Lower Bound on the Number of Discoverable Matches

In this section, we show that $(R_{\mathrm{tw}}, I_{\mathrm{tw}})$ is a special parallel reduction system for TW2 (see Definition 5.4.3). Therefore, we have to show that there are integers n_{min} and d, with $n_{min} \leq 19 \leq d$, and a constant $c' > 0$, such that for each connected B-labeled multigraph G with treewidth at most two that is given by some adjacency list representation, the following holds: if $|V(G)| + |E(G)| \geq n_{min}$ then G contains at least $c' \cdot (\,|V(G)| + |E(G)|\,)$ d-discoverable matches (with respect to d). Since a connected multigraph has $O(E(G))$ vertices, it suffices to show that there are at least $c' \cdot |E(G)|$ d-discoverable matches.

We use the same idea as for series-parallel graphs, only rule 19 gives some extra complications. Let $n_{min} = 2$ and $d = 20$ (note that d is the same as for series-parallel graphs, see page 179). A match to a rule in R_{tw} is d-discoverable if it either is a match to one of the

rules 1, 3 – 18, or 20 – 23, or it is a match to rule 2 or 19, and in the adjacency list of one of the terminals of this match, the edges of the match have distance at most 20.

Let G be a B-labeled multigraph $G = (V,E)$ given by some adjacency list representation. Recall from Chapter 8, page 180 that an edge e is bad if it has a parallel edge, but all its parallel edges have distance at least 21 in the adjacency lists of the end points of e.

A dangling edge e is called a *bad dangling edge* if it is incident with a vertex v that has two or more dangling edges, but in the adjacency list of v, these dangling edges have distance at least 21 to e. Note that an edge is bad if and only if it occurs in a match to rule 2, but not in a d-discoverable match to rule 2. A dangling edge is bad if and only if it occurs in a match to rule 19, but not in a d-discoverable match to rule 19.

Lemma 9.2.11. *Let $G = (V,E)$ be a* B*-labeled multigraph of treewidth at most two, given by some adjacency list representation. The graph G has at most $|E|/5$ bad edges and at most $|E|/10$ bad dangling edges.*

Proof. For the bound on the bad edges, see Lemma 8.2.7. Consider the dangling edges. Let $v \in V(G)$. If the adjacency list of v has length at most 20, then v does not have any bad dangling edges. If the adjacency list of v has length more than 20, then each 20 successive entries in the (cyclic) adjacency list contain at most one bad dangling edge. Hence there are at most $\deg(v)/20$ bad dangling edges in the adjacency list of v. If we sum over all vertices, we get that the number of dangling edges is at most $|E|/10$. □

Lemma 9.2.12. *There is a constant $c' > 0$ for which each connected* B*-labeled multigraph G of treewidth at most two with at least one edge has at least $c'|E(G)|$ d-discoverable matches.*

Proof. We use the same idea as in the proof of Lemma 8.2.8. Let $G = (V,E)$ be a B-labeled multigraph with at least one edge. Let $n = |V|$ and let $m = |E|$. Let de denote the number of dangling edges of G of which one end point has at least two dangling edges, i.e. the dangling edges which occur in a match to rule 19. We distinguish between three cases:

1. $m \geq 4n$,
2. $de \geq m/5$, and
3. $m < 4n$ and $de < m/5$.

Case 1. Suppose $m \geq 4n$. As G has treewidth at most two, the underlying simple graph has at most $2n$ edges (Lemma 2.2.6). This means that at least $m - 2n$ edges are parallel to another edge. At most $m/5$ of these are bad edges, hence at least $m - 2n - m/5 \geq 4m/5 - m/2 = 3m/10$ edges occur in a d-discoverable match to rule 2. This means that there are at least $3m/20$ d-discoverable matches to rule 2 in G.

Case 2. Suppose that $de \geq m/5$. Of the de dangling edges which occur in a match to rule 19, at most $m/10$ are bad. Hence at least $de - m/10 \geq m/10$ of these dangling edges occur in a d-discoverable match to rule 19. This means that there are at least $m/20$ d-discoverable matches to rule 19 in G.

Case 3. Suppose that $m < 4n$ and $de < m/5$. If G is a star, then G contains at least $m/20$ d-discoverable matches to rule 19.

Suppose G is not a star. Remove all dangling edges which occur in a match to rule 19 from G. Furthermore, for each pair u, v of vertices in G which have two or more edges between them, do the following. If u and v both have two or more neighbors, then remove all edges except one between u and v. If u or v has only one neighbor, then remove all but two edges between u and v. Let G' denote the resulting graph, and let $n' = |V(G')|$, $m' = |E(G')|$.

Note that, if G' contains a match to rule 2, then one of the terminals of this match has degree two, and hence this match is d-discoverable. Note furthermore that G' has no matches to rule 19: first all matches to rule 19 were removed, and after that, matches to rule 2 were removed without introducing new matches to rule 19.

We express n' and m' in terms of m: $n' = n - de > m/4 - m/5 = m/20$. As G is not a star, it follows that $n' \geq 2$. Furthermore, G' is connected, and hence $m' \geq n' - 1 \geq n'/2 \geq m/40$.

Note that $m' \geq 1$ (since G is not a star). By Lemma 9.2.10, G' contains at least $c \cdot m'$ matches. As we have argued before, these matches are all d-discoverable, and in each match to rule 2, one of the end points has degree at most two. Let M denote the set of all matches in G'. Each of these matches is either a d-discoverable match in G, or it is not a d-discoverable match in G. We call the first set the set of non-disturbed matches, denoted by M_{nd}, and the second the set of disturbed matches, denoted by M_{d}. Let M_{new} denote the set of d-discoverable matches in G which are not in G'. Note that the set of d-discoverable matches in G contains M_{nd} and M_{new}.

Consider a match $H \in M_{\text{d}}$. If H is a match to rule 2, then the terminal v of H which has degree two in G' has degree more than two in G. If H is a match to one of the other rules, then either H contains an inner vertex v which has dangling edges in G or which is incident with parallel edges in G, or v is a terminal which has degree $d \leq 8$ in G', but has degree more than d in G. In all cases, there is a vertex $v \in V(H)$ which has degree $d \leq 8$ in G' and has degree more than d in G.

Since v has larger degree in G than in G', it must be the case that in G, v has two or more dangling edges, or v is incident with parallel edges. If the adjacency list of v has length at most 20, then there are two edges incident with v which form a d-discoverable match to rule 2 or rule 19. Let H_v denote this match, note that $H_v \in M_{\text{new}} \cup M_{\text{nd}}$. If the adjacency list of v has length more than 20, then consider a sublist of length 20 of this list. If this sublist contains two or more dangling edges, then two of these form a d-discoverable match to rule 19 in G, and hence this match is in M_{new}. Let H_v denote this match. If the sublist contains at most one dangling edge, then 20 or 21 of the places in this sublist contain an edge e between v and a neighbor of v. As v has at most eight distinct neighbors, there must be at least two edges with the same end points in the sublist. Two of these edges correspond to a d-discoverable match to rule 2 in G. Let H_v again denote this match, and note that $H_v \in M_{\text{new}} \cup M_{\text{nd}}$.

Note that, as v has degree at most eight in G', it is contained in at most k matches in M and hence in M_{d} (Lemma 9.2.2). For each match H in M_{d}, account H to a match H_v of a vertex $v \in V(H)$ which has degree $d \leq 8$ in G' and degree more than d in G. In this way, each match to rule 19 in $M_{\text{new}} \cup M_{\text{nd}}$ has at most k matches accounted to it, and each match

to rule 2 in $M_{\text{new}} \cup M_{\text{nd}}$ has at most $2k$ matches accounted to it (at most k for each end point). Hence $|M_{\text{d}}| \leq 2k(|M_{\text{new}} \cup M_{\text{nd}}|) = 2k|M_{\text{new}}| + 2k|M_{\text{nd}}|$, and we can derive the following.

$$\begin{aligned} m &\leq 40m' \\ &\leq (40/c) \cdot |M| \\ &= (40/c) \cdot (\,|M_{\text{d}}| + |M_{\text{nd}}|\,) \\ &\leq (40/c) \cdot (\,2k|M_{\text{new}}| + (2k+1)|M_{\text{nd}}|\,) \\ &\leq (40(2k+1)/c) \cdot (\,|M_{\text{new}}| + |M_{\text{nd}}|\,) \end{aligned}$$

As G contains at least $|M_{\text{new}}| + |M_{\text{nd}}|$ d-discoverable matches, this means that G contains at least $c'|E|$ d-discoverable matches, with $c' = c/40(2k+1)$. □

Hence we have shown the following result.

Corollary 9.2.3. $(R_{\text{tw}}, I_{\text{tw}})$ *is a special parallel reduction system for* TW2.

9.2.4 The Construction Algorithms

In this section we complete the description of the special parallel constructive reduction system $(R_{\text{tw}}, I_{\text{tw}}, A_R^{\text{tw}}, A_I^{\text{tw}})$ for TW2 by describing algorithms A_R^{tw} and A_I^{tw}. Algorithm A_R^{tw} has to be such that it uses $O(1)$ time and is non-interfering, and algorithm A_I^{tw} must use $O(1)$ time. We construct A_R^{tw} and A_I^{tw} in such a way that the tree decomposition is a so-called special tree decomposition (see Definition 6.3.2).

Let $G = (V,E)$ be a connected B-labeled multigraph, and let $x,y \in V$. A set $W \subseteq V$ is an *x,y-separator* if $x,y \notin W$, and x and y are in different components of $G[V-W]$. An x,y-separator is called a *minimal x,y-separator* if no proper subset W' of W is an x,y-separator.

Definition 9.2.2 (Special Tree Decomposition). Let $G = (V,E)$ be a connected B-labeled multigraph with treewidth at most two. Let $TD = (T,X)$ be a tree decomposition of width two of G with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. Then TD is a *special tree decomposition* of G if it satisfies the following conditions,

1. For each vertex $u \in V$ there is a unique node i with $X_i = \{u\}$, called the node associated with u.
2. Each edge $e \in E$ with end points u and v has a node i with $X_i = \{u,v\}$ associated with it. Distinct edges have distinct associated nodes.
3. Let u be a cut vertex of G, let i denote the node associated with u. Then each component of $T[I - \{i\}]$ contains vertices of at most one component of $G[V - \{u\}]$.
4. Let e be a bridge of G with end points u and v and let i be the node associated with e. Then each component of $T[I - \{i\}]$ contains vertices of exactly one component of $(V, E - \{e\})$.
5. Let $u,v \in V$. If there is an edge between u and v, and $\{u,v\}$ is a minimal x,y-separator for some vertices x and y, then there is a node i associated with some edge between u and v such that x and y occur in different components of $T[I - \{i\}]$.

