THE HIGH PERFORMANCE GENERIC GRAPH COMPONENT LIBRARY

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

by

Lie-Quan Lee, B.S., M.S.

Andrew Lumsdaine, Director

Department of Computer Science and Engineering
Notre Dame, Indiana
September 1999

THE HIGH PERFORMANCE GENERIC GRAPH COMPONENT LIBRARY

Abstract

by

Lie-Quan Lee

In this thesis I present the Generic Graph Component Library (GGCL), a generic programming framework for graph data structures and graph algorithms. Following the theme of the Standard Template Library (STL), the graph algorithms in GGCL do not depend on the particular data structures upon which they operate, meaning a single algorithm can operate on arbitrary concrete representations of graphs. I describe the principal abstractions comprising the GGCL, the algorithms and data structures that it provides, and provide examples that demonstrate the use of GGCL to implement some common graph algorithms. Performance results are presented which demonstrate that the use of novel lightweight implementation techniques and static polymorphism in GGCL results in code which is significantly more efficient than similar libraries written using the object-oriented paradigm.

To my father,

my mother in memoriam,

and my wife Yun

CONTENTS

TABLES

FIGURES

ACKNOWLEDGEMENTS

I want to express my special appreication to my advisor, Dr. Andrew Lumsdaine. This work could not be done without his earnest guidance, invaluable suggestions and full support. I would like to take this opportunity to express my thanks to Jeremy G. Siek and his splendid ideas to GGCL. I would like to thank the all other members in the Laboratory of Scientific Computing, Jeff M. Squyres, Michael D. Mcnally, and Kinis Meyer.

I would like to express my appreciation to Dustin and Jennifer Lee, Jerry and Beverly Wei, Daniel and Shirley Meng, Rev. John and Grace Chao, Yamin Huang, and Bo Hu. Thank you for your spirtual guidance, encouragement in my low time, and assistance in my life.

CHAPTER 1

INTRODUCTION

The graph abstraction is widely used to model a large variety of structures and relationships in many areas such as transportation, scheduling, networks, robotics, VLSI design, compilers, database and software engineering. For example, a weighted graph can model airline flight schedules, with the airports as vertices and direct flights between two airports as edges whose weight is the distance between them. In the register allocation phase of a compiler, by constructing an undirected interference graph whose vertices represent temporary values and whose edges indicate pairs of temporaries that can not be assigned to the same register, register allocation, a very basic phase of the compiler, can be deduced as the classic graph coloring problem. Graph theory has been ubiquitous in sparse matrix computation ever since Seymour Parter used undirected graphs to model symmetric Gaussian elimination more than 30 years ago. Graph models of symmetric matrices and factorizations and algorithms on non-symmetric matrices, such as fill paths in Gaussian elimination, strongly connected components in irreducibility, bipartite matching, and alternating paths in linear dependence and structural singularity, not only make it easier to understand and analyze sparse matrix algorithms, but broaden the area of manipulating sparse matrices using existing graph algorithms and techniques [8]. Graph algorithms can be applied directly to various problem domains if the problems are properly modeled. Consequently, the implementation of graph algorithms is an important enterprise that can be greatly facilitated by the availability of high-quality software for realizing graph algo-

rithms. (By "high-quality" in this case we take to mean, such attributes as functionality, reliability, usability, efficiency, maintainability, and portability [18].)

There are several existing general purpose graph libraries, such as LEDA [17], the Graph Template Library (GTL) [6], Combinatorica [26], and Stanford GraphBase [14]. Sources such as Netlib [1] and [27] represent repositories of graph algorithms. These libraries and repositories represent a significant amount of potentially reusable algorithms and data structures. However, none of these libraries faithfully follows the *generic programming* paradigm [4] (also see Section 1.1) and are therefore far more rigid (and much less reusable) than necessary.

These libraries are inflexible in several respects. First, the user is restricted to the graph data structures provided by the library. Second, the graph algorithms often do not provide explicit mechanisms for extension, making it difficult or impossible for users to customize vanilla algorithms to meet their needs. Finally, the manner in which these libraries associate graph properties (such as color or weight) with a graph data structure is often inflexible and hard coded into the algorithms or data structures. Ultimately, these (and other) libraries are fundamentally limited in terms of their flexibility by their design and implementation.

1.1 Generic Programming

Recently, generic programming [4] has emerged as a powerful new paradigm for library development. The fundamental principle of generic programming is to separate algorithms from the concrete data structures on which they operate based on the underlying abstract problem domain concepts, allowing the algorithms and data structures to freely interoperate. That is, in a generic library, algorithms do not manipulate concrete data structures directly, but instead operate on abstract interfaces defined for entire equivalence classes of data structures. A single generic algorithm can thus be applied to any

particular data structure that conforms to the requirements of its equivalence class. In the celebrated Standard Template Library (STL) [15], the data structures are containers such as `vector`, `list`, `set` and `map`. Each of these container classes is a template, and can be instantiated to contain any type of object. Most importantly, each container has its *iterator* interface. Each container class defines an `iterator` type and member function `begin` and `end` which represent the first element of the container and the one-beyond-the-last element, respectively. `Iterator` is a generalization of pointer because the dereference of an iterator object gets the element value as a pointer does. *Iterators* form the abstract interface between algorithms and containers so that algorithms are able to be decoupled from containers. Each STL algorithm is written in terms of the iterator interface and as a result each algorithm can operate with any of the STL containers. The following is an example algorithm in STL which performs the operation for each iterator in the provided range.

```
template <class InputIterator, class OutputIterator,
          class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation op)
{
  for (; first != last; ++first, ++result)
    *result = op(*first);
  return result;
}
```

As shown above, algorithms are parameterized by the type of `iterator` so that they are not restricted to a single type of container. In addition, many of the STL algorithms are parameterized not only on the type of iterator used for traversal, but on the type of operation that is applied during the traversal. For example, the `transform()` algorithm shown above has a parameter for a `UnaryOperator` function object (functor). Function objects as a generalization of functions, allow abstraction not only over the types

of objects, but also over the operations that are being performed. Likewise, some of the STL containers are parameterized with function objects, such as the `Compare` template parameter for the `std::map` and `std::set` classes.

1.1.1 Concepts

The Generic Graph Component Library is expressed using terminology similar to that of the SGI STL [4]. In the parlance of the SGI STL, the set of requirements on a template parameter for a generic algorithm or data structure is called a *concept*. (Generic programming is sometimes referred to as "programming with concepts.") For example, the type of `first` and `last` in the above `transform()` example is required to compare two objects of that type for equality, to be possible to increment an object of that type, and to be possible to dereference an object of that type to obtain the object that it points to. The requirement set for the type of `first` and `last` is called InputIterator in the STL. Types that fulfill the requirements of a concept are said to *model* that concept. For example, pointer types such as `int*` meets the requirements of a InputIterator and can be used in `transform()`. The class types `std::vector<T>` and `std::list<T>` are models of the Container concept. Concepts can extend other concepts, which is referred to as *refinement*. We use a bold sans serif font for all concept identifiers.

For proper operation of `transform()`, we require that the type of the arguments `first` and `last` be models of the concept InputIterator. We note that the C++ language does not provide support for concept checking. That is, although we give the template parameter to `transform()` the name of InputIterator, the name is merely a placeholder. The C++ language does not enforce that the arguments passed to `transform()` actually *be* a model of InputIterator. Naturally, if the arguments do not model (or refine) InputIterator, it is likely that an error will occur when compiling that particular instan-

tiation of `transform()`, but that is not the same (semantically) as identifying that the instantiation itself is in error.

1.2 Generic Programming Process

As described by Stepanov, the generic programming process applied to a particular problem domain consists of the following basic steps:

1. Identify useful and efficient algorithms and other components.
2. Find their generic representation (i.e., parameterize each algorithm such that it makes the fewest possible requirements of the data on which it operates)
3. Derive a set of (minimal) requirements that allow these algorithms to run and to run efficiently
4. Construct a framework based on classifications of requirements

1.3 A Generic Graph Library

The domain of graphs and graph algorithms is a natural one for the application of generic programming. There are many kinds of graph representations, such as adjacency matrix, adjacency list, and dynamic pointer-based graphs and there also numerous graph algorithms such as Depth First Search (DFS), Breadth First Search (BFS), topological sort, connected components, Dijkstra's algorithm for single-soure shortest paths, Prim's algorithm and Kruskal's algorithm for minimum spanning trees, and Find/Union operation. In a generic graph library, we should be able to write each algorithm only once and use it with any graph data structure.

In addition, the algorithms should be flexible, so that algorithm *patterns* such as Depth First Search can be reused. For example, one may want to use DFS to traverse a graph and calculate whether vertices are reachable. In another situation, DFS could be used to

record the order of vertices. In yet another situation, one may want to use DFS to calculate reachability *and* the order of vertices. These requirements are similar to those of most general purpose libraries, which would perhaps suggest that the generic programming style of the STL might be directly applicable to the creation of a graph library.

However, there are important (and fundamental) differences between the types of algorithms and data structures in STL and the types of algorithms and data structures in a generic graph library. In particular, there are numerous ways in which edge and vertex properties (such as color and weight) are implemented and associated with vertices and edges. One way is to store properties in an array indexed by vertex ID. Another method, suitable for graphs with explicit storage for each vertex, is to store the properties inside the vertex data structure. Rather than imposing one approach over another, a generic graph library should provide an generic means for accessing the properties of a vertex or edge, regardless of the manner in which the properties are stored.

To accommodate the unique properties of graphs and graph algorithms, we introduce several concepts upon which the interface between graphs and graph algorithms will be built: Vertex, Edge, Visitor, and Decorator. The latter two concepts are similar in spirit to the "Gang of Four" [7] patterns Visitor and Decorator but are quite different in terms of implementation techniques.

In the following chapters we describe the design and implementation of the Generic Graph Component Library (GGCL) by applying the generic programming process to the graph domain. This library was designed and implemented from the ground up with generic programming as its fundamental paradigm. In the next chapter, we define the abstract graph interface and concepts used by GGCL in more detail. The generic graph algorithms in GGCL are described in Chapter 3, and Chapter 4 discusses the main implementation issues. Sparse matrix ordering algorithms as the first application of GGCL are discussed in the Chapter 5. Experimental results demonstrating the performance of GGCL

(and comparing the performance to several other graph libraries) are given in Chapter 6. After that, our conclusions are provided in Chapter 7. Finally, the GGCL Programmer's Guide is included in the Appendices.

CHAPTER 2

ABSTRACT GRAPH INTERFACE

As the first step of applying the process of generic programming to the graph domain, we identify that we need implement basic graph algorithms described in [3] which are Depth First Search (DFS), Breadth First Search (BFS), topological sort, connected components, Dijkstra's algorithm for single-source shortest paths, Prim's algorithm and Kruskal's algorithm for minimum spanning trees, Find/Union disjoint set operations. Let us look at classical BFS and consider how we can make it generic. Figure 2.1 shows the algorithm described in [5] by Cormen et al. The algorithm computes the distance of every vertex from starting vertex `s` and the predecessor of every vertex in the resulting BFS tree. During the graph traversing, the color of every vertex `u` $\in V$ is `color[u]` and a normal First-In First-Out queue object `Q` is used.

To be generic, we would like BFS can traverse any concrete graph data structures firstly. That can be achieved by parameterizing the type of input graph object. The type of vertex `s` is not necessarily parameterized. However, it should be able to know from the type of graph `G` somehow. We decide that we use a traits [19] class to get the type of vertex object. Secondly, the algorithm need to access the properties of a vertex such as color, distance, and predecessor. Those properties could be stored either in external arrays or inside the vertex objects. A generic algorithm requires a generic access mechanism. In fact, the textual description in Figure 2.1 has indicated a suitable solution. An STL functor-like mechanism, called Decorator, can be used here. The type of `color`

```
BFS(G, s)

//initialization
 for each vertex v ∈ V(G)
   do color[u] ← WHITE
      d[u] ← ∞
      π[u] ← NIL

//starting point
color[s] ← GRAY
d[s] ← 0
Q ← s

//main algorithm
 while (Q ≠ 0)
   //discover vertex u
   do u ← head[Q]
      for each v ∈ Adj[u]
        //process the edge (u→v)
        do if (color[v] == WHITE)
           then color[v] ← GRAY
                d[v] ← d[v] + 1
                π[v] ← u
                ENQUEUE(Q, v)
      DEQUEUE(Q)
      //finishing point for vertex u
      color[u] ← BLACK
```

Figure 2.1. The Breadth First Search algorithm description in the textbook. `d[u]` is the distance of vertex `u` from starting vertex `s`. π`[u]` is the predecessor of vertex `u`.

models Decorator, therefore, accessing color property of vertex `u` can be expressed as `color[u]`. So for `d` and π. We will identify the minimum set of requirements for a Decorator later. Thirdly, let us review the algorithm by focusing on the functionality to implement. The algorithm computes the distance and predecessor of every vertex exactly. It can not be more or less without re-implementing it. If we only want to compute predecessor information, we might use the algorithm but with unnecessary overhead of setting up the data structure for distance and computing it. On the other hand, if we want to compute the predecessor of every vertex and assign a level for all the vertices in BFS tree, we have to modify the algorithm. The modification is to substitute distance computation part with level assignment. However, the structure of the modified algorithm is the same as that of the algorithm in Figure 2.1. As we know, functors or function objects are used to abstract operations in STL so that STL algorithms can delay binding the concrete operations until instantiating time. A similar approach can be used in this situation except that we need several separate operations instead of one operation `operator()` only. We call it `Visitor`, in which `initialize()`, `start()`, `discover()`, `process()`, and `finish()` are defined. For example, we abstract the operation of setting the vertex color to be BACK at the finishing point in Figure 2.1 as operation `finish()` in a Visitor. We will give a formal definition of Visitor later. Finally, we parameterize the type of queue used in the algorithm to make additional reuse possible. Thus, a generic BFS could be prototyped as follows:

```
template <class Graph, class QType, class Visitor>
void BFS(Graph& G, graph_traits<Graph>::vertex_type s,
         Qtype& Q, Visitor visitor);
```

The complete implementation of the generic BFS algorithm in GGCL and the extensive reuse of the BFS pattern can be found in Chapter 3.

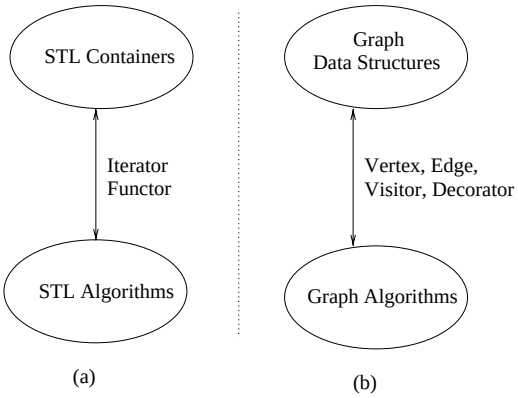


Figure 2.2. The analogy between the STL and the GGCL.

The domain of graph data structures and algorithms is in some respects more complicated than that of containers. The abstract iterator interface used by STL is not sufficiently rich to encompass the numerous ways that graph algorithms may traverse a graph. Instead, we formulate an abstract interface that serves the same purpose for graphs that iterators do for basic containers (though iterators still play a large role). Figure 2.2 depicts the analogy between the STL and the GGCL.

2.1 Formal Graph Definition

The appropriate abstract graph interface can be derived directly from the formal definition of a graph [5]. A graph *G* is a pair *(V,E)*, where *V* is a finite set and *E* is a binary relation on *V*. *V* is called a *vertex set* whose elements are called *vertices*. *E* is called an *edge set* whose elements are called *edges*. An edge is an ordered or unordered pair *(u,v)* where *u,v* $\in$ *V*. If *(u,v)* is and edge in graph *G*, then vertex *v* is *adjacent* to vertex *u*. Edge *(u,v)* is an *out-edge* of vertex *u* and an *in-edge* of vertex *v*. In a *directed* graph edges are ordered pairs while in a *undirected* graph edges are unordered pairs. In a *directed* graph an edge *(u,v)* leaves from the *source* vertex *u* to the *target* vertex *v*.

2.2 GGCL Concepts

The three main concepts necessary to define our graph interface are Graph, Vertex, and Edge. Each of our concept definitions derives directly from the formal graph definition. By design we have tried to keep the interface close to that of existing graph libraries and to the common graph algorithm notations.

2.2.1 Graph

The Graph concept merely contains a set of vertices and a set of edges and a tag to specify whether it is a directed graph or an undirected graph. Table 2.1 lists the Graph requirements, including its associated types. Note that the specific types of the sets are not specified. The only requirement is that *vertex set* be a model of ContainerRef and its `value_type` a model of Vertex. The *edge set* must be a model of ContainerRef and its `value_type` a model of Edge. The ContainerRef concept is very similar to the Container concept of the STL, except that the ContainerRef concept lacks the notion of "ownership", so making a copy of a ContainerRef object merely creates an alias to the same underlying container. Obviously, a reference to a Container object satisfies this requirements. Notice that all types are not necessary inside models of graph but deduced from traits class `graph_traits`. The function requirements are not the member functions of models but global functions.

2.2.2 Vertex

The Vertex concept provides access to the adjacent vertices, the out-edges of the vertex and optionally the in-edges. Table 2.2 lists the Vertex requirements, including its associated types. Similar to Graph concept, the Vertex concept requires that all the types are deduced from traits class `vertex_traits` and functions are global.

Table 2.1. The specification of the Graph concept. *X* is a model of Graph while *G* is an instance of *X*.

Expression	**Return Type**	**Description**
graph_traits< *X* >::vertex_type		A model of Vertex
graph_traits< *X* >::edge_type		A model of Edge
graph_traits< *X* >::vertices_type		A ContainerRef of vertices
graph_traits< *X* >::edges_type		A ContainerRef of edges
graph_traits< *X* >::direct_tag		directed or undirected tag
vertices(G)	vertices_type	The *vertex set* of graph G
edges(G)	edges_type	The *edge set* of graph G

Table 2.2. The specification of the Vertex concept. *X* is a model of Vertex while *u* is an instance of *X*.

Expression	**Return Type**	**Description**
vertex_traits< *X* >::edge_type		A model of Edge
vertex_traits< *X* >::vertexlist_type		The type for adj, ContainerRef
vertex_traits< *X* >::edgelist_type		The type for out_edge, ContainerRef
adj(u)	vertexlist_type	The adjacent vertices of u
out_edges(u)	edgelist_type	The out edges of vertex u
in_edges(u)	edgelist_type	The in edges of vertex u

Table 2.3. The specification of the Edge concept. X is a model of Edge. e is an instance of X.

Expression	Return Type	Description
`edge_traits`$< X >$`::vertex_type`		A model of Vertex
`source(e)`	`vertex_type`	The *source* vertex of edge `e`
`target(e)`	`vertex_type`	The *target* vertex of edge `e`

2.2.3 Edge

An Edge is an ordered or unordered pair of vertices. The elements comprising the Edge are the *source* vertex and the *target* vertex. In the unordered case it is just assumed that the position of the *source* and *target* vertices are interchangeable (and, correspondingly, that the Graph is undirected). Table 2.3 lists the Edge requirements. Similar to Graph concept, the Edge concept requires that the type is deduced from traits class `edge_traits` and functions are global.

The rest of the chapter gives the formal definitions of two concepts, Decorator and Visitor identified at the beginning of this chapter. They play an important role in the GGCL algorithms.

2.2.4 Decorator

As we mentioned, we would like to have a generic mechanism for accessing vertex and edge properties of a graph (e.g., color or weight) from within an algorithm. The generic access method is necessary to support the numerous ways in which the properties can be stored as well as the numerous ways in which access to that storage can be implemented. We give the name Decorator to this concept since it is similar to the intent of the "Gang of Four" Decorator pattern [7], which attaches additional responsibilities to an object dynamically.

Table 2.4 gives the definition of the Decorator concept. A Decorator looks like a functor, or function object. We use the method of `operator[]` instead of `operator()`

Table 2.4. The specification of the Decorator concept. X is a model of Decorator. d is an instance of X.

Expression	Return Type	Description
`decorator_traits`$< X >$`::value_type`		A type of object decorated
`d[u]`	`value_type`	The decorating property

since it is a better match for the commonly used graph algorithm notations. Similar to the Graph concept, the Decorator concept requires that the `value_type` be deduced from the `decorator_traits` class. Notice that there exists a fundamental difference between Decorator and RandomAccessIterator in the STL. The latter defines the method of `operator[]` with difference_type as the parameter type. However, the parameter type of the method of `operator[]` for Decorator is the type of object decorated, i.e., a model of Vertex or Edge.

2.2.5 Visitor

As we mentioned before, function objects or functors abstract the basic operations within algorithms and they can be used to generalize certain algorithms. In the same way that function objects are used to make STL algorithms more flexible, we use functor-like objects to make the graph algorithms more flexible. We use the name Visitor for this concept because the intent is similar to the well known visitor pattern [7]. We want to add operations to be performed on the graph without changing the source code for the graphs or for the generic algorithms.

Table 2.5 shows the definition of the Visitor concept. In the table, v is a visitor object, u and s are vertices, and e is an edge. As shown in the table, our Visitor is somewhat more complex than a function object, since there are several well defined entry points at which the user may want to introduce a call-back. For example, `discover()` is invoked when an undiscovered vertex is encountered within the algorithm. The `process()` method is

Table 2.5. The specification of the Visitor concept. Here *v*, *u*, or *e* is an instance of a model of Visitor, Vertex, or Edge, respectively.

Expression	Return Type	Description
`v.initialize(u)`	`void`	Invoked during initialization.
`v.start(u)`	`void`	Invoked at the beginning of algorithms.
`v.discover(u)`	`void`	Invoked when an undiscovered vertex is encountered.
`v.finish(u)`	`void`	Invoked when algorithms finish visiting a vertex.
`v.process(e)`	`bool`	Invoked when an edge is encountered.

invoked when an edge is encountered. The Visitor concept plays an important role in the GGCL algorithms.

The Decorator and Visitor concepts are used in the GGCL graph algorithm interfaces to allow for maximum flexibility. Below is the prototype for the GGCL depth first search algorithm, which includes parameters for both a Decorator and a Visitor object. There are two overloaded versions of the interface, the first one in which there is a default ColorDecorator. The default decorator accesses the color property directly from the graph vertices. This is analogous to the STL algorithms. For example, there are two overloaded versions of the `lower_bound()` algorithm. The default uses less-than operator defined for the element type, while the other version takes an explicit BinaryOperator functor argument for comparison operation.

```
template <class Graph, class Visitor>
void dfs(Graph& G, Visitor visit);

template <class Graph, class Visitor, class ColorDecorator>
void dfs(Graph& G, Visitor visit, ColorDecorator color);
```

CHAPTER 3

GENERIC GRAPH ALGORITHMS

The generic graph algorithms are written solely in terms of the abstract graph interface defined in the previous chapter. They do not make assumptions about the actual graph type or the underlying data structure. This enables a high degree of reuse for the algorithms.

