Artificial Intelligence 193 (2012) 186–216

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Concurrent forward bounding for distributed constraint optimization problems

Arnon Netzer, Alon Grubshtein *, Amnon Meisels

Dept. of Computer Science, Ben Gurion University of the Negev, P.O. Box 653, Be'er Sheva 84105, Israel

ARTICLE INFO

Article history:
Received 6 November 2011
Received in revised form 28 August 2012
Accepted 4 September 2012
Available online 7 September 2012

Keywords:
Distributed constraint optimization problems
Algorithms
ConcFB

ABSTRACT

A distributed search algorithm for solving Distributed Constraints Optimization Problems (DCOPs) is presented. The new algorithm scans the search space by using multiple search processes (SPs) that run on all agents concurrently. SPs search in non-intersecting parts of the global search space and perform Branch & Bound search. Each search process (SP) uses the mechanism of forward bounding (FB) to prune efficiently its part of the global search space. The Concurrent Forward-Bounding (ConcFB) algorithm enables all SPs to share their upper bound across all parts of the global search space. The number of concurrent SPs is controlled dynamically by the ConcFB algorithm, by performing dynamic splitting. Within each SP a dynamic variable ordering is employed in order to help control the balance of computational load among all agents and across different SPs. The ConcFB algorithm is evaluated experimentally and compared to all state of the art DCOP algorithms. The number of Non-Concurrent Logical Operations, Non-Concurrent Steps, the total number of messages sent and CPU time are used as performance metrics. The evaluation procedure considers different DCOP problem types with a varying number of agents and different constraint graphs. As problems become larger and denser, ConcFB is shown to outperform all other evaluated algorithms by 2–3 orders of magnitude in all performance measures. Further evaluations comparing different variants of ConcFB provide important insights into the working of the algorithm and reveals the contribution of its different components.

1. Introduction

The field of Distributed Constraint Reasoning provides a widely accepted framework for representing and solving Multi-Agent Systems (MAS) problems. In a distributed constraint problem each agent holds a set of variables representing its state. These variables take values from a finite domain and are subject to constraints. A distributed constraint algorithm defines an interaction protocol for coordinating a joint assignment of variables. Optimally solving constraint problems is NP-Hard in the general case [6].

Distributed Constraint Reasoning provide an elegant model for many everyday combinatorial problems that are distributed by nature. In these problems, independent computational entities, or agents, have partial knowledge of the problem. The distributed setting assumes that the agents are either incapable of disclosing private information or reluctant to do so [14,2]. Distributed Constraint Optimization Problems (DCOPs) were successfully applied to various MAS problems – coordinating mobile sensors [14,25], meeting and task scheduling [16], synchronization of traffic lights [13] and many others.

Recent years have seen a large number of different and interesting algorithms for optimally solving DCOPs. These include Synchronous Branch and Bound (SBB) [12], NCBB [4], ADOPT [21], ODPOP [23], BnB-ADOPT [26], OptAPO [10] and AFB [9].

* Corresponding author. Tel.: +972 54 7706194; fax: +972 8 6477650.
E-mail addresses: netzerar@cs.bgu.ac.il (A. Netzer), alongrub@cs.bgu.ac.il (A. Grubshtein), am@cs.bgu.ac.il (A. Meisels).

0004-3702/$ – see front matter
http://dx.doi.org/10.1016/j.artint.2012.09.002

Some algorithms use pseudo trees [21,23,26] (introduced in Section 2.2) and some attempt to asynchronously prune the search space [9]. The OptAPO algorithm partially centralizes the problem [10] and DPOP uses Dynamic Programming [23]. Despite these differences, all algorithms share two important properties: they all attempt to increase efficiency by increasing computational concurrency and they all attempt to promptly obtain good bounds to reduce the number of states visited in the search space.

The present paper presents a new approach towards finding an optimal solution to DCOPs. The proposed algorithm partitions the search space into non-intersecting subproblems somewhat similar to those described for DCSPs in [31]. Each subproblem involves all agents and is solved by the Synchronous Forward Bounding (SFB) algorithm. This choice of SFB stems from its synchronous nature and its powerful pruning abilities.

The agents take part in solving all independent subproblems concurrently by assigning unique identifiers and separate data structures to each subproblem. In this form, information from different areas of the search space can be used to achieve bounds faster. The Concurrent Forward Bounding (ConcFB) algorithm has the following important properties:

- High degree of concurrency. Much of the computational effort is performed in parallel and the algorithm terminates faster. Moreover, as is shown in Section 3, ConcFB controls the number of running concurrent search processes by dynamically splitting the remaining parts of the problem.
- Improves on former methods of Forward Bounding by sharing information between disjoint parts of the search space.
- Controls work load balancing by employing dynamic ordering heuristics. This is useful when computational effort is either costly or slow (weak mobile devices).

The efficiency of ConcFB is extensively evaluated against the state of the art algorithms where the number of Non-Concurrent Logical Operations, Non-Concurrent Steps, total number of messages sent and CPU time are used as performance metrics. An additional concurrent algorithm which combines multiple instances of SBB is introduced and its implementation is evaluated to provide further insights on the impact of concurrent algorithms.

The evaluation procedure considers different DCOP problem types with a varying number of agents and different constraint graphs. As problems become larger and denser, ConcFB is shown to outperform all other evaluated algorithms by *2–3 orders of magnitude* in all performance metrics. Further evaluations comparing different variants of ConcFB provide important insights into the working of the algorithm and reveals the contribution of its different components.

The remainder of this paper is structured as follows: Section 2 formally defines DCOPs and introduces some leading DCOP algorithms. Section 3 presents ConcFB in detail and Section 4 presents correctness and completeness proof for ConcFB. Section 5 describes enhancements to the basic ConcFB algorithm. The experimental evaluation and a discussion of the results is in Section 6. The conclusions are summarized in Section 7.

2. Distributed constraint optimization

2.1. Distributed Constraint Optimization Problem (DCOP)

Formally, a *DCOP* is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where:

1. $\mathcal{A}$ is a finite set of agents $A_1, A_2, \ldots, A_n$.
2. $\mathcal{X}$ is a finite set of variables $X_1, X_2, \ldots, X_m$. Each variable is held by a single agent but an agent may hold more than one variable.
3. $\mathcal{D}$ is a set of domains $D_1, D_2, \ldots, D_m$. Each domain D_i contains a finite set of values which can be assigned to the variable X_i.
4. $\mathcal{C}$ is a set of constraints. Each constraint $c \in \mathcal{C}$ defines a non-negative *cost* for every possible value combination of a set of variables, and is of the form:

$$C : D_{i_1} \times D_{i_2} \times \cdots \times D_{i_k} \to \mathbb{R}^+$$

A *binary constraint* is a constraint involving exactly two variables which takes the following form

$$C_{ij} : D_i \times D_j \to \mathbb{R}^+$$

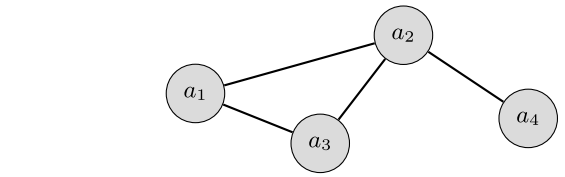
A *binary DCOP* is a DCOP in which all constraints are binary.

We say that a constraint c is applicable to a joint (partial or full) assignment a, if all variables involved in c take value in a.

To facilitate understanding, we make the following assumptions on the structure of DCOPs:

1. Each agent holds a single variable (the terms variable and agent will be used interchangeably).
2. DCOPs are assumed to be binary.

These are common assumptions in the DCOP literature (cf. [21,19,3,26]).



a_2 \ a_1	1	2	3	4
1	3	7	5	5
2	2	3	1	0

a_3 \ a_2	1	2
1	4	1
2	8	6

a_3 \ a_1	1	2	3	4
1	3	2	7	1
2	6	9	1	0

a_2 \ a_4	1	2
1	1	2
2	7	1

Fig. 1. A simple DCOP example with four agents.

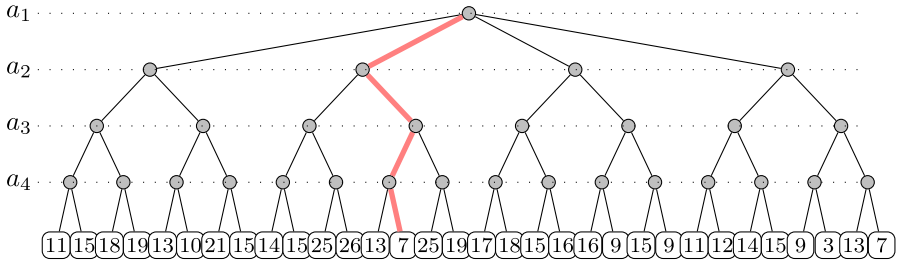


Fig. 2. The OR search tree of the DCOP depicted in Fig. 1.

Relations between interacting agents of a DCOP are often represented by graphs. Each node corresponds to an agent while an edge represents a constraint between two agents. The nature of these constraints is specified by a set of values corresponding to the agents' joint assignments. It is worth noting that unless explicitly stated otherwise, it is common to assume that the constraint structure of a DCOP does not limit communication between agents [27]. Thus agents can communicate with other agents and add links [29,27], agree on an ordering between them [30] or broadcast important information to other agents which are not directly connected to them via a constraint [9]. Fig. 1 presents a simple DCOP problem with four agents. The cost of each value combination is specified on the right hand side of the figure.

A DCOP algorithm proceeds by sending out messages and by adding or changing assignments to variables. An *assignment* is a $\langle variable, value \rangle$ pair. A set of assignments, in which each variable appears at most once, is called a *partial assignment*. The *cost* of a partial assignment, PA, is the aggregated constraint cost of all variables constituting the assignment PA. For example, the cost of the partial assignment $\langle a_1, 3\rangle\langle a_2, 2\rangle$ with respect to the DCOP of Fig. 1, is exactly 1. A *full assignment* is a partial assignment that includes all variables, and a *solution* is a full assignment of minimal cost. The solution to the above DCOP is $\langle a_1, 4\rangle\langle a_2, 2\rangle\langle a_3, 1\rangle\langle a_4, 2\rangle$ and its cost is 3.

One can also explore Constraint Reasoning problems by means of their underlying search space. Fig. 2 presents a tree representation of the search space of the DCOP in Fig. 1 (also referred to as the "generate-and-test" tree). In this tree each row corresponds to exactly one agent, and edges correspond to assignments. For convenience, each leaf of the search tree presents the cost of a full assignment. The set of all leaf nodes present all possible costs for every assignment combination. Throughout this paper we refer to the tree representation of a DCOP search space as the search tree or as the DCOP search tree and use these terms interchangeably.

A path from the root to a leaf in the search tree corresponds to a complete assignment. For example, the highlighted path in Fig. 2 corresponds to the assignment $\langle a_1, 2\rangle$ (the second out of four possible edges) $\langle a_2, 2\rangle$ (second out of two) and $\langle a_3, 1\rangle\langle a_4, 2\rangle$. The cost of this assignment is 7, which is higher than the optimal cost.

The search tree representation presented in Fig. 2 is also know as an OR search tree [17] of a constraint reasoning problem. It can be very useful in understanding different properties of constraint algorithms and their progress. The same search space can also be represented by an AND/OR search tree [17] which captures the idea of independent subproblems within the problem's search space. An AND/OR tree is guided by a pseudo tree [7].

A pseudo tree [8] is a spanning tree involving all agents of the problem with the following important property: any two nodes located in separate branches of the tree do not share a constraint. That is, different branches of the pseudo tree

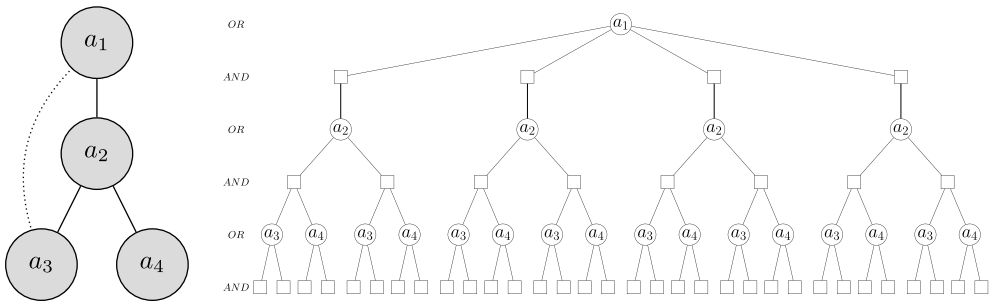


Fig. 3. A pseudo tree arrangement and the AND/OR tree corresponding to the simple DCOP example of Fig. 1.

represent independent DCOP subproblems. The arrangement of agents in a pseudo tree can increase concurrency and is used by some DCOP algorithms. Fig. 3 presents a pseudo tree arrangement and an AND/OR tree for the DCOP of Fig. 1.

General Cutset Conditioning [5,18] provides an alternative decomposition into independent subproblems. This approach is based on identifying a set of nodes that, once removed, would render the constraint graph cycle-free. That is, the method enumerates all possible instantiations of the cutset and can independently solve the remaining singly connected network in linear time. The performance gain from cutset conditioning is highly related to the constraint graph of a given instance and finding the minimal one is NP-complete.

2.2. DCOP algorithms

Recent years have seen a large number of algorithms for solving DCOPs. These vary in their approach towards finding minimal cost solutions. Successful DCOP algorithms are characterized by a high degree of concurrency and by methods for quick bounding of the search space.