6. For each two adjacent nodes $i, j \in I$, $||X_i| - |X_j|| = 1$, unless if $X_i = X_j = \{u, v\}$ and i and j are nodes associated with different edges between u and v.
7. For each $u, v \in V$, the nodes associated with edges between u and v induce a subtree of T.

We give a number of properties of a special tree decomposition.

Lemma 9.2.13. *Let $G = (V,E)$ be a* B*-labeled connected multigraph and let $TD = (T,X)$ be a special tree decomposition of G with $T = (I,F)$ and $X = \{X_i \mid i \in I\}$. TD satisfies the following properties.*

1. *No node with one or three vertices occurs twice, and if a node with two vertices u and v occurs t times, $t \geq 2$, then there are t edges between u and v.*
2. *For each $i \in I$, no two components of $T[I - \{i\}]$ contain vertices of the same component of $G[V - X_i]$.*
3. *Let $i, j \in I$ such that $\{i, j\} \in F$. Then $|X_i \cap X_j| \geq 1$, and either $X_i \subseteq X_j$ or $X_j \subseteq X_i$.*
4. *Let $i, j \in I$ such that $\{i, j\} \in F$, $X_i = \{u, v\}$ and $X_j = \{u, v, w\}$. Then w and u are in the same component of $G[V - \{v\}]$, and w and v are in the same component of $G[V - \{u\}]$.*

Proof.

1. Follows from conditions 1, 2 and 6 of Definition 9.2.2.

2. If two vertices u and v of the same component C of $G[V - X_i]$ occur in different components of $T[I - \{i\}]$, then X_i contains a vertex of a path in C from u to v.

3. If $X_i \cap X_j = \emptyset$, then G is not connected. Suppose $X_i \not\subseteq X_j$ and $X_j \not\subseteq X_i$. Let $u \in X_i - X_j$ and $v \in X_j - X_i$. Then $X_i \cap X_j$ is a u,v-separator. As i and j can not both have three vertices, $|X_i \cap X_j| = 1$. Let $a \in X_i \cap X_j$. Then a is a cut vertex and u and v are in different components of $G[V - \{a\}]$. Let k be the node associated with a. Then u and v occur in different components of $T[I - \{k\}]$. But then k must be between i and j, which is impossible, as i and j are adjacent.

4. Suppose w and u are not in the same component of $G[V - \{v\}]$. Then v is a cut vertex. In the same way as for case 3, this gives a contradiction. □

We use the following data structure for storing a special tree decomposition. We store a list containing all nodes of the tree decomposition. Each node i has an adjacency list which contains an entry for each neighbor of i. An entry for neighbor j in the adjacency list of i contains a pointer to j, the contents X_j of node j, and a pointer to the entry of i in the adjacency list of j. Furthermore, for each vertex and edge in the graph, we keep a pointer to the node associated with it.

Consider algorithm A_I^{tw}. Given a graph G consisting of one vertex v, A_I^{tw} simply constructs a tree decomposition of one node which contains v. Note that this tree decomposition satisfies conditions 1 – 7, and hence is a special tree decomposition.

Consider algorithm A_R^{tw}. The algorithm consists of 23 rounds which are executed consecutively. Each round corresponds to a rule in R_{tw}: if the algorithm gets rule r as input, then

it is only active in round r. Suppose the input of the algorithm is a rule $r = (H_1, H_2) \in R_{\text{tw}}$, a B-labeled multigraph $G = (V, E)$, a special tree decomposition $TD = (T, X)$ of G, a match $G_2 = (V_2, E_2, X)$ to H_2 in G, and a match $G_1 = (V_1, E_1, X)$ to H_1 with the same terminal set as G_2. Let $T = (I, F)$ and $X = \{X_i \mid i \in I\}$. We describe the algorithm per round.

Round 1. If r is not rule 1, then the algorithm is idle in this round. Suppose r is rule 1, and suppose $V_2 = \{a, b\}$, $E_2 = \{e\}$ and $V_1 = \{a, b, c\}$. See part I of Figure 9.4 (labelings of edges are not shown). Let i and j be the nodes of TD associated with a and b, and let k be the node associated with e.

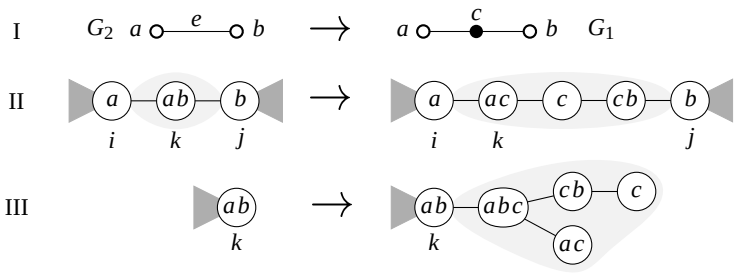


Figure 9.4. The construction for rule 1.

If e is a bridge in G, then a and b are cut vertices. The node k associated with e separates TD in different components corresponding to the components of $G' = (V, E - \{e\})$. Note that G' has exactly two components, one containing a and one containing b. By condition 4 of Definition 9.2.2, this means that the node k has degree two: it is adjacent to a node containing a and one containing b. Consider the path from node i to k in T. Let l be on this path, suppose $l \neq i$ and $l \neq k$. X_l contains a, and not b. If X_l contains a vertex $v \neq a$, then $v \notin X_k$, and hence there is no path from v to b which avoids a. Hence v and b are in different components of $G[V - \{a\}]$, while they occur in the same component of $T[I - \{i\}]$. This is a contradiction, and hence no node on the path from k to i contains another vertex than a (except k). Hence k and i are adjacent. In the same way, we can show that k and j are adjacent. This means that TD contains a subtree as shown in the left-hand side of part II of Figure 9.4. We replace this by the subtree shown in the right-hand side of part II (the light-gray parts of the tree decompositions are the parts that are involved in the modification). Note that the new edges are bridges and c is a cut vertex. Hence the new tree decomposition is a special tree decomposition.

Consider the case that e is not a bridge. Then node k does not necessarily have i and j as its only neighbors. If this is indeed not the case, we add an extra node l as new neighbor of k, with $X_l = \{a, b, c\}$, and we add some other nodes to fulfil conditions 1 – 7. See part III of Figure 9.4. Note that the new tree decomposition is indeed a special tree decomposition (c is not a cut vertex, the new edges are not bridges, and the sets $\{a, c\}$ and $\{c, b\}$ are not minimal x, y-separators).

Hence algorithm A_R^{tw} does the following. It looks if node k has as its only neighbors node i and k. If so, it applies the construction of part II of Figure 9.4. The new nodes are added, the contents of X_k is changed, and in the adjacency lists of i and j, the entries for node k are modified. Furthermore, the nodes with contents $\{a,c\}$, $\{b,c\}$ and $\{b\}$ are the nodes associated with the edge between a and c, the edge between b and c, and vertex c, respectively.

If k does not have only i and j as its neighbors, the algorithm applies the construction of part III of Figure 9.4. This can be done by adding the new nodes with their adjacency lists, and adding an entry for the new node adjacent to k at the end of the adjacency list of k. The new nodes are the nodes associated with the new edges and vertices.

It can be seen that this construction is correct. Furthermore, A_R^{tw} runs in $O(1)$ time for rule 1, and it is non-interfering: edge e is not involved in any other reduction at the same time, and hence node k is not involved in any other reductions that are performed in round 1. Nodes i and j may be involved in other applications of rule 1, however, only the contents of entries in the adjacency list of i and j are modified, and this can be done in different places of the adjacency list at the same time without concurrent reading or writing.

Round 2. If r is not rule 2, then the algorithm is idle in this round. Suppose r is rule 2. The construction is very simple. Let $V_1 = V_2 = \{a,b\}$, $E_2 = \{e\}$ and $E_1 = \{e,e'\}$, see part I of Figure 9.5. Let i be the node associated with e. Then we can apply the construction of part II of Figure 9.5. The newly added node is the node associated with edge e'. Note that condition 7 is satisfied, and hence TD is a special tree decomposition. It is easy to see that this construction can be done in $O(1)$ time and that round 2 is non-interfering (edge e is involved in only one reduction and hence so is node i).

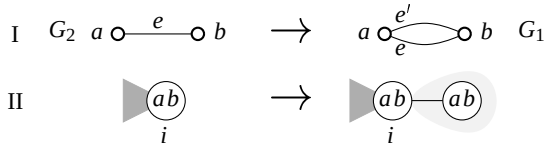


Figure 9.5. The construction for rule 2.

Round 3. Suppose r is rule 3 (otherwise, the algorithm is idle in this round). Let G_1 and G_2 be as depicted in part I of Figure 9.6. Let i, j and k be the nodes associated to edge e_1, e_2 and e_3, respectively. We show that T contains a subtree as depicted in the left-hand side of part II of Figure 9.6.

Note that there is no path from a to b in G which avoids both c and d, otherwise G contains a K_4 minor. Hence $\{c,d\}$ is a minimal a,b-separator. This shows that nodes i and j are in different components of $T[I - \{k\}]$. Let l be the (unique) node with $X_l = \{a,c,d\}$. Consider the path from k to l in T. Each node on this path contains c and d. Furthermore, only $X_k = \{c,d\}$, since there is only one edge between c and d. If there is a node $\{c,d,v\}$

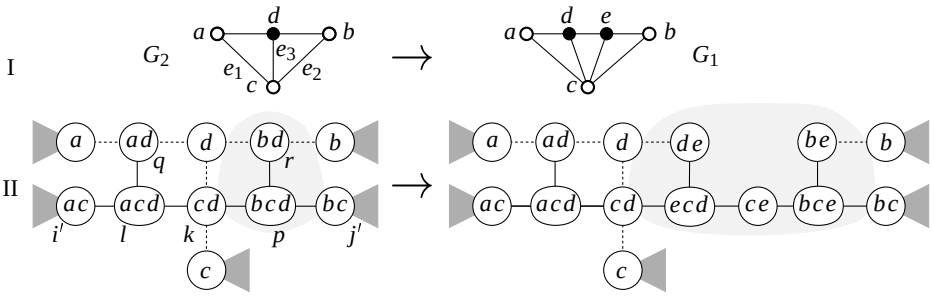


Figure 9.6. The construction for rule 3. Dashed lines denote possible adjacencies.

on this path for some $v \notin \{a,c,d\}$, then there must be another node $\{c,d\}$, since there are no two adjacent nodes with three vertices. This gives a contradiction, hence k and l are adjacent. Similarly, k is adjacent to the unique node p with $X_p = \{b,c,d\}$.