3.1 Breadth First Search Pattern

Our first example is the classic Breadth First Search algorithm. In GGCL we capture the essence of the Breadth First Search pattern in a generalized BFS algorithm, as shown in Figure 3.1. The `visitor` parameter provides flexibility in the kinds of actions performed during the BFS. There are several call-back points associated with the visitor, including `start()`, `discover()`, `process()`, and `finish()`. The `Q` parameter allows for different kinds of queues to be used. The `visited` functor allows algorithms to perform an action on subsequent encounters with a vertex after it is discovered. The initialization steps were moved to a separate function to accommodate the need for certain type-specific initializations.

In the `generalized_BFS()` algorithm we use the expression `out_edges(u)` to access the list of edges leaving vertex `u`. Iterators of this list are used to access each of the edges. That is equivalent to traverse the list of adjacent vertices. The algorithm also inserts each discovered vertex onto `Q` or, if the vertex has already been visited, invokes the `visited` functor. Target vertices are accessed through `target(e)`.

```
template <class Vertex, class QType,
        class Visitor, class Visited>
void generalized_BFS(Vertex s, QType& Q,
                 Visitor visitor, Visited visited)
{
  typedef typename vertex_traits<Vertex>::edge_type Edge;
  typename vertex_traits<Vertex>::edgelist_type::iterator ei;
  visitor.start(s);
  Q.push(s);
  while (! Q.empty()) {
    Vertex u = Q.front();
    Q.pop();
    visitor.discover(u);
    for (ei = out_edges(u).begin();
        ei != out_edges(u).end(); ++ei) {
      Edge e = *ei;
      if (visitor.process(e))
        Q.push(target(e));
      else
        visited(visitor, Q, ei);
    }
    visitor.finish(u);
  }
}
```

Figure 3.1. The generalized Breadth First Search algorithm.

The `generalized_BFS()` algorithm is ideal for reuse in other algorithms. Figure 3.2 gives an overview of the algorithms we have constructed so far using the `generalized_BFS`. A variation on the UML [13, 21] notation is used to represent the algorithms, visitor classes, and concepts. A solid box stands for an algorithm or a class. Dotted boxes are template arguments or concepts. The classes within a concept box are models of the concept. The notation <<`bind`>> indicates the binding of formal template arguments to concrete types. Unbound template arguments are marked with underscores, giving a notation for partial specialization.

In Figure 3.2 we can see how particular parameters are chosen in the creation the different algorithms. First, with regards to the queue type, the BFS algorithm in Figure 3.3 is constructed by using the STL `queue`, while Dijkstra's single-source shortest path and Prim's minimum spanning tree algorithms are constructed with a mutable priority queue (a priority queue with a decrease-key operation [5]). A customized queue is used with BFS in the Reverse Cuthill McKee sparse matrix ordering algorithm [10, 22].

Looking at the `Visitor` parameter, we see that the normal BFS algorithm uses the `bfs_visitor` which keeps track of the vertex colors. Dijkstra's and Prim's algorithms both use the `weighted_edge_visitor`, the only difference between them being the operator that is bound to BinaryOp parameter. Dijkstra's algorithm is implemented using a `plus` functor, and Prim's is implemented using the `project2nd` functor, which is just a binary operator that returns the 2nd argument. Figure 3.4 shows the GGCL implementation of Prim's minimum spanning tree algorithm while Figure 3.5 shows the GGCL implementation of Dijkstra's single source shortest path algorithm. The algorithms consist simply of some setup declarations, initialization and a call to `generalized_BFS`. The only difference between the two algorithms is the function object used inside `weighted_edge_visitor` whose implementation is shown in Figure 3.6.

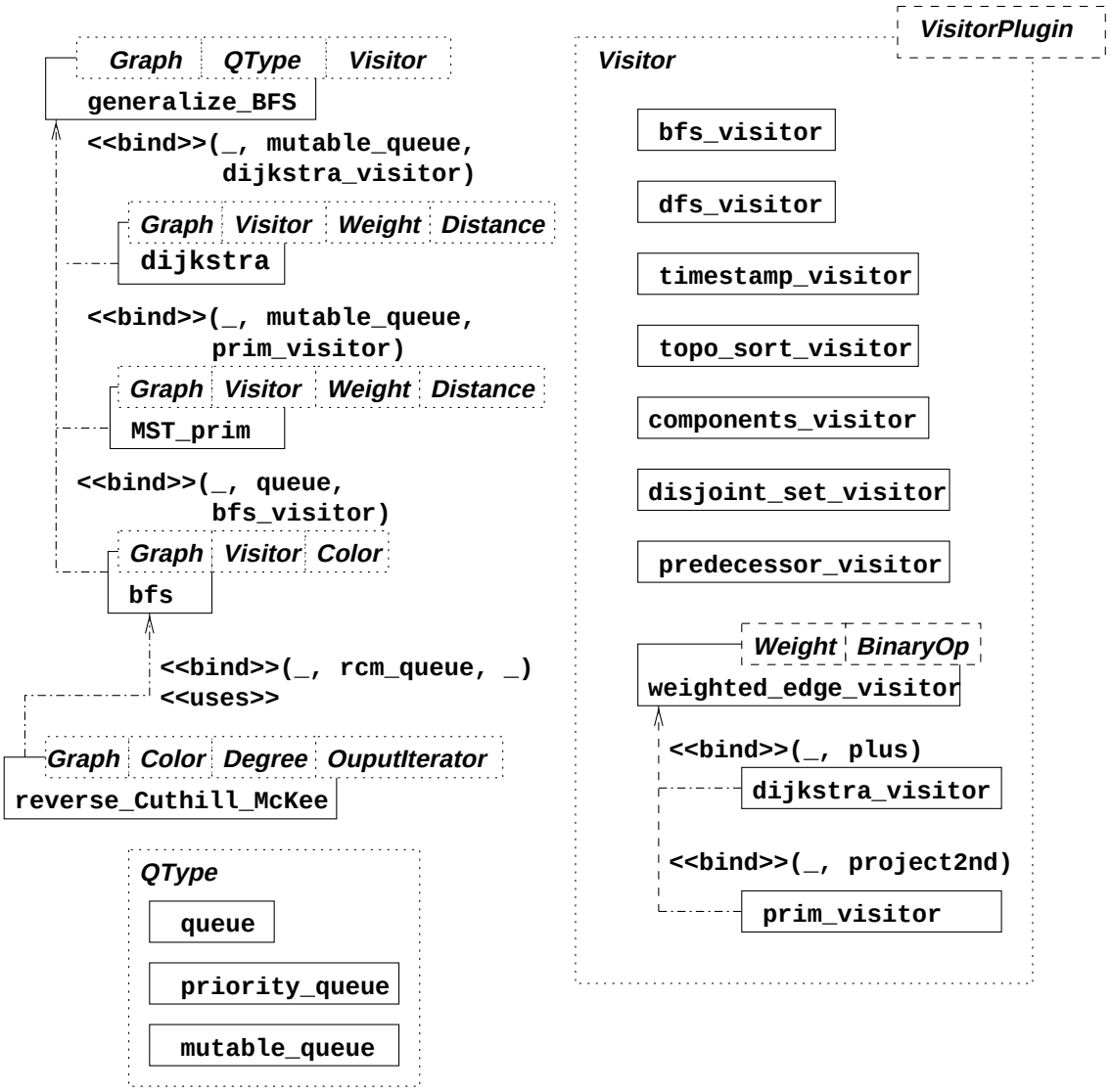


Figure 3.2. The BFS family of algorithms and the predefined set of visitors provided in GGCL.

```
template <class Graph, class Visitor, class ColorDecorator>
void bfs(Graph& G, graph_traits<Graph>::vertex_type s,
       Visitor visit, ColorDecorator color)
{
  typedef typename graph_traits<Graph>::vertex_type Vertex;
  std::queue<Vertex> Q;

  bfs_visitor<ColorDecorator, Visitor> visitor(color, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, null_operation());
}
```

Figure 3.3. The BFS algorithm in GGCL.

The `Visited` parameter is simply a null operation for the normal BFS algorithm, while in the Dijkstra's and Prim's algorithms it provides queue update by invoking the mutable priority queue' decrease-key operation.

3.2 Depth First Search Pattern

The Depth First Search is another fundamental traversal pattern in graph algorithms, and is a second source for reuse. Figure 3.7 depicts some algorithms that can be either directly derived from DFS, or that make use of it. The code example in Figure 3.8 gives the implementation of the topological sort algorithm, a classic example DFS algorithm reuse. The `topo_sort_visitor` merely outputs the vertex to the OutputIterator inside the `finish(u)` call-back.

The concise implementation of algorithms such as Prim's Minimum Spanning Tree and Topological Sort is enabled by the genericity of the GGCL algorithms, allowing us to exploit the reuse that is inherent in these graph algorithms in a concrete fashion.

Currently, the GGCL includes a basic set of algorithms: DFS, BFS, Dijksta's algorithm for the Shortest Path problem, Prim and Kruskal algorithms for Minimum Spanning

```
template <class Graph, class Visitor,
        class Distance, class Weight, class ID>
void prim(Graph& G, graph_traits<Graph>::vertex_type s,
        Visitor visit, Distance d, Weight w, ID id )
{
  typedef typename graph_traits<Graph>::vertex_type Vertex;
  typedef typename decorator_traits<Distance>::value_type D;
  typedef functor_less<Distance> Compare;
  typedef _project2nd<D,D> Project;

  Compare c(d);
  mutable_queue<Vertex, std::vector<Vertex>, Compare, ID>
    Q(G.num_vertices(), c, id);

  weighted_edge_visitor<Weight, Distance, Visitor, Project>
    visitor(w, d, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, queue_update());
}
```

Figure 3.4. The GGCL implementation of the Prim's Minimum Spanning Tree algorithm as a call to `generalized_BFS()`. The Dijkstra's Single-Source Shortest Path algorithm can be realized in the same way simply by using a different function object in place of `_project2nd<D,D>`.

```
template <class Graph, class Visitor,
        class Distance, class Weight, class ID>
void dijkstra(Graph& G, graph_traits<Graph>::vertex_type s,
           Visitor visit, Distance d, Weight w, ID id )
{
  typedef typename graph_traits<Graph>::vertex_type Vertex;
  typedef typename decorator_traits<Distance>::value_type D;
  typedef functor_less<Distance> Compare;

  Compare c(d);
  mutable_queue<Vertex, std::vector<Vertex>, Compare, ID >
    Q(G.num_vertices(), c, id);

  weighted_edge_visitor<Weight, Distance, Visitor, plus<D> >
    visitor(w, d, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, queue_update());
}
```

Figure 3.5. The GGCL implementation of the Dijkstra's Single-Source Shortest Path algorithm as a call to `generalized_BFS()`.

```
template <class Weight, class Distance,
        class Super, class BinaryOperator>
class weighted_edge_visitor : public Super {
   typedef typename decorator_traits<Distance>::value_type D;
public:
  //constructors

  template <class Edge>
  bool process(Edge e) {
    typedef typename decorator_traits<Weight>::value_type T;
    D du = d[source(e)];
    D dv = d[target(e)];
    bool ret = ( dv == numeric_limits<D>::max() );
    T wuv = w[e];
    if ( dv > op(du, wuv) ) {
      dv = op(du, wuv);
      d[target(e)] = dv;
      need_queue_update = !ret;
      Super::process(e);
    }
    return ret;
  }

  //other members

protected:
  Weight   w;
  Distance d;
  BinaryOperator op;
};
```

Figure 3.6. The implementation of `weight_edge_visitor` used in Dijkstra's algorithm and Prim's algorithm.

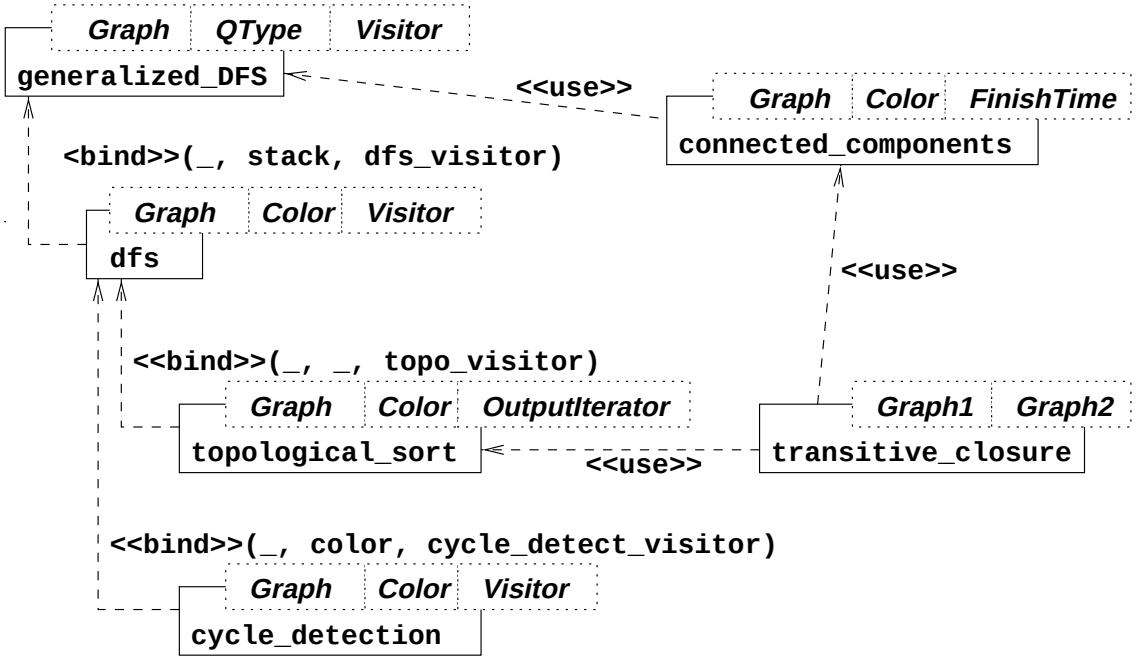


Figure 3.7. The family of DFS algorithms.

Tree, topological sort, and connected components. In addition we have implemented several graph algorithms for sparse matrix ordering, including the Reverse Cuthill McKee and the Minimum Degree algorithms. GGCL is an ongoing project and a number of generic graph algorithms are in the process of being implemented.

```
template <class Graph, class OutputIterator,
          class Visitor, class Color>
void topological_sort( Graph& G, OutputIterator result,
                       Visitor visitor, Color color) {
  topo_sort_visitor<OutputIterator, Visitor>
       topo_visit(c, visitor);
  dfs(G, topo_visit, color);
}

template <class OutputIterator, class Super>
struct topo_sort_visitor : public Super {
  //constructors . . .

  template <class Vertex>
  void finish(Vertex u) {
    *result = u; ++result;
    Super::finish(u);
  }
  OutputIterator result;
};
```

Figure 3.8. The GGCL implementation of the topological sort algorithm using DFS.

CHAPTER 4

GGCL IMPLEMENTATION

4.1 Graph Data Structure Implementation

The GGCL graph data structures are constructed in a layered manner to provide maximum flexibility and reuse. The layered architecture also provides several different points of customizability. At one end of the spectrum one can use the graphs provided by GGCL and make small modification with little effort. In the middle of the spectrum are graph types that can be pieced together from standard components such as lists and vectors. At the far end of the spectrum the user may already have their own data structure, and they just need to create a GGCL Graph compliant interface to his or her data structure.

4.1.1 Interfacing With External Graph Types

To demonstrate the ease of creating a GGCL interface for non-GGCL graph types, we constructed a Graph interface for LEDA graphs. The interface code is about 1 1/2 pages and it took approximately 1 man-hour to develop. Another testing case is to create an interface for a pointer-based graph data structure written in C-style. The code excerpt in Figure 4.1 is a typical pointer-based graph data structure in language of C. A `node` in this graph has a list of adjacent nodes and several properties. Function `make_node` serves to create a new `node`. The function `add_adj` is used to make a direct edge between two `nodes`.

```
/* Below possible pointer-based graph data structures in C. */
struct adj_list;

struct node {
  adj_list* adj_head;
  int color;
  int flag;
  int distance;
};

struct adj_list {
  node* cur;
  adj_list* next;
};

node* make_node(int color, int flag) {
  node* x = new node;
  x->color = color;
  x->flag = flag;
  x->adj_head = 0;
  return x;
}

/* x -> a */
void add_adj(node* a, node* x) {
  adj_list* l = new adj_list;
  l->cur = a;
  l->next = x->adj_head;
  x->adj_head = l;
}

void connect(node* x, node* y) {
  add_adj(y, x);
  add_adj(x, y);
}
```

Figure 4.1. An example of pointer-based graph data structures in C.

Figure 4.2 is the brief implementation of a class for Vertex. Constructors are omitted. The implementation lacks `vertexlist_type` class which will be similar to `edge-list_type`. Therefore, it is easy to create `vertexlist_type` class.

I also provide the a class confirming Edge concept in Figure 4.3. The template technique is not necessarily used here. However, it can deal with the problem of include dependency.

Finally, a simplified version of graph class is shown in Figure 4.4. Several required types are defined inside the class.

4.1.2 Composing Graphs From Standard Containers

The GGCL provides a framework for composing graphs out of standard containers such as STL `std::vector`, `std::list`, and matrices from the Matrix Template Library (MTL) [24], another generic component library we have developed. Of course, the composition mechanism will work for any STL Container compliant components, so this provides another avenue for extensibility by the user.

The set of graph configurations currently provided by GGCL are listed in Figure 4.5. Again, a solid box stands for a class. Dotted boxes are template arguments or concepts. The classes within a concept box are models of the concept.

Below is an example of defining an adjacency-list graph type whose vertices have an associated color and whose edges have an associated weight.

```
typedef graph<adjacency_list<vecT>, undirected,
              color_plugin<>, weight_plugin<int> > myGraph;
```

4.1.3 Graph Representation

The implementation framework centers around the main `graph` interface class and the GraphRepresentation concept. The `graph` interface class constructs the full graph interface based on the minimized interface exported by the GraphRepresentation con-

```
template < class Node >
struct pointwise_vertex {
  typedef pointwise_vertex<Node> self;
  typedef Node plugin_type;
  typedef pointwise_edge<self> edge_type;

  struct edgelist_type {
    struct iterator {
      iterator(Node* _s, adj_list* _d) : s(_s), adj(_d) {}
      iterator& operator++() { adj = adj->next; return *this; }
      bool operator != (iterator x) const
      { return s != x.s || adj != x.adj; }
      bool operator == (iterator x) const
      { return s == x.s && adj == x.adj; }
      edge_type operator*() { return edge_type(s, adj->cur); }
      Node* s;
      adj_list* adj;
    };
    iterator begin() { return iterator(_node, _node->adj_head); }
    iterator end() { return iterator(_node, 0); }
    Node* _node;
  };
  Node& plugin() { return *_node; }
protected:
  Node* _node;
};

template <class Node>
vertex_traits<pointwise_vertex<Node> >::edgelist_type
out_edges(pointwise_vertex<Node> u)
{ /*...*/ }
```

Figure 4.2. The sample implementation of vertex class for the pointer-based graph data structures.

```
template < class Vertex >
struct pointwise_edge {
  typedef typename Vertex::plugin_type Node;
  typedef Vertex vertex_type;

  pointwise_edge() : s(0), d(0) {}
  pointwise_edge(Node* _s, Node* _d) : s(_s), d(_d) {}

  Node* s;
  Node* d;
};

template < class Vertex >
Vertex source(pointwise_edge<Vertex> e) {
  return Vertex(e.s);
}

template < class Vertex >
Vertex target(pointwise_edge<Vertex> e) {
  return Vertex(e.d);
}
```

Figure 4.3. The sample implementation of edge class for the pointer-based graph data structures.

```
template < class Node, class Direct >
class pointwise_graph {
public:
  typedef pointwise_vertex<Node>      vertex_type;
  typedef pointwise_edge<vertex_type> edge_type;
  typedef ggcl::dynamic<>             rep_tag;
  typedef Direct                      direct_tag;

  pointwise_graph(Node* h) : head(h) {}
  vertex_type root() { return vertex_type(head); }
protected:
  Node* head;
};
```

Figure 4.4. The sample implementation of graph class for the pointer-based graph data structures.

cept. This allows full fledge GGCL Graphs to be constructed out of standard container components with very little work.

The GraphRepresentation concept is basically a 2D Container (a Container of Containers) coupled with four helper functions:

```
Iter2D get_target(Iter2D b, Iter1D i);
stored_edge& get_edge(Iter1D i);
bool add(EdgeList& elist, size_type vertex_num,
         const stored_edge& e);
void remove(EdgeList& elist, size_type vertex_num);
```

A 1D Container within a GraphRepresentation corresponds to the out-edge list for a particular vertex. In a model of 1D Container every element has a corresponding index conceptually. The elements do not have to be sorted by their index, and the indices do not necessarily have to start at `0`. The indices do not have to form a contiguous range. In actual implementation, the indices do not necessarily have to be stored.

In addition, there is a one-to-one correspondence between the 2D Iterator and the vertices of the graph.

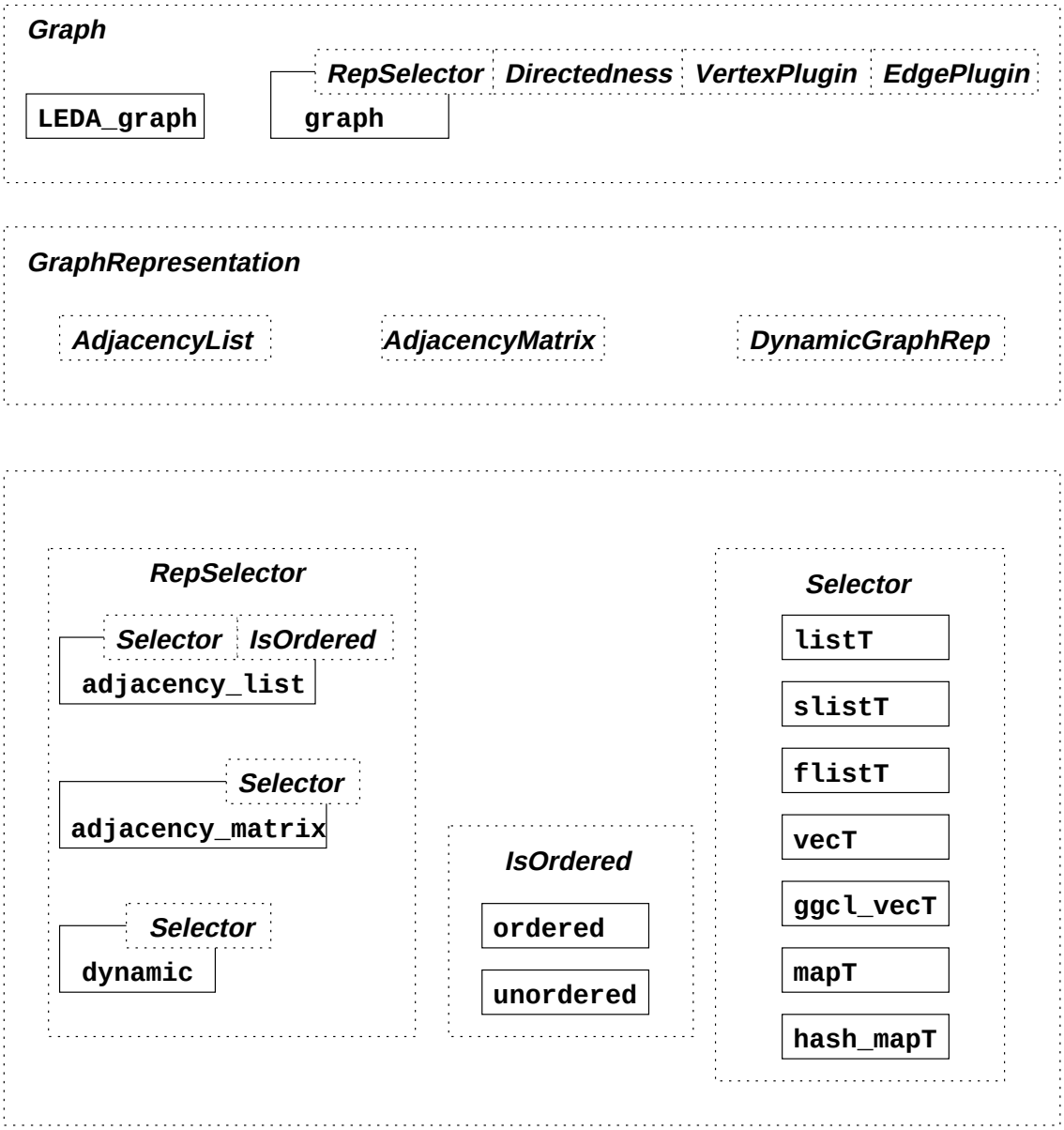


Figure 4.5. The Graph Components Provided By GGCL.