Synchronous Branch and Bound (SBB) [12,19] is the simplest DCOP algorithm. Often used as a baseline algorithm for evaluation, it conveys the general scheme and difficulties of most DCOP algorithms in a simple manner. SBB operates by passing a unique CPA message (Current Partial Assignment) between agents. An agent holding the CPA attempts to extend it by adding an assignment which has a lower cost than the current upper bound. If a suitable assignment is found, it is added to the CPA and the CPA is sent to the next in line agent. If there is no such assignment, the CPA is sent back to the last agent in the CPA. At any given moment, only the agent holding the CPA may perform an action. This naive implementation produces an algorithm in which agents are idle for most of the time. Recent algorithms significantly outperform SBB (cf. [19] for an in-depth introduction and discussion on DCOP algorithms):

- Asynchronous Distributed Optimization (ADOPT) [21] – ADOPT is an inherently asynchronous algorithm. Agents running ADOPT are pre-arranged in a pseudo tree structure (or execute some pre-processing protocol which results in one) and continuously search the solutions space in a *Best First* manner. At any given moment agents maintain a lower and upper bound of the current search and when these meet, a solution is found.
  A powerful extension to ADOPT was recently introduced in [26]. Unlike its predecessor, BNB-ADOPT applies a *Depth First* search which is shown to greatly improve ADOPT's performance.
  Both ADOPT and BNB-ADOPT exhibit a high degree of concurrency, and attain good bounds by employing strong heuristic functions. Although experimental evaluation indicates that these algorithms significantly improve (in terms of computational effort) the base line SBB algorithm, both suffer from a rapid growth in messages as the number of neighbors an agent is constrained with increases (the problem's density).
- Distributed Pseudotree Optimization Protocol (DPOP) [22] – the DPOP algorithm is based on Dynamic Programming instead of the standard Search approach (i.e. it is an inference algorithm and not a search algorithm). DPOP operates in three phases: In the first phase, agents are re-arranged in a pseudo tree structure. In the second phase (UTIL), agents propagate the optimal, aggregated utility of their subtrees from the leafs towards the root of the pseudo tree (bottom-up phase). Finally, in the third phase (VALUE) the optimal assignments are propagated from the root to the leafs (top-down phase) based on the aggregated calculation of the second phase.
  Unlike ADOPT, the three phase structure of DPOP ensures a low number of messages which is linear in the number of agents. However, DPOP message sizes are exponential in the induced width of the underlying pseudo tree.
  ODPOP [23] is an extension to DPOP which overcomes this problem by sending UTIL tuples instead of complete optimal assignment combinations. These tuples are ordered by quality and are sent out one at a time. Thus, ODPOP is a complete algorithm with polynomial (in the width of the tree) sized messages.
  Although initially designed to handle *open problems* [23], ODPOP requires no changes when applied to standard DCOPs. The inherently different approach to solving DCOPs makes evaluating the computational effort of DPOP and ODPOP difficult, and this performance measure was not specified by the authors [22,23]. Our experimental evaluation (Section 6) indicate that the number of logical operations taken by ODPOP is higher than those of SBB.

- Non-Commitment Branch and Bound (NCBB) [4] – NCBB is a pseudo tree extension of SBB which employs a non-committing scheme for improving its concurrency level. Taking advantage of the underlying tree structure, NCBB is capable of searching through different branches of the tree simultaneously. This is combined with an interesting exploration scheme in which an agent may inform its children of different assignment. This "non-commitment" to an assignment further increases the algorithm's concurrency level as it allows one descendant to continue its search with a different context while its sibling is still pursuing the previous one.
  In [26] NCBB's performance is evaluated against BnB-ADOPT and is shown to be slightly less efficient.
- Asynchronous Forward Bounding (AFB) [9] – AFB applies a hybrid approach to asynchronicity: the CPA is passed between agents and proceeds in synchronous steps but solution bounds are attained in an asynchronous manner.
  AFB agents refrain from sending large amounts of volatile data by synchronously extending the CPA. That is, the agent receiving the CPA will attempt to extend it (add its assignment) and pass it on to the next agent. After the agent passes the CPA to its successor the agent notifies all unassigned agents of its current assignment. Each unassigned agent infers a bound which is the lowest cost of the partial assignment. These values are computed concurrently and asynchronously, and responses are sent back to the originator of the request. This results in an asynchronous process of backwards costs propagation. Whenever the aggregated cost of the bound requests breaches the best known upper bound, the requesting agent generates a new CPA with a revised assignment (or backtracks).
  The AFB algorithm, which synchronously assigns values to variables but asynchronously compute lower bounds on solution costs, was shown to produce better results in terms of network load and computational effort than those of ADOPT and DPOP, on most problem instances [9].

3. Concurrent forward bounding

Concurrent Forward Bounding (ConcFB) applies SFB (Synchronous Forward Bounding) as its means to attain good lower bounds while utilizing multiple concurrent search processes to speed up the search. ConcFB partitions the original *search space* to disjoint parts and concurrently search through each part with SFB. This means that multiple instances of the SFB algorithms are executed by *all* agents, and each instance searches through a distinct part of the search space. Global information such as the best upper bound is shared across all instances and further improves the algorithms pruning abilities.

The following features of ConcFB should be noted:

- It does not rely on a pseudo tree arrangement of agents. This simplifies the use of agent reordering heuristics which usually provide a significant performance boost to DCSP algorithms [20,33,34] (Section 5.1).
- ConcFB partitions the *search space* and do not rely on the structure of the constraints. The generated subproblems involve all agents but differ in the domain of at least one agent's variable (Section 3.2).
- Although the search process within each subproblem proceeds in synchronous steps (Section 3.1) there is no synchronization between the ongoing processes of different subproblems. This means that the *agents* act asynchronously and concurrently.

The components of ConcFB are first described separately – Section 3.1 discusses the forward bounding search technique, and Section 3.2 discusses concurrent search of distributed constraint reasoning problems. Section 3.3 provides a detailed overview of the ConcFB algorithm and Section 3.4 presents a ConcFB trace on the example DCOP problem of Fig. 1.

3.1. Synchronous Forward Bounding (SFB)

Synchronous Forward Bounding is highly related to the AFB algorithm presented in [9]. Similar to AFB, SFB agents pass a unique CPA message between them and only the agent that is holding the CPA may assign new values to it. SFB differs from AFB in that the agents do not proceed with an assignment prior to receiving all lower bounds of unassigned agents. This results in a synchronization of the algorithm which may help reduced propagation of irrelevant data at the cost of reduced concurrency. Note that ConcFB combines SFB with a Concurrent framework (Section 3.2) to increase its concurrency level.

Fig. 4 presents a simplistic Forward Bounding assignment scheme.[1]

SFB agents wait for messages which trigger different responses. Whenever an agent a_i receives a CPA, it first attempts to extend it by assigning a value to its variable (lines 1–4). It then sends out (or broadcasts) a message with the updated partial assignment to all unassigned agents. This *LB_Request* message triggers a calculation of the minimal cost for each agent which is sent back to a_i (lines 5–6). Unlike AFB, progress is blocked until all *LB_Report* messages are received and aggregated from all unassigned agents (lines 7–8). This blocks further execution and may result in long periods of idle waiting by agents. If the sum of this aggregated cost and the current cost of the CPA is lower than the current upper bound, then the CPA may be further extended by the next agent and the proper message is sent forward. If, however, this cost is

[1] Important details specifying the actions taken when the CPA is filled, the content of different message types and other relevant information is omitted for the sake of clarity.

```
1  CPA ← m.CPA;
2  if not Domain.isEmpty? then
3      CPA.add(Local_Assignment);
4      CPA.cost ⟵ CPA.cost + assignment's cost;
5      for a_j ∈ CPA.unassigned do
6          send(a_j, LB_Request);
7      Wait until all unassigned agents reply with an LB_Report message;
8      LB ⟵ aggregated sum of all LB_Report;
9      if CPA.cost + LB < Upper_Bound then
10         send(Next_Agent, CPA)
11     else
12         CPA.remove(Local_Assignment);
13         CPA.cost ⟵ CPA.cost − assignment's cost;
14         Remove Local_Assignment from domain;
15         call assign_CPA(m);
16 else
17     send(Last_Assigned_Agent, Backtrack_CPA);
```

Fig. 4. assign_CPA(msg) – A simplistic pseudocode for the assign method of SFB. This code is executed upon the reception of a message containing the CPA.

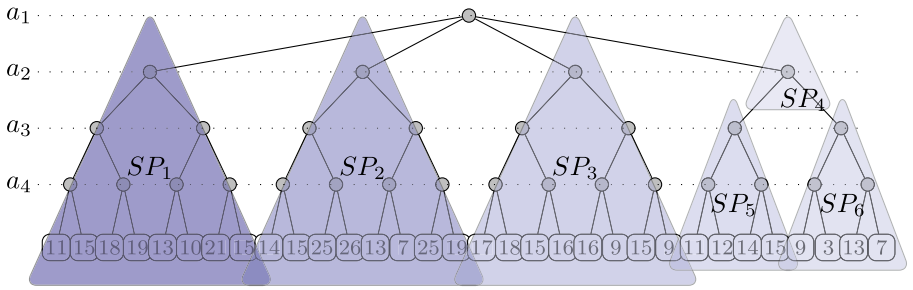


Fig. 5. Six disjoint partitions of the search space.

not lower than the current upper bound, a new assignment is attempted and the process repeats itself (lines 9–15). When the domain of the agent is emptied a backtrack is triggered and the current CPA (without an assignment to a_i) is sent to the last assigning agent (e.g. a_{i-1}) (lines 16–17).

3.2. Concurrent search

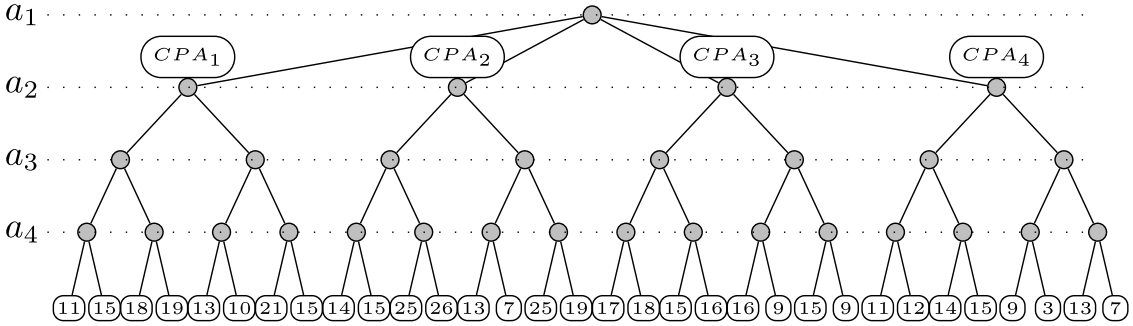
Concurrent search for solving DCSPs was presented by [31]. The basic idea in concurrent search algorithms is to *logically* partition the search space into non-intersecting parts. This allows the same set of agents to participate in distinct constraint search processes. Concurrent search is a powerful framework which may be applied to any DCSP algorithm [31].

In the simplest partitioning scheme, each assignment of the initializing agent is used by a different Search Process.[2] This results in a number of search processes which is exactly the same as the domain size of the initializing agent.

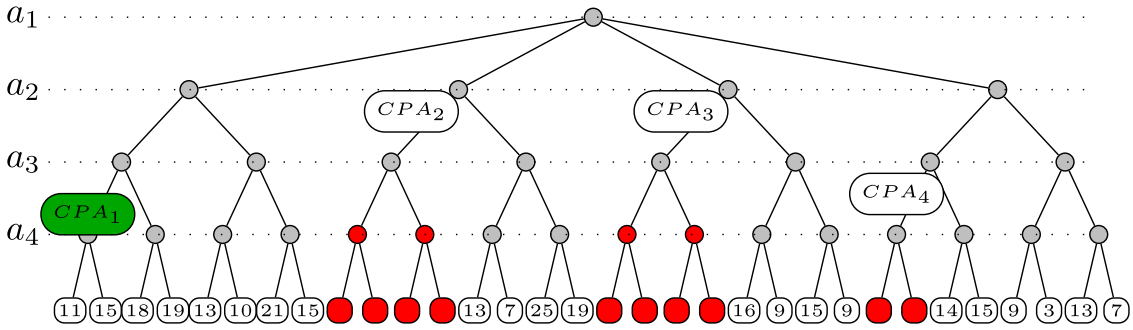
Concurrent search may also be adapted to DCOPs. However, unlike DCSPs which seek a single consistent solution, in a concurrent DCOP framework the minimal cost solution over all subproblems is reported. Fig. 5 provides a graphical representation of a possible partition scheme to the problem of Fig. 1. The search space is partitioned into six disjoint segments SP_1–SP_6 induced by the different domain of agent a_1 and a further split of SP_4 based on the domain of agent a_2. In each one of these subproblems, a_1 owns a single value in its variable domain. In SP_1 the domain of a_1 is 1, in SP_2 it is 2, and so forth. Having partitioned the original problem any complete DCOP algorithm may be applied to solve subparts SP_1, SP_2, SP_3, SP_5 and SP_6 to extract the minimal cost solution. The minimal cost solutions of SP_5 and SP_6 should be combined by the agent initiating the split of SP_4 (agent a_2).

It is important to point out that all agents in concurrent search may participate in more than a single subproblem. Fig. 5 demonstrates this idea: each one of the four agents participate in multiple search processes. This is in contrast to the subproblems resulting from a pseudo tree arrangement of agents, where each agent participate in one subproblem and results are merged by a higher priority agents.