Consider the path from l to i in T. Each node on this path contains a and c. Let i' be the node on this path which has $X_{i'} = \{a,c\}$ and is the node closest to l for which this holds. Then i' and l are adjacent. Similarly, node p is adjacent with a node j' with $X_{j'} = \{b,c\}$. Consider the node q with $X_q = \{a,d\}$. This node must be adjacent to node l. Similarly, the node r associated with edge $\{b,d\}$ must be adjacent to node p.

Consider the nodes l and p. Both these nodes are adjacent to three other nodes, and, by property 3 of Lemma 9.2.13 and condition 7 of Definition 9.2.2 they can not be adjacent to any other node. Consider nodes k, q and r. By property 4 of Lemma 9.2.13, none of these nodes can be adjacent to another node containing three vertices. Furthermore, the only nodes containing one vertex that can be adjacent to these nodes are the nodes associated with a, b, c, and d. The node associated with a may be adjacent to node q, the node associated with b may be adjacent to node r, the node associated with c may be adjacent to k, and the node associated with d must be adjacent to either k, q, or r. Hence we have the subtree depicted in the left-hand side of part II of Figure 9.6. The possible adjacencies of the nodes associated with a, b, c, and d are denoted by dashed lines. We replace a part of this subtree by a new subtree, as is depicted in the right-hand side of part II of Figure 9.6. The dashed lines again denote possible adjacencies, which are the same as in the left-hand side. Note that the new tree decomposition satisfies conditions 1 – 7 of Definition 9.2.2, and hence it is a special tree decomposition.

It is easy to see that this construction can be done in $O(1)$ time: find the node associated with edge $\{c,d\}$, and search for the structure of the subtree. Then replace the proper subtree by the new subtree. It can also be seen that round 3 of algorithm A_R^{tw} is non-interfering: the replaced nodes are not involved in any other application of rule 3. Only the node associated with b and node j' may be involved in other applications, but for these nodes, only the contents of one entry in the adjacency list are changed, and this can be done in parallel for

different entries.

Rounds 4 – 18. The constructions for rules 4 – 18 are similar to the construction for rule 3, so we do not describe them. The rules of which the right-hand side contains a chordless four-cycle are a bit different, as there are two possibilities for the structure of the tree decomposition. As an example, we depict the two possible constructions for rule 6 in Figure 9.7.

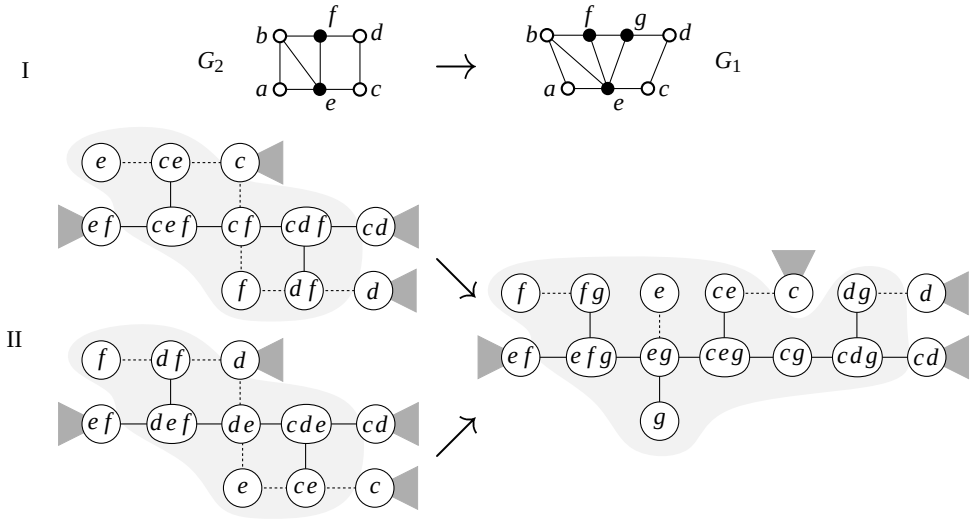


Figure 9.7. The construction for rule 6. Dashed lines denote possible adjacencies.

Round 19. Suppose r is rule 19. Let G_1 and G_2 be as depicted in part I of Figure 9.8 (labelings of edges are not shown). Note that edge e is a bridge, and hence the tree decomposition contains a subtree as depicted in the left-hand side of part II of Figure 9.8 (see also round 1). We replace this subtree by the subtree depicted in the right-hand side of part II of Figure 9.8. Note the resulting tree decomposition is special.

In order to make the algorithm non-interfering in round 19, the construction is as follows. We make two new nodes with contents $\{a,c\}$ and $\{c\}$, respectively, which are adjacent to each other. Furthermore, we make a new entry in the adjacency list of the node associated with a. The new entry is for the node associated with edge $\{a,c\}$. It is added between the entry for the node associated with edge e and its right neighbor in the list. In this way, we can make sure that no two constructions for rule 19 try to modify the same entry of the adjacency list of node i.

Round 20. Suppose r is rule 20. Let G_1 and G_2 be as depicted in part I of Figure 9.9. Let i denote the node associated with a. We apply the construction depicted in part II of Figure 9.9.

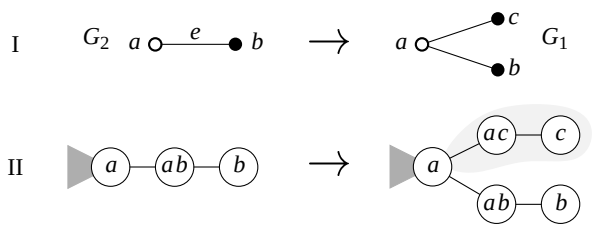


Figure 9.8. The construction for rule 19.

Note that the resulting tree decomposition is special.

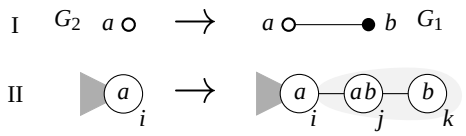


Figure 9.9. The construction for rule 20.

In order to make the algorithm non-interfering in round 20, the construction is as follows. Let H be a B-labeled terminal multigraph such that $G = G_2 \oplus H$. Note that, in $G_1 \oplus H$, vertex a has degree at most eight. We divide round 20 into eight subrounds, which are executed subsequently. First, edge e computes its rank k in the adjacency list of a in $G_1 \oplus H$. Then, in subround k, the construction is applied by adding new nodes j and l with $X_j = \{a,b\}$ and $X_l = \{a\}$ which are adjacent to each other, and making j adjacent to i: an extra entry for node j is added at the end of the adjacency list of node i. In each subround, at most one such construction that involves vertex a is applied, and hence the algorithm is non-interfering and runs in $O(1)$ time.

Rounds 21 – 23. Rules 21, 22 and 23 are very similar to each other. We give the construction only for rule 21. Let G_1 and G_2 be as depicted in part I of Figure 9.10. As edge e is a bridge, the tree decomposition must contain a subtree as depicted in the left-hand side of part II of Figure 9.10. This subtree is replaced by the subtree depicted in the right-hand side of this figure. It is easy to see that the new tree decomposition is special, and that the construction can be done in $O(1)$ time and is non-interfering.

This completes the description of algorithm A_R^{tw}. We have shown the following result.

Theorem 9.2.1. *S_{tw} is a special parallel constructive reduction system for* TW2.

9.3 Algorithms

In the previous section, we have given a special parallel constructive reduction system for the problem TW2. Theorem 6.3.1 (with the modifications for multigraphs as given in Section 5.4)

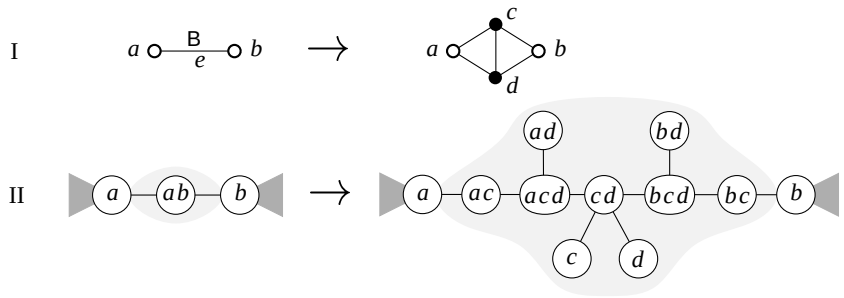


Figure 9.10. The construction for rule 21.

shows that this system immediately gives an algorithm for TW2 that runs in $O(\log m \log^* m)$ time on an EREW PRAM and in $O(\log m)$ time on a CRCW PRAM, in both cases with $O(m)$ operations and space.

We can use this algorithm for the same problem, but without requiring that the input graph is connected. In that case, we use a technique similar to the technique of Bodlaender and Hagerup [1995]: from each connected component of the graph we select one vertex. Then we add a new dummy vertex to the graph, and make all selected vertices adjacent to this dummy vertex. The new graph is connected, and has treewidth at most two if and only if the original graph has treewidth at most two. Now we solve the problem on the new connected graph with the reduction system given in the previous section. After that, we remove the dummy vertex from all nodes it occurs in, and the resulting tree decomposition is a tree decomposition of width at most two of the input graph.

The only problem is how to select a vertex from each connected component of the graph. We use the reduction system $(R_{\text{tw}}, I_{\text{tw}})$ for this, with a small extension: we add reduction rule 24 as depicted in Figure 9.11 to R_{tw}. Let R be the new set of reduction rules. It is easy to see that (R, I_{tw}) is a special parallel reduction system for TW2 on input graphs which do not have to be connected (see Definition 5.4.3).

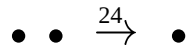


Figure 9.11. The extra rule 24.

Now we first make a copy of the input graph, and then apply a reduction algorithm on this copy, based on (R, I_{tw}). Each time the new rule is applied, we remove a connected component of the graph, by removing the only vertex that is left from this component. We then mark the vertex in the original graph that corresponds to this removed vertex. This marked vertex is the selected vertex of its connected component. If the input graph has treewidth at most two, then it is reduced to the graph in I_{tw} in $O(\log(n+m)\log^*(n+m))$ time

on an EREW PRAM and in $O(\log(n+m))$ time on a CRCW PRAM, both with $O(n+m)$ operations (note that it is not necessarily the case that $n = O(m)$ if the graph is not connected). Furthermore, in the original graph, each connected component has exactly one marked vertex. If the input graph has treewidth three or more, then the reduction algorithm will not succeed to reduce the graph within the given time bounds, and it stops in time, returning false. This gives the following result.