The `get_target()` helper function is necessitated because the GGCL `graph` must be able to derive the target vertex from an edge object, through the information provided by the GraphRepresentation. The `get_edge()` function provides a generic access method to the extra edge properties stored within an edge list, and the `add()` and `remove()` methods provide a generic interface for adding and removing edges from a vertex.

The GraphRepresentation is further refined into three sub concepts, the AdjacencyList, AdjacencyMatrix, and DynamicGraphRep.

The AdjacencyList concept corresponds to a "sparse" or "compressed" representation of a graph. As such, further requirements are added to the 2D Container of the GraphRepresentation. For a model of AdjacencyList the inner container must be a variable-sized Container whose `value_type` is the `size_type` for a vertex if the graph has no extra edge-associated data, or a `std::pair<size_type,stored_edge>` where the `stored_edge` is the type of an object containing any extra edge-associated data such as weight.

Technically, the edge information of an AdjacencyList graph can be stored in order by vertices or a nature order which is the order by adding edges on creating a graph. This is not an part of concept but it is convenient to allow users do both as they may want to. We implement it by providing a template argument `IsOrdered` in selector class `adjacency_list` as shown in Figure 4.5.

The AdjacencyMatrix concept corresponds to a "dense" representation of a graph, with boolean values for all vertex pairs, to mark them as connected or not. Thus, adding or removing an edge is simply by marking the corresponding boolean true or false.

The DynamicGraphRep concept requires its models to have a head pointer and explicitly stored vertex objects. Through the stored vertex it is able to access adjacent vertices.

4.1.4 Custom Graph Representations

As an example of constructing customized models of GraphRepresentation, we show how one can build an AdjacencyList using `std::vector` and `std::list`. The various parts of the GraphRepresentation are injected into the GGCL `graph` class by constructing a graph representation class. This is a class that defines the four helper functions mentioned above (as static member functions), and also defines `graphrep_type`, which is the 2D Container of the GraphRepresentation. Figure 4.6 lists the implementation. One merely has to compose a couple of container types and fill in a few short functions. The `add()` and `remove()` methods are not depicted, but they are each approximately 5 lines.

4.2 Decorator Implementation

In some situations the particular property of vertices or edges is strongly associated with the graph and exists for the lifetime of the graph. For instance, the distance property could fall into this category. In other situations the property is only needed for a particular algorithm. Typically one would want to store a color property externally, since it may only be needed for a particular algorithm invocation. Thus there are two categories of decorators, *interior decorators* and *exterior decorators*. For exterior decorators, the decorating properties are stored outside of the graph object (they are passed directly to the GGCL algorithm) and the decorator will access the externally stored data indexed by the vertex or edge ID. On the other hand, if the decorating properties are stored inside of the graph object, the decorator consults the vertex or the edge objects to obtain the decorating property. Figure 4.7 shows the predefined models Decorator in GGCL.

The interface of a decorator as defined in the previous chapter, is very concise. One is the type `value_type` for the property and the other is the member method `operator[]` to access the property. For example, The weight of an edge `e` could be accessed by a

```
//Define a tag for the custom graph representation.
struct my_graphrep_tag { };

template < class StoredEdge >
class graph_representation_gen< StoredEdge, my_graphrep_tag > {
  typedef std::list<pair<size_t, StoredEdge> > EdgeList;
  typedef EdgeList::iterator Iter1D;
  typedef std::vector<EdgeList>::iterator Iter2D;
public:
  typedef adjacency_list<my_graphrep_tag> rep_tag;
  typedef std::vector<EdgeList> graphrep_type;

  static Iter2D get_target(Iter2D b, Iter1D i)
    { return b + (*i).first; }
  static StoredEdge* get_edge(Iter1D i)
    { return &((*i).second); }
  static bool add(EdgeList& elist, size_t vertex_num,
                  const StoredEdge& e);
  static void remove(EdgeList& elist, size_t vertex_num);
};

//Use the above representation to create a graph type.
typedef graph< adjacency_list< my_graphrep_tag > > MyGraph;
```

Figure 4.6. An example of constructing a GGCL Graph.

model of WeightDecorator `w` as `w[e]` Other properties such as color and distance properties could be accessed as a similar way.

4.2.1 Internally Stored Properties: Vertex and Edge Plugins

For internal properties, the graph class provides optional parameterized storage plugins for both vertices and edges. This allows the user to plug in storage for an arbitrary set of decorating properties. For example, a graph with internally stored edge weights and color and distance properties for vertices could be defined as follow:

```
typedef color_plugin<distance_plugin<> > VPlugins;
typedef graph<adjacency_list<>, undirected,
              VPlugins, weight_plugin<int> > myGraph;
```

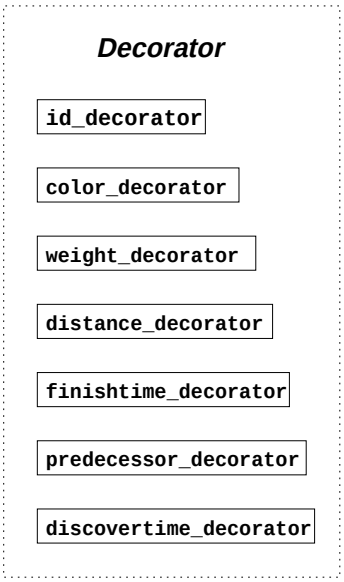


Figure 4.7. The predefined models of Decorator in GGCL.

The mixin technique [23] of parameterized inheritance is used to implement the layering of vertex and edge plugins. Normally, superclasses are defined at subclass definition time. However, mixins are opposite they are classes without specific superclass definition. With the template techniques, the implementation of mixin is very simple. For example, the definition of the above class `color_plugin` looks like:

```
template < class Super >
class color_plugin : public Super {
  //...
};
```

The advantage of static ploymorphism makes plugin classes extremely easy to extend.

Figure 4.7 shows the decorators that are provided in GGCL. We have also created a mechanism so that users can easily create new custom storage plugins for decorating properties with user-defined names.

4.2.2 RandomAccessIterator Issue

I mentioned in the Chapter 2 that the concept of RandomAccessIterator is different from that of Decorator. However, they are similar in the following matter: They both

provides `value_type`. and they both provide access method to memory but they do not own the memory. With a very simple wrapper class for a model of RandomAccessIterator, it can be a model of Decorator. Thus, I provide a traits mechanism [19] to let users be able to use models of RandomAccessIterator directly where a Decorator is required in the GGCL algorithms.

The specific mechanism is as follows. First, a type `category` is defined for Decorator through `decorator_traits` class. Namely, let `decorator_traits<Component>::category` be `decorator_tag` or `random_access_iterator_tag`, respectively, if the `Component` is a model of Decorator or RandomAccessIterator. Second, define the following class:

```
template < class Component,
     class Category = decorator_traits<Component>::category >
struct IglueD {
  typedef Component type;
};

//specialization for Decorator
template < class Component >
struct IglueD <Component, decorator_tag> {
  typedef Component type;
};

//specialization for RandomAccessIterator
template < class Component >
struct IglueD <Component, std::random_access_iterator_tag> {
  typedef random_access_iterator_decorator<Component> type;
};
```

Here the partial specialization is used to distinguish the two cases. In the first case, `Component` is a model of Decorator. The the other case, `Component` is a model of RandomAccessIteraor and a wrapper class is `typedef`ed to be `type` to promise that the `type` is a model of Decorator. Finally, `IglueD<Component>::type` should be a model of Decorator always as long as `Component` is either a model of Decorator or a model of RandomAccessIterator.

4.3 Visitor Implementation

To implement a model of Visitor one defines a class conforming to the Visitor concept and fills in the call-back methods (`discover()`, `process()`, etc.). Figure 4.8 shows the model of Visitor used to create the normal BFS algorithm from the `generalized_BFS`. This class is responsible for keeping track of the vertex colors.

As in the decorator plugins, the mixin technique [23] is used to make visitors more extensible. This is the reason for the `Base` template argument, which allows visitors to be layered through inheritance, giving an arbitrary number of visitors a chance to perform actions during the algorithm (each call-back method must invoke in inherited call-back in addition to performing its own actions). If one wished to recreate the textbook BFS algorithm shown previously, which calculates distances and predecessors, one would call `bfs` with a distance and predecessor visitor. The GGCL has helper functions defined for creating the standard visitors. (They are like `make_pair()` function which creates a `std::pair` object in the STL.)

```
bfs(G, s, visit_distance(d, visit_predecessor(p)));
```

where `G` is a graph object, `s` the starting vertex, `d` an instance of distance decorator, and `p` an instance of predecessor decorator.

```
template < class Color, class Base = null_visitor >
class bfs_visitor : public Base {
  typedef decorator_traits<Color>::value_type color_type;
public:
  // constructors . . .
  template <class Vertex>
  void initialize(Vertex u) {
    color[u] = color_traits<color_type>::white();
    Base::initialize(u);
  }

  template <class Vertex>
  void start(Vertex u) {
    color[u] = color_traits<color_type>::gray();
    Base::start(u);
  }

  template <class Vertex>
  void finish(Vertex u) {
    color[u] = color_traits<color_type>::black();
    Base::finish(u);
  }

  template <class Edge>
  bool process(Edge e) {
    if ( is_undiscovered(target(e)) ) {
      color[target(e)] = color_traits<color_type>::gray();
      Base::process(e);
      return true;
    }
    return false;
  }

  template <class Vertex>
  bool is_undiscovered(Vertex u) {
    return (color[u] == color_traits<color_type>::white());
  }
protected:
  Color color;
};
```

Figure 4.8. An example model of the Visitor concept.

CHAPTER 5

SPARSE MATRIX ORDERING ALGORITHMS

As mentioned in the introduction, graph theory is an ideal tool in sparse matrix techniques. As the first application of GGCL to sparse matrix ordering, I implemented several sparse matrix ordering algorithms. This also serves to examine how well GGCL abstract interface behaves in the "real world" applications.

5.1 Graphs and Sparse Matrices

As a graph is a way of representing a binary relation between objects , the nonzero pattern of a sparse matrix of a linear system can be modeled with a graph *G(V,E)*, whose *n* vertices in *V* represent the *n* unknowns. its edges represent the binary relations established by the equations in the following manner. There is an edge from vertex *i* to vertex *j* when A_ij is nonzero. Thus, when a matrix has a symmetric nonzero pattern, the corresponding graph is undirected.

A row permutation of sparse matrix *A* is to change the order of equations while a column permutation is to relabel (reorder) the unknowns. A symmetric permutation corresponds to applying the same permutation to both row and column. This operation is typical because the diagonal elements often are large. From the point view of graph theory, finding permutation matrix the in first step of solving a symmetric linear system mentioned above corresponds to relabeling the vertices of the graph without altering the edges.

5.2 Sparse Matrix Ordering Algorithms

The process for solving a sparse symmetric positive definite linear system, $Ax = b$, can be divided into four stages as follows:

Ordering: Find a permutation P of matrix A,

Symbolic factorization: Set up a data structure for Cholesky factor L of PAP^T,

Numerical factorization: Decompose PAP^T into LL^T,

Triangular system solution: Solve $LL^TPx = Pb$ for x.

Because the choice of permutation P will directly determine the number of fill-in elements (elements present in the non-zero structure of L that are not present in the non-zero structure of A), the ordering has a significant impact on the memory and computational requirements for the latter stages. However, finding the optimal ordering for A (in the sense of minimizing fill-in) has been proven to be NP-complete [29] requiring that heuristics be used for all but simple (or specially structured) cases.

An widely used but rather simple ordering algorithm is a variant of the Cuthill-McKee orderings. It also can be used as a preordering method to improve ordering in more sophisticated methods such as minimum degree algorithms [11].

5.2.1 Reverse Cuthill-McKee Ordering Algorithm

The original Cuthill-McKee ordering algorithm is primarily designed to reduce the profile of a matrix [10]. George discovered that the reverse ordering often turned out to be superior to the original ordering in 1971. I described RCM algorithms in the graph language:

1. *Finding a starting vertex*: Determine a starting vertex *r* and assign $x_1 \leftarrow r$.

2. *Main part*: For $i = 1, \ldots, N$, find all the unnumbered neighbors of the vertex x_i and number them in increasing order of degree.

3. *Reversing ordering*: The reverse Cuthill-McKee ordering is given by $y_1, \ldots, y_N$ where y_i is x_{N-i+1} for $i = 1, \ldots, N$.

At the first step, a good starting vertex needs to be determined. the study by George and Liu [10] showed that a pair of vertices which are at maximum or near maximum "distance" apart are good ones. They also proposed an algorithm to find such a starting vertex in [10].

My implementation of RCM is quite concise because many components from GGCL can be reused. The key part of step one is a custom queue type with BFS as shown in Figure 5.1. The main algorithm has a simple BFS-like structure although I can not reuse BFS directly because the algorithm is required a local priority (increasing order of degree of all unnumbered neighbors).

5.2.2 Minimum Degree Ordering Algorithm

Developing algorithms for high-quality orderings has been an active research topic for many years. The pattern of ordering algorithms in wide use are based on a greedy approach such that the ordering is chosen to minimize some quantity at each step of a simulated n-step symmetric Gaussian elimination process. The algorithms using such an approach are typically distinguished by their greedy minimization criteria [20].

In graph terms, the basic ordering process used by most greedy algorithms is as follows:

1. *Start:* Construct undirected graph G^0 corresponding to matrix A

2. *Iterate:* For $k = 1, 2, \ldots,$ until $G^k = \emptyset$ do:

   - Choose a vertex v^k from G^k according to some criterion

```
template <class Graph, class Vertex, class Color, class Degree>
int
pseudo_peripheral_pair(Graph& G, Vertex u, Vertex& w,
                       Color c, Degree d) {
  typedef typename IglueD<Degree>::type DegreeDecorator;
  rcm_queue<Vertex, DegreeDecorator> Q(d);
  bfs(G, u, Q, null_visitor(), c);
  w = Q.spouse();
  return Q.eccentricity();
}

template <class Graph, class Color, class Degree>
typename graph_traits<Graph>::vertex_type
find_starting_node(Graph& G, Color c, Degree d) {
  typedef typename graph_traits<Graph>::vertex_type Vertex;
  Vertex r = *(vertices(G).begin());
  Vertex x, y;

  int eccentricity_r, eccentricity_x;
  eccentricity_r = pseudo_peripheral_pair(G, r, x, c, d);
  eccentricity_x = pseudo_peripheral_pair(G, x, y, c, d);

  while (eccentricity_x > eccentricity_r) {
    r = x;
    eccentricity_r = eccentricity_x;
    x = y;
    eccentricity_x = pseudo_peripheral_pair(G, x, y, c, d);
  }

  return r;
}
```

Figure 5.1. The GGCL implementation of `find_starting_node`. The key part `pseudo_peripheral_pair` is BFS with a custom queue type virtually.

- Eliminate v^k from G^k to form G^{k+1}

The resulting ordering is the sequence of vertices $\{v^0, v^1, \ldots\}$ selected by the algorithm.

One of the most important examples of such an algorithm is the *Minimum Degree* algorithm. At each step the minimum degree algorithm chooses the vertex with minimum degree in the corresponding graph as v^k. A number of enhancements to the basic minimum degree algorithm have been developed, such as the use of a quotient graph representation, mass elimination, incomplete degree update, multiple elimination, and external degree. See [11] for a historical survey of the minimum degree algorithm.

The GGCL implementation of the Minimum Degree algorithm closely follows the algorithmic descriptions of the one in [11, 16]. The implementation presently includes the enhancements for mass elimination, incomplete degree update, multiple elimination, and external degree.

In particular, I create a graph representation to improve the performance of the algorithm. It is based on a templated "vector of vectors." The vector container used is an adaptor class built on top the STL `vector` class. Particular characteristics of this adaptor class include the following:

- Erasing elements does not shrink the associated memory. Adding new elements after erasing will not need to allocate additional memory.

- Additional memory is allocated efficiently on demand when new elements are added (doubling the capacity every time it is increased). This property comes from STL vector.

Note that this representation is similar to that used in Liu's implementation, with some important differences due to dynamic memory allocation. With the dynamic memory allocation we do not need to over-write portions of the graph that have been eliminated, allowing for a more efficient graph traversal. More importantly, information about the

elimination graph is preserved allowing for trivial symbolic factorization. Since symbolic factorization can be an expensive part of the entire solution process, improving its performance can result in significant computational savings.

The overhead of dynamic memory allocation could conceivably compromise performance in some cases. However, in practice, memory allocation overhead does not contribute significantly to run-time for our implementation as shown in the next chapter because it is not done very often and the cost gets amortized.

CHAPTER 6

PERFORMANCE

Efficiency is typically advertised as yet another advantage of generic programming — and these claims are not simply hype. The efficiency that can be gained through the use of generic programming and high-level performance optimization techniques (which themselves can be expressed in a generic fashion) is astonishing. For example, the Matrix Template Library, a generic linear algebra library written completely in C++, is able to achieve performance as good as or better than vendor-tuned math libraries [24].

For many of the efficient graph data structures in GGCL, vertex and edge objects that model the GGCL interface concepts are not explicitly stored. Rather, only partial information is stored. The GGCL interface layer constructs full vertex and edge objects on the fly from this information. These objects are extremely light-weight, and have been designed so that a modern C++ compiler will optimize the small objects away altogether. We call a light-weight object such as this a Mayfly because of its very short lifetime. We discussed the Mayfly as a design pattern for high performance computing in [25].

Additionally, the flexibility within the GGCL is derived exclusively from static polymorphism, not from dynamic polymorphism. As a result, all dispatch decisions are made at compile time, allowing the compiler to inline every function in the GGCL graph interface. Hence the “abstraction penalty” of the GGCL interface is completely eliminated. The machine instructions produced by the compiler are equivalent to what would be produced from hand-coded graph algorithms in C or Fortran.

6.1 Comparison to General Purpose Libraries

Using a concise predefined implementation of adjacency list graph representation in GGCL following the concepts we described in Section 4, we compare the performance of `bfs`, `dfs`, and `dijkstra` algorithms with those in LEDA(version 3.8), a popular object-oriented graph library [17], and those in GTL [6]. We did not perform comparison between GGCL and Combinatorica [26] we mentioned previously since it is written in Mathematica.

Our experiments compare the performance of three algorithms: `bfs`, `dfs`, and `dijkstra`. The `bfs` algorithm calculates the distance and the predecessor for every reachable vertex from a starting vertex. The `dfs` algorithm calculates the discovery time and finishing time of vertices. The `dijkstra` algorithm calculates the distance and the predecessor of every vertex from a starting vertex.

Figure A.1, Figure 6.2 and Figure 6.3 show the results for those algorithms applied to randomly generated graphs having a varying number of edges and a varying number of vertices. Because GTL does not have a Dikstra's algorithm to compare to, it is not in Figure 6.3. All results were obtained on a Sun Microsystems Ultra 30 with the UltraSPARC-II 296MHz microprocessor. For these experiments, GGCL is 5 to 7 times faster than LEDA.

6.2 Comparison to Special Purpose Library

In addition, we demonstrate the performance of a GGCL-based implementation of the multiple minimum degree algorithm [16] using selected matrices from the Harwell-Boeing collection [12] and the University of Florida's sparse matrix collection [2]. Our tests compare the execution time of our implementation against that of the equivalent SPARSPAK Fortran algorithm (GENMMD) [9]. For each case, our implementation and GENMMD produced identical orderings. Note that the performance of our implementation is essen-

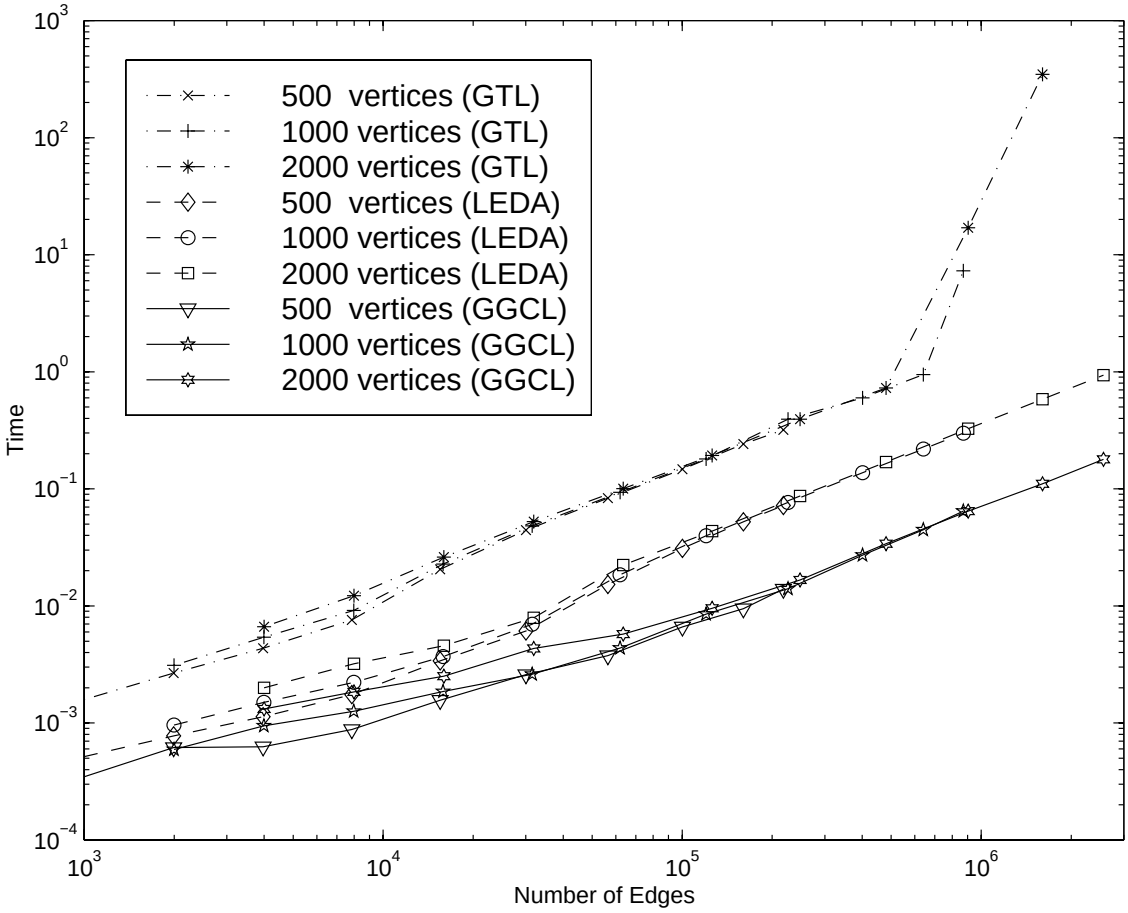


Figure 6.1. Performance comparison of the `bfs` algorithm in GGCL with that in LEDA and in GTL. Every curve represents a graph with fixed number of vertices and with varied number of edges.