[2] Represented as an OR node at the top of the tree, cf. [17].



(a) Concurrent SBB.



(b) Pruning multiple SPs in Concurrent SBB. Colored nodes are pruned when the CPA reaches agent 3 and 4.

Fig. 6. Snapshots of a Concurrent SBB run.

Information on the progress of each SP is maintained in a *Search Process* (SP) data structure maintained by each agent for each SP. An SP is a single search process and in the remainder of this paper we use these terms interchangeably. Each SP includes a unique identifier and maintains data on the global ordering of agents. All relevant messages and data structures of the underlying search algorithm are stamped by the relevant SP identifier and each agent holds a designated data structure recording information on every SP. In particular each agent maintains information on its current domain with respect to every SP (a value v_j maybe pruned from the domain of agent a_i in the context of SP_x but still be valid with respect to SP_y).

Before presenting ConcFB in detail, consider an example run of Concurrent SBB on the DCOP of Fig. 1. In this example the search space is partitioned into four SPs in which a_1 holds a single value in its domain (as shown in Fig. 5). ConcSBB begins by generating four different CPAs instead of the unique one used in standard SBB. In each SP, agent a_1 assigns its only value to one of the CPAs and sends it to agent a_2 (Fig. 6(a)). Upon receiving a CPA message, a_2 examines its identifier (the SP ID code) and associates it with the corresponding local SP. If a_2 is unfamiliar with the ID code, a new local SP is instantiated. The agent then proceeds to extend each one of the CPAs and continue its SBB run. At some point in time, the CPA of SP_1 holds a full assignment. A full assignment serves as a new upper bound which can be used to prune the search space. This new bound is relevant to the current SP, but it can also be used in other SPs to prune the search space. If agents 3 and 4 are notified of the new bound before proceeding to assign values the bound found by SP_1 can be used to prune assignments of SP_2, SP_3 and SP_4 (Fig. 6(b)).

3.3. Concurrent Forward Bounding (ConcFB)

The present work applies SFB to each individual search process resulting in Concurrent Forward Bounding (ConcFB). The SFB algorithm has two important features that make it suitable for concurrent search: powerful pruning abilities and synchronous progress. In particular, the latter feature facilitate the introduction of dynamic reordering of agents and dynamic splitting of the search space (discussed in Section 5).

The first agent in ConcFB begins its operation by partitioning the search space into disjoint parts. For each assignment in its domain a Search Process (SP) is initiated. These SPs represent a partition upon which a search algorithm (SFB) is executed. When all search algorithms conclude, the first agent reports the minimal cost solution found in these subproblems.

It is worth noting, that in order to improve pruning, upper bounds may be shared across search processes. That is, whenever a new upper bound is found, it is used to prune search on all SPs. Since an agent may participate in multiple SPs at any given time, means to identify and manage the execution of any search procedure carried out on any of its SPs are introduced.

The following subsections provides a detailed account of ConcFB. We first introduce the main data structures. Next, the messages used in the algorithm are presented. Finally Section 3.3.3 describes the pseudocode of each one of ConcFB's functions.

3.3.1. Data structures

The main data structures used in ConcFB are:

CPA A CPA (Current Partial Assignment) maintains all values of currently assigned agents and the resulting cost of the joint assignment. That is, it contains a set of pairs of the form $\langle Agent, Value \rangle$ and an integer value which is the aggregated sum of costs of all constraints applicable to the CPA. Every CPA is associated with a Search Process (discussed below) and shares with it the same unique identifier.

LB_List The LB_List is a vector of Lower Bounds (LBs) reported by all *unassigned agents* with respect to a given CPA.

Splits A list of all SPs diverging from a Search Process (not an agent). In the basic implementation of ConcFB the splits data structure is used only by the root search process.

SP The SP (Search Process) data structure lies at the heart of ConcFB. Each agent holds an SP for each logical search process it takes part in (see Fig. 5). An SP is instantiated whenever a new CPA is received by the agent. Every SP contains a unique ID, the current assignment the agent takes with respect to the specific SP along with its cost and the agent's current domain (again, with respect to the specific SP). Additionally, the SP holds the CPA that triggered its creation, the list of LBs (LB_List) and the splits diverging from the SP.

SP_List A list of all currently active SPs, held by each agent.

3.3.2. Messages

ConcFB uses five types of messages to transfer information and requests between agents:

CPA A message containing a *CPA* and an *LB_List*, sent by an agent extending the CPA on a given search process to an unassigned agent.

BT_CPA A backtrack message, notifying an agent that a CPA received needs to be backtracked.

LB_Request A message containing a *CPA*, sent to an agent asking it to calculate and return a Lower Bound for the given *CPA*.

LB_Report A message sent as a reply to **LB_Request**, reporting a Lower Bound from a given agent for a given *CPA*.

UB A broadcast message informing all agents on a newly reached Upper Bound.

These messages are limited in size and will at most contain a message type, an SP identifier, a CPA, an LB_List vector and possibly a few integer values (bounds).

3.3.3. Pseudocode

Fig. 7, Main(): The pseudocode of ConcFB's main procedure is described in Fig. 7. It starts with the *initializing agent* creating a *Root_SP* (line 3), and in it a data structure to hold the search process that would be created by splitting the domain of the initializing agent (line 4). The *initializing agent* calls the Init_SP() function to create the search processes and starts the search (line 5). The main loop (line 6) continuously looks for incoming messages (line 7), and dispatches them according to the message type to the appropriate functions (lines 8–19).

Fig. 8, Init_SP(): The Init_SP() function, described in Fig. 8, loops through all possible assignments in the domain (line 1). Each value in the domain is assigned to a different Search Process which is created with a unique SP_ID. The generated SP contains a single relevant value (line 2). All SPs are added to a list in the *Root_SP* data structure (line 3), and the Assign_CPA() function is invoked multiple times – once for each SP.

Fig. 9, Receive_CPA(): Upon receiving a *CPA* message, the function Receive_CPA() is called. In Receive_CPA() an agent creates a new SP data structure with the *SP_ID* of the received *CPA* and the entire domain of the current agent (line 1). A copy of the received *CPA* and *LB_List* is saved in the SP (lines 2,3), and the current agent's LB is removed from the *LB_List* (line 4). The removal of the agent's LB from the *LB_list* is crucial as it will be replaced later by the agent's actual cost according to its assignment. The newly created SP is added to a list of all SPs maintained by the agent (line 5), and Assign_CPA() is called.

Fig. 10, Receive_BT_CPA(): In Receive_BT_CPA() the agents need to identify the Search Process for which the BT_CPA message was sent. This is done using the *SP_List* and according to the ID retrieved from the message (line 1). The current assignment of the agent for the specific Search Process is removed from its domain (lines 2, 3). If the domain of the SP is exhausted then a Backtrack() is called. Alternatively an Assign_CPA() is called to try another assignment (lines 4–7).

Fig. 11, Receive_LB_Request(): When an agent responds to one of the *LB_Request* messages, it calculates its lower bound with respect to the *CPA* in the message (line 1). Formally the lower bound is defined as $LB = \min_{d \in Domain}\{\sum_{x_i \in CPA.vars} Cost(x_i^a, d)\}$. The lower bound is the minimum cost that can be achieved by summing all the constraints between any assignment in the agents domain and the given *CPA*. This lower bound is sent back to the *LB_Request* message originator as an LB_Report message (line 2).

```
1  done ⟵ false
2  if Initializing_Agent then
3      Root_SP := new SP(SP_ID(root), domain)
4      Root_SP.splits := new Split_Set()
5      Init_SP()
6  while not done do
7      msg ⟵ Get_Next_Msg()
8      switch msg.type do
9          case CPA:
10             Receive_CPA(msg)
11         case BT_CPA:
12             Receive_BT_CPA(msg)
13         case LB_Request:
14             Receive_LB_Request(msg)
15         case LB_Report:
16             Receive_LB_Report(msg)
17         case Upper_Bound:
18             if msg.UB < this.UB then
19                 this.UB ⟵ msg.UB
20         case Terminate:
21             done ⟵ true
22 return this.UB
```

Fig. 7. Main().

```
1  for i ⟵ 1 to domain.size do
2      SP_id := new SP(SP_ID(i), domain[i])
3      Root_SP.splits.add(SP_id)
4      Assign_CPA(SP_id)
```

Fig. 8. Init_SP().

```
1  SP_id := new SP(msg.sp_id, domain)
2  SP_id.CPA ⟵ msg.CPA
3  SP_id.Org_LB_List ⟵ msg.LB_List
4  SP_id.Org_LB_List.remove(LB of current agent)
5  SP_list.add(SP_id)
6  Assign_CPA(SP_id)
```

Fig. 9. Receive_CPA(msg).

```
1  SP_id ⟵ SP_list.get(msg.sp_id)
2  ca ⟵ SP_id.Current_Assignment
3  SP_id.domain.remove(ca)
4  if SP_id.domain.isEmpty? then
5      Backtrack(SP_id)
6  else
7      Assign_CPA(SP_id)
```

Fig. 10. Receive_BT_CPA(*msg*).

```
1 LB = min_{d∈Domain}{∑_{x_i∈msg.CPA.vars} Cost(x_i^a, d)}
2 send(LB_Report, msg.origin, LB)
```

Fig. 11. Receive_LB_Request(msg).

```
1  SP_id ⟵ SP_list.get(msg.sp_id)
2  SP_id.LB_List[msg.origin] ⟵ msg.LB
3  if received LB_Report messages from all unassigned neighbors then
4      if SP_id.CPA.cost + SP_id.Current_Assignment_Cost + ∑ SP_id.LB_List[i] < this.UB then
5          CPA ⟵ SP_id.CPA ∪ SP_id.Current_Assignment
6          send(CPA, Next_Agent(), CPA and SP_id.LB_List)
7      else
8          Receive_BT_CPA(msg)
```

Fig. 12. Receive_LB_Report(msg).

Fig. 12, Receive_LB_Report(): The Receive_LB_Report() function is used to collect LB_Report messages and decide if the current assignment of a given Search Process does not violate the UB. When LB_Report messages are received the agent needs to find out to which Search Process it belongs (line 1). The received LB is then entered into the *LB_List* of the appropriate SP either as a new entry or as an update to a previous LB from that agent (line 2). If LB_Reports were received for the specific SP from all unassigned neighbors (line 3) then all the Forward Bounding information has been received in order to decide whether the current assignment should be used to extend the *CPA*.

The cost of the *CPA* is added to the current assignment of the Search Process and to the sum of the Lower Bounds in the LB_List. The *CPA* cost stands for the cost collected by all assigned agents, the current assignment cost is the cost added by the current agent, and the sum of LB_List is a lower bound for the cost that would be added by all unassigned agents. If the sum of all the costs is smaller than the known Upper Bound (line 4) then a *CPA* is created by adding the current assignment to the Search Process *CPA* (line 5) and a CPA message is sent to the next agent, with the newly created *CPA* and the updated LB_List (line 6). If, on the other hand, the calculated sum of costs is bigger than the known Upper Bound, then the current assignment cannot extend the CPA to a solution and a Receive_BT_CPA() is called.

Fig. 13, Assign_CPA(): The Assign_CPA function is called whenever an agent tries to assign a new value to a given Search Process. If the domain of the specific Search Process is empty (no more value to assign) then a Backtrack() is called (line 21). Otherwise, the next best assignment is picked from the domain (lines 2–3) and the cost of this assignment is calculated and stored (line 4).[3] The cost of an assignment is the sum of the costs of all constraints between the assignment and the assignments in the *CPA*. Formally $Cost = \sum_{x_i \in CPA.vars} Cost(x_i^a, assignment)$.

The cost of the *CPA* is added to the current assignment cost and to the sum of the *LB_List*, and compared to the known Upper Bound (line 5). Note that the sum of the *LB_List* represents the lower bounds of unassigned agents with respect to the *CPA*, without the current agent assignment. If the sum of costs is larger or equal to the Upper Bound then the assignment cannot extend the *CPA* to a better solution. In this case the assignment is removed from the domain of the specific Search Process (line 6). If the removal of the assignment exhausts the domain, then a Backtrack() is called (line 8). Otherwise, the Assign_CPA() is called again to assign a new value (line 10).

If the sum of costs is smaller then the Upper Bound, then if this is the last agent and the assignment is a full assignment (line 12) a new Upper Bound has been found. In this case the new Upper Bound value is broadcast to all agents (line 14) and a Backtrack() is called. If this is not the last agent, a copy of the *Org_Lb_List* is created (line 17) and a request is sent to all unassigned neighbors to report their Lower Bound on the *CPA* after the new assignment was added to it (lines 18, 19). Note that LB_Request only needs to be sent to unassigned neighbors and not all unassigned agents, since unassigned agents which are not neighbors of the current agent will not change their Lower Bound due to current agent assignment. Note also that the *LB_List* must be initialized to the *Org_LB_List* before any Lower Bounds are collected for a new assignment, since Lower Bound updates for the previous assignment of the current agent are no longer valid and must be discarded.

Fig. 14, Backtrack(): When an agent decides to backtrack, then if this is the *Initializing Agent* the specific Search Process is removed from the *Root_SP.splits* list (line 2). If all Search Processes ended (line 3) the search is complete. The current Upper Bound is the solution and a terminate message is broadcast to all agents (lines 3–6). If this is not the *Initializing Agent* then a BT_CPA message is sent to the previous agent (lines 8, 9).