Theorem 9.3.1. *There is a parallel algorithm which checks whether a given (*B*-labeled) multigraph G has treewidth at most two, and if so, returns a tree decomposition of width at most two of G. The algorithm uses $O(n+m)$ operations and space, and $O(\log(n+m))$ time on a CRCW PRAM, or $O(\log(n+m)\log^*(n+m))$ time on an EREW PRAM.*

If the input graph $G = (V,E)$ is simple, then we can use the same preprocessing step as described in Chapter 8 for series-parallel graphs (see page 185) to check whether $|E| \leq 2|V|$. This step takes $O(\log n)$ time and $O(n)$ operations on an EREW or CRCW PRAM. If the preprocessing step does not return false, then we know that $|E| \leq 2|V|$, and hence we have the following result.

Theorem 9.3.2. *There is a parallel algorithm which checks whether a given simple graph G has treewidth at most two, and if so, returns a tree decomposition of width at most two of G. The algorithm uses $O(n)$ operations and space, and $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ time on an EREW PRAM.*

In Section 2.2.3, we have argued that many (constructive) optimization and decision problems can be solved on graphs of bounded treewidth in $O(\log n)$ time with $O(n)$ operations on an EREW PRAM, if a tree decomposition of bounded width of the input graph is given (with $O(n)$ nodes). These problems include all (constructive) decision problems that are MS-definable. Together with Theorem 9.3.2, this shows that all these problems can be solved within the same resource bounds as the problem of finding a tree decomposition of width at most two, if the input graph has treewidth at most two.

One of the problems which can be solved if a tree decomposition of bounded width of the input graph is given, is the pathwidth problem: given a graph G and an integer constant k, check whether G has pathwidth at most k, and if so, find a path decomposition of width at most k of the graph [Bodlaender and Hagerup, 1995]. Hence we have the following result.

Theorem 9.3.3. *Let $k \geq 1$ be an integer constant. There is a parallel algorithm which checks whether a given graph G has treewidth at most two and pathwidth at most k, and if so, returns a path decomposition of width at most k of G. The algorithm uses $O(n)$ operations and space, and $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ time on an EREW PRAM.*

Note that the theorem also holds for multigraphs, if we replace n by $n+m$ in the time and operations bounds. As graphs of pathwidth at most two also have treewidth at most two, the theorem implies that we can find a path decomposition of width at most two of a graph, if one exists, within the same resource bounds.

Chapter 10

Conclusions

In this chapter we summarize the results presented in this thesis, and we give some remarks and directions for further research.

In Chapter 3, we give a complete characterization of partial two-paths, and we use this characterization to obtain a linear time algorithm for building a path decomposition of width at most two of a graph, if one exists. This algorithm has no hidden constants and is easier to implement than the linear time algorithm of Bodlaender [1996a] for recognizing partial k-paths for any fixed positive integer k. This makes it potentially more practical than Bodlaender's algorithm for the case that $k = 2$.

An obvious generalization of the results of Chapter 3 would be to find a characterization of partial k-paths for $k \geq 3$, and to make an efficient and practical algorithm for recognizing partial k-paths, based on this characterization. It seems however that this is not feasible with the method used in Chapter 3, since the characterization of partial k-paths is already quite complicated for the case $k = 2$ and the complexity grows quickly with k.

In Chapter 4, we discuss two problems that originate from molecular biology, namely k-INTERVALIZING SANDWICH GRAPHS (k-ISG) and k-UNIT-INTERVALIZING SANDWICH GRAPHS (k-UISG). We give an algorithm for 2-ISG that runs in linear time in the number of vertices of the input graph, and an algorithm for 3-ISG that runs in quadratic time in the number of vertices. Furthermore, we show that for $k \geq 4$, k-ISG is NP-complete. We also give an algorithm for 3-UISG which is linear in the number of edges of the graph. Kaplan et al. [1994] have given an $O(n^{k-1})$ algorithm for k-UISG. Hence our algorithm is more efficient for the case that $k = 3$. The given algorithms for 3-ISG and 3-UISG are based on the characterization of partial two-paths as presented in Chapter 3. The algorithms have no hidden constants, but they consist of an extensive case analysis, thus making the algorithms very large. It would be nice to find more compact algorithms running in the same time bounds.

Unfortunately, in most practical instances of k-ISG and k-UISG, k lies between five and fifteen. For these cases, there is (probably) no polynomial time algorithm for k-ISG. For k-UISG there is an $O(n^{k-1})$ algorithm, but if $k = 10$ for example, this algorithm is not practical. Besides, k-UISG is $W[1]$-hard [Kaplan et al., 1994], which means that there is probably no algorithm which solves k-UISG in $O(n^c)$ time where c is a constant which does not depend on k. Thus the results for k-ISG and k-UISG are not very helpful and it might be interesting to take a closer look at the original problems as they occur in biology. It might for instance

be that problems which model the practical situation more accurately have more efficient algorithms, or there might be practical situations in which extra constraints hold that make the problems easier.

In Chapters 5 – 9 we discuss reduction algorithms. Chapter 5 contains an overview of results in Arnborg et al. [1993], Bodlaender [1994] and Bodlaender and Hagerup [1995], presented in a uniform setting. We give definitions of reduction systems for both decision problems and optimization problems. We also give a number of conditions for reduction systems which ensure the existence of efficient reduction algorithms based on these systems. For the sequential case, these algorithms use linear time and space. In parallel they use $O(n)$ operations and space, and $O(\log n \log^* n)$ time on an EREW PRAM or $O(\log n)$ time on a CRCW PRAM. We also apply these results to problems on graphs of bounded treewidth: we show that for all finite index decision problems and for all finite integer index optimization problems on graphs of bounded treewidth there exist reduction systems for which the efficient reduction algorithms can be used, and these systems can be automatically generated in most cases.

In Chapter 6 we extend the results of Chapter 5 to constructive decision and optimization problems: we define constructive reduction systems, and we give efficient sequential and parallel algorithms based on constructive reduction systems that satisfy some additional constraints. These algorithms do not only decide problems or find optimal values for optimization problems, but they also find an (optimal) solution for the problem, if one exists. They use the same resource bounds as the reduction algorithms presented in Chapter 5. We again apply these results to graphs of bounded treewidth: we show that all finite (integer) index problems of which the solution domain satisfies some extra constraints have constructive reduction systems which admit efficient algorithms.

In Chapter 7, we apply the results of Chapters 5 and 6 to a number of constructive optimization problems on graphs of bounded treewidth. We also show that a number of optimization problems which are MS-definable, are not finite integer index, which means that the technique presented in Chapters 5 and 6 can not be applied to all MS-definable optimization problems.

In Chapters 5 and 6, we have shown that efficient reduction algorithms can be used for a large class of problems on graphs of bounded treewidth. However, we did not investigate any other graph classes: it might well be that the efficient reduction algorithms can be used for problems on other classes of graphs, as long as the graphs are sparse (i.e. the number of edges of each graph must be at most a constant times the number of vertices). It would be interesting to find such classes of graphs, and the problems for which these algorithms can be used.

The reduction algorithms presented in Chapters 5 and 6 are simple, and not hard to implement. As long as the number of reduction rules in a reduction system is not too large, these algorithms are probably also efficient in practice. For a large class of problems on graphs of bounded treewidth, a set of reduction rules can automatically be generated. The size of this set may become very large, and the generation process may take a long time. It would be interesting to find out how long this process actually takes and to see whether it can be

made quicker. It is also interesting to find out how large the set of reduction rules can get, and to try to keep the set of generated rules as small as possible. This might for instance be done by applying self-reduction on the reduction rules: if a reduction rule r contains a match to another reduction rule r', then rule r can be removed (as soon as rule r is applicable to some graph, rule r' is also applicable). If a set of reduction rules is large, then a way to improve the efficiency of a reduction algorithm based on this set is to use the structure of the reduction rules. Consider for instance the set of reduction rules for series-parallel graphs presented in Figure 8.3: rules 3 – 18 in this set are all very much alike. This means that if the algorithm tries to find a match to one of these rules, it does not have to do this for each rule independently, but it can make use of the fact that many of these rules have the same subgraphs.

As mentioned in Chapter 2, all MS-definable decision problems are of finite index, and thus can be solved on graphs of bounded treewidth using an efficient reduction algorithm (apply the technique from Chapter 5). For optimization problems, this does not hold: we have shown in Chapter 7 that there are MS-definable optimization problems which are not of finite integer index. It might be interesting to find out whether there is a method with which all MS-definable optimization problems can be solved by using a type of reduction algorithm. It is also interesting to find a language like MSOL to define optimization problems which are finite integer index.

The constructive reduction algorithms presented in Chapter 6 can be applied to constructive decision problems which are of finite index and constructive optimization problems which are of finite integer index, as long as the structure of solutions is suitable. This is the case for many problems, but for instance not for k-TREEWIDTH and k-PATHWIDTH. It would be interesting to extend the methods presented in Chapter 6 such that they can also be applied to these problems. A start with this is made for instance for 2-TREEWIDTH in Chapter 9.

In Chapters 8 and 9, we give efficient parallel reduction algorithms for the problem of finding an sp-tree of a graph, if it is series-parallel, and the problem of finding a tree decomposition of width at most two of a graph, if it has treewidth at most two. We do this by giving constructive reduction systems for both problems, which can then be used in the efficient parallel constructive reduction algorithm as given in Chapter 6. Both algorithms improve in efficiency on previously known parallel algorithms for these problems. The reduction system for 2-TREEWIDTH is an extension of the reduction system for series-parallel graphs, since series-parallel graphs have treewidth at most two. It would be interesting to extend the reduction system for 2-TREEWIDTH to a reduction system for 3-TREEWIDTH or even k-TREEWIDTH for any fixed k. These reduction systems however might become too large to be practical, and thus new techniques might be necessary to turn the algorithms into practical algorithms (as is also described above).

Using a result from Bodlaender and Hagerup [1995] and the parallel algorithm for 2-TREEWIDTH, we also have an efficient parallel algorithm for 2-PATHWIDTH: first make a tree decomposition of width at most two of the graph, then apply a procedure from Bodlaender and Hagerup [1995] to construct a path decomposition of width at most two. Although this procedure runs in $O(\log n)$ time with $O(n)$ operations on an EREW PRAM, it is

not very practical. Therefore, it would be nice to find a more direct parallel algorithm for 2-PATHWIDTH, for instance by building a constructive reduction system and applying the results of Chapter 6.