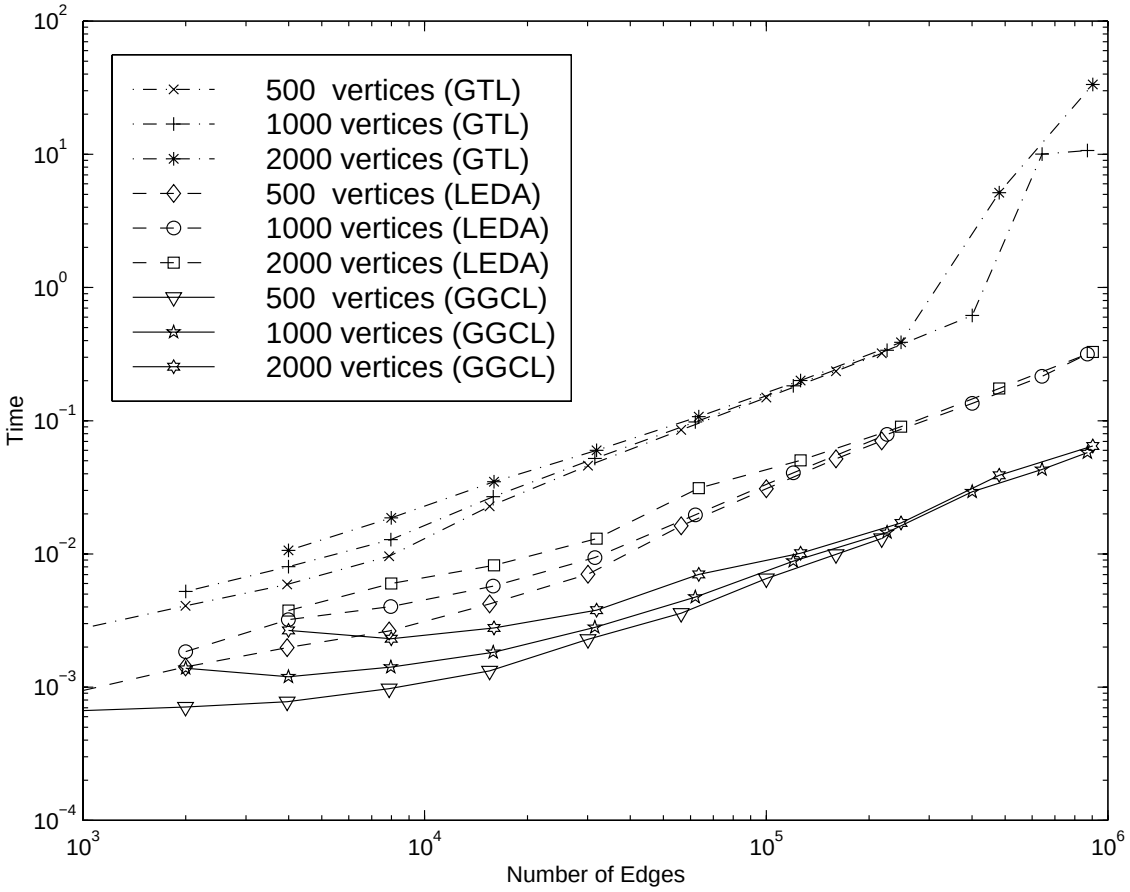


Figure 6.2. Performance comparison of the `dfs` algorithm in GGCL with that in LEDA and in GTL. Every curve represents a graph with fixed number of vertices and with varied number of edges.

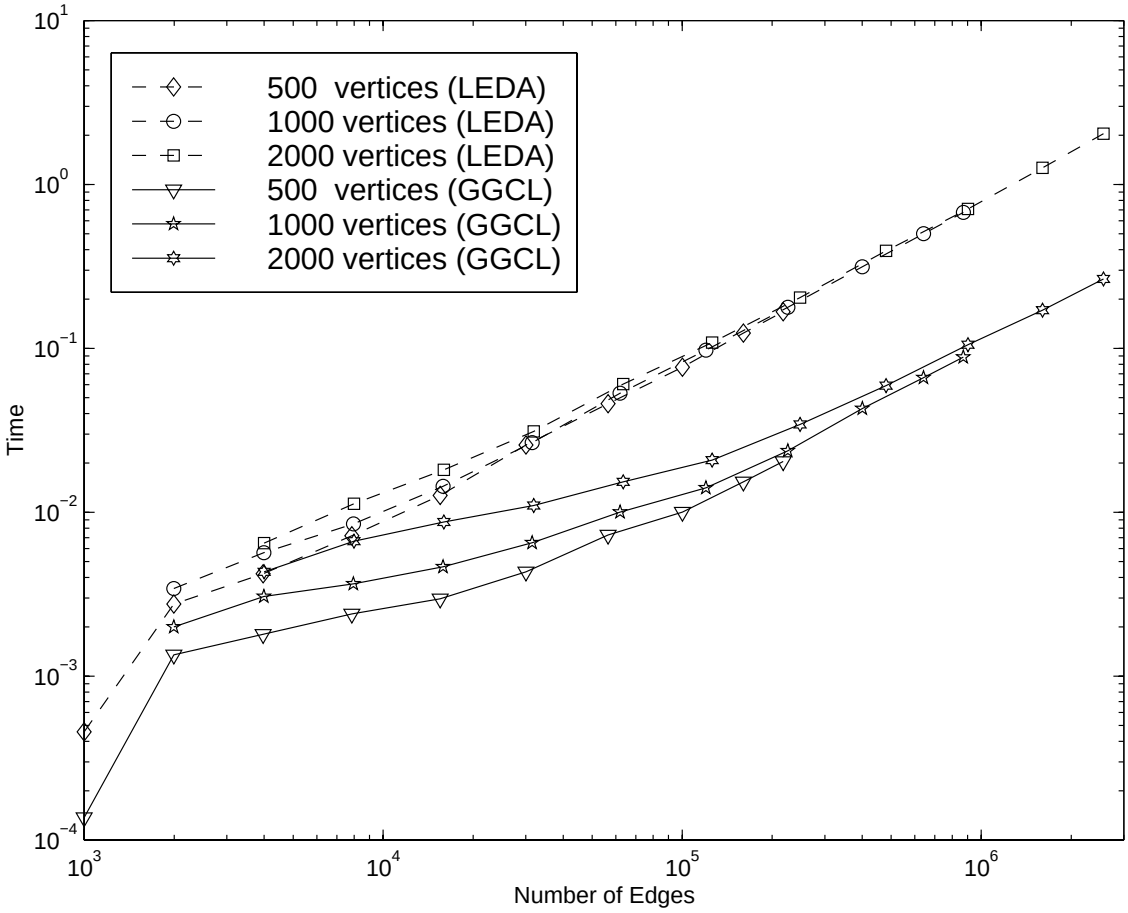


Figure 6.3. Performance comparison of the `dijkstra` algorithm in GGCL with that in LEDA. Every curve represents a a graph with fixed number of vertices and with varied number of edges.

Table 6.1. Performance comparison of minimum degree algorithms. Test matrices and ordering time in seconds, for GENMMD (Fortran) and GGCL (C++) implementations of minimum degree ordering. Also shown are the matrix order (n) and the number of off-diagonal non-zero elements (nnz).

Matrix	n	nnz	GENMMD	GGCL
BCSPWR09	1723	2394	0.00728841	0.007807
BCSPWR10	5300	8271	0.0306503	0.033222
BCSSTK15	3948	56934	0.13866	0.142741
BCSSTK18	11948	68571	0.251257	0.258589
BCSSTK21	3600	11500	0.0339959	0.039638
BCSSTK23	3134	21022	0.150273	0.146198
BCSSTK24	3562	78174	0.0305037	0.031361
BCSSTK26	1922	14207	0.0262676	0.026178
BCSSTK27	1224	27451	0.00987525	0.010078
BCSSTK28	4410	107307	0.0435296	0.044423
BCSSTK29	13992	302748	0.344164	0.352947
BCSSTK31	35588	572914	0.842505	0.884734
BCSSTK35	30237	709963	0.532725	0.580499
BCSSTK36	23052	560044	0.302156	0.333226
BCSSTK37	25503	557737	0.347472	0.369738
CRYSTK02	13965	477309	0.239564	0.250633
CRYSTK03	24696	863241	0.455818	0.480006
CRYSTM03	24696	279537	0.293619	0.366581
CT20STIF	52329	1323067	1.59866	1.59809
PWT	36519	144794	0.312136	0.383882
SHUTTLE_EDDY	10429	46585	0.0546211	0.066164
NASASRB	54870	1311227	1.34424	1.30256

tially equal to that of the Fortran implementation and even surpasses the Fortran implementation in a few cases.

6.3 Template Issues

There are several issues that often come up in libraries that make heavy use of C++ templates and advanced language features, such as code size, compile times, ease of debugging, and compiler portability. For template libraries such as GGCL, code size is very much dependent on how the library is used. If a particular code only uses a few GGCL

Table 6.2. Comparison of executable sizes for `bfs`, `dfs`, and `dijkstra` implemented in GTL, LEDA and GGCL.

Package Name	Executable Size (KBytes) `bfs`	`dfs`	`dijkstra`
GTL	151	151	/
LEDA	842	841	857
GGCL	33	30	30

algorithms and graph types, then the executable size will actually be much smaller than it would be using typical libraries. With a template library, only the functions that are actually used are included. On the other hand, with a traditional library, the whole object module will be linked in even though only one function in the module may be used. To demonstrate these effects, we compare the size of sample executables of `bfs`, `dfs`, and `dijkstra` algorithms in GTL, LEDA, and GGCL in Table 6.2. All are compiled by egcs-1.1.2 using the same compilation options. (Similar results are obtained for other compilers and architectures.) Of course, with a template library like GGCL it is very easy to instantiate redundant functionality which may unnecessarily increase the executable size, so users with large projects should be cognizant of this issue. There are techniques one can use to reduce this effect by explicity instantiating template functions in object files that can be shared.

Long compilation times are often cited as a drawback to template libraries, especially those that use expression templates [28]. Since GGCL does not use expression templates, and the overall code size of GGCL is moderate, we have not experienced severe problems in this regard. In addition, many compilers provide precompiled header mechanisms to improve compile times for template libraries.

Another concern for users of template libraries are the almost impenetrable error messages that occur when the library is misused (e.g., when a template parameter type does not model the appropriate concept). We have recently addressed this problem with some

template techniques that cause the arguments to a library call to be checked up front with regards to the type requirements. With this mechanism the resulting error messages are much more informative.

Lastly, compiler portability is currently an issue for libraries that use the more advanced features of C++. GGCL currently compiles with egcs, Metrowerks CodeWarrior, Intel C++, SGI MIPSpro, KAI C++, and other Edison Design Group based compilers. We foresee some difficulty porting to Visual C++ because of its lack of standards conformance. Since the C++ standard has been finalized, we fully expect that language conformance problems will cease to be a significant issue in the near future.

CHAPTER 7

CONCLUSION AND AVAILABILITY

7.1 Conclusion

In this thesis, I applied the emerging paradigm of generic programming to the important problem domain of graphs and graph algorithms. Our resulting framework, the Generic Graph Component Library, is a collection of generic algorithms and data structures that interoperate through the abstract graph interface comprised of Vertex, Edge, Visitor, and Decorator concepts. The generic GGCL algorithms allow basic algorithm patterns to be applied in different ways to build up more complicated graph algorithms, resulting in significant code reuse. Similarly, since GGCL algorithms are independent of the underlying graph representation, custom graph representation implementation can be mixed and matched with GGCL graph algorithms. Since our C++ implementation of the generic programming paradigm makes heavy use of static (compile-time) polymorphism, there is no run-time overhead associated with the powerful abstractions provided by GGCL. Experimental results demonstrate that the GGCL executes significantly faster than LEDA, a well-known object-oriented graph library, and can even compete with high performance Fortran code.

7.2 Availability

The source code and complete documentation for the GGCL can be downloaded from the GGCL home page at

`http://lsc.nd.edu/research/ggcl`

BIBLIOGRAPHY

[1] Netlib repository. http://www.netlib.org/.

[2] University of Florida sparse matrix collection. http://www-pub.cise.ufl.edu/∼davis/sparse/.

[3] M. J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1999.

[4] M. H. Austern. *Generic Programming and the STL*. Addison Wesley Longman, Inc, October 1998.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] M. Forster, A. Pick, and M. Raitner. *Graph Template Library*. http://www.fmi.uni-passau.de/Graphlet/GTL/.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addiaon Wesley Publishing Company, October 1994.

[8] A. George, J. R. Gilbert, and J. W. Liu, editors. *Graph Theory and Sparse Matrix Computation*. Springer-Verlag New York, Inc, 1993.

[9] A. George and J. W. H. Liu. User's guide for SPARSPAK: Waterloo sparse linear equations packages. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1980.

[10] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Computational Mathematics. Prentice-Hall, 1981.

[11] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[12] R. G. Grimes, J. G. Lewis, and I. S. Duff. User's guide for the harwell-boeing sparse matrix collection. User's Manual Release 1, Boeing Computer Services, Seattle, WA, October 1992.

[13] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process*. Addison-Wesley, 1999.

[14] D. E. Knuth. *Stanford GraphBase: a platform for combinatorial computing*. ACM Press, 1994.

[15] M. Lee and A. Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.

[16] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transaction on Mathematical Software*, 11(2):141–153, 1985.

[17] K. Mehlhorn and S. Naeher. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[19] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[20] E. G. Ng and P. Raghavan. Performance of greedy ordering heuristics for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, To appear.

[21] Object Management Group. *UML Notation Guide*, version 1.1 edition, September 1997. http://www.rational.com/uml/.

[22] Y. Saad. *Iterative Methods for Sparse Minear System*. PWS Publishing Company, 1996.

[23] Y. Samaragdakis and D. Batory. Implementing layered designs with mixin layers. In *The Europe Conference on Object-Oriented Programming*, 1998.

[24] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[25] J. G. Siek and A. Lumsdaine. Mayfly: A pattern for light-weight generic interfaces. In *PLOP99*, 1999. Accepted.

[26] S. Skiena. *Implementing Discrete mathematics*. Addion-Wesley, 1990.

[27] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc, 1998.

[28] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[29] M. Yannanakis. Computing the minimum fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, 1981.

APPENDIX A

GRAPHS

A.1 Concepts

A.1.1 Graph

Description

The **Graph** concept merely contains a set of vertices and a set of edges and a tag to specify whether it is a directed graph or an undirected graph.

Notations

- `X` A type that is a model of Edge
- `G` An object of the `X`

Table A.1: Expression semantics of concept Graph

Expression	Description
`graph_traits < X > ::vertex_type`	Vertex type
`graph_traits < X > ::edge_type`	Edge type
`graph_traits < X > ::vertices_type`	The return type of `vertices()`
`graph_traits < X > ::edges_type`	The return type of `edges()`
`vertices(G)`	To return a ContainerRef object held all vertices in the graph.

Expression	Description
`edges(G)`	To return a ContainerRef object held all edges in the graph.

Table A.2: Function specification of concept Graph

Prototype	Description
`vertices_type vertices(G)`	To return a ContainerRef object held all vertices in the graph.
`edges_type edges(G)`	To return a ContainerRef object held all edges in the graph.

Models

- graph
- LEDA_graph

Notes

Global functions instead of member functions are chosen to make the concept more general. `ContainerRef` is similar to the `Container` concept except that the former lacks the notion of "ownership", so making a copy of a `ContainerRef` object merely creates an alias to the same underlying container. Obviously, a reference to a `Container` object satisfies this requirements

A.2 Graph type selectors

A.2.1 adjacency_list

Description

To choose a graph type whose representation is the adjacency list. See GraphRepresentation for details about the concept of a graph representation. The concrete graph representation is selected by the template arguemnt `ConcreteRep`. The data stored in OneD part can be ordered or unorder with respect to the vertex. The second template argument is used to choose ordered or unordered. Here are several examples of adjacency lists and the example code to use them.

```
typedef adjacency_list < listT, ordered > GraphRep;
typedef adjacency_list < slistT, ordered > GraphRep2;
typedef adjacency_list < flistT, unordered > GraphRep3;
typedef adjacency_list < vecT, unordered > GraphRep4;
typedef adjacency_list < mapT > GraphRep5;
typedef adjacency_list < hash_mapT > GraphRep6;
typedef adjacency_list < ggcl_vecT, ordered > GraphRep7;
typedef graph< GraphRep1 >  Graph1;
typedef graph< GraphRep2, undirected > Graph2;
typedef graph< GraphRep3, directed > Graph3;
```

The Table A.3 describes the concrete graph representation associated with the predefined selectors.

Although the concrete graph representations selectd by the predefined selectors are indeed two-dimensional, users are able to use those concrete graph representations other than those predefined ones. For example, users have a model of twod container which is

Table A.3. Concrete graph representations

selector	ordered/unordered	concrete graph rep to select
listT	both	ggcl_vec <std::list >
slistT	both	ggcl_vec <std::slist >
flistT	both	ggcl_vec <flist >
vecT	both	ggcl_vec <std::vector >
mapT	ordered only	ggcl_vec <std::map >
hashmapT	ordered only	ggcl_vec <std::hash_map >
ggcl_vecT	both	ggcl_vec <ggcl_vec >

called `compressed2D`. The following will create a custom adjacent_list graph type and it is able to be used in GGCL.

```
struct compT {};  //define a custom selector

template < class StoredEdge, class IsOrdered >
class graph_representation_gen < adjacency_list
                          < compT, IsOrdered > > {
  typedef compressed2D < StoredEdge >   graphrep_type;

  template <class Iter>
  static StoredEdge* get_edge(Iter i);

  template < class RandomAccessIter, class ForwardIter >
  static RandomAccessIter
  get_target(RandomAccessIter b, ForwardIter i);

  template < class OneD, class size_type >
  static bool add(OneD& c, size_type j,
              const StoredEdge& val);

  template < class OneD, class size_type >
  static bool remove(OneD& c, size_type j);
};

typedef graph < adjacency_list < compT, unordered >,
             undirected > Graph;
```

Definition

tags.h

Table A.4: Template parameters of class `adjacency_list`

Parameter	Default	Description
`ConcreteRep`		the concrete representation type selector
`IsOrdered`		ordered or unordered
`ConcreteRep`	`ggcl_vecT`	concrete representation type selector
`IsOrdered`	`ordered`	to store edges in order or not see ordered and unordered

Table A.5: Members of `adjacency_list`

Declaration	Description	Where Defined
`enum type = ADJACENCY_LIST, isOrdered = IsOrdered::type`		
`concrete_rep_type`	concrete representation type selector	
`is_ordered_type`	type of graph representation for ordered or unordered storage.	

A.2.2 adjacency_matrix

Description

To choose a graph type whose representation is adjacency matrix. The adjacency matrix graph is ordered implicitly. Adding or removing an edge takes constant time. However, the traversing an adjacency matrix graph is not so efficient as traversing an adjacency list graph.

Currently, the selected OneD container is required to be a model of RandomAccessContainer. See sgi stl documentation for the concept of RandomAccessContainer. Thus, vecT and ggcl_vecT are the only two predefined selectors now althrough users could provides their own ones.

Table A.6: Template parameters of class `adjacency_matrix`

Parameter	Default	Description
`ConcreteRep`	`vecT`	concrete representation type selector

Table A.7: Members of `adjacency_matrix`

Declaration	Description	Where Defined
`enum type = ADJACENCY_MATRIX, isOrdered = ORDERED`		
`concrete_rep_type`		
`is_ordered_type`		

A.2.3 directed

Description

The tag for directed graph

A.2.4 dynamic

Description

To choose a graph type whose representation is dynamic. Thus, only `head` of a graph is directly available from the graph class through `root()` method.

Table A.8: Template parameters of class `dynamic`

Parameter	Default	Description
`ConcreteRep`	`vecT`	OneD part selector
`IsOrdered`	`ordered`	OneD part selector

Table A.9: Members of `dynamic`

Declaration	Description	Where Defined
`enum type = DYNAMIC, isOrdered = IsOrdered::type`		
`concrete_rep_type`		
`is_ordered_type`		

A.2.5 undirected

Description

The tag for undirected graph

A.3 Graph classes

A.3.1 LEDA_Graph

Description

GGCL algorithms are truely generic. Users are able to use it as long as the data structures used meet the cooresponded concepts. This is one of three classes to meet the graph concept for LEDA's graph data structure. Here is a brief example to use them:

```
GRAPH < int, int > _G;  //This is the LEDA's graph object.
//...
//use GGCL algorithms
typedef LEDA_Graph< GRAPH < int, int > > Graph;
Graph G(_G);
bfs(G, ...);
```

Example

In bfs_leda.cc:

```
GRAPH<int,int> _G;
//LEDA graph data
typedef LEDA_Graph< GRAPH<int,int> > Graph;
Graph G(_G);

typedef Graph::vertex_type Vertex;
Vertex s = *(G.vertices().begin());

ggcl_vec<Vertex> p(G.num_vertices());
ggcl_vec<Graph::size_type> d(G.num_vertices());
ggcl_vec<default_color_type> color(G.num_vertices());

bfs(G, s, visit_distance(mapfun(d),
        visit_predecessor(mapfun(p))), mapfun(color));
```

Definition

LEDA.h

Table A.10: Members of `LEDA_Graph`

Declaration	Description	Where Defined
`vertex_type`		Graph
`edge_type`		Graph
`size_type`		
`rep_tag`		Graph
`struct vertices_type`		Graph
`LEDA_Graph (LEDAG& _G)`		
`vertices_type vertices ()`		Graph
`size_type num_vertices () const`		

See also

LEDA_Vertex, LEDA_Edge

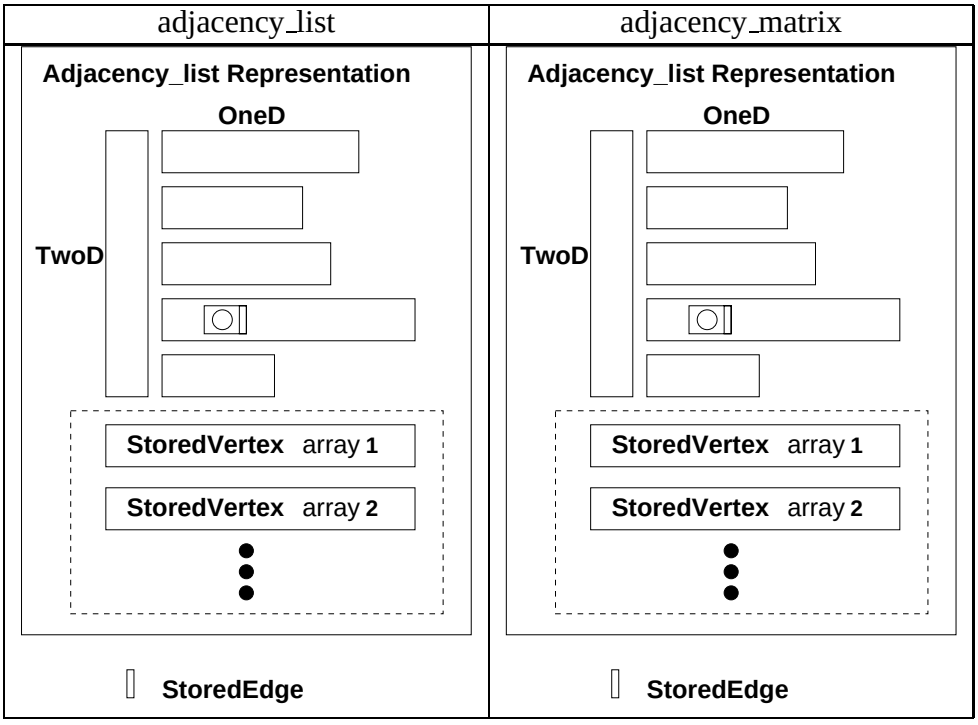
A.3.2 graph

Description

This is the GGCL implemention of GGCL Graph concept. The adjacency_list and adjacency_matrix representations are referred to this class. Dynamic represetnation graphs is refered to another class which is the specialization. A graph object with any graph representation is able to remove edges, add edges, remove vertices, and add vertices. The following pictures give you an overview of a graph with adjacency_list or adjacency_matrix represetation. Specially, The two dimensional structure of **TwoD** and **OneD** are conceptual in the pictures. The implementation could be in a segment of contigous memory depend on the concrete graph representation data structure.