[3] Dynamic ordering of assignments can be easily implemented by calculating, sorting and storing all assignments' costs. This requires some (minor) additional computation and further improves performance.

```
1  if not SP_id.domain.isEmpty? then
2      ca ⟵ next best assignment from SP_id.domain
3      SP_id.Current_Assignment ⟵ ca
4      SP_id.Current_Assignment_Cost ⟵ ∑_{x_i ∈ msg.CPA.vars} Cost(x_i^a, ca)
5      if SP_id.CPA.cost + SP_id.Current_Assignment_Cost + ∑ SP_id.Org_LB_List[i] ⩾ this.UB then
6          SP_id.domain.remove(ca)
7          if SP_id.domain.isEmpty? then
8              Backtrack(SP_id)
9          else
10             Assign_CPA(SP_id)
11     else
12         if current agent is the last agent in SP_id then
13             this.UB ⟵ SP_id.CPA.cost + SP_id.Current_Assignment_Cost
14             Broadcast(Upper_Bound, this.UB)
15             Backtrack(SP_id)
16         else
17             SP_id.SP_List ⟵ SP_id.Org_SP_List
18             foreach a_i ∈ SP_id.CPA.Unassigned_Neighbor do
19                 send(LB_Request, a_i, SP_id.CPA ∪ SP_id.Current_Assignment)
20 else
21     Backtrack(SP_id)
```

Fig. 13. Assign_CPA(SP_{id}).

```
1  if Initializing_Agent then
2      Root_SP.splits.remove(SP_id)
3      if Root_SP.splits.isEmpty? then
4          Solution ⟵ this.UB
5          done ⟵ true
6          Broadcast(Terminate, null)
7  else
8      a_i ⟵ SP_id.CPA.Last_Assigned_Agent()
9      send(BT_CPA, a_i, SP_id.sp_id)
```

Fig. 14. Backtrack(SP_{id}).

3.4. ConcFB run trace

We next present a detailed trace of ConcFB on the simple DCOP problem presented in Fig. 1.

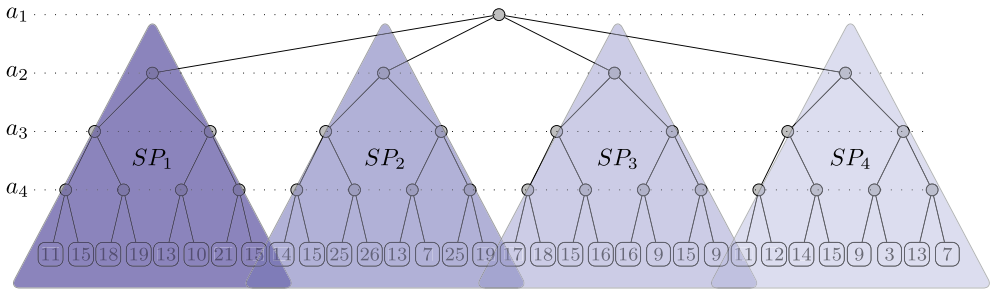
Agents begin by setting their upper bound value to ∞. All agents but the first (agent a_1) remain idle and await incoming messages.

Agent a_1 begins by generating four distinct search processes and assigns a different domain value to each one. These Search Processes carry a unique identifier (SP_1 to SP_4 in our example in Fig. 15a) used by all agents. Agent a_1 then assigns its value in each search process and broadcasts an LB_Request message to agents a_2 to a_4 (a total of 12 messages – one message per agent per search process). Note that the domain of a_1 includes a single distinct value in each SP and thus agent a_1 have four distinct assignments – one assignment per SP.

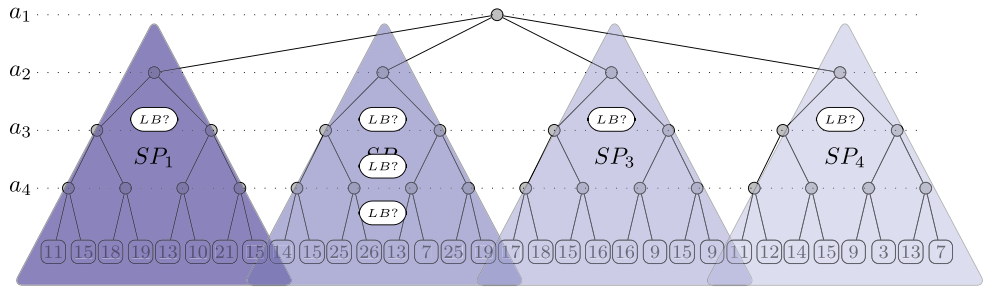
Agents receiving these requests will compute the minimal cost assignment and reply with its value as their LB. Thus, agent a_2 will respond with 2 for the message with the ID SP_1, and respond with a 0 (in a different message) when receiving a_1's message with ID SP_4. In contrast agent a_4 will return 0 for all SPs since it has no constraint with a_1.

It is important to note that although ConcFB proceeds in synchronous steps there is no guarantee on the timing of messages and computation by other agents. In some SPs a_1 may receive LB replies from all neighbors and proceed onward before all other bounds are received from other SPs. Let us assume then, that a_1 received the following five messages[4]: $\langle SP_1, \langle a_2, LB = 2\rangle\rangle$, $\langle SP_2, \langle a_2, LB = 3\rangle\rangle$, $\langle SP_2, \langle a_3, LB = 2\rangle\rangle$, $\langle SP_2, \langle a_4, LB = 0\rangle\rangle$ and the message $\langle SP_4, \langle a_3, LB = 0\rangle\rangle$, as depicted in Fig. 15b.

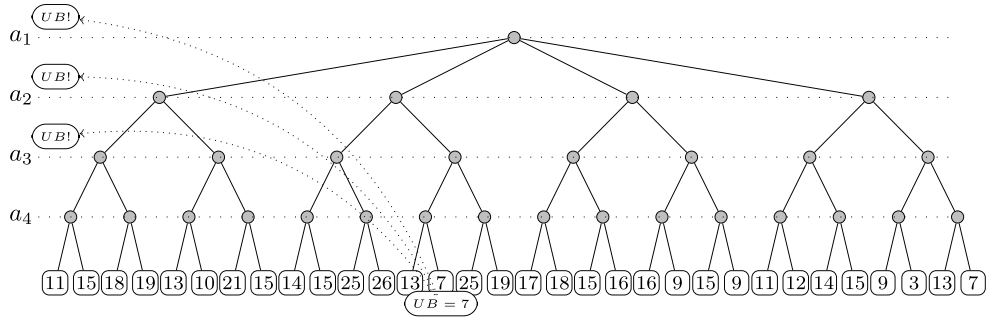
[4] We use the following format for messages $\langle SP_{id}, \langle sender, value\rangle\rangle$.



(a) Four SPs which act on non-intersecting parts of the search space. Each SP executes an instance of SFB.



(b) An LB request is made to each agent within each SP. At this point, agents a_3 and a_4 received the request in the context of SP_2 only.



(c) When a new upper bound is found its value is shared with all other agents which will apply it in all SPs.

Fig. 15. Run trace of ConcFB.

At this point agent a_1's SP_2 computes the sum of lower bounds (which is 5) and since it is lower than the initial upper bound of ∞ it sends out the SP_2 CPA to agent a_2. Having received the CPA of SP_2, agent a_2 now attempts to extend it. Unlike the previous case agent a_2 now has 2 domain value in this SP to choose from. It will assign the second one (best one) which is 2 and broadcast LB requests from a_3 and a_4 (note that in the background other agents keep responding to agent a_1's LB request. Specifically, agent a_2 may respond to such requests for other SPs as well but these are omitted for the sake of clarity).

Agents a_3 and a_4 respond to a_2's request by sending $\langle SP_2, \langle a_3, LB = 3\rangle\rangle$ and $\langle SP_2, \langle a_4, LB = 1\rangle\rangle$. As before, the resulting cost is lower than the initial upper bound and the CPA is passed onward. It is easy to see that at this point that SP_2 execution will proceed until a new upper bound of 7 is reached by agent a_4. This information is broadcast to all other agents (Fig. 15c) and the execution within SP_2 continues.

At this point let us focus on the progress of SP_1 (while assuming that execution of other SPs on all agents continues). Assume that at this point a_1 already received the following LB replies: $\langle SP_1, \langle a_2, LB = 2\rangle\rangle$, $\langle SP_1, \langle a_3, LB = 3\rangle\rangle$, $\langle SP_1, \langle a_4, LB = 0\rangle\rangle$. The resulting joint cost is 5 and as it is lower than the current upper bound of 7, a_1 sends the SP_1 CPA to agent a_2. The agent will choose its best assignment (2) and send an LB request to agents a_3 and a_4. Their responses are $\langle SP_1, \langle a_3, LB = 3\rangle\rangle$ and $\langle SP_1, \langle a_4, LB = 1\rangle\rangle$ which result in a cost equal to that of the bound (when combined with the constraint cost of a_1 and a_2's assignments). Agent a_2 next attempts its other assignment $a_2 = 1$ and resend LB requests to the unassigned agents a_3 and a_4. Again the bound is breached and a backtrack message is generated by agent a_2 to a_1. Agent a_1 receives the backtrack message and being the initializing agent terminates SP_1 by removing it from its split set.

A similar interaction concurrently occurs for other SPs: most of SP_3's search space is pruned by agent a_2 and the bound value of 7, SP_2 fails to locate a bound lower than 7 and SP_4 finds the optimal solution with a cost of 3.

4. Correctness of ConcFB

To prove the correctness of ConcFB we first prove that it terminates and then prove that upon termination the value of the upper bound UB is the same as the cost of the optimal solution.

To prove that ConcFB terminates one needs to prove that it will never go into an endless loop. To do so, we will consider the algorithm's states $s \in S$, where $S = \langle S_1 \times S_2 \times \cdots \times S_k \rangle$ – the Cartesian product of all SP states. A state $s_i \in S_i$ includes all agents, their current assignment and domain. The following lemma proves that the same state is not generated more than once.

Lemma 1 *(Unique states). A state S of ConcFB is never repeated.*

Proof. Begin by noting the following simple observation:

Observation 1. Any two states of different SPs $s_i \in S_i$, $s_j \in S_j$, $i \neq j$, will never be identical.

Observation 1 stems from the fact that the partitioning scheme of ConcFB (discussed in Section 3.2) generates disjoint search spaces assigned to the different SPs. In other words, the domain of the initializing agent is divides into different non-intersection parts (single values from its domain – line 2 of Init_SP()) which means that the assignment of agent a_1 in s_i and s_j can never be the same value.

The above observation implies that one needs only consider the states of the individual SPs to prove the lemma. That is, it is enough to show that in any given SP, the SFB algorithm never repeats any of its states.

Consider a specific search process, SP_{id}. Assume by negation that a_i is the highest priority agent (first agent in the order of assignments) that by acting repeats a previously visited state. Specifically, repeating a state means that the partial assignment (the CPA) is duplicated. Any new assignment added to the CPA is selected in the Assign_CPA function. This function is invoked from either one of the following functions:

- Init_SP() – This function is only invoked once – at the beginning of ConcFB's run. This means that no other prior CPA existed for SP_{id} (or any of its sibling SPs for that matter). Hence one must conclude that the SP_{id}'s CPA is identical to the CPA generated for some other search process, SP_j. However, since Init_SP() assigns a single unique value to each one of the SPs domain (line 2 of Init_SP()) and since new CPA values are added based on the current domain (line 2 of Assign_CPA()) we must conclude that the CPA generated for SP_{id} is different than that of SP_j in contradiction to our false assumption.
- Receive_CPA() – The Receive_CPA() function is invoked whenever a higher priority agent a_j (where $j < i$) sends a CPA message to a_i (line 10 of Main() and line 6 of Receive_LB_Report()). A duplicated CPA generated by a_i includes the same assignments to all of its variables and therefore the first j assignments must be the same. The fact that Receive_LB_Report() is the only trigger for a forward sent CPA message and is in turn triggered by messages received following line 18 of Assign_CPA() implies that agent a_j was the one to generate a duplicate CPA. Since Receive_CPA() does not change any domain value or assignment value (the state s_{id}) this contradicts the assumption that a_i is the highest priority agent which repeats a state.
- Receive_BT_CPA() – If Assign_CPA() is invoked following line 7 of Receive_BT_CPA(), lines 1–4 are also executed. Specifically, line 3 which removes assignments from the current domain of SP_{id} without adding new ones. As a result, line 2 of Assign_CPA() can never generate a duplicate CPA unless a value is returned to the domain of SP_{id}. This, however, only occurs in line 1 of Receive_CPA() which contradicts our assumption. □

Theorem 1 *(Termination). Every run of ConcFB terminates.*

Proof. A DCOP search algorithm will terminate if the following conditions hold:

- The number of states it goes through is finite.
- It does not examines the same state more than once (i.e. it does not fall into endless loops).
- The algorithm maintains progress. That is, it moves from one state to another within a finite amount of time.

The first condition is trivially met by the definition of a DCOP (Section 2.1), and the second one immediately follows from Lemma 1.

Following Observation 1 it suffices to examine the progress of a single SP to see that the third condition holds.

Consider the state $s_a \in S_i$. This state can proceed to some other state $s_b \in S_i$ whenever the Assign_CPA() and Receive_BT_CPA() functions are executed (assignment change or domain value removal) by some agent. Let a_i be the last agent in s_a to add an assignment to the CPA.