Appendix A

Graph Problems

In this appendix, we give definitions of the graph problems that are used in this thesis. Most graph problems are defined both for simple graphs and for multigraphs. If the problems are mentioned in Garey and Johnson [1979], we give the number of the problem in this book between square brackets.

For almost all of the problems defined in this appendix, there are two decision variants and one optimization variant. In the first decision variant, say PROBLEM, the instance consists of a graph G and an integer k, and we ask whether a solution exists for the graph which has value at least or at most k. In the second decision variant, the integer is taken to be some constant k, and the instance consists of a graph only. The question is the same as for the first decision variant, and the problem is denoted by k-PROBLEM for any fixed integer k. In the optimization variant, the instance is a graph, and we ask for the maximum or minimum value of the integer k for which the decision problem has a 'yes' answer. The problem is denoted by MAX PROBLEM or MIN PROBLEM, respectively (unless stated otherwise).

In this appendix, we only give definitions of the first variant of the decision problem: the definitions of the other two variants follow directly. Each of the defined problems has a constructive version, which follows directly from the context.

For definitions of tree decompositions and treewidth, see Definition 2.2.1. For path decompositions and pathwidth, see Definition 2.2.2.

TREEWIDTH
Instance: A graph $G = (V, E)$, integer $k \geq 1$.
Question: Does G have treewidth at most k, i.e. does G have a tree decomposition of width at most k?

PATHWIDTH
Instance: A graph $G = (V, E)$, integer $k \geq 1$.
Question: Does G have pathwidth at most k, i.e. does G have a path decomposition of width at most k?

A *Hamiltonian circuit* in a graph G is a simple cycle in G containing all vertices of G. A *Hamiltonian path* in a graph G is a path in G containing all vertices of G.

HAMILTONIAN CIRCUIT [GT37]
Instance: A graph $G = (V, E)$.
Question: Does G contain a Hamiltonian circuit?

HAMILTONIAN PATH [GT39]
Instance: A graph $G = (V, E)$.
Question: Does G contain a Hamiltonian path?

An *independent set* of a graph G is a set $W \subseteq V(G)$ such that no two vertices in W are adjacent in G.

INDEPENDENT SET [GT20]
Instance: A graph $G = (V, E)$, integer $k \geq 0$.
Question: Does G have an independent set of cardinality at least k?

For any integer $k \geq 1$, a *k-coloring* of a graph G is a partition $(V_1, \ldots, V_k)$ of $V(G)$ such that for each i, V_i is an independent set of G.

COLORABILITY [GT4]
Instance: A graph $G = (V, E)$, an integer $k \geq 1$.
Question: Does G have a k-coloring?

The minimization problem in which we ask for a minimum value of k for which a k-coloring exists, is denoted by CHROMATIC NUMBER.

For definitions of layouts and bandwidth, see Definition 2.3.2.

BANDWIDTH [GT40]
Instance: A graph $G = (V, E)$, an integer k.
Question: Does G have bandwidth at most k, i.e. does G have a layout of bandwidth at most k?

The following problem is defined for all integers $d \geq 0$.

INDUCED d-DEGREE SUBGRAPH
Instance: A graph $G = (V, E)$, integer $k \geq 0$.
Question: Is there a set $S \subseteq V$ such that all vertices in $G[S]$ have degree at most p, and $|S| \geq k$?

For $d = 0$, this is the INDEPENDENT SET problem.

VERTEX COVER [GT1]
Instance: A graph $G = (V, E)$, an integer $k \geq 1$.
Question: Is there a set $S \subseteq V$ such that for each edge $\{v, w\} \in E(G)$, $v \in S$ or $w \in S$, and $|S| \leq k$?

The following problem is defined for any fixed integer $p \geq 1$.

p-DOMINATING SET
Instance: A graph $G = (V, E)$, an integer $k \geq 1$.
Question: Is there a set $S \subseteq V$ such that all vertices in $V - S$ have at least p neighbors in S, and $|S| \leq k$?.

For $p = 1$, this is the DOMINATING SET problem, numbered [GT2].

A *cut* in a graph $G = (V, E)$ is a partition (V_1, V_2) of V.

LARGE CUT [ND16]
Instance: A graph $G = (V,E)$, integer $k \geq 1$.
Question: Does G have a cut (V_1,V_2) such that $|\{\{v,w\} \in E \mid v \in V_1 \wedge w \in V_2\}| \geq k$?

The corresponding maximization problem is called MAX CUT.

PARTITION INTO CLIQUES [GT15]
Instance: A graph $G = (V,E)$, an integer $k \geq 1$.
Question: Is there a partition $(V_1,\ldots,V_s)$ of V in which for each i, $1 \leq i \leq s$, $G[V_i]$ is a complete graph, and $s \leq k$?

COVERING BY CLIQUES [GT17]
Instance: A graph $G = (V,E)$, an integer $k \geq 1$.
Question: Is there a set $\{V_1,\ldots,V_s\}$, in which for each i, $1 \leq i \leq s$, $V_i \subseteq V$, $G[V_i]$ is a complete graph, and for each edge $e \in E$, there is an i, $1 \leq i \leq s$, such that $e \in E(G[V_i])$, and furthermore, $s \leq k$?

HAMILTONIAN CIRCUIT COMPLETION [GT34]
Instance: A graph $G = (V,E)$, an integer $k \geq 0$.
Question: Is there a set $F \subseteq \{\{u,v\} \mid u,v \in V\}$, such that $G' = (V,E \cup F)$ contains a Hamiltonian circuit, and $|F| \leq k$?

HAMILTONIAN PATH COMPLETION
Instance: A graph $G = (V,E)$, an integer $k \geq 0$.
Question: Is there a set $F \subseteq \{\{u,v\} \mid u,v \in V\}$, such that $G' = (V,E \cup F)$ contains a Hamiltonian path, and $|F| \leq k$?.

A *spanning tree* of a graph $G = (V,E)$ is a subgraph $T = (V,F)$ of G which is a tree.

LEAF SPANNING TREE [ND2]
Instance: A graph $G = (V,E)$, an integer $k \geq 1$.
Question: Is there a spanning tree of G in which at least k vertices have degree one?

LONG PATH [ND29]
Instance: A graph $G = (V,E)$, an integer $k \geq 1$.
Question: Does G have a path of length at least k?

The corresponding maximization problem is called LONGEST PATH.

LONG CYCLE [ND28]
Instance: A graph $G = (V,E)$, an integer $k \geq 1$.
Question: Does G have a cycle of length $k \geq 1$?

The corresponding maximization problem is called LONGEST CYCLE.

The following seven problems are only used in Chapter 4. For definitions of sandwich graphs, k-intervalizations, k-unit-intervalizations, and the (proper) pathwidth and bandwidth of sandwich graphs, see Section 4.1 and Section 4.3.

INTERVALIZING SANDWICH GRAPHS (ISG)
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$.
Question: Is there a k-intervalization of S?

INTERVALIZING COLORED GRAPHS (ICG)
Instance: A simple graph $G = (V, E)$, an integer $k \geq 1$ and a k-coloring c for G.
Question: Is there a k-intervalization of G and c?

UNIT-INTERVALIZING SANDWICH GRAPHS (UISG)
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$.
Question: Is there a k-unit-intervalization of S?

UNIT-INTERVALIZING COLORED GRAPHS (UICG)
Instance: A simple graph $G = (V, E)$, an integer $k \geq 1$ and a k-coloring c for G.
Question: Is there a k-unit-intervalization of G and c?

SANDWICH PATHWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$.
Question: Does S have pathwidth at most $k-1$?

SANDWICH PROPER PATHWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, an integer $k \geq 1$.
Question: Does S have proper pathwidth at most $k-1$, i.e. is there a proper path decomposition of S?

SANDWICH BANDWIDTH
Instance: A sandwich graph $S = (V, E_1, E_2)$, integer $k \geq 1$.
Question: Does S have bandwidth at most $k-1$, i.e. is there a legal layout of bandwidth at most $k-1$ of S?

The following two problems are only used in Chapter 8. For definitions of source-sink labeled graphs, series-parallel graphs and sp-trees, see Section 2.3.3.

SOURCE-SINK LABELED SERIES-PARALLEL GRAPH
Instance: A source-sink labeled multigraph (G, s, t).
Question: Is (G, s, t) series-parallel, i.e. is there an sp-tree for (G, s, t)?

For directed input graphs, this problem is denoted by DLSPG, for indirected input graphs by LSPG.

SERIES-PARALLEL GRAPH
Instance: A multigraph G.
Question: Is G series-parallel, i.e. is there an sp-tree for G?

For directed input graphs, this problem is denoted by DSPG, for indirected input graphs by SPG.

The following problem is only used in Chapter 9 (for definitions of a labeled multigraph and the treewidth of such a graph, see Section 9.1).

TREEWIDTH AT MOST TWO (TW2)
Instance: A connected labeled multigraph G.
Question: Does G have treewidth at most two, i.e. is there a tree decomposition of width at most two of G?

References

ABRAHAMSON, K. R., N. DADOUN, D. G. KIRKPATRICK, AND T. PRZYTYCKA [1989]. A simple parallel tree contraction algorithm. *J. Algorithms 10*, 287–302.

ABRAHAMSON, K. R. AND M. R. FELLOWS [1993]. Finite automata, bounded treewidth and well-quasiordering. In N. Robertson and P. Seymour (Eds.), *Proceedings of the AMS Summer Workshop on Graph Minors, Graph Structure Theory*, Volume 147 of Contemporary Mathematics, pp. 539–564. American Math. Soc., Providence, Rhode Island.

AGARWALA, R. AND D. FERNÁNDEZ-BACA [1993]. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pp. 140–147. IEEE Computer Science Press, Los Alamitos, California.

ARNBORG, S. [1985]. Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey. *BIT 25*, 2–23.

ARNBORG, S., D. G. CORNEIL, AND A. PROSKUROWSKI [1987]. Complexity of finding embeddings in a *k*-tree. *SIAM J. Alg. Disc. Meth. 8*, 277–284.

ARNBORG, S., B. COURCELLE, A. PROSKUROWSKI, AND D. SEESE [1993]. An algebraic theory of graph reduction. *J. ACM 40*, 1134–1164.

ARNBORG, S., J. LAGERGREN, AND D. SEESE [1991]. Easy problems for tree-decomposable graphs. *J. Algorithms 12*, 308–340.