Here are several simple examples of defining graphs:

Table A.11. Overview of adjacency list and adjacency matrix graphs




```
typedef graph <>  Graph1;

typedef graph < adjacency_matrix <> > Graph2;

typedef on_vertex_color_plugin
        < on_vertex_distance_plugin< id_plugin<> > >
      VertexPlugin;

typedef graph < dynamic <>,
            directed, VertexPlugin > Graph3;

typedef graph < adjacency_list < vectT, unordered >,
            undireted, color_plugin<>,
            Weight< double > > Graph4;
```

Example

In bfs_1.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed, color_plugin<> >  Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(1, 4);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(2, 4);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

/* Array to store predecessor (parent) of each vertex */
std::vector<Vertex> p(G.num_vertices());

/* Array to store distances from the source to each vertex */
std::vector<Graph::size_type> d(G.num_vertices());

/* The source vertex */
Vertex s = *vertices(G).begin();

bfs(G, s, visit_distance(mapfun(d),
                    visit_predecessor(mapfun(p))));
```

Definition

graph.h

Table A.12: Template parameters of class `graph`

Parameter	Default	Description
`rep_t`	`adjacency_list<>`	graph representation selector

Parameter	Default	Description
direct_t	directed	graph type selector. Two possible types: directed and undirected
StoredVertexPlugin	off_vertex_default_plugin<>	Stored Vertex type
StoredEdgePlugin	no_plugin	Stored edge type

Table A.13: Members of graph

Declaration	Description	Where Defined
StoredEdge		
edgeplugin_type		
storedvertex_type		
direct_tag		
graphrep_gen		
vertex_type	vertex type	
const_vertex_type	constant vertex type	
edge_type	edge type	
const_edge_type	constant edge type	
size_type		
vertices_type		
const_vertices_type		
edges_type		
const_edges_type		
graph ()		

Declaration	Description	Where Defined
`graph (size_type n)`	n is number of vertices in the graph	
`graph (std::pair<size_type, size_type>* edges, size_type numedges, size_type n, const edgeplugin_type& ep = edgeplugin_type() )`		
`num_v (n)`		
`bool add_edge (size_type i, size_type j, const edgeplugin_type& ep=edgeplugin_type())`	add an edge i -> j for directed graph or an edge i - j for undirected graph	
`bool add_directed_edge (size_type i, size_type j, const edgeplugin_type& ep=edgeplugin_type())`		
`int remove_edge (size_type i, size_type j)`		
`void remove_all_edges (size_type i)`		
`void remove_vertex (size_type i)`	remove vertex whose id is i	
`void add_vertex ()`	add a new vertex	
`vertices_type vertices ()`		
`const_vertices_type vertices () const`		
`const size_type num_vertices () const`		
`edges_type edges ()`		
`const_edges_type edges () const`		
`void print () const`	get a vertex type from	

A.3.3 graph

Description

This is the partial specialization of graph class for dynamic representation. The following picture gives you an overview of a graph with a dynamic representation. This class does not take care emory managemnet of dynamic_node. Thus add a new vertex without any new edges will not affect the class. This is the reason why there is no `add_vertex()` method here.

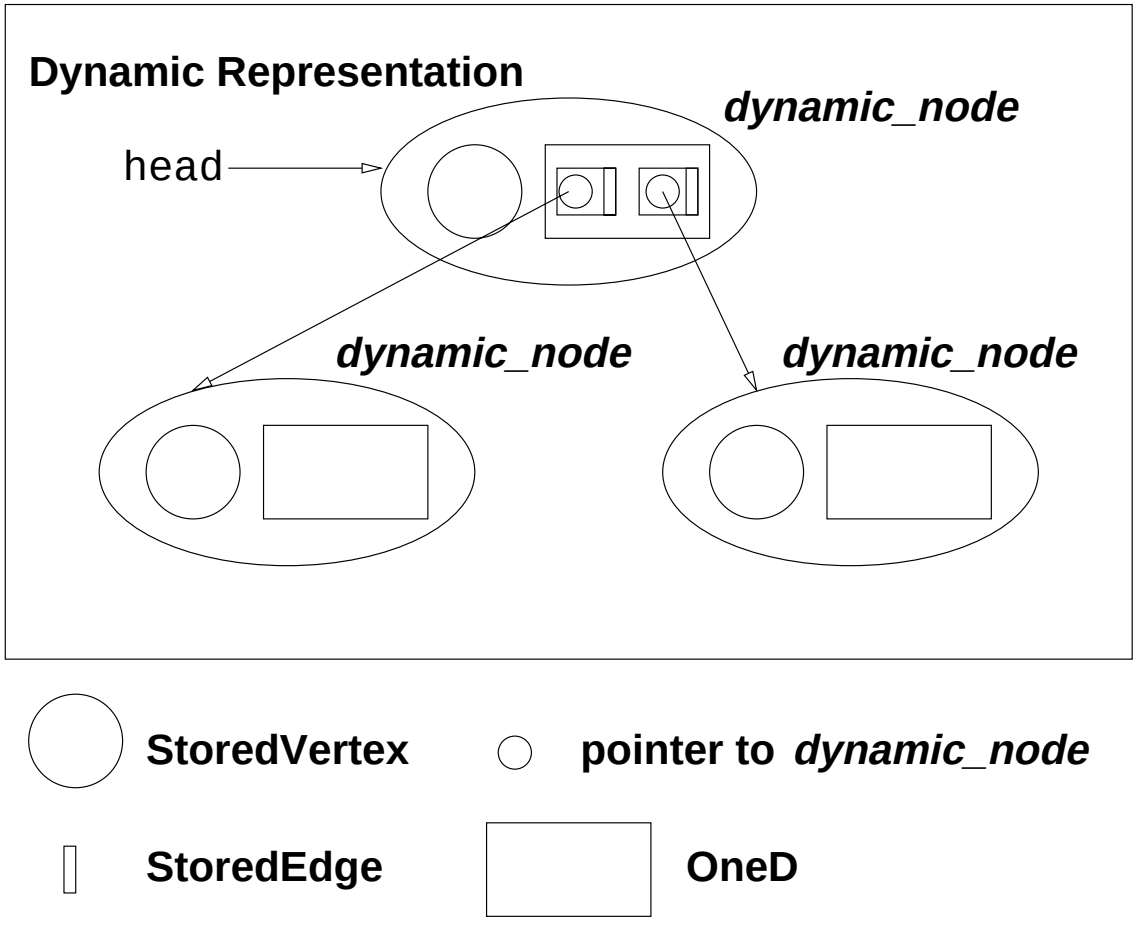


Figure A.1. Overview of the dynamic graph

Unlike the adjacency_list graph, here the `Storedvertex` is stored inside the `dynamic_node`. Here is an example to define a dynamic graph:

```
typedef on_vertex_color_plugin
           < on_vertex_distance_plugin< id_plugin<> > >
      VertexPlugin;
typedef graph< dynamic < vecT, unordered >, directed,
            VertexPlugin, no_plugin> Graph;
```

See examples for how to create a wraper for a pointwise graph data structure so that GGCL algorithms can be applied.

Example

In dynamic.cc:

```
typedef on_vertex_color_plugin< on_vertex_distance_plugin
          < id_plugin<> > > VertexPlugin;

typedef graph<dynamic<listT, unordered>,
          directed, VertexPlugin> Graph;

typedef Graph::storedvertex_type DynamicVertex;
typedef Graph::vertex_type Vertex;

DynamicVertex* head = new DynamicVertex();

Graph G(head);

DynamicVertex* v1 = new DynamicVertex();
DynamicVertex* v2 = new DynamicVertex();

G.add_edge(head, v1);
G.add_edge(head, v2);

DynamicVertex* v3 = new DynamicVertex();
DynamicVertex* v4 = new DynamicVertex();

G.add_edge(v2, v3);
G.add_edge(v2, v4);

DynamicVertex* v5 = new DynamicVertex();

G.remove_edge(v2, v3);
G.add_edge(v1, v3);
G.add_edge(v1, v5);


bfs(G, G.root(), visit_distance( distance_decorator<Vertex>() ),
   color_decorator<Vertex>() );

cout << head->distance() << " " << v1->distance() << " "
    << v2->distance() << " " << v3->distance() << " "
    << v4->distance() << " " << v5->distance() << endl;
```

More examples can be found in pointwise.cc

Definition

dynamic.h

Table A.14: Members of `graph`

Declaration	Description	Where Defined
`rep_tag`	representation tag.	
`graphrep_gen`		
`graphrep_type`	graph representation type	
`storedvertex_type`	stored vertex type.	
`vertex_type`	vertex type	
`const_vertex_type`		
`edge_type`	edge type	
`const_edge_type`		
`direct_tag`	direct tag	
`edgeplugin_type`	stored edge type	
`graph ()`		
`graph (storedvertex_type* _h)`		
`bool add_edge (storedvertex_type* u, storedvertex_type* v, const edgeplugin_type& ep=edgeplugin_type())`	add an edge from vertex `u` to vertex `v`.	
`void remove_edge (storedvertex_type* i, storedvertex_type* j)`	remove the edge from vertex `u` to vertex `v`.	
`void remove_vertex (storedvertex_type* u)`	remove vertex `u`, currently only the function works only for undircted graph.	
`vertex_type root ()`	the root of the graph	

Declaration	Description	Where Defined
`const_vertex_type root () const`		

A.4 Graph functions

A.4.1 vertices

Prototype

```
template <class Graph>
graph_traits<Graph>::vertices_type vertices(Graph& G) ;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all vertices in graph `G`. it is a part of graph concept.

Definition

graph.h

A.4.2 vertices

Prototype

```
template <class Graph>
graph_traits<Graph>::const_vertices_type vertices(const Graph& G)
;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all vertices in graph `G`. it is a part of graph concept. This is for a constant graph object.

Definition

graph.h

A.4.3 edges

Prototype

```
template <class Graph>
graph_traits<Graph>::edges_type edges(Graph& G) ;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all edges in graph G. it is a part of graph concept.

Definition

graph.h

A.4.4 edges

Prototype

```
template <class Graph>
graph_traits<Graph>::const_edges_type edges(const Graph& G) ;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all edges in graph G. it is a part of graph concept. This is for a constant graph object.

Definition

graph.h

APPENDIX B

GRAPH REPRESENTATIONS

B.1 Concepts

B.1.1 GraphRepresentation

Description

A graph can be represented in several ways. The way how to representate a graph is called the graph representation. GGCL graphs can be three categories of graph representations: adjacency matrix, adjacency list, and dynamic pointer based representation.

The **GraphRepresentation** concept is basically a TwoD Container (conceptually it is a Container of Containers although it is not necessary to implement it as a Container of Containers such as `vector < vector < int > >`.) coupled with four helper functions. we often refer the TwoD Container as concrete graph representation. A OneD Container within a GraphRepresentation corresponds to the out-edge list for a particular vertex. In addition, there is a one-to-one correspondence between the TwoD Iterator and the vertices of the graph.

Table B.1: Expression semantics of concept GraphRepresentation

Expression	Description
`get_target(b, i)`	To deduce a TwoDIterator from a TwoDIterator and a OneDIterator. Used for deriving the `target(e)` from an edge `e`.

Expression	Description
`get_edge(i)`	An access method to the extra edge information stored within an edge list
`add(elist, v, storededge)`	To add an edge
`remove(elist, v)`	To remove an edge

Table B.2: Function specification of concept GraphRepresentation

Prototype	Description
`TwoDIterator get_target(TwoDIterator b, OneDIterator i)`	To deduce a TwoDIterator from a TwoDIterator and a OneDIterator. Used for deriving the `target(e)` from an edge `e`.
`stored_edge* get_edge(OneDIterator i)`	An access method to the extra edge information stored within an edge list
`bool add(EdgeList& elist, size_type vertex, const stored_edge& e)`	To add an edge
`void remove(EdgeList& elist, size_type vertex)`	To remove an edge

Models

- adjacency_list
- adjacency_matrix
- dynamic

B.2 Graph representation type selectors

B.2.1 flistT

Description

flist(Fortran list) is chosen as the OneD part of the graph representation

B.2.2 ggcl_vecT

Description

ggcl_vec, a model of random access container, is chosen as the OneD part of the graph representation

B.2.3 hash_mapT

Description

std::hash_map(sgi stl extension) is chosen as the OneD part of the graph representation. Thus, it is always ordered. If the `StoredEdge` type is `no_plugin` in the graph, which indicates no extera stored edge information, std::hash_set(sgi stl extension) is chosen as the OneD part.

B.2.4 listT

Description

std::list is chosen as the OneD part of the graph representation

B.2.5 mapT

Description

std::map is chosen as the OneD part of the graph representation. Thus, it is always ordered. If the `StoredEdge` type is `no_plugin` in the graph, which indicates no extera stored edge information, std::set is chosen as the OneD part.

B.2.6 ordered

Description

The tag for the graph representation. The adjacent vertices of any vertex are in order with respect to the vertex. Mulitple edges between two vertices are not allowed. Thus, if adding an edge which is already there, the edge is overwitten. Adding an edge is implemented by two steps: First step is to search the position to add using `lower_bound`. The second step is insert the edge using `insert` defined in OneD. Removing an edge also has two steps. The first step is to search the position, which is the same as adding an edge. The second step is to call `erase` defined in OneD. Thus, time complexity depends on the container to choose for OneD. If there will have a lot of adding and removing edge operations, it is recommended to use ordered graph to represent it.

Table B.3: Members of `ordered`

Declaration	Description	Where Defined
`enum type = ORDERED`		

B.2.7 slistT

Description

std::slist(sgi stl extention) is chosen as the OneD part of the graph representation

B.2.8 unordered

Description

The tag for the graph representation. The adjacent vertices of any vertex are not in order with respect to the vertex. Mulitple edges between two vertices may exist. Adding an edge takes constant time. Meanwhile, removing an edge takes linear (to the length of the OneD part) time. In the worse case it could be linear to the number of vertices in the

graph. It is recommended to use unordered graph if the graph will have no removing-edges operations.

Table B.4: Members of `unordered`

Declaration	Description	Where Defined
`enum type = UNORDERED`		

B.2.9 vecT

Description

std::vector is chosen as the OneD part of the graph representation

APPENDIX C

VERTICES

C.1 Concepts

C.1.1 Vertex

Description

Vertex provides access to the adjacent vertices, the out-edges of the vertex and optionally the in-edges.

Notations

- X A type that is a model of Vertex
- u An Object of type X

Table C.1: Expression semantics of concept Vertex

Expression	Description
`vertex_traits < X > ::edgelist_type`	the return type of `adj()`
`vertex_traits < X > ::vertexlist_type`	the return type of `out_edges()`
`out_edges(u)`	To return a ContainerRef object held all out-edges

Expression	Description
`in_edges(u)`	Optional. To return a ContainerRef object held all in-edges
`adj(u)`	To return a ContainerRef object held all adjacent vetices.

Table C.2: Function specification of concept Vertex

Prototype	Description
`edgelist_type out_edges(u)`	To return a ContainerRef object held all out-edges
`edgelist_type in_edges(u)`	Optional. To return a ContainerRef object held all in-edges
`vertexlist_type adj(u)`	To return a ContainerRef object held all adjacent vetices.

Models

- vertex
- LEDA_vertex

Notes

Global functions instead of member functions are chosen to make the concept more general. `ContainerRef` is similar to the `Container` concept except that the former lacks the notion of "ownership", so making a copy of a `ContainerRef` object merely creates an alias to the same underlying container. Obviously, a reference to a `Container` object satisfies this requirements

C.2 Vertex classes

C.2.1 LEDA_Vertex

Description

This is one of three classes to make LEDA's graph data structure work under GGCL algorithms. It is to implement the Vertex concept.

Definition

LEDA.h

Table C.3: Members of `LEDA_Vertex`

Declaration	Description	Where Defined
`size_type`		
`edge_type`		
`vertex_type`		
`LEDA_Vertex ()`		
`LEDA_Vertex (node _v)`		
`LEDA_Vertex (const LEDA_Vertex& x)`		
`struct edgelist_type`		Vertex
`struct vertexlist_type`		Vertex
`vertexlist_type adj ()`		Vertex
`edgelist_type out_edges ()`		Vertex
`size_type id () const`		

See also

LEDA_Edge, LEDA_Graph

C.2.2 vertex

Description

The GGCL implementation of Vertex concept. See GGCL graph class. The convenient way to refer a vertex type is through graph type. Here is an example:

```
typedef graph< adjacency_list<>, directed > Graph;
Graph G(n);
...
typedef Graph::vertex_type Vertex;
typedef Graph::edge_type  Edge;
//Edge e
...
Vertex u = e.source();
```

Definition

vertex.h

Table C.4: Members of `vertex`

Declaration	Description	Where Defined
`enum type = CONST`		
`graph`	the graph type for this vertex object	
`direct_tag`	the tag to indicate the graph is directed or undirected.	
`graphrep_type`	the graph representation type	
`edgelist_type`		Vertex
`vertexlist_type`		Vertex

Declaration	Description	Where Defined
`edge_type`	The edge type of current graph object associated with.	
`size_type`		
`GraphRepPtr`		
`plugin_type`	the type of stored vertex	
`StoredVertexPtr`		
`gr_iterator`		
`vertex ()`		
`vertex (gr_iterator i, GraphRepPtr m, StoredVertexPtr vp)`		
`vertex (const self& x)`		
`vertex& operator= (const self& x)`		
`vertexlist_type adj () const`	return a container object held all adjacent vertices	Vertex
`edgelist_type out_edges () const`	return a container object held all out-edges	Vertex
`plugin_type* plugin ()`	Stored vertex.	
`const plugin_type* plugin () const`		
`bool operator!= (const self& x) const`		
`bool operator== (const self& x) const`		
`gr_iterator iter`		
`GraphRepPtr matrix`		

C.3 Vertex functions

C.3.1 adj

Prototype

```
template <class Vertex>
vertex_traits<Vertex>::vertexlist_type adj(Vertex u) ;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all adjacent vertices of the vertex. This is a part of Vertex concept.

Definition

vertex.h

C.3.2 out_edges

Prototype

```
template <class Vertex>
vertex_traits<Vertex>::edgelist_type out_edges(Vertex u) ;
```

Description

This is a global function to return an instance of a model of ContainerRef which holds all out edges of the vertex. This is a part of Vertex concept.

Definition

vertex.h

APPENDIX D

EDGES

D.1 Concepts

D.1.1 Edge

Description

An **Edge** is a pair of vertices, one is the source vertex and the other is the target vertex. In the unordered case It is just assumed that the position of the source and target vertices are interchangeable.

Notations

- `X` A type that is a model of Edge
- `e` An object of the `X`

Table D.1: Expression semantics of concept `Edge`

Expression	Description
`edge_traits < X > ::vertex_type`	Vertex type
`source(e)`	source vertex. Notice it is a global function.
`target(e)`	target vertex. Notice it is a global function.

Table D.2: Function specification of concept Edge

Prototype	Description
`vertex_type source(e)`	source vertex. Notice it is a global function.
`vertex_type target(e)`	target vertex. Notice it is a global function.

Models

- edge
- LEDA_edge

Notes

Global functions instead of member functions are chosen to make the concept more general. `ContainerRef` is similar to the `Container` concept except that the former lacks the notion of "ownership", so making a copy of a `ContainerRef` object merely creates an alias to the same underlying container. Obviously, a reference to a `Container` object satisfies this requirements

D.2 Edge classes

D.2.1 LEDA_Edge

Description

This is one of three classes to make LEDA's graph data structure work under GGCL algorithms. It is to implement the Edge concept.

Definition

LEDA.h

Table D.3: Members of `LEDA_Edge`

Declaration	Description	Where Defined
`vertex_type`		
`LEDA_Edge ()`		
`LEDA_Edge (edge _e)`		
`vertex_type source () const`		Edge
`vertex_type target () const`		Edge

See also

LEDA_Vertex, LEDA_Graph

D.2.2 edge

Description

This is the GGCL implementation of Edge. Extra information for the edge can be accessed through StoredEdge. See GGCL graph class. The convenient way to refer an edge type is through graph type. Here is an example:

```
typedef graph< adjacency_list<>, directed > Graph;
...
typedef Graph::edge_type Edge;
...
```

Definition

edge.h

Table D.4: Members of `edge`

Declaration	Description	Where Defined
`vertex_type`	vertex type	Edge
`storededge_type`	Stored edge type	
`edge ()`		
`edge (gr_iterator s, gr_-iterator d, GraphRepPtr mf, EdgePlugin* eplug, rep_iterator _i, StoredVertexPtr vplug)`		
`edge (const edge& x)`		
`edge& operator= (const edge& x)`	assignment operator	
`vertex_type source () const`	source vertex of the edge	Edge
`vertex_type target () const`	target vertex of the edge	Edge
`storededge_type* plugin ()`	Stored edge	
`const storededge_type* plugin () const`		

D.2.3 stored_edge

Description

This is the base class for StoredEdge in the adjacency_matrix representation. GGCL uses it internally.