If a_i is the first agent and it has just commenced its run, it will call Init_SP() from the main function, which will in turn invoke Assign_CPA(). Assign_CPA() will broadcast requests to all unassigned agents (line 19, Assign_CPA()) and when

all request arrive it will either add its assignment to the CPA (line 5, Receive_LB_Report()) and move to state s_b or execute Receive_BT_CPA() in line 8. Receive_BT_CPA() will remove the offending value from the domain (line 3, Receive_BT_CPA()) and as a result the SP will move on to state s_b.

Otherwise, a_i assigns a value to SP_i's partial assignment in line 5 of Receive_LB_Report() only. It will then proceed to send the CPA to agent a_j. This message will eventually be received by agent a_j which will call Receive_CPA (line 10 of Main()). After updating relevant SP data structures Assign_CPA() will be executed. This can result in different outcomes:

1. A new assignment will be examined by a_j which will result in a breach of the upper bound (line 5 of Assign_CPA()). The agent will remove the offending value (line 6) which will change the current SP's state.
2. If the agent is the last agent, the Backtrack function will be invoked (line 15). A BT_CPA message will be sent to agent a_i (line 9, Backtrack()) which will invoke Receive_BT_CPA() (line 12 of Main()). As before, this function will remove the last assigned value of a_i from its domain (line 3 of Receive_BT_CPA()) and the SP's state will change to s_b.
3. The CPA and the potential assignment are broadcast to all unassigned agents and a_j resumes its Main() function. Unassigned agent will receive the new LB_Request message, calculate their LB value and send it to a_j (without removing any value from the domain or changing their assignment. That is, without changing the SP's state). Agent a_j will receive its peers replies and once the aggregated cost of all unassigned agents is calculated the agent will either update its current assignment (Receive_LB_Report(), line 5) or else execute Receive_BT_CPA and remove the potential assignment from its domain. In either case, the result would be a new state. □

Next one needs to prove that the value returned by ConcFB upon completion is indeed the optimal cost of a full assignment.

Theorem 2 *(Optimality). Upon termination, the cost of the upper bound* (*the local variable UB*) *is equal to that of the minimal cost assignment.*

Proof. ConcFB termination is initiated by the first agent (lines 5–6 of the Backtrack() function). The termination message will only be broadcast after all SPs are exhausted (line 3 of Backtrack()) so no other types of messages are generated (and hence *UB* will not be re-set after all SPs are exhausted). To prove optimality one must verify that whenever a new complete assignment with minimal cost is generated it is recorded, and that no valid assignments were pruned during search.

An upper bound cost is valid only if its corresponding assignment includes all agents and if its cost is lower than the current upper bound. This can only occur when the last unassigned agent successfully assigns a new value of lower cost. Lines 11–14 of the Assign_CPA() function manage this functionality. If the condition on lines 7 and 8 of this function hold, then a new upper bound is recorded by the last agent and all other agents are notified and consequently record this value (lines 17–19 in Main()).

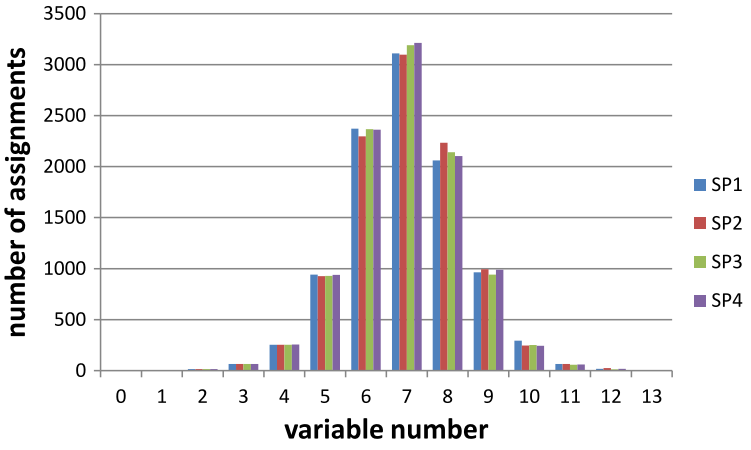
To prove that no other valid values are pruned one must examine all cases where an agent changes a CPA or skips a value.

A value may be skipped in line 5 of Assign_CPA() or as a result of line 4 of Receive_LB_Report(). In both cases the value is skipped when the combined cost of the CPA, the current assignment and the minimal cost of future (unassigned) agents is greater than the current cost of *UB*. Clearly the cost of the current assignment will not lead to a solution of lower cost than that of *UB* at termination and hence this value may be skipped over.

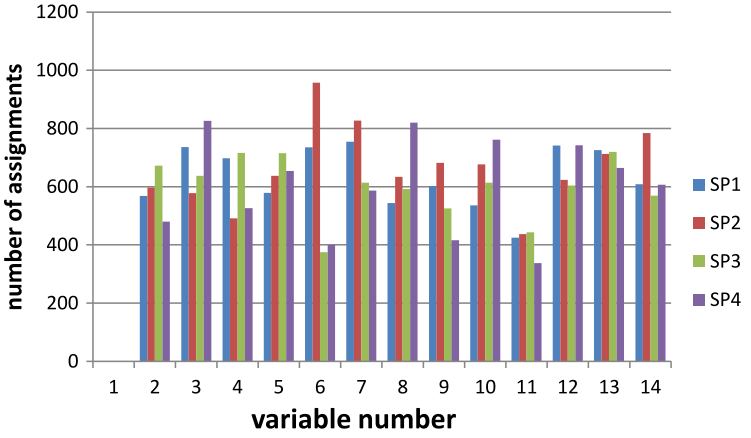
Let us now consider all cases which lead to a change of value:

- Assign_CPA() is invoked in line 4 of Init_SP(). Since this is the initial assignment of an SP and no prior value existed there is no risk that this call prunes any part of the search space.
- Receive_CPA()'s line 6 invokes Assign_CPA(). In this case an agent A_i receives a new CPA from a higher priority agent after some change of value by its predecessors. Since A_i did not yet pick an assignment at this point, any assignment it will make will not lose any potential solutions.
- Assign_CPA() is called from line 7 of Receive_BT_CPA() as a result of a lower priority agent's message. When a BT_CPA is sent from a lower priority agent one can conclude that the entire subspace originating from the current CPA was fully explored and as a result any change of value will not result in lost solutions.
- Assign_CPA() is called from line 7 of Receive_BT_CPA() as a result of a "message" from the agent to itself (line 15 of Assign_ CPA() followed by line 9 of Backtrack()). This case will occur when an agent reaches a full CPA and must examine the cost of other assignments. Since the cost of each full CPA is examined and recorded (when a new *UB* value is found) there is no risk of pruning solutions.
  Note that a similar situation can occur in line 8 of Receive_LB_Report(). However, this was considered as a value skipping condition – discussed above.

Finally, one must pay attention to situations in which a CPA is discarded. This can occur whenever an SP is fully explored ("exhausted") – in line 2 of Backtrack(). However, since the CPA is discarded only after the SP it belonged to was exhausted, no part of the subspace is skipped or pruned.



(a) CPA assignments per SP, static lexicographic ordering.



(b) CPA assignments per SP with random ordering.

Fig. 16. Work load distribution with random ordering and with static lexicographic ordering.

In conclusion, whenever a value is skipped, changed, or a CPA is discarded no better solutions are lost. Therefore, at termination ConcFB reports the lowest possible cost. □

5. Enhancements

This section describes further enhancements to the basic ConcFB algorithm. Dynamic variable orderings and dynamic splitting are introduced, and the pseudocode to incorporate these enhancements is presented.

5.1. Variable ordering

Dynamic reordering was shown to significantly boost performance for DCSP algorithms. However, there are a limited number of works discussing dynamic reordering for DCOPs (a notable exception is [24] which focuses on reordering pseudo tree based algorithms and may therefore be applied to DCOPs). The present paper presents two dynamic variable ordering heuristics which utilize the synchronous search carried out on each SP. One heuristic is aimed at improving the concurrency of the algorithm, and the other is designed to improve the algorithm's pruning ability by the use of a "fail first" heuristic.

5.1.1. Random ordering

To illustrate the motivation for random variable ordering one can examine the work load distribution between the agents during a ConcFB run. Fig. 16(a) shows a histogram of the number of CPA assignments per agent during the run of ConcFB with four Search Processes and twelve agents. It is clear that most of the assignments are done by agents 5–10. The first four agents are doing little work, being located at the higher levels of the search tree. Agents 10–12 are doing little work due to effective pruning.

To achieve a balanced work load distribution among the agent, one can use a random ordering scheme. The order of variables in each search process (after the *Initiating Agent*) is randomly selected. The only change needed in the pseudocode is in the Receive_LB_Report() function, where the random selection of the next agent is inserted (line 6 in Fig. 17).

```
1 SP_id ⟵ SP_list.get(msg.sp_id)
2 SP_id.LB_List[msg.origin] ⟵ msg.LB
3 if received LB_Report messages from all unassigned neighbors then
4     if SP_id.CPA.cost + SP_id.Current_Assignment_Cost + ∑ SP_id.LB_List[i] < this.UB then
5         CPA ⟵ SP_id.CPA ∪ SP_id.Current_Assignment
6         Next_Agent ⟵ random unassigned agent
7         send(CPA, Next_Agent, CPA and SP_id.LB_List)
8     else
9         Receive_BT_CPA(msg)
```

Fig. 17. Receive_LB_Report(msg) – random ordering.

Fig. 16(b) shows the same setup run with random variable ordering. One can clearly see the more balanced distribution among all agents.

5.1.2. Fail first ordering

A different approach to dynamic ordering is to use reordering not for the purpose of improving concurrency, but in order to improve pruning. One can use a "fail first" heuristic that is aimed at reaching the Upper Bound as soon as possible. A naive approach to "fail first" variable ordering would be to set the next agent in the ordering as the agent whose reported LB value is maximal. However, ordering agents in this fashion is not expected to change performance. The reason for that is that all unassigned agents are guaranteed to receive at least the same LB values as those received by the agent selecting the next one in the order. Hence their expected impact can not be distinguished based on this information.

To asses the possible impact of future agents on the total cost we consider unassigned agents' average cost with other unassigned agents. That is, we let each agent reporting its LB, calculate the additional average cost with others, assuming its LB assignment is used.

Let X_i be a variable owned by agent A_i, which received an LB_Request message and let X_i^{lb} be the assignment that produced the current LB for its LB_Report() message. X_j represents a variable j and X_j^a its assignment. We write $|Dom(X_j)|$ to specify the size of the complete domain of variable j. Agents precedence ordering is based on a calculated heuristic value:

$$h_{CPA}^{X_i} = \sum_{X_j \notin CPA} \frac{\sum_{a \in Dom(X_j)} Cost(X_j^a, X_i^{lb})}{|Dom(X_j)|}.$$

$h_{CPA}^{X_i}$ sums the average costs of unassigned variables with respect to A_i's LB assignment. This value is then sent back along with the Lower Bound. Note that the average cost of every agent with all other agents can be calculated in a preprocess stage in polynomial time.

Whenever an agent receives an LB_Report message, it registers the h values along with the lower bound. When an agent selects its next agent, it selects the one with the highest h value. Figs. 18 and 19 show the code updates needed for the "fail first" heuristic dynamic ordering.

5.1.3. Correctness of ConcFB with variable ordering

Theorem 3. *ConcFB with variable ordering terminates and is complete.*

Proof. As before, the initializing agent partitions the search space to disjoint parts. As a result one needs only consider the correctness of a single SP (see Observation 1).

We use induction over the number of agents to prove that ConcFB with variable ordering maintains correctness and rely on the fact that the highest priority agent in all ordering remains the same – the initializing agent. The base case of our induction includes a single agent. In this case, ordering is static and the search is both complete and it terminates. Next, assume that our induction assumption is true for any DCOP with $k < n$ agents and consider a DCOP with n agents.

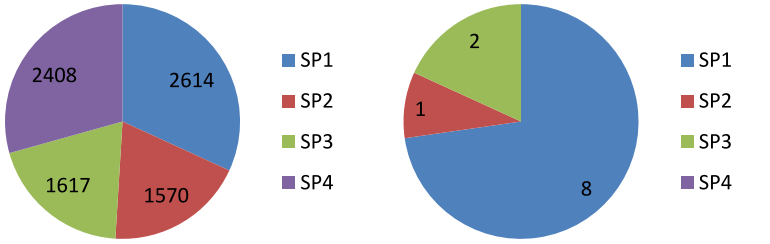
Since the initializing agent will never change its position within the ordering, it will assign its value and send it to one of the next (unassigned) agents. The remaining DCOP has $n-1$ agents and its initial order is set by the initializing agent's choice. By the induction assumption the remaining DCOP is complete and will terminate. That is, if a new upper bound is found it will be shared with all other agents (on all SPs) and its termination will result in a BT_CPA() message sent to the initializing agent. The initializing agent will either attempt a different assignment (invoke Assign_CPA() in line 7 of Receive_BT_CPA()) or backtrack and end the search in this SP (lines 1–6 of Backtrack()). Whenever a new successful assignment is made the initializing agent again send it to one of the unassigned agents. As before the remaining DCOP has $n-1$ agents and applying the search on it is both complete and terminates. This number of times this process repeats itself is exactly based on the number of values in the initializing agent's domain, and thus the process must end within a finite

1 $LB = \min_{d \in Domain} \{\sum_{x_i \in msg.CPA.vars} Cost(x_i^a, d)\}$
2 $d_{lb} \longleftarrow$ assignment that produced the LB
3 $h = \sum_{X_j \notin CPA} \frac{\sum_{a \in Dom(X_j)} Cost(X_j^a, X_i^{lb})}{|Dom(X_j)|}$
4 send(LB_Report, msg.origin, LB, h)

Fig. 18. Receive_LB_Request(msg) – heuristic ordering.