ARNBORG, S. AND A. PROSKUROWSKI [1986]. Characterization and recognition of partial 3-trees. *SIAM J. Alg. Disc. Meth. 7*, 305–314.

BERN, M. W., E. L. LAWLER, AND A. L. WONG [1987]. Linear time computation of optimal subgraphs of decomposable graphs. *J. Algorithms 8*, 216–235.

BODLAENDER, H. L. [1988a]. Dynamic programming algorithms on graphs with bounded tree-width. In T. Lepistö and A. Salomaa (Eds.), *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, Volume 317 of Lecture Notes in Computer Science, pp. 105–119. Springer-Verlag, Berlin.

BODLAENDER, H. L. [1988b]. NC-algorithms for graphs with small treewidth. In J. van Leeuwen (Ed.), *Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science WG'88*, Volume 344 of Lecture Notes in Computer Science, pp. 1–10. Springer-Verlag, Berlin.

BODLAENDER, H. L. [1993]. A tourist guide through treewidth. *Acta Cybernetica 11*, 1–23.

BODLAENDER, H. L. [1994]. On reduction algorithms for graphs with small treewidth. In J. van Leeuwen (Ed.), *Proceedings of the 19th International Workshop on Graph-*

Theoretic Concepts in Computer Science WG'93, Volume 790 of Lecture Notes in Computer Science, pp. 45–56. Springer-Verlag, Berlin.

BODLAENDER, H. L. [1996a]. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput. 25*, 1305–1317.

BODLAENDER, H. L. [1996b]. A partial *k*-arboretum of graphs with bounded treewidth. Technical Report UU-CS-1996-02, Department of Computer Science, Utrecht University, Utrecht.

BODLAENDER, H. L. AND B. DE FLUITER [1995]. Intervalizing *k*-colored graphs. In Z. Fülöp and F. Gécseg (Eds.), *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, Volume 944 of Lecture Notes in Computer Science, pp. 87–98. Springer-Verlag, Berlin. Full version appeared as Tech. Rep. UU-CS-1995-15, Department of Computer Science, Utrecht University.

BODLAENDER, H. L. AND B. DE FLUITER [1996a]. On intervalizing *k*-colored graphs for DNA physical mapping. *Disc. Appl. Math. 71*, 55–77.

BODLAENDER, H. L. AND B. DE FLUITER [1996b]. Parallel algorithms for series parallel graphs. In J. Diaz and M. Serna (Eds.), *Proceedings of the 4th Annual European Symposium on Algorithms ESA'96*, Volume 1136 of Lecture Notes in Computer Science, pp. 277–289. Springer-Verlag, Berlin.

BODLAENDER, H. L. AND B. DE FLUITER [1996c]. Reduction algorithms for constructing solutions in graphs with small treewidth. In J.-Y. Cai and C. K. Wong (Eds.), *Proceedings of the 2nd Annual International Conference on Computing and Combinatorics COCOON'96*, Volume 1090 of Lecture Notes in Computer Science, pp. 199–208. Springer-Verlag, Berlin.

BODLAENDER, H. L., R. G. DOWNEY, M. R. FELLOWS, M. T. HALLETT, AND H. T. WAREHAM [1995]. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences 11*, 49–57.

BODLAENDER, H. L., M. R. FELLOWS, AND M. HALLETT [1994]. Beyond *NP*-completeness for problems of bounded width: Hardness for the *W*-hierarchy. In *Proceedings of the 26th Annual Symposium on Theory of Computing*, pp. 449–458. ACM Press, New York.

BODLAENDER, H. L., M. R. FELLOWS, AND T. J. WARNOW [1992]. Two strikes against perfect phylogeny. In W. Kuich (Ed.), *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Volume 623 of Lecture Notes in Computer Science, pp. 273–283. Springer-Verlag, Berlin.

BODLAENDER, H. L., J. R. GILBERT, H. HAFSTEINSSON, AND T. KLOKS [1995]. Approximating treewidth, pathwidth, and minimum elimination tree height. *J. Algorithms 18*, 238–255.

BODLAENDER, H. L. AND T. HAGERUP [1995]. Parallel algorithms with optimal speedup for bounded treewidth. In Z. Fülöp and F. Gécseg (Eds.), *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, Volume 944 of Lecture Notes in Computer Science, pp. 268–279. Springer-Verlag, Berlin.

BODLAENDER, H. L. AND T. KLOKS [1993]. A simple linear time algorithm for triangulating three-colored graphs. *J. Algorithms 15*, 160–172.

BODLAENDER, H. L. AND T. KLOKS [1996]. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms 21*, 358–402.

BODLAENDER, H. L., T. KLOKS, AND D. KRATSCH [1993]. Treewidth and pathwidth of permutation graphs. In A. Lingas, R. Karlsson, and S. Carlsson (Eds.), *Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, Volume 700 of Lecture Notes in Computer Science, pp. 114–125. Springer-Verlag, Berlin.

BODLAENDER, H. L. AND R. H. MÖHRING [1993]. The pathwidth and treewidth of cographs. *SIAM J. Disc. Meth. 6*, 181–188.

BODLAENDER, H. L., R. B. TAN, D. M. THILIKOS, AND J. VAN LEEUWEN [1995]. On interval routing schemes and treewidth. In M. Nagl (Ed.), *Proceedings of the 21th International Workshop on Graph-Theoretic Concepts in Computer Science WG'95*, Volume 1017 of Lecture Notes in Computer Science, pp. 181–186. Springer-Verlag, Berlin.

BOOTH, K. S. AND G. S. LUEKER [1976]. Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-tree algorithms. *J. Comp. Syst. Sc. 13*, 335–379.

BORIE, R. B., R. G. PARKER, AND C. A. TOVEY [1991]. Deterministic decomposition of recursive graph classes. *SIAM J. Disc. Meth. 4*, 481–501.

BORIE, R. B., R. G. PARKER, AND C. A. TOVEY [1992]. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica 7*, 555–581.

BRYANT, R. L., M. R. FELLOWS, N. G. KINNERSLEY, AND M. A. LANGSTON [1987]. On finding obstruction sets and polynomial-time algorithms for gate matrix layout. In *Proceedings of the 25th Allerton Conference on Communication, Control and Computing*, pp. 397–298.

CHANDRASEKHARAN, N. AND S. T. HEDETNIEMI [1988]. Fast parallel algorithms for tree decomposing and parsing partial *k*-trees. In *Proceedings of the 26th Annual Allerton Conference on Communication, Control, and Computing*, pp. 283–292.

COOK, W. AND P. D. SEYMOUR [1993]. An algorithm for the ring-routing problem. Bellcore technical memorandum, Bellcore.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST [1989]. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts.

CORNEIL, D. G., H. KIM, S. NATARAJAN, S. OLARIU, AND A. P. SPRAGUE [1995]. Simple linear time recognition of unit interval graphs. *Inform. Proc. Letters 55*, 99–104.

COURCELLE, B. [1990]. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation 85*, 12–75.

COURCELLE, B. AND J. LAGERGREN [1996]. Equivalent definitions of recognizability for sets of graphs of bounded tree-width. *Mathematical Structures in Computer Science 6*, 141–166.

COURCELLE, B. AND M. MOSBAH [1993]. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comp. Sc. 109*, 49–82.

DE FLUITER, B. AND H. L. BODLAENDER [1997]. Intervalizing sandwich graphs. Technical Report UU-CS-1997-04, Department of Computer Science, Utrecht University, Utrecht.

DEO, N., M. S. KRISHNAMOORTY, AND M. A. LANGSTON [1987]. Exact and approximate solutions for the gate matrix layout problem. *IEEE Trans. Computer Aided Design 6*, 79–84.

DINNEEN, M. J. [1995]. *Bounded Combinatorial Width and Forbidden Substructures*. Ph.D. thesis, University of Victoria.

DOWNEY, R. G. AND M. R. FELLOWS [1995]. Fixed-parameter tractability and completeness I: Basic results. *SIAM J. Comput. 24*, 873–921.

DUFFIN, R. J. [1965]. Topology of series-parallel graphs. *J. Math. Anal. Appl. 10*, 303–318.

ELLIS, J. A., I. H. SUDBOROUGH, AND J. TURNER [1994]. The vertex separation and search number of a graph. *Information and Computation 113*, 50–79.

EPPSTEIN, D. [1992]. Parallel recognition of series parallel graphs. *Information and Computation 98*, 41–55.

FELLOWS, M. R., M. T. HALLETT, AND H. T. WAREHAM [1993]. DNA physical mapping: Three ways difficult (extended abstract). In T. Lengauer (Ed.), *Proceedings of the 1st Annual European Symposium on Algorithms ESA'96*, Volume 726 of Lecture Notes in Computer Science, pp. 157–168. Springer-Verlag, Berlin.

FELLOWS, M. R. AND M. A. LANGSTON [1989]. An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pp. 520–525. IEEE Computer Science Press, Los Alamitos, California.

FELLOWS, M. R. AND M. A. LANGSTON [1992]. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM J. Disc. Meth. 5*, 117–126.

GAREY, M. R., R. L. GRAHAM, D. S. JOHNSON, AND D. E. KNUTH [1978]. Complexity results for bandwidth minimization. *SIAM J. Appl. Math. 34*, 477–495.

GAREY, M. R. AND D. S. JOHNSON [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company.

GAVRIL, F. [1974]. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Comb. Theory Series B 16*, 47–56.

GOLDBERG, P. W., M. C. GOLUMBIC, H. KAPLAN, AND R. SHAMIR [1995]. Four strikes against physical mapping of DNA. *J. Computational Biology 2*(1), 139–152.

GOLUMBIC, M. C., H. KAPLAN, AND R. SHAMIR [1994]. On the complexity of DNA physical mapping. *Advances in Applied Mathematics 15*, 251–261.

GRANOT, D. AND D. SKORIN-KAPOV [1991]. NC algorithms for recognizing partial 2-trees and 3-trees. *SIAM J. Disc. Meth. 4*(3), 342–354.

GURARI, E. M. AND I. H. SUDBOROUGH [1984]. Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. *J. Algorithms 5*, 531–546.

HAGERUP, T. [1988]. On saving space in parallel computation. *Inform. Proc. Letters 29*, 327–329.

HARARY, F. [1969]. *Graph Theory*. Addison-Wesley Publishing Company, Reading, Massachusetts.

HE, X. [1991]. Efficient parallel algorithms for series-parallel graphs. *J. Algorithms 12*, 409–430.