Definition

graph.h

Table D.5: Members of `stored_edge`

Declaration	Description	Where Defined
`stored_edge ()`		

Declaration	Description	Where Defined
`stored_edge (const Plugin& p)`		
`stored_edge (const stored_edge& s)`		
`bool connected`	true if the edge exist in the graph, false otherwise.	

D.3 Edge functions

D.3.1 source

Prototype

```
template <class Edge>
edge_traits<Edge>::vertex_type source(Edge e) ;
```

Description

This is to return the source vertex of the edge. This is a part of Edge concept.

Definition

edge.h

D.3.2 target

Prototype

```
template <class Edge>
edge_traits<Edge>::vertex_type target(Edge e) ;
```

Description

This is to return the target vertex of the edge. This is a part of Edge concept.

Definition

edge.h

APPENDIX E

DECORATORS

E.1 Concepts

E.1.1 Decorator

Description

Decorator provides a generic method to access vertex and edge properties, such as color and weight, from within an algorithm. There are two categories of decorators:

Interior Decorator: The decorating properties are stored outside of the graph object (they are passed directly to the GGCL algorithm) and the decorator will access the externally stored data indexed by the vertex or edge ID.

Exterior Decorator: The decorating properties are stored inside of the graph object. The decorator consults the vertex or the edge objects to access the decorating property.

Notations

- `X` A type that is a model of Decorator
- `d` An object of type `X`
- `u` An object of a model of Edge or Vertex

Table E.1: Expression semantics of concept Decorator

Expression	Description
`decorator_traits < X > ::value_type`	the type of object accessed by the decorator.
`decorator_traits < X > ::reference`	
`d[u]`	access the decorating property of Vertex or Edge u.

Table E.2: Function specification of concept Decorator

Prototype	Description
`reference operator[](Vertex u)`	access the decorating property of Vertex or Edge u.

Models

- id_decorator
- color_decorator
- distance_decorator
- in_degree_decorator
- out_degree_decorator
- degree_decorator
- parent_decorator

- predecessor_decorator
- discover_time_decorator
- finish_time_decorator
- weight_decorator

E.2 Decorator classes

E.2.1 dummy_decorator

Description

This is to provide a dummy decorator. The `operator[](Vertex v)` always return the same one regardless any varied `Vertex v`.

Definition

decorator.h

Table E.3: Members of `dummy_decorator`

Declaration	Description	Where Defined
`iterator_category`		
`value_type`		
`difference_type`		
`pointer`		
`reference`		
`dummy_decorator ()`		
`dummy_decorator (value_type cc)`		
`dummy_decorator (const dummy_-decorator& x)`		
`template <class Vertex> reference operator[] (Vertex v)`		

Declaration	Description	Where Defined

`template <class Vertex>`
`const value_type& operator[]`
`(Vertex v) const`		

E.2.2 id_decorator

Description

This decorator is to provide a method to get Vertex ID. The `v` must be a valid vertex(`v != Vertex()` held) , otherwise, a running time error happens.

Definition

decorator.h

Table E.4: Members of `id_decorator`

Declaration	Description	Where Defined
`iterator_category`	the type to distinguish RandomAccessIterator and Decorator	
`value_type`		
`difference_type`		
`pointer`		
`reference`		

`template <class Vertex>`
`Vertex::size_type operator[]`
`(Vertex v)`		

`template <class Vertex>`
`Vertex::size_type operator[]`
`(Vertex v) const`		

E.2.3 random_access_iterator_decorator

Description

This is pretty much the same as container_decorator except this is for RandomAccessIterator instead of a random access container.

Definition

decorator.h

Table E.5: Members of `random_access_iterator_decorator`

Declaration	Description	Where Defined
`value_type`		
`iterator_category`		
`difference_type`		
`pointer`		
`reference`		
`random_access_iterator_decorator (RandomAccessIterator cc, const IDfunc& _id = IDfunc() )`		
`random_access_iterator_decorator (const random_access_iterator_-decorator& x)`		
`template <class Vertex> reference operator[] (Vertex v)`		
`template <class Vertex> const value_type& operator[] (Vertex v) const`		

E.2.4 weight_decorator

Description

this is a decorator for accessing weight of edges.

Definition

decorator.h

Table E.6: Members of `weight_decorator`

Declaration	Description	Where Defined
`value_type`	weight type	
`iterator_category`	the type to distinguish RandomAccessIterator and Decorator	
`difference_type`		
`pointer`		
`reference`		
`reference operator[] (Edge e)`	access method for weight of Edge e.	
`const value_type& operator[] (Edge e) const`		

E.3 Decorator functions

E.3.1 mapfun

Prototype

```
template <class Container>
container_decorator<Container> mapfun(Container& c) ;
```

Description

This is a utility to create an instance of Exterior Decorator. If there is:

```
std::vector < default_color_type > color(G.num_vertices());
```

then the `mapfun(color)` will return an instance of exterior decorator for color properities.

Definition

decorator.h

APPENDIX F

VISITORS

F.1 Concepts

F.1.1 Visitor

Description

Visitor is the STL functor-like object to make the graph algorithms more flexible. There are several predefined models of visitor.

Notations

`X`	A type that is a model of Visitor
`visitor`	An object of type `X`
`u`	An object of a model of Vertex
`e`	An object of a model of Edge

Table F.1: Expression semantics of concept Visitor

Expression	Description
`visitor.initialize(u)`	Invoked during initialization.
`visitor.start(u)`	Invoked at the beginning of algorithms.
`visitor.discover(u)`	Invoked when an undiscovered Vertex u is encountered.

Expression	Description
`visitor.finish(u)`	Invoked when algorithms finish visiting the Vertex u.
`visitor.process(e)`	Invoked when the edge e is traversed.

Table F.2: Function specification of concept Visitor

Prototype	Description
`void initialize(Vertex u)`	Invoked during initialization.
`void start(Vertex s)`	Invoked at the beginning of algorithms.
`void discover(Vertex u)`	Invoked when an undiscovered Vertex u is encountered.
`void finish(Vertex u)`	Invoked when algorithms finish visiting the Vertex u.
`bool process(Edge e)`	Invoked when the edge e is traversed.

Models

- dfs_visitor
- distance_visitor
- predecessor_visitor
- timestamp_visitor
- bfs_visitor
- weighted_edge_visitor
- components_visitor

- topo_sort_visitor

Notes

The implementation of a visitor should always have a Super visitor as a template and whose default arguement is recommended to be `null_visitor`. It is also recommended that visitor is inherited from Super visitor.

F.2 Visitor classes

F.2.1 bfs_visitor

Description

This is the visitor used inside the BFS algorithm

Definition

bfs_visitor.h

Table F.3: Template parameters of class `bfs_visitor`

Parameter	Default	Description
`DistanceDecorator`		distance decorator
`Base`	`null_visitor`	a Super Visitor
`FocusOnEdge`	`false`	a boolean template to determine whether an edge encountered will be always visited (by invoking process(e)) or not

Table F.4: Members of `bfs_visitor`

Declaration	Description	Where Defined
`bfs_visitor ()`		
`bfs_visitor (ColorDecorator c, const Base& b)`		
`bfs_visitor (const bfs_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`	set the color of vertex `u` as white and invoke the `Base::initialize(u)`.	Visitor
`template <class Vertex> void start (Vertex u)`	set the color of vertex `u` as gray and invoke the `Base::start(u)`.	Visitor
`template <class Vertex> void finish (Vertex u)`	set the color of vertex `u` as black and invoke `Base::finish(u)`	Visitor

Declaration	Description	Where Defined
`template <class Edge>` `bool process (Edge e)`	If the target `v` of edge `e` has not been discovered yet, it grays the `v` and invokes the `Base::process(e)` and return true, otherwise, there are two cases. If `DocusOnEdge` is true, it invokes `Base::process(e)` and return false. Otherwise, it return false only.	Visitor
`template <class Vertex>` `bool is_undiscovered (Vertex u)`	To indicate whether Vertex u is discovered or not.	

F.2.2 components_visitor

Description

Using this visitor to record which components a vertex is attributed to during the second DFS traversal in the strongly connected components algorithm.

Definition

connected_components.h

Table F.5: Template parameters of class `components_visitor`

Parameter	Default	Description
`ComponentsDecorator`		Components Decorator
`Base`	`null_visitor`	a Super Visitor

Table F.6: Members of `components_visitor`

Declaration	Description	Where Defined
`comp_type`		
`components_visitor (ComponentsDecorator _c, const Base& b=Base())`		
`components_visitor (const components_visitor& x)`		
`template <class Vertex> void discover (Vertex u)`	record which components for Vertex `u`	
`void set_count (comp_type _count)`	set the count of components to let the algorithm interact with the visitor	

F.2.3 dfs_visitor

Description

This visitor is used inside the DFS algorithm.

Definition

dfs_visitor.h

Table F.7: Template parameters of class `dfs_visitor`

Parameter	Default	Description
`ColorDecorator`		Color Decorator
`Base`	`null_visitor`	a Super Visitor
`FocusOnEdge`	`false`	a boolean template to determine whether an edge encountered will be always visited (by invoking process(e)) or not

Table F.8: Members of `dfs_visitor`

Declaration	Description	Where Defined
`color_type`		
`dfs_visitor (ColorDecorator c, const Base& b)`		
`dfs_visitor (const dfs_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`	set the color of vertex `u` as white and invoke the `Base::initialize(u)`.	Visitor
`template <class Vertex> void start (Vertex u)`	set the color of vertex `u` as gray and invoke the `Base::start(u)`.	Visitor
`template <class Vertex> void discover (Vertex u)`	set the color of vertex `u` as gray and invoke the `Base::discover(u)`.	Visitor
`template <class Vertex> void finish (Vertex u)`	void operation.	Visitor

Declaration	Description	Where Defined
`template <class Edge>` `bool process (Edge e)`	If the target `v` of edge `e` has not been discovered yet, it grays the `v` and invokes the `Base::process(e)` and return true, otherwise, there are two cases. If `DocusOnEdge` is true, it invokes `Base::process(e)` and return false. Otherwise, it return false only.	Visitor
`template <class Vertex>` `bool is_undiscovered (Vertex u)`	check whether Vertex u is undiscovered by checking the color of u.	
`template <class Vertex>` `bool is_finished (Vertex u)`	check whether visiting Vertex u is finished by checking whether the color is black or not. If so, invoke `Base::finish(u)` and return true otherwise return false only.	

F.2.4 distance_visitor

Description

This visitor is used to calculate the distance for every verterx from the reference vertex (source).

Definition

distance_visitor.h

Table F.9: Template parameters of class `distance_visitor`

Parameter	Default	Description
`DistanceDecorator`		distance decorator
`Base`	`null_visitor`	a Super Visitor

Table F.10: Members of `distance_visitor`

Declaration	Description	Where Defined
`distance_visitor ()`		
`distance_visitor (DistanceDecorator dist)`		
`distance_visitor (DistanceDecorator dist, const Base& x)`		
`distance_visitor (const distance_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`	Initialize the distance of vertex `u` and invoke `Base::initialize(u)`.	Visitor

Declaration	Description	Where Defined
`template <class Vertex>` `void start (Vertex s)`	Set the distance of vertex `s` to be zero and invoke `Base::start(s)`.	Visitor
`template <class Edge>` `bool process (Edge e)`	If the target `v` of `e` have not been set a distance, `d[e.target()] = d[e.source()] + 1`, invoke the `Base::process(e)` and return true. Otherwise, invoke the `Base::process(e)` and return false.	Visitor

F.2.5 level_visitor

Description

This is the the visitor to set the level for every vertex. The level of starting vertices is zero.

Definition

level_decorator.h

Table F.11: Members of `level_visitor`

Declaration	Description	Where Defined
`level_visitor (LevelDecorator l, const Super& x = Super())`		
`level_visitor (const level_visitor& x)`		

Declaration	Description	Where Defined
`template <class Vertex> void initialize (Vertex u)`		
`template <class Edge> bool process (Edge e)`		
`LevelDecorator level`		

F.2.6 null_visitor

Description

This is a visitor to provide only the standard visitor interface. All methods are emtpy.

Definition

util.h

Table F.12: Members of `null_visitor`

Declaration	Description	Where Defined
`null_visitor ()`		
`null_visitor (const null_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`		Visitor
`template <class Vertex> void start (Vertex s)`		Visitor
`template <class Vertex> void discover (Vertex s)`		Visitor
`template <class Vertex> void finish (Vertex s)`		Visitor
`template <class Edge> bool process (Edge e)`		Visitor

F.2.7 predecessor_visitor

Description

This is a visitor to record the predecessor of vertex discovered in the graph algorithms.

Definition

predecessor_visitor.h

Table F.13: Template parameters of class `predecessor_visitor`

Parameter	Default	Description
`PredecessorDecorator`		a predecessor decorator with `Vertex operator[](Vertex)` defined.
`Base`		a super visitor

Table F.14: Members of `predecessor_visitor`

Declaration	Description	Where Defined
`predecessor_visitor ()`		
`predecessor_visitor (PredecessorDecorator _p)`		
`predecessor_visitor (PredecessorDecorator _p, const Base& b)`		
`predecessor_visitor (const predecessor_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`	Initialize the predecessor of vertex `u` and invoke `Base::initialize(u)`	
`template <class Vertex> void start (Vertex s)`	Set the predecessor of vertex `s` as a null vertex and invoke `Base::start(s)`	

Declaration	Description	Where Defined
`template <class Edge>` `bool process (Edge e)`	Set `p[target(e)] = source(e)` and invoke `Base::process(e)`	

F.2.8 timestamp_visitor

Description

This visitor is to record the discover time and finish time of vertices during graph traversal. Notice that only one timer for both time.

Definition

timestamp_visitor.h

Table F.15: Template parameters of class `timestamp_visitor`

Parameter	Default	Description
`DiscoverTime`		discover time decorator
`FinishTime`		finish time decorator
`Base`	`null_visitor`	a Super Visitor

Table F.16: Members of `timestamp_visitor`

Declaration	Description	Where Defined
`timestamp_visitor ()`		
`timestamp_visitor (DiscoverTime disc, FinishTime fin)`		
`timestamp_visitor (DiscoverTime disc, FinishTime fin, const Base& b)`		

Declaration	Description	Where Defined
`timestamp_visitor (const timestamp_visitor& x)`		
`template <class Vertex> void discover (Vertex u)`	Increment timer, set the discover time for vertex `u` and invoke `Base::discover(u)`	Visitor
`template <class Vertex> void finish (Vertex u)`	Increment timer, set the finish time for vertex `u` and invoke `Base::finish(u)`	Visitor

F.2.9 topo_sort_visitor

Description

This is to record the vertex in topological order.

Definition

topological_sort.h

Table F.17: Template parameters of class `topo_sort_visitor`

Parameter	Default	Description
`OutputIterator`		output iterator
`Base`		Super Visitor

Table F.18: Members of `topo_sort_visitor`

Declaration	Description	Where Defined
`topo_sort_visitor (OutputIterator _iter, Base x)`		
`template <class Vertex> void finish (Vertex& u)`		

F.2.10 weighted_edge_visitor

Description

This is a generalization of the kind of visitor used inside Dijkstra's and Prim's algorithms.

This is also used for the min-max paths problem.

Definition

weighted_edge_visitor.h

Table F.19: Template parameters of class `weighted_edge_visitor`

Parameter	Default	Description
`WeightDecorator`		weight decorator
`DistanceDecorator`		distance decorator
`Base`		Super visitor
`BinaryOperator`	`std::plus`	std::plus for dijkstra and _-project2nd for prim

Table F.20: Members of `weighted_edge_visitor`

Declaration	Description	Where Defined
`weighted_edge_visitor ()`		

Declaration	Description	Where Defined
`weighted_edge_visitor (WeightDecorator wf, DistanceDecorator df, const Base& b)`		
`weighted_edge_visitor (WeightDecorator wf, DistanceDecorator df, BinaryOperator binop, const Base& b)`		
`weighted_edge_visitor (const weighted_edge_visitor& x)`		
`template <class Vertex> void initialize (Vertex u)`	Initialize the distance of vertex `u` and invoke `Base::initialize(u)`.	Visitor
`template <class Vertex> void start (Vertex s)`	Set the distance of vertex `s` to be zero and invoke `Base::start(s)`.	Visitor
`template <class Edge> bool process (Edge e)`	If the target `v` of `e` have not been set a distance, update it and return true. Otherwise, update the distance if need and return false.	Visitor
`bool need_update_queue ()`	To indicate whether update queue operatoion need to perform	

F.3 Visitor functions

F.3.1 visit_distance

Prototype

```
template <class Distance>
distance_visitor<IglueD<Distance>::type, null_visitor> visit_-
distance(Distance d) ;
```

Description

To return an instance of distance visitor with Distance and null_visitor as template arguments, like `make_pair` return a `pair` object in the STL.

Definition

distance_visitor.h

Requirements on types

- Distance - an instance of a distance decorator or a RandomAccessIterator.

F.3.2 visit_distance

Prototype

```
template <class Distance, class SuperVisitor>
distance_visitor<IglueD<Distance>::type, SuperVisitor> visit_-
distance(Distance d, SuperVisitor b) ;
```

Description

return an instance of distance visitor with Distance and SuperVisitor as template arguments.

Definition

distance_visitor.h

Requirements on types

- Distance d - an instance of a distance decorator or a RandomAccessIterator.
- SuperVisitor b- an instance of a visitor.

F.3.3 visit_level

Prototype

```
template < class LevelDecorator >
level_visitor<LevelDecorator> visit_level(LevelDecorator level) ;
```

Description

Definition

level_visitor.h

F.3.4 visit_level

Prototype

```
template < class LevelDecorator, class Super>
level_visitor<LevelDecorator, Super> visit_level(LevelDecorator
level, const Super& b) ;
```

Description

Definition

level_visitor.h

F.3.5 visit_predecessor

Prototype

```
template <class Predecessor>
predecessor_visitor<IglueD<Predecessor>::type, null_visitor>
visit_predecessor(Predecessor p) ;
```

Description

This function returns an instance of predecessor_visitor with Predecessor and null_visitor as template arguments. The former can be a model of PredecessorDecorator or a model of RandomAccessIterator.

Definition

predecessor_visitor.h

F.3.6 visit_predecessor

Prototype

```
template <class Predecessor, class BaseVisitor>
predecessor_visitor<IglueD<Predecessor>::type, BaseVisitor>
visit_predecessor(Predecessor p, BaseVisitor b) ;
```

Description

This function returns an instance of predecessor_visitor with Predecessor and BaseVisitor as template arguments. The former can be a model of PredecessorDecorator or a model of RandomAccessIterator.

Definition

predecessor_visitor.h

F.3.7 visit_timestamp

Prototype

```
template <class DiscoverTime, class FinishTime>
timestamp_visitor<IglueD<DiscoverTime>::type,
IglueD<FinishTime>::type, null_visitor> visit_-
timestamp(DiscoverTime d, FinishTime f) ;
```

Description

To return an instance of timestamp_visitor with no super visitor.

Definition

timestamp_visitor.h

F.3.8 visit_timestamp

Prototype

```
template <class DiscoverTime, class FinishTime, class Base>
timestamp_visitor<IglueD<DiscoverTime>::type,
IglueD<FinishTime>::type, Base> visit_timestamp(DiscoverTime
d, FinishTime f, const Base& b) ;
```

Description

To return an instance of timestamp_visitor with super visitor `Base`.

Definition

timestamp_visitor.h

F.3.9 visit_bfs

Prototype

```
template <class Color, class SuperVisitor>
bfs_visitor<IglueD<Color>::type, SuperVisitor> visit_bfs(Color
c, SuperVisitor b) ;
```

Description

It takes two arguments and return an instance of bfs_visitor. The type of the first one could be either a model of ColorDecorator or a model of RandomAccessIterator. The second one is the model of Visitor.

Definition

bfs_visitor.h

APPENDIX G

ALGORITHMS

G.1 GGCL algorithms

G.1.1 _generalized_init

Prototype

```
template <class Graph, class Visitor>
void _generalized_init(Graph& G, Visitor visit) ;
```

Description

We spearate the initializtion step from main algorithms in case users want to call main algorithms mulitple times. If the `G` is a model of dynamic graph reperesentation, This function does nothing. Otherwise, `visit.initialize(u)` gets invoked for every vertex `u` in the graph `G`.

Definition

bfs.h

G.1.2 _generalized_BFS

Prototype

```
template <class Vertex, class QType, class Visitor, class
VisitedFunc>
void _generalized_BFS(Vertex s, QType& Q, Visitor visit,
VisitedFunc visited) ;
```

Description

A generalized BFS algorithm with all argument types templatized. The initialization step is **not** included. If users want it, users can either call `_generalized_init` first then call this function, or use `generalized_BFS` which includes the initialization step. We separate the initializtion step from main algorithms in case users want to call main algorithms multiple times.

Definition

bfs.h

See also

_generalized_init, generalized_BFS

G.1.3 generalized_BFS

Prototype

```
template <class Graph, class QType, class Visitor, class
VisitedFunc>
void generalized_BFS(Graph& G, typename graph_-
traits<Graph>::vertex_type s, QType& Q, Visitor visit,
VisitedFunc visited) ;
```

Description

A generalized BFS algorithm with all argument types templatized. The initialization step is included.

Definition

bfs.h

See also

_generalized_init, _generalized_BFS

G.1.4 bfs

Prototype

```
template <class Graph, class QType, class Visitor, class Color >
void bfs(Graph& G, typename graph_traits<Graph>::vertex_type s,
QType& Q, Visitor visit, Color c) ;
```

Description

Three versions of overloaded BFS algorithms are provided in GGCL.

In the first version, the arguments are Graph `G`, its starting vertex `s` and a visitor object `visit` only. The Graph type and visitor type are templatized. This version requires the usage of interior color_decorator.

In the second version, the arguements are the all three ones in the first version plus a templatized decorator object `color` to access the color properity of vertices. The version are able to use exterior decorator or interior decorator for color properity. With the interior decorator, the same requirement is applied.

In the third version, the arguments are the all four ones in the second version plus a templatized Queue type object `Q` to provide extra flexibility.

This is the third version.

Definition

bfs.h

Example

In bfs_3.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed > Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(1, 4);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(2, 4);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

std::vector<default_color_type> color(G.num_vertices());
std::vector<Vertex>          p(G.num_vertices());
std::vector<Graph::size_type>  d(G.num_vertices());

Vertex s = *vertices(G).begin();
std::queue<Vertex> Q;
bfs(G, s, Q, visit_distance(mapfun(d),
           visit_predecessor(mapfun(p))), mapfun(color));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT

Press, 1990, P. 470

G.1.5 bfs

Prototype

```
template <class Graph, class Visitor, class ColorDecorator>
void bfs(Graph& G, typename graph_traits<Graph>::vertex_type s,
Visitor visit, ColorDecorator color) ;
```

Description

Three versions of overloaded BFS algorithms are provided in GGCL.

In the first version, the arguments are Graph `G`, its starting vertex `s` and a visitor object `visit` only. The Graph type and visitor type are templatized. This version requires the usage of interior color_decorator.

In the second version, the arguements are the all three ones in the first version plus a templatized decorator object `color` to access the color properity of vertices. The version are able to use exterior decorator or interior decorator for color properity. With the interior decorator, the same requirement is applied.