1 $SP_{id} \longleftarrow SP\_list$.get($msg.sp\_id$)
2 $SP_{id}.LB\_List[msg.origin] \longleftarrow msg.LB$
3 $SP_{id}.h\_List[msg.origin] \longleftarrow msg.h$
4 **if** *received LB_Report messages from all unassigned neighbors* **then**
5 **if** $SP_{id}.CPA.cost + SP_{id}.Current\_Assignment\_Cost + \sum SP_{id}.LB\_List[i] < this.UB$ **then**
6 $CPA \longleftarrow SP_{id}.CPA \cup SP_{id}.Current\_Assignment$
7 $Next\_Agent \longleftarrow$ agent with highest h
8 send(CPA, $Next\_Agent$, CPA and $SP_{id}.LB\_List$)
9 **else**
10 Receive_BT_CPA(msg)

Fig. 19. Receive_LB_Report(msg) – heuristic ordering.



(a) Total number of assignments per SP. (b) Total number of upper bounds reached per SP.

Fig. 20. Work load per search process.

amount of time. Its correctness stems from the fact that during each phase the minimal cost solution which includes all agents is found and broadcast to all agents. □

5.2. Dynamic splitting

Fig. 20(a) presents the assignments statistics of each of the 4 SPs that were created during the run time of ConcFB. It includes the total number of CPA assignments, per each of the SPs. There are 14 variables in the example with an average of 4 neighbors per variable, and a domain size of 4.

One can see that the distribution of CPA assignments is not uniform among the search processes. In this example SP_2 has about half the number of CPA assignments than that of SP_1. This will result in SP_2 exhausting its subspace much earlier than other SPs, and lowering the degree of concurrency of the algorithm. Spawning more SPs to search the subspace of SP_1 will balance the work load among all search processes, and ensure that enough SPs are active at each time during the search.

To increase the level of concurrency, a dynamic split request is passed to the agent which spawned the current SP which in turn partitions the existing SP.[5] Such a dynamic splitting heuristic for the DCSP case is described in [31]. In [31] the heuristic measures the number of assignments in a given SP, and when this value exceeds a given threshold a split is requested. However, in the DCOP setting, dynamic splitting may also be used to focus the search efforts in promising subspaces which are more likely to produce Upper Bounds. Fig. 20(b) presents the distribution of the number of Upper Bounds found per SP for the same runtime example. As can be seen, the number of Upper Bounds found can vary significantly between SPs. In this case, SP_1 reaches most of the upper bounds and partitioning it into smaller subspaces is expected to improve the overall pruning of the algorithm.

[5] This corresponds to an OR node which represents the split point to the original OR tree.

Split depth	$n = 14,\ D = 5,\ k = 4$		$n = 10,\ D = 10,\ k = 4$	
	Max SPs	NCCC	Max SPs	NCCC
n	18	474 936	21	201 010
$n-1$	40	471 936	42	202 593
$n-2$	92	453 880	88	199 194
$n-3$	230	453 907	228	195 304
$n-4$	646	459 084	785	202 718
$n-5$	2049	474 936	1896	210 025
$n-6$	6679	483 030	–	–
$n-7$	15 125	491 874	–	–
$n-8$	22 151	495 933	–	–

Fig. 21. The maximal number of concurrent SPs and the number of NCCC as a function of split depth request, averaged over 50 executions with n agents, a domain size D and k neighbors per agent.

While there are numerous partitioning heuristics, we propose a dynamic split mechanism in which agents residing in the lower parts of the search tree send split requests for every search process that reaches them. The *split request* is sent to the agent that originated the search process, asking it to re-split. This heuristic is aimed at focusing search efforts on promising parts of a SP, as implied by Fig. 16(a). Determining the threshold depth for sending a split request can have a significant impact on the number of search process. In extreme cases over splitting may result in thrashing and increased memory load of agents.

The impact of the split depth's value on the maximal number of concurrent SPs and the total number of NCCCs is presented in Fig. 21. We say that two or more SPs are concurrent if at a given moment both SPs are active. The table measures split depth relative to the total number of agents, such that a split depth of n means that a split request is generated only by the last agent. The table shows an exponential increase in the maximal number of concurrent SPs as the split depth decreases and that the number of NCCCs is minimal for split depths of roughly $n-3$. To understand these results, first consider split depth values which are greater than $n-3$. An increase in the split depth decreases the maximal number of concurrent SPs and concurrency in general. The number of NCCCs will therefore increase as less work is being done simultaneously. In contrast, when lowering the split depth's value the number of concurrent SPs dramatically increases and each agent must process requests from multiple, relatively shallow, SPs. These requests interfere with the agents progress towards new solutions which lie at the bottom of the search tree and as a result the number of NCCCs increase.

5.2.1. Integrating dynamic splitting to ConcFB

To identify the originator of the Search Process, and keep track of search processes, a unique *SP_ID* is constructed from a list of pairs. That is, $SP\_ID = \langle Agent, Counter \rangle$ where *Agent* is the *ID* of the creating agent, and *Counter* is incremented for every search process spawned by that agent. The originator agent of a search process can always be retrieved by looking at the *Agent* field of the *SP_ID*.

To request dynamic splitting a new type of message is used:

Split_Request A split request is sent to an agent asking it to split a specific Search Process.

The main function is updated to respond to this type of message and upon receiving *Split_Request* it calls a new function *Receive_Split_Request*() (lines 17, 18 in Fig. 22).

In *Receive_Split_Request*() (Fig. 23) the agent retrieves the Search process that needs to be split (line 1). If the relevant SP domain has more than one value that was not assigned yet (line 2), then the current search process can be split. A new *SP_ID* is created in line 3, a new SP is created with the new *SP_ID*, and with half of the splitting Search Process domain (line 4). The new SP is added to the father SP *Splits* list (line 5), it is added to the agent *SP_list* (line 6) and *Assign_CPA*() is called for the new SP (line 7).

If the current domain of the father SP is too small to split, and if the current agent is not too deep in the search tree (line 9), a *Split_Request* is sent to the next agent down, trying to split the SP deeper in the search tree (line 10).

In addition to the new *Receive_LB_Report*() function, Receive_CPA() needs to be updated in order to support dynamic splitting. The updated routine is presented in Fig. 24. In line 5 the newly created search process adds itself to its own splits list. In lines 7, 8 the depth of the search tree is checked, and if a given depth is reached, a *Split_Request* is sent. A good heuristic for the target depth was found experimentally to be the bottom quarter of the search tree.

The last change required in the basic ConcFB pseudocode in order to support dynamic splitting, is crucial to maintaining correctness. The *Backtrack*() routine must be updated to make sure that all Search Processes spawned from a given *Father_SP* ended their search, before *Father_SP* can backtrack (Fig. 25). The *Father_SP* of the backtracked SP is located in line 8. The backtracked SP is removed from its father splits list (line 9), and if this list is empty then the *Father_SP* backtracks.

5.2.2. Correctness of ConcFB with dynamic splitting

Theorem 4. *ConcFB with dynamic splits terminates and is complete.*

```
1  done ⟵ false
2  if Initializing_Agent then
3      Root_SP := new SP(SP_ID(root), domain)
4      Root_SP.splits := new Split_Set()
5      Init_SP()
6  while not done do
7      msg ⟵ Get_Next_Msg()
8      switch msg.type do
9          case CPA:
10             Receive_CPA(msg)
11         case BT_CPA:
12             Receive_BT_CPA(msg)
13         case LB_Request:
14             Receive_LB_Request(msg)
15         case LB_Report :
16             Receive_LB_Report(msg)
17         case Split_Request:
18             Receive_Split_Request(msg)
19         case Upper_Bound:
20             if msg.UB < this.UB then
21                 this.UB ⟵ msg.UB
22         case Terminate:
23             done ⟵ true
24 return this.UB
```

Fig. 22. Main() – dynamic splitting.

```
1  sp_current ⟵ SP_list.get(msg.sp_id)
2  if sp_current.Current_Domain.Size > 1 then
3      SP_ID := new SP_ID(serial, counter)
4      sp_new := new SP(SP_ID, sp_current.Current_Domain.Split_Domain())
5      sp_current.splits.add(sp_new)
6      SP_list.add(sp_new)
7      Assign_CPA(sp_new)
8  else
9      if Not last agent then
10         send Split to next agent;
```

Fig. 23. Receive_Split_Request(*msg*) – dynamic split.

```
1  SP_id := new SP(msg.sp_id, domain)
2  SP_id.CPA ⟵ msg.CPA
3  SP_id.Org_LB_List ⟵ msg.LB_List
4  SP_id.Org_LB_List.remove(LB of current agent)
5  SP_id.splits ⟵ msg.sp_id
6  SP_list.add(SP_id)
7  if SP_id.CPA.Number_Of_Assigned_Agents > Split_Depth then
8      send Split to msg.sp_id.originator;
9  Assign_CPA(SP_id)
```

Fig. 24. Receive_CPA(msg) – dynamic split.

Proof. Proving that ConcFB with dynamic splits maintains correctness requires proving that the union of all subparts equals the entire SP's space, that the search on all subparts of an SP are complete and terminate and that the algorithm do not conclude the search of the original SP before all subparts conclude.

It is easy to see that a split request partitions the search space into disjoint parts which include all values of the splitting agent's current domain (line 4 of Receive_Split_Request()).

```
1  if Initializing_Agent then
2      Root_SP.splits.remove(SP_id)
3      if Root_SP.splits.isEmpty? then
4          solution ⟵ this.UB
5          done ⟵ true
6          Broadcast(Terminate, null)
7  else
8      Father_SP ⟵ father SP of SP_id
9      Father_SP.splits.remove(SP_id)
10     if Father_SP.splits.isEmpty() then
11         a_i ⟵ SP_id.CPA.Last_Assigned_Agent()
12         send(BT_CPA, a_i, SP_id.sp_id)
```

Fig. 25. Backtrack(SP_{id}) – dynamic split.

To address the second point, it is enough to see that by generating sub-SPs from different values in the domain of the original SP one creates disjoint search (sub)spaces (cf. Observation 1). Next, by following the correctness of SFB the search on individual sub-SPs is complete and terminates.

A complete search through each sub-SP is insufficient. To maintain correctness of ConcFB, the algorithm must not conclude the search of an SP until all subsearches conclude.

The dynamic split version of ConcFB achieves this by requiring that all sub-SPs are completed prior to sending a BT_CPA message to the last assigned agent (lines 10–12 of Fig. 25). Thus, whenever an agent executes the Backtrack() function of Fig. 25 it removes the current SP from its *splits* list (which may contain only the existing SP if the agent did not receive a split request at an earlier stage), and hold messages to its predecessor until all sub-SPs are terminated. This guarantees that only after the search process is thoroughly examined a report acknowledging this fact is sent. □

6. Experimental evaluation

The experimental evaluation is divided into four subsections. The first subsection deals with a comparison between ConcFB and other state of the art DCOP algorithms. The second, explores performance in a network environment where the cost of communication is significantly higher than the cost of computation. An evaluation of the concurrent search approach is detailed in the third subsection where we compare ConcFB with a concurrent search version of SBB. Finally, we analyze ConcFB and quantify the impact of each component in the algorithm.

In all experiments the evaluated ConcFB variant included dynamic fail first reordering and dynamic splitting with the split depth heuristic set to $n-2$.

6.1. Algorithm performance

Two performance measures are routinely used to evaluate DCOP algorithms: run-time in the form of Non-Concurrent Constraints Checks (NCCCs) [32], and network load measured as the total number of messages sent [15,28].

We focus the present evaluation on a comparison of ConcFB to leading DCOP algorithms – BnB-ADOPT [26], AFB-CBJ [9], ODPOP [23] and BnB-ADOPT+ [11] – a recent improvement of BnB-ADOPT which removes redundant messages (note that BnB-ADOPT+'s computational effort in terms of NCCCs is exactly the same as BnB-ADOPT).

Although DPOP is often used for solving DCOPs, we find it impractical for the problems we experimented on, mainly due to its exponential sized messages. As discussed in Section 2.2, ODPOP [23] provides as a remedy for the exponential sized message problem of DPOP on regular DCOPs.

The main computational operation of ODPOP is the comparison of combinations of assignments, sent to each computing agent by its offspring in the pseudo tree [23]. This operation is performed by each agent in order to find an assignment for itself that is optimal with respect to the assignments of its ancestors.

The introduction of ODPOP requires that all run time experimental evaluations are given in terms of non-concurrent logical operations (NCLO). For ODPOP these are compatibility checks, and for BnB-ADOPT and ConcFB they are constraints checks.

In a recent paper, BnB-ADOPT was shown to be superior to both ADOPT and NCBB [26]. The present evaluation uses BnB-ADOPT with DP2 as a preprocessing phase for h values [1]. Synchronous branch & bound algorithm (SBB) with best value assignment is used as a reference algorithm. The AFB variant used in our evaluation is the best known one – AFB-CBJ. In AFB-CBJ, back-jumps to the offending agent are used instead of simple backtrack steps. This provides a potential speedup of the original algorithm [9].

Following [26] we applied a Distributed DFS protocol to all pseudo tree based algorithms. Communication overhead of this protocol was not added to the comparison.