HE, X. AND Y. YESHA [1987]. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation 75*, 15–38.

HSU, W.-L. [1993]. A simple test for interval graphs. In *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science WG'92*, Volume 657 of Lecture Notes in Computer Science, pp. 11–16. Springer-Verlag, Berlin.

JÁJÁ, J. [1992]. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts.

JOHNSON, D. S. [1985]. The NP-completeness column: An ongoing guide. *J. Algorithms 6*, 434–451.

JOHNSON, D. S. [1987]. The NP-completeness column: An ongoing guide. *J. Algorithms 8*, 285–303.

JUNGCK, J. R., G. DICK, AND A. G. DICK [1982]. Computer-assisted sequencing, interval graphs, and molecular evolution. *BioSystems 15*, 259–273.

KANNAN, S. AND T. WARNOW [1990]. Inferring evolutionary history from DNA sequences. In *Proceedings of the 31rd Annual Symposium on Foundations of Computer Science*, pp. 362–371. IEEE Computer Science Press, Los Alamitos, California.

KANNAN, S. AND T. WARNOW [1992]. Triangulating 3-colored graphs. *SIAM J. Disc. Meth. 5*, 249–258.

KAPLAN, H. AND R. SHAMIR [1996]. Pathwidth, bandwidth and completion problems to proper interval graphs with small cliques. *SIAM J. Comput. 25*, 540–561.

KAPLAN, H., R. SHAMIR, AND R. E. TARJAN [1994]. Tractability of parameterized completion problems on chordal and interval graphs: Minimum fill-in and physical mapping. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 780–791. IEEE Computer Science Press, Los Alamitos, California.

KARP, R. M. [1993]. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, pp. 278–285. ACM Press, New York.

KIKUNO, T., N. YOSHIDA, AND Y. KAKUDA [1983]. A linear algorithm for the domination number of a series-parallel graph. *Disc. Appl. Math. 5*, 299–311.

KINNERSLEY, N. G. AND M. A. LANGSTON [1994]. Obstruction set isolation for the gate matrix layout problem. *Disc. Appl. Math. 54*, 169–213.

KLOKS, T. [1994]. *Treewidth. Computations and Approximations*, Volume 842 of Lecture Notes in Computer Science. Springer-Verlag, Berlin.

KORNAI, A. AND Z. TUZA [1992]. Narrowness, pathwidth, and their application in natural language processing. *Disc. Appl. Math. 36*, 87–92.

KORTE, N. AND R. H. MÖHRING [1989]. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput. 18*, 68–81.

LAGERGREN, J. [1991]. *Algorithms and Minimal Forbidden Minors for Tree-decomposable Graphs*. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden.

LAGERGREN, J. [1996]. Efficient parallel algorithms for graphs of bounded tree-width. *J. Algorithms 20*, 20–44.

LAGERGREN, J. AND S. ARNBORG [1991]. Finding minimal forbidden minors using a finite congruence. In J. Leach Albers, B. Monien, and B. Rodríguez Artalejo (Eds.), *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, Volume 510 of Lecture Notes in Computer Science, pp. 532–543. Springer-Verlag, Berlin.

LAURITZEN, S. J. AND D. J. SPIEGELHALTER [1988]. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological) 50*, 157–224.

MATOUŠEK, J. AND R. THOMAS [1991]. Algorithms finding tree-decompositions of graphs. *J. Algorithms 12*, 1–22.

MCMORRIS, F. R., T. WARNOW, AND T. WIMER [1994]. Triangulating vertex-colored graphs. *SIAM J. Disc. Meth. 7*(2), 296–306.

MEHLHORN, K. [1984]. *Sorting and Searching*, Volume 1 of Data Structures and Algorithms. Springer-Verlag, Berlin.

MÖHRING, R. H. [1990]. Graph problems related to gate matrix layout and PLA folding. In E. Mayr, H. Noltemeier, and M. Sysło (Eds.), *Computational Graph Theory, Computing Suppl. 7*, pp. 17–51. Springer-Verlag, Berlin.

MONIEN, B. [1986]. The bandwidth minimization problem for caterpillars with hair length 3 is NP-complete. *SIAM J. Alg. Disc. Meth. 7*, 505–512.

RAMACHANDRAMURTHI, S. [1994]. *Algorithms for VLSI Layout Based on Graph Width Metrics*. Ph.D. thesis, Computer Science Department, University of Tennessee, Knoxville, Tennessee, USA.

REED, B. [1992]. Finding approximate separators and computing tree-width quickly. In *Proceedings of the 24th Annual Symposium on Theory of Computing*, pp. 221–228. ACM Press, New York.

ROBERTSON, N. AND P. D. SEYMOUR [1983]. Graph minors. I. Excluding a forest. *J. Comb. Theory Series B 35*, 39–61.

ROBERTSON, N. AND P. D. SEYMOUR [1984]. Graph minors. III. Planar tree-width. *J. Comb. Theory Series B 36*, 49–64.

ROBERTSON, N. AND P. D. SEYMOUR [1985]. Graph minors — a survey. In I. Anderson (Ed.), *Surveys in Combinatorics*, pp. 153–171. Cambridge Univ. Press.

ROBERTSON, N. AND P. D. SEYMOUR [1986a]. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms 7*, 309–322.

ROBERTSON, N. AND P. D. SEYMOUR [1986b]. Graph minors. V. Excluding a planar graph. *J. Comb. Theory Series B 41*, 92–114.

ROBERTSON, N. AND P. D. SEYMOUR [1986c]. Graph minors. VI. Disjoint paths across a disc. *J. Comb. Theory Series B 41*, 115–138.

ROBERTSON, N. AND P. D. SEYMOUR [1988]. Graph minors. VII. Disjoint paths on a surface. *J. Comb. Theory Series B 45*, 212–254.

ROBERTSON, N. AND P. D. SEYMOUR [1990a]. Graph minors. IV. Tree-width and well-quasi-ordering. *J. Comb. Theory Series B 48*, 227–254.

ROBERTSON, N. AND P. D. SEYMOUR [1990b]. Graph minors. IX. Disjoint crossed paths. *J. Comb. Theory Series B 49*, 40–77.

ROBERTSON, N. AND P. D. SEYMOUR [1990c]. Graph minors. VIII. A Kuratowski theorem for general surfaces. *J. Comb. Theory Series B 48*, 255–288.

ROBERTSON, N. AND P. D. SEYMOUR [1991a]. Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory Series B 52*, 153–190.

ROBERTSON, N. AND P. D. SEYMOUR [1991b]. Graph minors. XVI. Excluding a non-planar graph. Manuscript.

ROBERTSON, N. AND P. D. SEYMOUR [1994]. Graph minors. XI. Distance on a surface. *J. Comb. Theory Series B 60*, 72–106.

ROBERTSON, N. AND P. D. SEYMOUR [1995a]. Graph minors. XII. Excluding a non-planar graph. *J. Comb. Theory Series B 64*, 240–272.

ROBERTSON, N. AND P. D. SEYMOUR [1995b]. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory Series B 63*, 65–110.

ROBERTSON, N. AND P. D. SEYMOUR [1995c]. Graph minors. XIV. Extending an embedding. *J. Comb. Theory Series B 65*, 23–50.

ROBERTSON, N. AND P. D. SEYMOUR [1996]. Graph minors. XV. Giant steps. *J. Comb. Theory Series B 68*, 112–148.

ROSE, D. J., R. E. TARJAN, AND G. S. LUEKER [1976]. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput. 5*, 266–283.

SANDERS, D. P. [1996]. On linear recognition of tree-width at most four. *SIAM J. Disc. Meth. 9*(1), 101–117.

SCHEFFLER, P. [1987]. Linear-time algorithms for NP-complete problems restricted to partial *k*-trees. Report R-MATH-03/87, Karl-Weierstrass-Institut Für Mathematik, Berlin, GDR.

SCHEFFLER, P. [1989]. *Die Baumweite von Graphen als ein Maß für die Kompliziertheit algorithmischer Probleme*. Ph.D. thesis, Akademie der Wissenschaften der DDR, Berlin.

TAKAHASHI, A., S. UENO, AND Y. KAJITANI [1994]. Minimal acyclic forbidden minors for the family of graphs with bounded path-width. *Disc. Math. 127*(1/3), 293–304.

TAKAMIZAWA, K., T. NISHIZEKI, AND N. SAITO [1982]. Linear-time computability of combinatorial problems on series-parallel graphs. *J. ACM 29*, 623–641.

THORUP, M. [1995]. Structured programs have small tree-width and good register allocation. Technical Report DIKU-TR-95/18, Department of Computer Science, University of Copenhagen, Denmark.

VALDES, J., R. E. TARJAN, AND E. L. LAWLER [1982]. The recognition of series parallel digraphs. *SIAM J. Comput. 11*, 298–313.

VAN DER GAAG, L. C. [1990]. *Probability-Based Models for Plausible Reasoning*. Ph.D. thesis, University of Amsterdam.

WIMER, T. V. [1987]. *Linear Algorithms on k-Terminal Graphs*. Ph.D. thesis, Dept. of Computer Science, Clemson University.

Acknowledgments

Many people have contributed to the completion of this thesis. In this section I would like to thank all these people for their input and support.

Hans Bodlaender has introduced me to the fascinating area of treewidth and pathwidth. I thank him for being a stimulating supervisor: it has been a pleasure to work with a leading researcher in the field.

I thank Jan van Leeuwen for carefully reading a draft version of the manuscript and suggesting many improvements. Also I thank Jacques Verriet for voluntarily reading the entire thesis on his way to Japan (and back). Jacques has also pointed out a lot of corrections and improvements. I also thank Emile Aarts, Mike Fellows, Torben Hagerup and Mark Overmars for taking place in the reviewing committee.

I thank all my colleagues and ex-colleagues for providing a pleasant working environment. I am especially grateful to Thomas Arts for being a friend. He listened to all my complaints (and complained to me too) and drank lots of tea with me (with chocolate or cookies whenever we were *not* on a diet). I also want to thank Chritiene Aarts, Chantal Wentink and René van Oostrum for being such companionable office roommates.

I want to thank my family and friends for their moral support. Especially, I thank my parents and my grand-mother for having unconditional belief in me and my abilities. I want to thank my sister Inge de Fluiter for designing the thesis cover with me.

Last but not least I thank Hans van Antwerpen for being my partner in life. He simply makes life more enjoyable.