In the third version, the arguments are the all four ones in the second version plus a templatized Queue type object `Q` to provide extra flexibility.

This is the second version.

Definition

bfs.h

Complexity

linear

Example

In bfs_2.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed > Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(1, 4);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(2, 4);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

std::vector<default_color_type> color(G.num_vertices());
std::vector<Vertex> p(G.num_vertices());
std::vector<Graph::size_type> d(G.num_vertices());

Vertex s = *vertices(G).begin();

bfs(G, s, visit_distance(mapfun(d),
        visit_predecessor(mapfun(p))), mapfun(p));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT

Press, 1990, P. 470

G.1.6 bfs

Prototype

```
template <class Graph, class Visitor>
void bfs(Graph& G, typename Graph::vertex_type s, Visitor visit)
;
```

Description

Three versions of overloaded BFS algorithms are provided in GGCL.

In the first version, the arguments are Graph `G`, its starting vertex `s` and a visitor object `visit` only. The Graph type and visitor type are templatized. This version requires the usage of interior color_decorator.

In the second version, the arguements are the all three ones in the first version plus a templatized decorator object `color` to access the color properity of vertices. The version are able to use exterior decorator or interior decorator for color properity. With the interior decorator, the same requirement is applied.

In the third version, the arguments are the all four ones in the second version plus a templatized Queue type object `Q` to provide extra flexibility.

This is the first version.

Definition

bfs.h

Preconditions

- G has to have a color_plugin as a part of StoredVertexPlugin at least so that it is valid to use the interior color_decorator. See the example below.

Complexity

linear

Example

In bfs_1.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed, color_plugin<> >  Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(1, 4);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(2, 4);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

/* Array to store predecessor (parent) of each vertex */
std::vector<Vertex> p(G.num_vertices());

/* Array to store distances from the source to each vertex */
std::vector<Graph::size_type> d(G.num_vertices());

/* The source vertex */
Vertex s = *vertices(G).begin();

bfs(G, s, visit_distance(mapfun(d),
                    visit_predecessor(mapfun(p))));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 470

G.1.7 connected_components

Prototype

```
template < class Graph, class Visitor, class
ComponentsDecorator, class ColorDecorator >
decorator_traits<ComponentsDecorator>::value_type connected_-
components(Graph& G, Visitor v, ComponentsDecorator c,
ColorDecorator color) ;
```

Description

Using DFS to construct the algorithm. If the `G` is an directed graph, the algorithm computes the strongly connected components of the graph assuming interior decorators `DiscoverTimeDecorator` and `FinishTimeDecorator` defined. Otherwise, the `G` is undirected graph, the algorithm compute the connected components for undirected graphs.

Definition

connected_components.h

G.1.8 connected_components

Prototype

```
template < class Graph, class Visitor, class ComponentsDecorator
>
decorator_traits<ComponentsDecorator>::value_type connected_-
components(Graph& G, Visitor v, ComponentsDecorator c) ;
```

Description

Using DFS to construct the algorithm. Assuming an interior decorator of color_decorator is able to be used. If the `G` is an directed graph, the algorithm computes the strongly connected components assuming interior decorators of `DiscoverTimeDecorator` and `FinishTimeDecorator`. otherwise, the `G` is undirected graph and the algorithm compute the connected components for undirected graphs.

Definition

connected_components.h

Example

In connected_components.cc:

```
using namespace ggcl;
typedef discover_time_plugin< finish_time_plugin
                    < color_plugin<> > > VertexPlugin;
typedef graph < adjacency_list < ggcl_vecT >,
             directed, VertexPlugin >  Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

std::vector<int> c(G.num_vertices());
int num = connected_components(G, null_visitor(), mapfun(c));
```

G.1.9 _generalized_DFS

Prototype

```
template <class Vertex, class QType, class Visitor, class
VisitedFunc>
void _generalized_DFS(Vertex u, QType& Q, Visitor& visitor,
VisitedFunc visited) ;
```

Description

A generalized DFS algorithm with all argument types templatized. The initialization step is **not** included. If users want it, users can call `_generalized_init` first then call this function. We separate the initializtion step from main algorithms in case users want to

call main algorithms multiple times. With the `stack` as the Qtype here, the algorithm performs the normal DFS.

Definition

dfs.h

See also

_generalized_init

G.1.10 dfs

Prototype

```
template <class Graph, class Visitor, class Color>
void dfs(Graph& G, Visitor visitor_, Color c) ;
```

Description

This is non-recursive version of DFS. It is implemented by _generalized_DFS with a `stack` as `Qtype`. Notice that the order of the discovering vertices in adjacenct vertices of one vertex is reverse comparing to the recursive version. For example: **v0** have its adjacent vertices **v1** and **v2**. **v1** has not adjacent vertices while **v2** has an adjacent vertex **v3**. The recursive version of DFS will have **v0 v1 v2 v3** as a sequence of discovering vertices. However, the non-recursive version will have **v0 v2 v3 v1** instead. We choose the nonrecursive version because it runs fast.

Two versions of overloaded DFS algorithm are provided.

In the first version, the arguments are Graph `G` and a visitor object `visit` only. The Graph type and visitor type are templatized. This version requires the usage of interior color_decorator.

In the second version, the arguements are the all three ones in the first version plus a templatized decorator object `color` to access the color properity of vertices. The version

are able to use exterior decorator or interior decorator for color properity. With the interior decorator, the same requirement is applied.

This is the second version.

Definition

dfs.h

Example

In dfs_2.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed > Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

std::vector<Graph::size_type> dt(G.num_vertices());
std::vector<Graph::size_type> ft(G.num_vertices());
std::vector<default_color_type> color(G.num_vertices());

dfs(G, visit_timestamp(mapfun(dt), mapfun(ft)),
   mapfun(color));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 478

G.1.11 dfs

Prototype

```
template <class Graph, class Visitor>
void dfs(Graph& G, Visitor visitor) ;
```

Description

In the first version, the arguments are Graph `G` and a visitor object `visit` only. The Graph type and visitor type are templatized. This version requires the usage of interior color_decorator.

In the second version, the arguements are the all three ones in the first version plus a templatized decorator object `color` to access the color properity of vertices. The version are able to use exterior decorator or interior decorator for color properity. With the interior decorator, the same requirement is applied.

This is the first version.

Definition

dfs.h

Preconditions

- G has to have a color_plugin as a part of StoredVertexPlugin at least so that it is valid to use the interior color_decorator. See the example below.

Example

In dfs_1.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT >,
            directed, color_plugin<> >  Graph;

Graph G(5);
G.add_edge(0, 2);
G.add_edge(1, 1);
G.add_edge(1, 3);
G.add_edge(2, 1);
G.add_edge(2, 3);
G.add_edge(3, 1);
G.add_edge(3, 4);
G.add_edge(4, 0);
G.add_edge(4, 1);

typedef Graph::vertex_type Vertex;

std::vector<Graph::size_type> dt(G.num_vertices());
std::vector<Graph::size_type> ft(G.num_vertices());

dfs(G, visit_timestamp(mapfun(dt), mapfun(ft)));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT

Press, 1990, P. 478

G.1.12 dijkstra

Prototype

```
template <class Graph, class Visitor, class Distance, class
Weight, class ID >
void dijkstra(Graph& G, typename graph_traits<Graph>::vertex_type
s, Visitor visit, Distance d, Weight w, const ID& id ) ;
```

Description

Dijkstra's Algorithm solves the single-source shortest-paths problem on a weighted, directed graph G for the case in which all edge weights are nonnegative. This algorithm does not check whether edge weights are nonnegative or not.

Three overloaded versions of the algoithm are provided.

The first version has four arguements with templatized types:: the Graph `G`, source vertex `s`, a visitor object `visit` and a Distance Decorator obejct `d`. It requires id_decorator and weight_decorator to work.

The second version has five arguments: all the four arguments in the first version plus a templatized weight decorator object `w`.

The third version has the size arguments: all the five arguments in the second version plus a templatized ID object `id`.

This is the third version.

Definition

dijkstra.h

G.1.13 dijkstra

Prototype

```
template <class Graph, class Visitor, class Distance, class
Weight >
void dijkstra(Graph& G, typename graph_traits<Graph>::vertex_type
s, Visitor visit, Distance d, Weight w ) ;
```

Description

Dijkstra's Algorithm solves the single-source shortest-paths problem on a weighted, directed graph G for the case in which all edge weights are nonnegative. This algorithm does not check whether edge weights are nonnegative or not.

Three overloaded versions of the algoithm are provided.

The first version has four arguements with templatized types:: the Graph `G`, source vertex `s`, a visitor object `visit` and a Distance Decorator obejct `d`. It requires id_decorator and weight_decorator to work.

The second version has five arguments: all the four arguments in the first version plus a templatized weight decorator object `w`.

The third version has the size arguments: all the five arguments in the second version plus a templatized ID object `id`.

This is the second version.

Definition

dijkstra.h

G.1.14 dijkstra

Prototype

```
template <class Graph, class Visitor, class Distance>
void dijkstra(Graph& G, typename graph_traits<Graph>::vertex_type
s, Visitor visit, Distance d) ;
```

Description

Dijkstra's Algorithm solves the single-source shortest-paths problem on a weighted, directed graph G for the case in which all edge weights are nonnegative. This algorithm does not check whether edge weights are nonnegative or not.

Three overloaded versions of the algoithm are provided.

The first version has four arguements with templatized types:: the Graph `G`, source vertex `s`, a visitor object `visit` and a Distance Decorator obejct `d`. It requires id_decorator and weight_decorator to work.

The second version has five arguments: all the four arguments in the first version plus a templatized weight decorator object `w`.

The third version has the size arguments: all the five arguments in the second version plus a templatized ID object `id`.

This is the first version.

Definition

dijkstra.h

Example

In dijkstra.cc:

```
using namespace ggcl;
typedef graph < adjacency_matrix < ggcl_vecT >, undirected,
  off_vertex_default_plugin<>, Weight<int> > Graph;
typedef Graph::vertex_type Vertex;
typedef Graph::edgeplugin_type::weight_type weight_type;

Graph G(5);

G.add_edge(0, 2, Weight<weight_type>(1));
G.add_edge(1, 1, Weight<weight_type>(2));
G.add_edge(1, 3, Weight<weight_type>(1));
G.add_edge(1, 4, Weight<weight_type>(2));
G.add_edge(2, 1, Weight<weight_type>(7));
G.add_edge(2, 3, Weight<weight_type>(3));
G.add_edge(3, 4, Weight<weight_type>(1));
G.add_edge(4, 0, Weight<weight_type>(1));
G.add_edge(4, 1, Weight<weight_type>(1));

std::vector<Vertex> p(G.num_vertices());
std::vector<Graph::size_type> d(G.num_vertices());

Vertex s = *(vertices(G).begin());

dijkstra(G, s, visit_predecessor(mapfun(p)), mapfun(d) );
```

G.1.15 kruskal

Prototype

```
template < class Graph, class OutputIterator, class Rank, class
Parent, class Weight >
void kruskal(Graph& G, OutputIterator c, Rank rank, Parent p,
Weight w) ;
```

Description

This is a greedy algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. The output will be a set of edges.

Two overloaded version of Kruskal's algorithm are provided.

The first version has four templatized arguments: Graph `G`, OutputIterator `c`, a rank decorator object `rank` and a parent decorator object `parent`. The version requires to use weight_decorator.

The second version has one more additional argument which is a templatized type weight decorator.

This is the second version.

Definition

kruskal.h

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.16 kruskal

Prototype

```
template < class Graph, class OutputIterator, class Rank, class
Parent >
void kruskal(Graph& G, OutputIterator c, Rank rank, Parent p ) ;
```

Description

This is a greedy algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. The output will be a set of edges.

Two overloaded version of Kruskal's algorithm are provided.

The first version has four templatized arguments: Graph `G`, OutputIterator `c`, a rank decorator object `rank` and a parent decorator object `parent`. The version requires to use weight_decorator.

The second version has one more additional argument which is a templatized type weight decorator.

This is the first version.

Definition

kruskal.h

Example

In kruskal.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT, unordered >,
  undirected, off_vertex_default_plugin<>, Weight<int> >
  Graph;
typedef Graph::edgeplugin_type::weight_type weight_type;

Graph G(5);

G.add_edge(0, 2, Weight<weight_type>(1));
G.add_edge(1, 1, Weight<weight_type>(2));
G.add_edge(1, 3, Weight<weight_type>(1));
G.add_edge(1, 4, Weight<weight_type>(2));
G.add_edge(2, 1, Weight<weight_type>(7));
G.add_edge(2, 3, Weight<weight_type>(3));
G.add_edge(3, 4, Weight<weight_type>(1));
G.add_edge(4, 0, Weight<weight_type>(1));
G.add_edge(4, 1, Weight<weight_type>(1));

typedef Graph::edge_type Edge;
typedef Graph::vertex_type Vertex;

typedef std::vector<Edge> container;
std::vector<Edge> c;
c.reserve(G.num_vertices());
std::vector<Vertex> p;
std::vector<int> rank;

kruskal(G, std::back_insert_iterator<container>(c),
       mapfun(rank), mapfun(p));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.17 prim

Prototype

```
template <class Graph, class Visitor, class Distance, class
Weight, class ID>
void prim(Graph& G, typename graph_traits<Graph>::vertex_type s,
Visitor visit, Distance d, Weight w, ID id ) ;
```

Description

This is Prim's algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. There are four overloaded functions:

The first version has three arguments only with the templatized types: Graph `G`, a start vertex `s`, and a visitor object `visit` which could record information such as parent of every vertex in MST on return. The version requires to use interior distance_decorator, weight_decorator, id_decorator.

The second version has four arguments: the three ones in the first version with one additional templatized distance decorator `d` so that it is possible to use exterior decorator for diatance.

The third version has one more argument comparing to the second version. The additional one is a weight decorator object `w` with a templatized type.

The fourth version has one more argument comparing to the third version. The ID decorator is templatized and need to supply from users.

Output: visit records the information in MST

Input: undirected Graph `G`, starting vertex `s`, `visit` Vistor, and ID decorator `id`.

This is the fourth version.

Definition

prim.h

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.18 prim

Prototype

```
template <class Graph, class Visitor, class Distance, class
Weight>
void prim(Graph& G, typename graph_traits<Graph>::vertex_type s,
Visitor visit, Distance d, Weight w) ;
```

Description

This is Prim's algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. There are four overloaded functions:

The first version has three arguments only with the templatized types: Graph `G`, a start vertex `s`, and a visitor object `visit` which could record information such as parent of every vertex in MST on return. The version requires to use interior distance_decorator, weight_decorator, id_decorator.

The second version has four arguments: the three ones in the first version with one additional templatized distance decorator `d` so that it is possible to use exterior decorator for diatance.

The third version has one more argument comparing to the second version. The additional one is a weight decorator object `w` with a templatized type.

The fourth version has one more argument comparing to the third version. The ID decorator is templatized and need to supply from users.

Output: visit records the information in MST

Input: undirected Graph `G`, starting vertex `s`, `visit` Vistor, and ID decorator `id`.

This is the third version.

Definition

prim.h

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.19 prim

Prototype

```
template <class Graph, class Visitor, class Distance>
void prim(Graph& G, typename graph_traits<Graph>::vertex_type s,
Visitor visit, Distance d) ;
```

Description

This is Prim's algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. There are four overloaded functions:

The first version has three arguments only with the templatized types: Graph `G`, a start vertex `s`, and a visitor object `visit` which could record information such as parent of every vertex in MST on return. The version requires to use interior distance_decorator, weight_decorator, id_decorator.

The second version has four arguments: the three ones in the first version with one additional templatized distance decorator `d` so that it is possible to use exterior decorator for diatance.

The third version has one more argument comparing to the second version. The additional one is a weight decorator object `w` with a templatized type.

The fourth version has one more argument comparing to the third version. The ID decorator is templatized and need to supply from users.

Output: visit records the information in MST

Input: undirected Graph `G`, starting vertex `s`, `visit` Vistor, and ID decorator `id`.

This is the second version.

Definition

prim.h

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.20 prim

Prototype

```
template <class Graph, class Visitor>
void prim(Graph& G, typename graph_traits<Graph>::vertex_type s,
Visitor visit) ;
```

Description

This is Prim's algorithm to calculate the Minimum Spanning Tree for an undirected graph with weighted edges. There are four overloaded functions:

The first version has three arguments only with the templatized types: Graph `G`, a start vertex `s`, and a visitor object `visit` which could record information such as parent of every vertex in MST on return. The version requires to use interior distance_decorator, weight_decorator, id_decorator.

The second version has four arguments: the three ones in the first version with one additional templatized distance decorator `d` so that it is possible to use exterior decorator for diatance.

The third version has one more argument comparing to the second version. The additional one is a weight decorator object `w` with a templatized type.

The fourth version has one more argument comparing to the third version. The ID decorator is templatized and need to supply from users.

Output: visit records the information in MST

Input: undirected Graph `G`, starting vertex `s`, `visit` Vistor, and ID decorator `id`.

This is the first version.

Definition

prim.h

Example

In prim.cc:

```
using namespace ggcl;
typedef graph < adjacency_list < ggcl_vecT, unordered >,
  undirected, distance_plugin<>, Weight<int> > Graph;
typedef Graph::edgeplugin_type::weight_type weight_type;

Graph G(5);

G.add_edge(0, 2, Weight<weight_type>(1));
G.add_edge(1, 1, Weight<weight_type>(2));
G.add_edge(1, 3, Weight<weight_type>(1));
G.add_edge(1, 4, Weight<weight_type>(2));
G.add_edge(2, 1, Weight<weight_type>(7));
G.add_edge(2, 3, Weight<weight_type>(3));
G.add_edge(3, 4, Weight<weight_type>(1));
G.add_edge(4, 0, Weight<weight_type>(1));
G.add_edge(4, 1, Weight<weight_type>(1));
std::vector<Graph::vertex_type> p(G.num_vertices());
prim(G, *(vertices(G).begin()), visit_predecessor(mapfun(p)));
```

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 505

G.1.21 recursive_DFS

Prototype

```
template <class Graph, class Visitor, class Color>
void recursive_DFS(Graph& G, Visitor v, Color c) ;
```

Description

This is the recursive version of DFS algorithm. We provide another version of DFS which is based on stack. We provide both algorithms because they are different algorithms even though they are for the same problem. If user only wants to calculate discovering and finsihing time, the stack version is faster than recursive version. However, the recursive version will be faster if user want tree edges only.

Definition

recursive_DFS.h

G.1.22 DFS_visit

Prototype

```
template < class Vertex, class Visitor >
void DFS_visit(Vertex u, Visitor& v) ;
```

Description

The main component for recursive version of DFS and other DFS related algorithms patterns.

Definition

recursive_DFS.h

See also

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990, P. 478

G.1.23 topological_sort

Prototype

```
template < class Graph, class OutputIterator, class Visitor,
class Color >
void topological_sort(Graph& G, OutputIterator iter, Visitor
myvisit, Color color) ;
```

Description

Applying dfs to perform topological sorts of directed acyclic graphs(DAG). The algorithm does not check whether the input graph is a DAG. There are three overloaded functions:

The first version has two arguments only: Graph `G` and an OutputIterator `iter` to hold the vertcies in topological order. The version requires to use interior color_decorator.

The second version has one more argument: a visitor object `myvisit` to provide ability to compute more information on finding topological order. The version requires to use interior color_decorator.

The third version has one more argument comparing to the second version. The templatized color decorator makes it possible to use exterior color decorator if need.

This is the third version.

Definition

topological_sort.h

Example

In topo_sort_2.cc:

```
using namespace ggcl;
typedef discover_time_plugin< finish_time_plugin<
      distance_plugin<> > > VertexPlugin;

typedef graph < adjacency_list < ggcl_vecT, unordered >,
  directed, VertexPlugin, Weight<int> > Graph;

typedef Graph::vertex_type Vertex;

std::pair<size_t,size_t> edges[7] = { Pair(0,1),
                                Pair(2,4), Pair(2,5),
                                Pair(0,3), Pair(1,4),
                                Pair(4,3), Pair(5,5) };
Graph G(edges, 7, 6);
typedef std::vector< Vertex > container;
container c;
null_visitor null_v;
std::vector<default_color_type> color(G.num_vertices());
topological_sort(G, std::back_insert_iterator<container>(c),
             null_v, mapfun(color));
```

G.1.24 topological_sort

Prototype

```
template <class Graph, class OutputIterator, class Visitor>
void topological_sort(Graph& G, OutputIterator iter, Visitor
myvisit) ;
```

Description

Applying dfs to perform topological sorts of directed acyclic graphs(DAG). The algorithm does not check whether the input graph is a DAG. There are three overloaded functions:

The first version has two arguments only: Graph `G` and an OutputIterator `iter` to hold the vertcies in topological order. The version requires to use interior color_decorator.

The second version has one more argument: a visitor object `myvisit` to provide ability to compute more information on finding topological order. The version requires to use interior color_decorator.

The third version has one more argument comparing to the second version. The templatized color decorator makes it possible to use exterior color decorator if need.

This is the second version.

Definition

topological_sort.h

Example

In topo_sort.cc:

```
using namespace ggcl;
typedef discover_time_plugin< finish_time_plugin<
      color_plugin<distance_plugin<> > > > VertexPlugin;

typedef graph < adjacency_matrix < ggcl_vecT >, directed,
  VertexPlugin, Weight<int> > Graph;

typedef Graph::vertex_type Vertex;

std::pair<size_t,size_t> edges[7] = { Pair(0,1), Pair(2,4),
                                Pair(2,5),
                                Pair(0,3), Pair(1,4),
                                Pair(4,3), Pair(5,5) };
Graph G(edges, 7, 6);
typedef std::vector< Vertex > container;
container c;
null_visitor null_v;
typedef std::back_insert_iterator<container> OutputIterator;
topological_sort(G, OutputIterator(c), null_v);
```

G.1.25 topological_sort

Prototype

```
template <class Graph, class OutputIterator>
void topological_sort(Graph& G, OutputIterator iter) ;
```

Description

Applying dfs to perform topological sorts of directed acyclic graphs(DAG). The algorithm does not check whether the input graph is a DAG. There are three overloaded functions:

The first version has two arguments only: Graph `G` and an OutputIterator `iter` to hold the vertcies in topological order. The version requires to use interior color_decorator.

The second version has one more argument: a visitor object `myvisit` to provide ability to compute more information on finding topological order. The version requires to use interior color_decorator.

The third version has one more argument comparing to the second version. The templatized color decorator makes it possible to use exterior color decorator if need.

This is the first version.

Definition

topological_sort.h

Example

In topo_sort.cc:

```
using namespace ggcl;
typedef discover_time_plugin< finish_time_plugin<
      color_plugin<distance_plugin<> > > > VertexPlugin;

typedef graph < adjacency_matrix < ggcl_vecT >, directed,
  VertexPlugin, Weight<int> > Graph;

typedef Graph::vertex_type Vertex;

std::pair<size_t,size_t> edges[7] = { Pair(0,1), Pair(2,4),
                                Pair(2,5),
                                Pair(0,3), Pair(1,4),
                                Pair(4,3), Pair(5,5) };
Graph G(edges, 7, 6);
typedef std::vector< Vertex > container;
container c;
null_visitor null_v;
typedef std::back_insert_iterator<container> OutputIterator;
topological_sort(G, OutputIterator(c), null_v);
```

G.1.26 transpose

Prototype

```
template <class Graph1, class Graph2>
void transpose(const Graph1& G1, Graph2& G2) ;
```

Description

To get the transpose of a directed graph. The transpose of a directed graph G = (V, E) is the graph GT = (V, ET), where ET = (v, u) in V x V: (u, v) in E. i.e., GT is G with all its edges reversed.