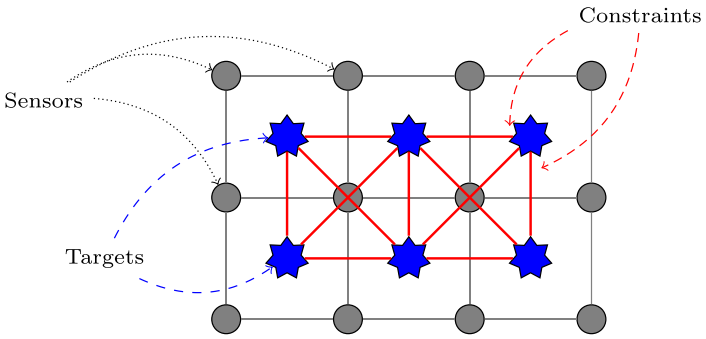


Fig. 26. A sensor network example.

Our evaluation setup included three problem types – Graph Coloring, Random DCOPs and sensor networks. In each setup 50 instances were generated for each set of parameter configuration and the reported results were averaged over all runs.

- **Graph Coloring:** This setup reconstructs the Graph Coloring tests presented in [26]. In this setup, each vertex corresponds to an agent. An edge between two vertices corresponds to a constraint. Constraint costs are randomly and uniformly selected from the range of 0 to 10 000 and each vertex degree is set to 3 (randomly selected). The evaluation included several different setups where the number of agents used was varied across setups from 5 to 15. Each agent's domain include three possible values (colors). All problems instances are connected graphs.
- **Random problems:** Random DCOPs include a set of agents randomly constrained with one another. This setup includes unstructured problems which are denser than the Graph Coloring problems and in which, if not specified otherwise, constraints costs were uniformly sampled in the range of 0 to 10 000. The number of agents, graph density, agents' domain sizes and cost ranges were varied to assess the algorithms robustness to different setups. As before, problems instances include connected graphs only.
- **Sensor networks:** The sensor network setup models a target tracking system [26,16]. Sensors jointly attempt to track targets taking into account the availability of other sensors, the number of sensors required to track a target and the sensors spatial configuration. In this setup, agents are the targets, their domains are the time slots when they are being tracked and the constraint are between adjacent targets. The cost of assigning a time slot to a target that is also assigned to an adjacent target is infinity (or a sufficiently large enough number) since the same sensor cannot track both targets during the same time slot. Similar to [26], the cost of tracking a target is in the range of 0 to 100, and the cost of a target untracked during any time slot is 100.
  Fig. 26 depicts a sensor network example. In this example, there are 6 targets tracked by a grid of 3×4 sensors. Targets are only constrained with other adjacent targets. In the evaluation of this setup the number of targets varied in the range of 5 to 14 over a grid of $3 \times x$ sensors (where the number of sensors varied to accommodate the number of targets).

Fig. 27 presents performance measurements in the Graph Coloring setup. One can see that ConcFB outperforms all algorithms but ODPOP on both metrics. It is about twice as efficient in terms of NCLOs, and provides a significant improvement in terms of messages over BnB-ADOPT and BnB-ADOPT+. ConcFB also outperforms AFB-CBJ by at least an order of magnitude in both metrics. ODPOP on the other hand, trades high computational effort with lower network load. Thus, although it uses less messages, its run time in terms of NCLOs is a hundred times slower.

The results in Fig. 27(a) are in accordance with those reported by [26] for BnB-ADOPT. Since the BnB-ADOPT messaging scheme is similar to that used by ADOPT [21], the resulting high number of messages is not surprising – as was already found by [9].

Our results also reaffirms the low number of messages used by ODPOP. It indicates the high computational effort associated with the process of combining assignments and the trade off between computational effort and network load.

The initial experiments conducted on Random DCOPs introduced a dramatic increase in the NCLO count of ODPOP. As problems became more constrained ODPOP's performance rapidly degraded in comparison to other algorithms. Fig. 28 presents a small random problem with 8 variables and a domain size of 4. As the number of constrained neighbors per agent increased from 3 to 5 the number of NCLOs required by ODPOP rapidly increased. Applying ODPOP on larger problems did not return within a reasonable time, and was therefore removed from the full sized Random DCOP setup.

Fig. 29 presents the results of the random problems with 12 agents and a domain size of 5. The number of constrained neighbors per agent was varied across setups from 3 to 7. In this setup the NCLO measure is actually equivalent to NCCC, for all algorithms. One can see that ConcFB outperforms BnB-ADOPT by 2 to 3 orders of magnitude and AFB-CBJ and BnB-ADOPT+ by 1 to 2 orders of magnitudes, on both metrics.

These experiments show that BnB-ADOPT and BnB-ADOPT+ do not scale as well as ConcFB and AFB-CBJ to higher density problems. To understand this difference in performance one must examine two important aspects of BnB-ADOPT/+:

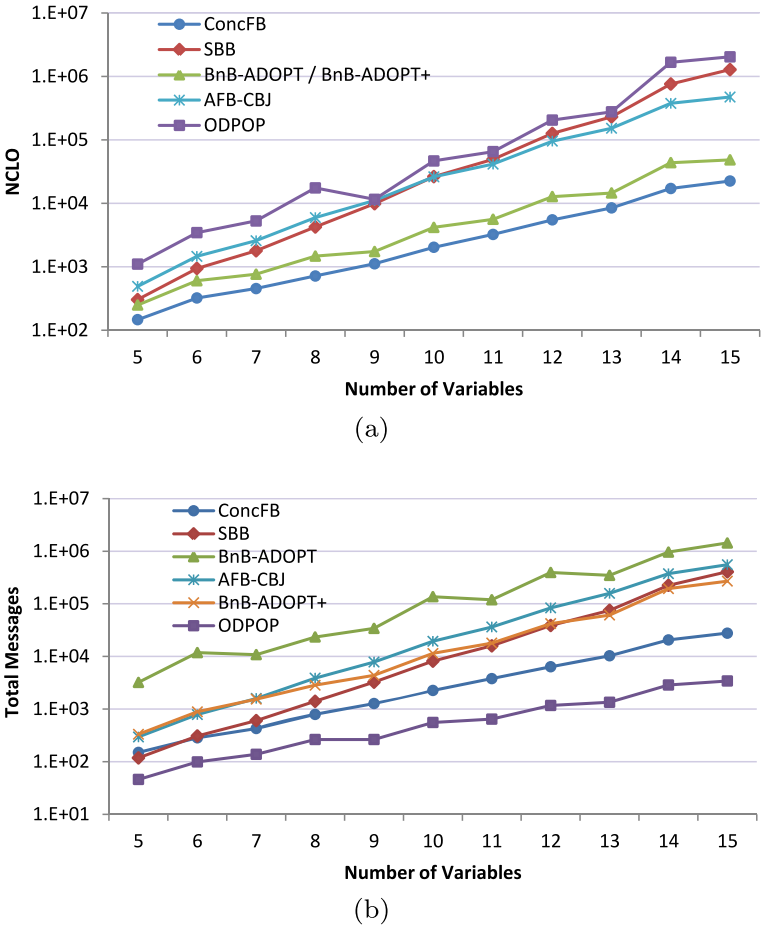


Fig. 27. Experimental results for the Graph Coloring setup (5 to 15 agents, domain size of 3 and density value 3).

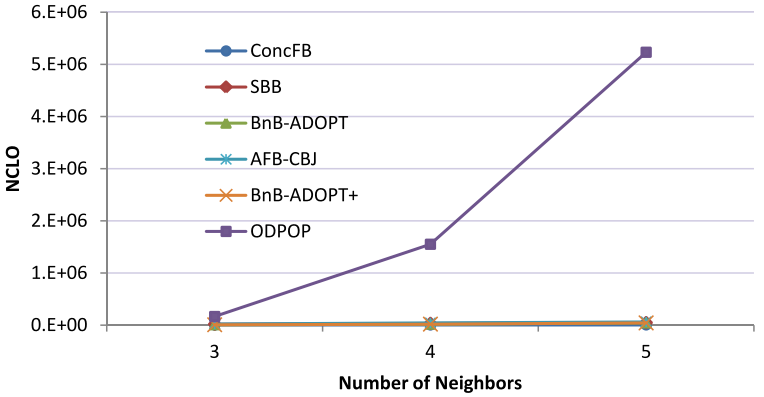
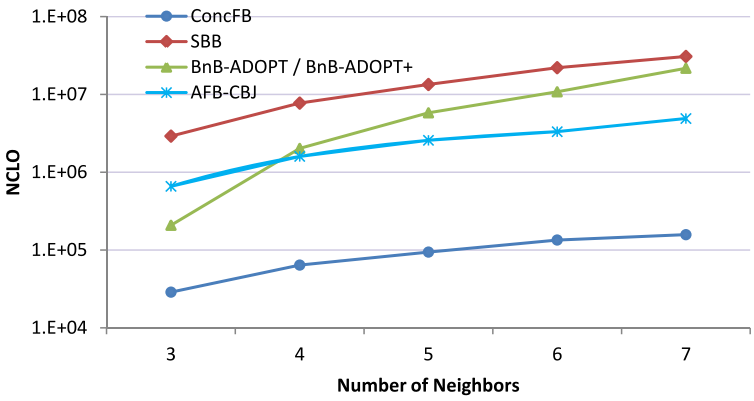


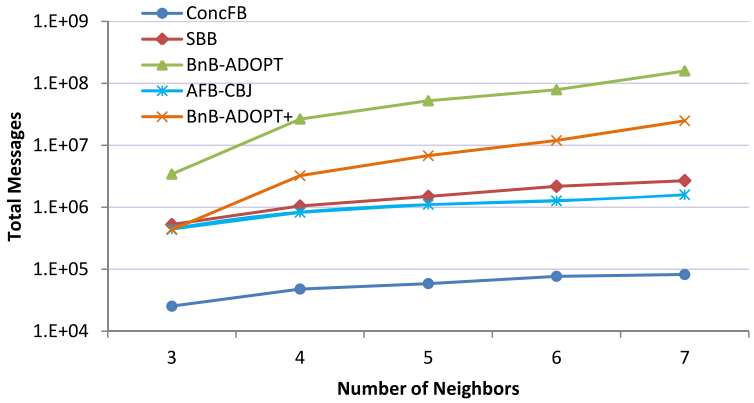
Fig. 28. Experimental results for small random problems (8 agents, domain size of 4 and varying number of neighbors).

- Both rely on a pseudo tree arrangement of agents. As the density value increases, a "wide" pseudo tree becomes less likely and as a result concurrency level decreases.
- BnB-ADOPT/+'s inability to reuse bounds. In these algorithms, when a CPA change does not violate the descendant's assignment, all bounds are still valid. As constraint density increases, the probability that a CPA change will not violate any constraint decreases and bounds have to be recalculated.

The relative robustness of ConcFB to various changes in the problem parameters is demonstrated in Figs. 30 and 31. In these experiments the averaged performance of ConcFB, BnB-ADOPT+, AFB and SBB was compared on problems with 12



(a)



(b)

Fig. 29. Experimental results for random problems (12 agents, domain size of 5 and varying number of neighbors).

agents, each with 4 neighbors and varying domain sizes or costs structure. In the first setup, the domain size was varied in the range of 3 to 8 (Fig. 30) and in the second one it was set to 5 and the costs were uniformly selected from the range 0 to x, where $x = \{1, 10, 100, 1000, 10\,000\}$ (Fig. 31). The results clearly validate ConcFB's superiority over other state of the art algorithms. ConcFB's significant improvement over other algorithms in NCLOs and the number of messages is maintained even in the presence of these changes.

Fig. 32 presents the performance of ConcFB, SBB, BnB-ADOPT+ and AFB as a function of the number of targets tracked in the sensor network problem. These results are consistent with the evaluation of the other setups reaffirming the advantage ConcFB holds over other algorithms.

All run-time measurements of DCOP algorithms use a non-concurrent metric such as NCCCs or NCLOs, taking into account the concurrency of agents' computations. However, these pure measures count steps of non-concurrent computation and are not influenced by details of implementation. In order to provide some insight on the cost of agents' actions which go beyond the most common logical operation – for example, operations on complex data structures – the CPU run-time has been measured in one of the experiments.

Fig. 33 presents the *total* CPU run time of ConcFB, SBB, BnB-ADOPT/+ and AFB. Although all experiments were conducted on the same platform (4 Cores Intel i5, 2.9 GHz, 4 GB RAM and 64 bits Windows 7) measuring the relative performance helps to reduce some of the potential effects that the hardware may have on the evaluation. The results indicate that the total time required for BnB-ADOPT/+ is relatively higher than other algorithms. One possible explanation is that this may be due to the computational overhead involved in processing messages and priority merges which are not measured by NCLOs. Pruning irrelevant messages by BnB-ADOPT+ seems to reduce this computational overhead. An alternative explanation for these results can be the high concurrency level of the BnB-ADOPT versions which results in multiple computations carried out at the same time and a large thread management overhead, which increase the CPU run time.