Samenvatting

Veel problemen uit de praktijk kunnen worden gemodelleerd als optimaliserings- of beslissingsproblemen op grafen. Denk bijvoorbeeld aan het probleem waarbij een koerier een aantal pakketjes moet afleveren op verschillende adressen in het land. De manager van de koerier wil dat hij een zo kort mogelijke route aflegt die begint en eindigt bij het koeriersbedrijf, en die alle adressen aandoet. Het probleem om zo'n kortste route te vinden is het zogenaamde *handelsreizigersprobleem*. De invoer kan worden gemodelleerd als een graaf, waarbij elke knoop in de graaf een adres vertegenwoordigt en elke kant tussen twee knopen de weg tussen de corresponderende adressen. Elke kant heeft een gewicht dat aangeeft hoe lang de corresponderende weg is. Het probleem is dan om een cykel in de graaf te vinden die alle knopen bevat en waarvoor de som van de gewichten van alle kanten in de cykel minimaal is.

Helaas is het zo dat de meeste problemen op grafen die praktische problemen modelleren lastig zijn in die zin, dat er waarschijnlijk geen efficiënte algoritmen zijn die deze problemen oplossen. Formeel gezegd zijn deze problemen NP-lastig. Het handelsreizigersprobleem is een voorbeeld hiervan. Een manier om hiermee om te gaan is om te kijken of er in het probleem uit de praktijk een structuur zit die maakt dat het probleem efficiënter is op te lossen. Het kan bijvoorbeeld zo zijn dat het gegeven probleem in het algemeen lastig is, maar dat de grafen die in de praktijk voorkomen een dusdanige structuur hebben dat er wel een efficiënt algoritme voor het probleem bestaat.

Een voorbeeld van een prettige graafstructuur is de *boomstructuur*: het blijkt dat veel graafproblemen die in het algemeen lastig zijn, een efficiënt algoritme hebben wanneer de graaf een boom is. Helaas is de boomstructuur erg beperkt: er zijn maar weinig praktische problemen die kunnen worden gemodelleerd als problemen op bomen. In dit proefschrift kijken we daarom naar een generalisatie van deze structuur, en dat is de boom*achtige* structuur: we kijken naar grafen met *boombreedte* hooguit k of *padbreedte* hooguit k, waarbij k een positief geheel getal is.

Intuïtief gezien geeft de boombreedte van een graaf de mate aan waarin de graaf op een boom lijkt: hoe groter de gelijkenis, hoe kleiner de boombreedte. Met een graaf van boombreedte k kan een boom worden geassocieerd waarbij elke knoop van de boom correspondeert met een deelgraaf van de graaf op een zodanige manier dat

- elke knoop en elke kant van de graaf in tenminste een knoop van de boom voorkomt, en
- voor elke knoop v in de graaf geldt dat de knopen in de boom die v bevatten een verbonden deelboom vormen.

Zo'n boom bestaande uit deelgrafen wordt een *boomdecompositie* van de graaf genoemd. De breedte van de boomdecompositie is het maximaal aantal knopen van de graaf dat in één

knoop van de boomdecompositie voorkomt, min één. De boombreedte van een graaf is de minimale breedte over alle boomdecomposities van de graaf (een boom heeft boombreedte één). Een *paddecompositie* van een graaf is een boomdecompositie die de vorm heeft van een pad. De padbreedte van een graaf is de minimale breedte over alle paddecomposities van de graaf. Dus de boombreedte van een graaf is altijd ten hoogste gelijk aan z'n padbreedte.

Voor veel problemen zoals het handelsreizigersprobleem is er een efficiënt algoritme op grafen met kleine boombreedte. Het blijkt dat er veel praktische graafproblemen zijn waarvoor de invoergraaf een kleine boombreedte heeft. Bij al deze problemen helpt dit gegeven bij het vinden van een efficiënter algoritme. Deze algoritmen maken meestal gebruik van een boomdecompositie van de graaf met kleine breedte. Daarom is het nodig om eerst zo'n boomdecompositie van de graaf te vinden. Hiervoor zijn efficiënte algoritmen beschikbaar, zowel sequentieel als parallel.

Helaas is het zo dat veel algoritmen op grafen met een kleine boombreedte alleen in theorie efficient zijn: de looptijd van de algoritmen is vaak exponentieel in de boombreedte van de graaf. Dit geldt bijvoorbeeld voor de algoritmen voor het vinden van een boom- of paddecompositie van breedte hooguit k van een graaf, waarbij k constant is.

Het doel van dit proefschrift is om efficiënte sequentiële en parallelle algoritmen te ontwerpen voor problemen op grafen met een kleine boom- of padbreedte. Het doel is om algoritmen te ontwerpen die niet alleen theoretisch efficiënt zijn, maar die ook in praktische toepassingen efficiënt kunnen zijn.

Het proefschrift is als volgt georganiseerd. Hoofdstuk 1 geeft een inleiding. In hoofdstuk 2 worden formele definities van boom- en padbreedte gegeven, en wordt een aantal eigenschappen en bekende resultaten over grafen met een kleine boom- en padbreedte gegeven. Verder worden definities gegeven die worden gebruikt in de rest van het proefschrift.

In hoofdstuk 3 geven we een volledige karakterisatie van grafen met padbreedte twee. Deze karakterisatie wordt vervolgens gebruikt voor een efficiënt sequentieel algoritme dat beslist of een graaf padbreedte ten hoogste twee heeft en, als dat zo is, een paddecompositie van minimale breedte vindt. De karakterisatie wordt ook gebruikt in de algoritmen die zijn beschreven in hoofdstuk 4.

Hoofdstuk 4 gaat over twee problemen welke hun oorsprong vinden in de moleculaire biologie. In beide problemen bestaat de invoer uit een aantal copieën van een DNA string welke in fragmenten zijn opgedeeld. Voor elk paar van fragmenten is informatie beschikbaar over de overlap tussen die twee fragmenten: óf we weten dat de fragmenten zeker overlappen, óf we weten dat ze zeker niet overlappen, óf we weten niets. Met behulp van deze informatie moet de volledige overlap informatie tussen elk tweetal fragmenten worden berekend, dat wil zeggen dat voor elk tweetal fragmenten moet worden berekend of ze wel of niet overlappen. Dit probleem heet k-INTERVALIZING SANDWICH GRAPHS of k-ISG, waarbij k het aantal copieën is dat is gefragmenteerd. In de tweede variant is ook nog bekend dat alle fragmenten gelijke lengte hebben. Deze variant heet k-UNIT-INTERVALIZING SANDWICH GRAPHS of k-UISG. De invoer van beide problemen kan worden gemodelleerd als een graaf. Het blijkt dat de volledige overlap informatie alleen kan worden berekend wanneer die graaf padbreedte ten hoogste k heeft, waarbij k weer het aantal copieën is. In Hoofdstuk 4 geven we een

kwadratisch algoritme voor 3-ISG, en we bewijzen dat k-ISG NP-moeilijk is wanneer $k \geq 4$. Verder geven we een lineair algoritme voor 3-UISG.

Hoofstukken 5 – 9 gaan over een speciaal soort algoritmen, namelijk *reductie algoritmen*. Een reductie algoritme is een algoritme waarin een reeks reducties wordt uitgevoerd op de invoergraaf. Het gedrag van de reducties is beschreven in een verzameling van reductie regels, welke afhangen van het probleem waarvoor het algoritme is. Wanneer de reductie regels aan bepaalde voorwaarden voldoen kan het reductie algoritme lineaire tijd gebruiken (of logaritmische tijd in het geval van een parallel reductie algoritme). De reductie algoritmen zijn eenvoudig: de moeilijkheden van het probleem zitten verstopt in de verzameling reductie regels, en niet in het algoritme.

Er zijn hele klassen van problemen op grafen met begrensde boombreedte waarvoor een verzameling van reductie regels kan worden geconstrueerd. Het voordeel van reductie algoritmen voor het oplossen van problemen op grafen met begrensde boombreedte is dat er geen boomdecompositie van de graaf nodig is: de algoritmen werken direct op de graaf.

In hoofdstuk 5 geven we een overzicht van de bestaande theorieën over reductie algoritmen. We combineren verschillende bestaande ideeën en presenteren ze als een geheel. Dit hoofdstuk is tevens een inleiding voor hoofdstukken 6 – 9.

Reductie algoritmen hebben als nadeel dat ze alleen optimaliserings- en beslissingsproblemen kunnen oplossen: bij een optimaliseringsprobleem wordt alleen de optimale waarde teruggegeven, maar niet een oplossing waarvoor de waarde optimaal is. Bij beslissingsproblemen wordt alleen het antwoord ‘ja’ of ‘nee’ gegeven, maar als het antwoord ‘ja’ is wordt geen oplossing gegeven. In hoofdstuk 6 breiden we de theorie van reductie algoritmen uit naar *constructieve reductie algoritmen* welke ook een (optimale) oplossing teruggeven, mits er een is. We laten zien dat voor veel problemen op grafen met begrensde boombreedte waarvoor reductie algoritmen kunnen worden toegepast, ook de constructieve reductie algoritmen kunnen worden toegepast.

In hoofdstuk 7 passen we de theorieën welke zijn gepresenteerd in hoofdstukken 5 en 6 toe op een aantal optimaliseringsproblemen.

In hoofdstukken 8 en 9 gebruiken we de theorieën uit hoofdstuk 6, aangevuld met nieuwe ideeën, om efficiënte, constructieve parallelle reductie algoritmen te verkrijgen voor de volgende twee aanverwante problemen:

- gegeven een graaf, bepaal of hij series-parallel is, en zo ja, vind dan een ‘sp-boom’ van de graaf,
- gegeven een graaf, bepaal of hij boombreedte hooguit twee heeft, en zo ja, maak een boomdecompositie van breedte twee van de graaf.

In hoofdstuk 10 vatten we de resultaten uit dit proefschrift nog eens samen, en geven we wat richtingen aan voor verder onderzoek.

Appendix A bevat een opsomming van definities van alle graafproblemen welke worden gebruikt in het proefschrift.

Curriculum Vitae

Babette Lucie Elisabeth de Fluiter

6 mei 1970
Geboren te Leende.

augustus 1982 - juni 1988
VWO aan het Hertog Jan College te Valkenswaard.

september 1988 - april 1993
Studie Technische Informatica aan de Technische Universiteit Eindhoven. Doctoraal diploma cum laude behaald op 29 april 1993.

juni 1993 - juni 1997
Onderzoeker-in-Opleiding aan de Vakgroep Informatica, Universiteit Utrecht. In dienst van de Stichting Informatica Onderzoek in Nederland (SION) van de Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO).

Index

R