Definition

transpose.h

APPENDIX H

PLUGINS

H.1 Concepts

H.1.1 interior_vertex_plugin

Description

The interior vertex plugin is to provide the a way to store the vertex properityes. There are two categories of it. One is off_vertex_plugin which stored vertex properities inside the graph object but out of the vertex. The off_vertex_plugin is meanful to adjacency_list and adjacency_matrix graphs. On the other hand, if the graph is dynamic representation, it is better that vertex properities are stored inside the vertex and on_vertex_plugin provides that.

Table H.1: Expression semantics of concept interior_vertex_plugin

Expression	Description
`x.NAME(u)`	To access the properity of vertex whose ID is u.

Table H.2: Function specification of concept interior_vertex_plugin

Prototype	Description
`NAME_type& NAME(size_type u)`	To access the properity of vertex whose ID is u.

Models

- off_vertex_plugin
- on_vertex_plugin

H.1.2 off_vertex_plugin

Description

The off-vertex plugin is to provide off-vertex storage for vertex properities inside a graph object. Using interior decorators to access the vertex properities. The four functions defined below will be invoked automatically when a graph add/remove an edge/vertex.

Refinement of

interior_vertex_plugin

Notations

- X A type that is a model of off_vertex_plugin
- x An object of type X
- u An object of a model of Vertex

Table H.3: Expression semantics of concept off_vertex_plugin

Expression	Description
`x.NAME(u)`	To access the properity of vertex whose ID is u.
`x.add_vertex(u)`	invoked when a graph object add a vertex.
`x.remove_vertex(u)`	invoked when a graph object remove a vertex.

Table H.4: Function specification of concept off_vertex_plugin

Prototype	Description
`NAME_type& NAME(size_type u)`	To access the properity of vertex whose ID is u.
`void add_vertex(size_type u)`	invoked when a graph object add a vertex.
`void remove_vertex(size_type u)`	invoked when a graph object remove a vertex.

Models

- distance_plugin
- color_plugin
- out_degree_plugin
- in_degree_plugin
- degree_plugin

- discover_time_plugin
- finish_time_plugin

Notes

it is recommended to inherit from default_plugin on implementing a user-defined off_-vertex plugin.

H.1.3 on_vertex_plugin

Description

The on-vertex plugin is to provide the storage for vertex properities on the vertex.

Refinement of

interior_vertex_plugin

Table H.5: Expression semantics of concept on_vertex_plugin

Expression	Description
`x.NAME(u)`	To access the properity of vertex whose ID is u.

Table H.6: Function specification of concept on_vertex_plugin

Prototype	Description
`NAME_type& NAME(size_type u)`	To access the properity of vertex whose ID is u.

Models

- on_vertex_color_plugin
- id_plugin
- on_vertex_distance_plugin

H.2 Plugin classes

H.2.1 color_plugin

Description

This is the off-vertex plugin to provide the storage of vertex color properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use color_decorator to access the distance properity.

```
typedef color_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            irected, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
color_decorator < Vertex > color;
color[s] = color_traits< color_type >::black();
```

Definition

vertex_plugin.h

Table H.7: Template parameters of class `color_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_-default_plugin<>`	a super off vertex plugin

Parameter	Default	Description
`Container`	`std::vector<typename StoragePlugin::size_type>`	a container type to hold all vertices' colors

Table H.8: Members of `color_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_-plugin
`color_type`		off_vertex_-plugin
`color_plugin (size_type n)`		
`color_type& color (size_type u)`		off_vertex_-plugin
`const color_type& color (size_type u) const`		
`void add_vertex (size_type u)`		off_vertex_-plugin
`void remove_vertex (size_type u)`		off_vertex_-plugin

H.2.2 degree_plugin

Description

This is the off-vertex plugin to provide the storage of vertex degree properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use degree_-decorator to access the distance properity.

```
typedef degree_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            undirected, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
degree_decorator < Vertex > deg;
cout << deg[s] << endl;
```

Definition

vertex_plugin.h

Table H.9: Template parameters of class `degree_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_default_plugin<>`	a super off vertex plugin
`Container`	`std::vector<typename StoragePlugin::size_type>`	a container type to hold all vertices'-degrees

Table H.10: Members of `degree_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_plugin
`degree_type`		off_vertex_plugin
`degree_plugin ()`		

Declaration	Description	Where Defined
`degree_plugin (size_type n)`		
`const degree_type& degree (size_type u) const`		
`degree_type& degree (size_type u)`		off_vertex_plugin
`void add_edge (size_type u, size_type v)`		off_vertex_plugin
`void remove_edge (size_type u, size_type v)`		off_vertex_plugin
`void add_vertex (size_type u)`		off_vertex_plugin
`void remove_vertex (size_type u)`		off_vertex_plugin

H.2.3 discover_time_plugin

Description

This is the off-vertex plugin to provide the storage of vertex discover-time properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use discover_time_decorator to access the distance properity.

```
typedef discover_time_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            undirected, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
discover_time_decorator < Vertex > dt;
cout << dt[s] << endl;
```

Definition

vertex_plugin.h

Table H.11: Template parameters of class `discover_time_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_default_plugin<>`	a super off vertex plugin
`Container`	`std::vector<typename StoragePlugin::size_type>`	a container type to hold all vertices' discover time

Table H.12: Members of `discover_time_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_plugin
`discover_time_type`		off_vertex_plugin
`discover_time_plugin ()`		
`discover_time_plugin (size_type n)`		
`const discover_time_type& discover_time (size_type u) const`		
`discover_time_type& discover_time (size_type u)`		off_vertex_plugin
`void add_vertex (size_type u)`		off_vertex_plugin

Declaration	Description	Where Defined
`void remove_vertex (size_type u)`		off_vertex_-plugin

H.2.4 distance_plugin

Description

This is the off-vertex distance plugin to provide the storage of vertex distance properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use distance_decorator to access the distance properity.

```
typedef distance_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            directed, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
distance_decorator < Vertex > d;
d[s] = 0;
```

Definition

vertex_plugin.h

Table H.13: Template parameters of class `distance_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_-default_plugin<>`	a super off vertex plugin
`Container`	`std::vector<typename StoragePlugin::size_type>`	a container type to hold all vertices' distances

Table H.14: Members of `distance_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_-plugin
`distance_type`		off_vertex_-plugin
`distance_plugin (size_type n)`		
`distance_type& distance (size_type u)`		off_vertex_-plugin
`const distance_type& distance (size_type u) const`		
`void add_vertex (size_type u)`		off_vertex_-plugin
`void remove_vertex (size_type u)`		off_vertex_-plugin

H.2.5 finish_time_plugin

Description

This is the off-vertex plugin to provide the storage of vertex discover-time properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use discover_-time_decorator to access the distance properity.

```
typedef finish_time_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            undirected, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
finish_time_decorator < Vertex > ft;
cout << ft[s] << endl;
```

Definition

vertex_plugin.h

Table H.15: Template parameters of class `finish_time_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_-default_plugin<>`	a super off vertex plugin
`Container`		a conatiner type to hold all vertices' finish-time - std::vector<typename Storage-Plugin::size_type>

Table H.16: Members of `finish_time_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_-plugin
`finish_time_type`		off_vertex_-plugin

Declaration	Description	Where Defined
`finish_time_plugin ()`		
`finish_time_plugin (size_type n)`		
`const finish_time_type& finish_time (size_type u) const`		
`finish_time_type& finish_time (size_type u)`		off_vertex_plugin
`void add_vertex (size_type u)`		off_vertex_plugin
`void remove_vertex (size_type u)`		off_vertex_plugin

H.2.6 id_plugin

Description

This is to provide an on vertex plugin so that every vertex can have an ID. Use this for dyanmic graph only.

```
typedef on_vertex_color_plugin< id_plugin<> > VertexPlugin;
typedef graph < dynamic < vecT, unordered >,
            directed, VertexPlugin > Graph;
```

Definition

vertex_plugin.h

Table H.17: Members of `id_plugin`

Declaration	Description	Where Defined
`id_type`		
`id_type& id ()`		

Declaration	Description	Where Defined
`id_type& id () const`		
`id_type _id`		

H.2.7 in_degree_plugin

Description

This is the off-vertex plugin to provide the storage of vertex in degree properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use in_degree_-decorator to access the distance properity.

```
typedef in_degree_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            directed, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
in_degree_decorator < Vertex > in_d;
cout << in_d[s] << endl;
```

Definition

vertex_plugin.h

Table H.18: Template parameters of class `in_degree_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_-default_plugin<>`	a super off vertex plugin

Parameter	Default	Description
`Container`		a conatiner type to hold all vertices' in-degrees - std::vector<typename Storage-Plugin::size_type>

Table H.19: Members of `in_degree_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_-plugin
`in_degree_type`		off_vertex_-plugin
`in_degree_plugin (size_type n)`		
`const in_degree_type& in_degree (size_type u) const`		
`in_degree_type& in_degree (size_-type u)`		off_vertex_-plugin
`void add_edge (size_type u, size_-type v)`		off_vertex_-plugin
`void remove_edge (size_type u, size_type v)`		off_vertex_-plugin
`void add_vertex (size_type u)`		off_vertex_-plugin
`void remove_vertex (size_type u)`		off_vertex_-plugin

H.2.8 off_vertex_default_plugin

Description

This class is the default stored vertex plugin and will be the base of other off_vertex plugin.

Definition

vertex_plugin.h

Table H.20: Members of `off_vertex_default_plugin`

Declaration	Description	Where Defined
`size_type`		
`off_vertex_default_plugin ()`		
`off_vertex_default_plugin (size_type n)`		
`void add_edge (size_type u, size_type v)`		off_vertex_plugin
`void remove_edge (size_type u, size_type v)`		off_vertex_plugin
`void add_vertex (size_type u)`		off_vertex_plugin
`void remove_vertex (size_type u)`		off_vertex_plugin

H.2.9 on_vertex_color_plugin

Description

This is to provide an on vertex plugin so that every vertex can have a color properity. Use this for dyanmic graph only. Interior decorator `color_decorator` is used to access this properity.

```
typedef on_vertex_color_plugin < > VertexPlugin;
typedef graph < dynamic <>,
            undirected, VertexPlugin > Graph;
//. . .
typedef Graph::vertex_type Vertex;
Vertex u = G.root();
color_decorator< Vertex > color;
color[u] = color_traits< color_type >::grey();
```

Definition

vertex_plugin.h

Table H.21: Template parameters of class `on_vertex_color_plugin`

Parameter	Default	Description
`Super`	`id_plugin<>`	a super on vertex plugin

Table H.22: Members of `on_vertex_color_plugin`

Declaration	Description	Where Defined
`color_type`		on_vertex_-plugin
`on_vertex_color_plugin ()`		

Declaration	Description	Where Defined
`default_color_type& color (typename Super::id_type i = 0)`		on_vertex_-plugin
`default_color_type& color (typename Super::id_type i = 0) const`		
`default_color_type _color`		

H.2.10 on_vertex_distance_plugin

Description

This is to provide an on vertex plugin so that every vertex can have a distance properity. Use this for dyanmic graph only. Interior decorator `distance_decorator` is used to access this properity.

```
typedef on_vertex_distance_plugin < > VertexPlugin;
typedef graph < dynamic <>,
            directed, VertexPlugin > Graph;
//. . .
typedef Graph::vertex_type Vertex;
Vertex u = G.root();
distance_decorator< Vertex > d;
d[u] = 0;
```

Definition

vertex_plugin.h

Table H.23: Template parameters of class `on_vertex_distance_plugin`

Parameter	Default	Description
`Super`	`id_plugin<>`	a super on vertex plugin

Table H.24: Members of on_vertex_distance_plugin

Declaration	Description	Where Defined
distance_type		on_vertex_-plugin
on_vertex_distance_plugin ()		
distance_type& distance (typename Super::id_type i = 0)		on_vertex_-plugin
distance_type& distance (typename Super::id_type i = 0) const		
distance_type _distance		

H.2.11 out_degree_plugin

Description

This is the off-vertex plugin to provide the storage of vertex out degree properity inside a graph object. Use this for adjacency_list and adjacency_matrix graphs. Use out_degree_-decorator to access the distance properity.

```
typedef out_degree_plugin<> VerexPlugin;
typedef graph < adjacency_list<>,
            directed, VerexPlugin > Graph;
//...
typedef Graph::vertex_type Vertex;
Vertex s = *(vertices(G).begin());
out_degree_decorator < Vertex > out_d;
cout << out_d[s] << endl;
```

Definition

vertex_plugin.h

Table H.25: Template parameters of class `out_degree_plugin`

Parameter	Default	Description
`StoragePlugin`	`off_vertex_default_plugin<>`	a super off vertex plugin
`Container`		a conatiner type to hold all vertices' out-degrees - std::vector<typename Storage-Plugin::size_type>

Table H.26: Members of `out_degree_plugin`

Declaration	Description	Where Defined
`size_type`		off_vertex_plugin
`out_degree_type`		off_vertex_plugin
`out_degree_plugin (size_type n)`		
`const out_degree_type& out_degree (size_type u) const`		
`out_degree_type& out_degree (size_type u)`		off_vertex_plugin
`void add_edge (size_type u, size_type v)`		off_vertex_plugin
`void remove_edge (size_type u, size_type v)`		off_vertex_plugin

Declaration	Description	Where Defined
`void add_vertex (size_type u)`		off_vertex_-plugin
`void remove_vertex (size_type u)`		off_vertex_-plugin

H.2.12 weight_plugin

Description

This is to provide the storage for weight information for every edge. It is stored on the edge as a plugin. Use weight_decorator to access weight for an edge.

```
typedef graph < adjacency_list<>, undirected,
            interior_default_plugin<>, Weight<int> > Graph;
//...
typedef Graph::edge_type Edge;
// ... e is a valid edge from graph G
weight_decorator < Edge > w;
cout << w[e] << endl;
```

Definition

edge_plugin.h

Table H.27: Members of `weight_plugin`

Declaration	Description	Where Defined
`weight_type`		
`weight_plugin ()`		
`weight_plugin (T t, const Plugin& p = Plugin() )`		
`weight_plugin (const weight_-plugin& W)`		

Declaration	Description	Where Defined
`const weight_type& weight () const`		
`weight_type& weight ()`		

APPENDIX I

FUNCTION OBJECTS

I.1 classes

I.1.1 first_equal

Description

This is a function object. It tests the truth or falsehood of two objects whose types are std::pair. If `f` is an object of `first_equal` and `x` and `y` are two pair objects Then `f(x, y)` returns true if `x.first == y.first` and false otherwise.

Definition

functor.h

Table I.1: Members of `first_equal`

Declaration	Description	Where Defined
`first_argument_type`		
`second_argument_type`		
`result_type`		
`bool operator() (const PairType& a, const PairType& b) const`		

I.1.2 first_less

Description

This is a function object. It tests the truth or falsehood of two objects whose types are std::pair. If `f` is an object of `first_less` and `x` and `y` are two pair objects Then `f(x, y)` returns true if `x.first < y.first` and false otherwise.

Definition

functor.h

Table I.2: Members of `first_less`

Declaration	Description	Where Defined
`template < class PairType > bool operator() (const PairType& a, const PairType& b) const`		

I.1.3 functor_equal

Description

This is a function object. It tests the truth or falsehood of two objects' decorating properties through decorator. If `f` is an object of `functor_equal` < D > and `x` and `y` are two objects with decorator `d` which is an object of `D`, then `f(x, y)` returns true if `d[x] == d[y]` and false otherwise.

Table I.3: Template parameters of class `functor_equal`

Parameter	Default	Description
`Decorator`		a model of Decorator

Table I.4: Members of `functor_equal`

Declaration	Description	Where Defined
`functor_equal (const Decorator& df = Decorator())`		
`template <class Vertex> bool operator() (const Vertex& u, const Vertex& v) const`	return `d[u] == d[v]`	

I.1.4 functor_greater

Description

This is a function object. It tests the truth or falsehood of two objects' decorating properties through decorator. If `f` is an object of `functor_greator` < D > and `x` and `y` are two objects with decorator `d`, which is an object of `D`, then `f(x, y)` returns true if `d[x]` > `d[y]` and false otherwise.

Definition

functor.h

Table I.5: Template parameters of class `functor_greater`

Parameter	Default	Description
`Decorator`		a model of Decorator

Table I.6: Members of `functor_greater`

Declaration	Description	Where Defined
`functor_greater (const Decorator& df = Decorator())`		
`template <class Vertex> bool operator() (const Vertex& u, const Vertex& v) const`	return `d[u] > d[v]`	

I.1.5 functor_less

Description

This is a function object. It tests the truth or falsehood of two objects' decorating properties through decorator. If `f` is an object of `functor_less` < D > and `x` and `y` are two objects with decorator `d` which is an object of `D`, then `f(x, y)` returns true if `d[x]` < `d[y]` and false otherwise.

Definition

functor.h

Table I.7: Template parameters of class `functor_less`

Parameter	Default	Description
`Decorator`		a model of Decorator

Table I.8: Members of `functor_less`

Declaration	Description	Where Defined
`functor_less (const Decorator& df = Decorator())`		
`template <class Vertex> bool operator() (const Vertex& u, const Vertex& v) const`	return `d[u]` < `d[v]`	

I.1.6 null_operation

Description

This functor provides three overloaded versions of `operator()` function with one argument, two arguments, and three arguments. The function body is empty to provide the null operation.

Definition

visited.h

Table I.9: Members of `null_operation`

Declaration	Description	Where Defined
`template < class A, class B, class C > void operator() ( const A& a, const B& b, const C& c ) const`		
`template < class A, class B > void operator() ( const A& a, const B& b ) const`		
`template < class A> void operator() ( const A& a ) const`		

I.1.7 queue_update

Description

This functor is used for dijkstra amd prim. It provides a operation to check if it needs perform queue update. if so, do it.

Definition

visited.h

Type requirements

- Visitor must be a model of Visitor to provide the method `need_update_queue()`.
- Qtype must be a model of queue who has method of `update(v)` where `v` has a type of `Qtype::value_type`.

Table I.10: Members of `queue_update`

Declaration	Description	Where Defined
`template < class Visitor, class Qtype, class Iter > void operator() ( Visitor& visitor, Qtype& Q, Iter ei ) const`		

APPENDIX J

MATRIX ORDERING

J.1 Utility classes

J.1.1 Marker

Description

This class is to provide a generalization of coloring which has complexity of amortized constant time to set all vertices' color back to be white. It implemented by simply increasing a tag.

Definition

mmd_aux.h

Table J.1: Members of `Marker`

Declaration	Description	Where Defined
`Marker (Decorator _data, size_type _num)`		
`void init (size_type n)`		
`void mark_done (size_type node)`		
`bool is_done (size_type node)`		
`void mark_tag (size_type node)`		
`void mark_mtag (size_type node)`		
`bool is_tagged (size_type node) const`		

Declaration	Description	Where Defined
`bool is_not_tagged (size_type node) const`		
`bool is_mtagged (size_type node) const`		
`void increment_tag ()`		
`void set_mtag (value_type mdeg0)`		
`void set_tag_as_mtag ()`		
`void print (size_type n)`		

J.1.2 Stacks

Description

This to use a single array for multiple stacks. It was used in Fortran code orginally because of its efficiency.

Definition

mmd_aux.h

Table J.2: Members of `Stacks`

Declaration	Description	Where Defined
`Stacks (Decorator _data)`		
`class stack`	stack	
`stack operator[] (size_type i)`	To return a stack object with the head provided.	
`stack make_stack ()`	To return a stack object	

J.1.3 ordered_stacks

Description

This is a bucket sorter virtually. It is used inside of mmd algorithms.

Table J.3: Members of `ordered_stacks`

Declaration	Description	Where Defined
`ordered_stacks (Decorator _head, Decorator _next, Decorator _prev)`		
`ordered_stacks (const ordered_-stacks& x)`		
`void init (size_type n)`		
`class stack`		
`stack operator[] (size_type i)`		
`stack operator[] (size_type i) const`		
`void remove (size_type i)`		
`void mark_need_update (size_type i)`		
`bool need_update (size_type i)`		
`const value_type null ()`		
`bool outmatched_or_done (size_type i)`		
`void mark (size_type i)`		
`void print (size_type n)`		

J.2 Functions

J.2.1 psuedo_peripheral_pair

Prototype

```
template <class Graph, class Vertex, class Color, class Degree>
int psuedo_peripheral_pair(Graph& G, const Vertex& u, Vertex& w,
Color c, Degree d) ;
```

Description

To compute an approximated peripheral for a given vertex.

Definition

RCM.h

J.2.2 find_starting_node

Prototype

```
template <class Graph, class Color, class Degree>
graph_traits<Graph>::vertex_type find_starting_node(Graph& G,
Color c, Degree d) ;
```

Description

This is to find a good starting node for the RCM algorithm. The ”good” is in sense of the ordering generated by RCM.

Definition

RCM.h

See also

Alan George and Joseph W-H Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, 1981, Page 62.

J.2.3 reverse_Cuthill_McKee

Prototype

```
template < class Graph, class RandomAccessContainer, class
Color, class Degree >
void reverse_Cuthill_McKee(Graph& G, RandomAccessContainer&
iperm, Color c, Degree d) ;
```

Description

A starting vertex is computed by `find_starting_node`. This algorithm does not require user to provide a starting vertex to compute RCM ordering.

Definition

RCM.h

J.2.4 reverse_Cuthill_McKee

Prototype

```
template < class Graph, class RandomAccessContainer, class Color
>
void reverse_Cuthill_McKee(Graph& G, RandomAccessContainer&
iperm, Color c) ;
```

Description

A starting vertex is computed by `find_starting_node`. This algorithm does not require user to provide a starting vertex to compute RCM ordering. An interior DegreeDecorator is required.

Definition

RCM.h

J.2.5 reverse_Cuthill_McKee

Prototype

```
template <class Graph, class RandomAccessContainer>
void reverse_Cuthill_McKee(Graph& G, RandomAccessContainer&
iperm) ;
```

Description

A starting vertex is computed by `find_starting_node`. This algorithm does not require user to provide a starting vertex to compute RCM ordering. Assume that an interior DegreeDecorator and a ColorDecorator are available.

Definition

RCM.h

J.2.6 mmd

Prototype

```
template<class Graph, class DecoratorI, class DecoratorP, class
DecoratorQ>
void mmd(Graph& G, DecoratorI inverse_perm, DecoratorP perm,
DecoratorQ qsize, int delta=0) ;
```

Description

The implementation presently includes the enhancements for mass elimination, incomplete degree update, multiple elimination, and external degree.

Definition

mmd.h

See also

Alan George and Joseph W. H. Liu, The Evolution of the Minimum Degree Ordering Algorithm, SIAM Review, 31, 1989, Page 1-19