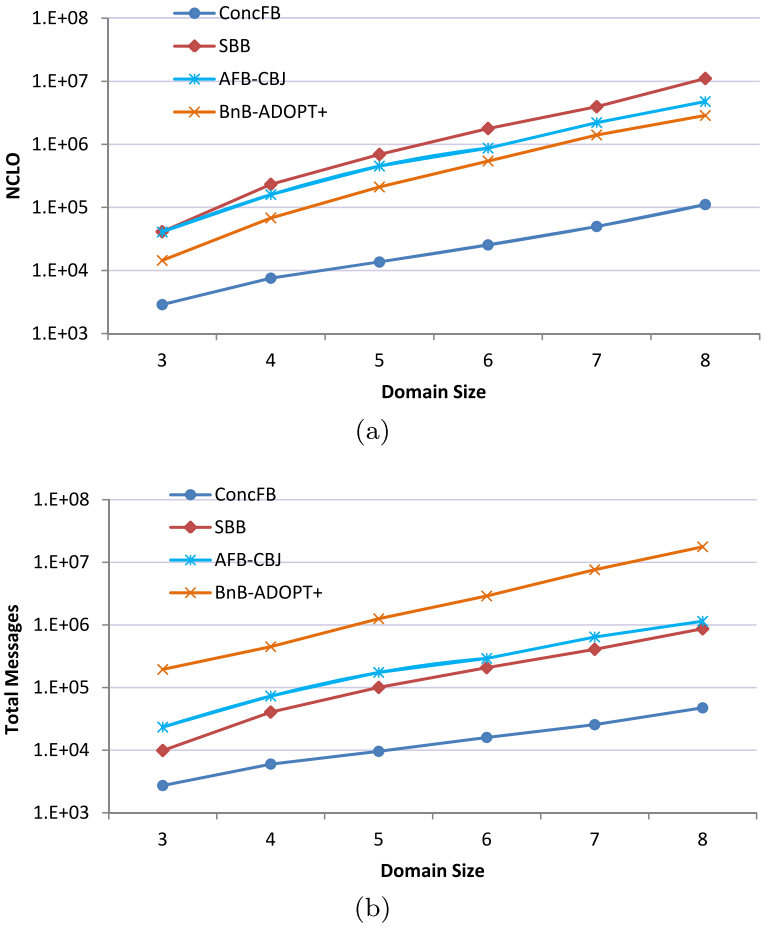


Fig. 30. Experimental results for random problems with 12 agents, 4 neighbors and domains of various sizes.

6.2. Measuring performance in high latency networks

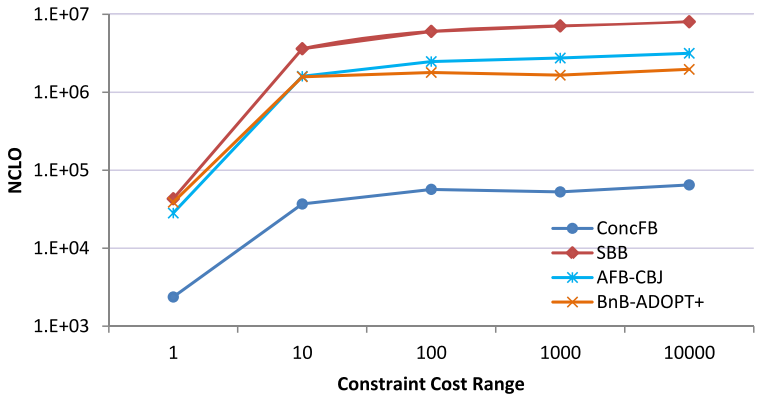
In many realistic settings communication time is dominant over computation time. In these settings a more appropriate evaluation metric is the number of non-concurrent steps (NC-Steps). This metric details the length of the longest chain of messages between agents [9,26].

Fig. 34 presents the number of NC-Steps performed by SBB, AFB, BnB-ADOPT, BnB-ADOPT+ and ConcFB. In the graph coloring setting (Fig. 34(a)), BnB-ADOPT+, ODPOP and ConcFB the number of NC-Steps is comparable, however, on the larger problems with higher density values, ConcFB's longest chain of messages is at least an order of magnitude shorter than that of all other leading algorithms (Fig. 34(b)). It should also be noted that in these problems ODPOP failed to complete within a reasonable time and was therefore omitted (cf. Fig. 28).

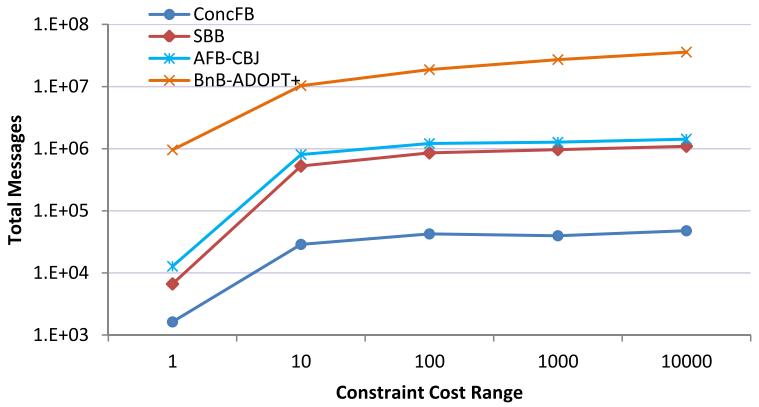
One can see that BnB-ADOPT+ improvement over BnB-ADOPT is consistent with its improvement in the total number of message (Fig. 27(b) and Fig. 29(b)). In contrast, despite SBB's significantly lower number of messages when compared to BnB-ADOPT and BnB-ADOPT+ (Fig. 29(v)) it is outperformed by BnB-ADOPT+ in terms of NC-Steps. This is due to the fact the SBB operates in a synchronous (sequential) manner whereas BnB-ADOPT and BnB-ADOPT+ are capable of sending messages concurrently.

6.3. Evaluating concurrent search

We next proceeded to evaluate the concurrent search approach by comparing ConcFB against SBB and its concurrent search implementation. We extended the Synchronous Branch and Bound algorithm so that multiple instances of this algorithm are executed on each search process. Each search process applied a different random (static) ordering and the resulting algorithm is referred to as ConcSBB. The performance evaluation of SBB, ConcFB and ConcSBB with the split depth heuristic set to n is presented in Fig. 35. Each data point represents the averaged results of 50 random problems with 12 agents, domain size of 5 and varying number of neighbors.



(a)



(b)

Fig. 31. Experimental results for random problems with 12 agents, domain size of 5 and 4 neighbors. In these problems, integer cost ranges were varied in the interval 0 to x, where $x = \{1, 10, 100, 1000, 10\,000\}$.

Not surprisingly, ConcFB outperforms ConcSBB in both measures by roughly two orders of magnitude. ConcSBB's performance is significantly better than that of SBB in terms of NCLOs but not in terms of network load. This latter point, where the large number of messages generated by ConcSBB is inconsistent with its improvement over SBB, reveals the intricate relation between concurrent search and the underlying algorithm applied to each SP.

In this setup, the majority of messages are either due to an extension of the CPA or a backtracking messages for both SBB and ConcSBB. These message types are correlated with the covered parts of the search area. An equal amount of messages implies that on the average ConcSBB covers the same percentage of the search space as SBB. The improvement in ConcSBB's NCLO over SBB demonstrates its ability to concurrently cover the same space.

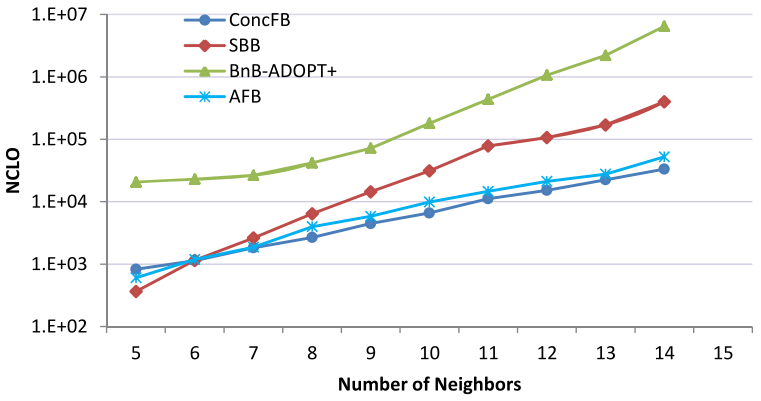
Note that despite the introduction of multiple upper bounds from different search processes a concurrent search algorithm does not necessarily cover smaller parts of the search space. This is due to unknown distribution of the costs and upper bounds within the search space.

It is worth noting that unlike ConcFB, ConcSBB is much more susceptible to search process thrashing when applying the split depth heuristic presented in Section 5.2. The limited pruning ability of ConcSBB results in significantly more assignments in lower parts of the search tree. As a result the number of split requests in ConcSBB is expected to substantially grow as the split depth is decreased, as presented in Fig. 36.

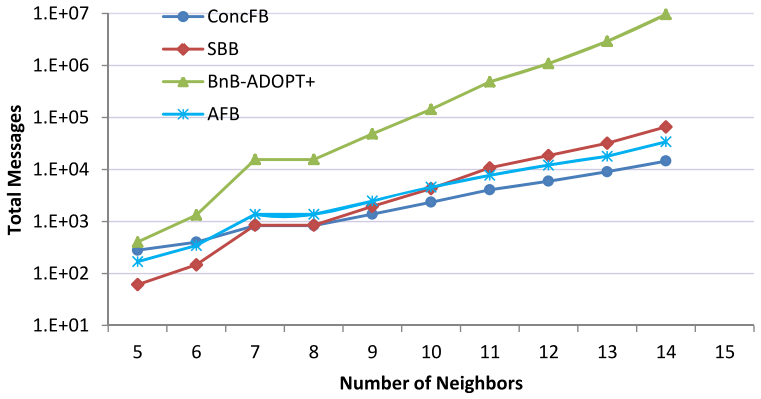
6.4. Analysis of ConcFB

Our analysis of ConcFB's components and their contribution yielded four variants of the algorithm:

- **FB:** A single synchronous forward bound search process. No concurrent search with static lexicographical ordering and No dynamic splitting.



(a)



(b)

Fig. 32. Experimental results for the sensor network problem, with varying number of tracked targets.

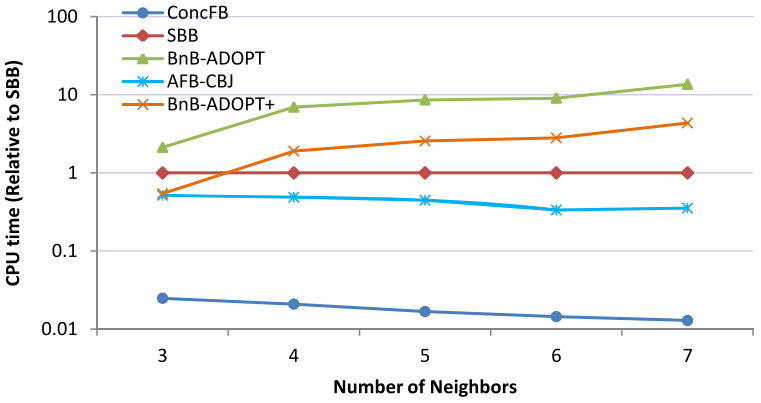
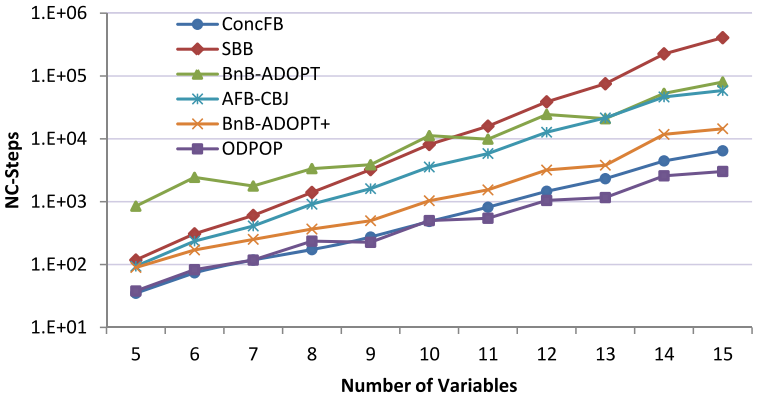
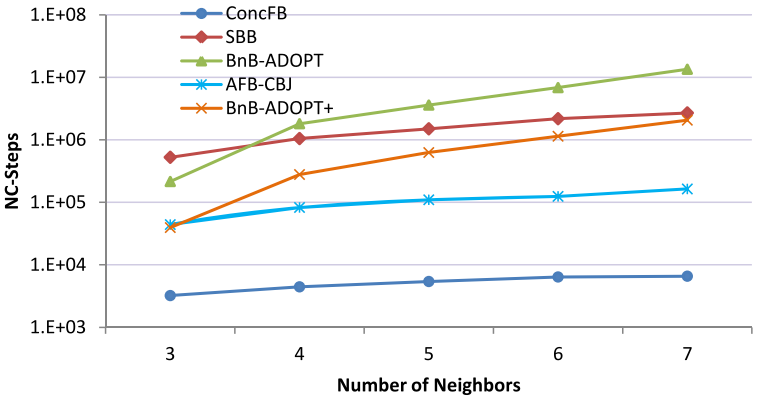


Fig. 33. CPU run time for random problems (12 agents, domain size of 5 and varying number of neighbors).

- **ConcFB-LO:** A multiple search process, forward bounding search algorithm, with lexicographical ordering. No dynamic splitting was used by this variant.
- **FB-DR:** Similar to FB but with the heuristic fail first dynamic reordering of agents.
- **ConcFB-DR:** A multiple search process, forward bounding search algorithm, with the heuristic fail first dynamic reordering. No dynamic splitting was used by this variant.
- **ConcFB-Full:** A fully featured ConcFB algorithm.



(a)



(b)

Fig. 34. The number of NC-Steps performed on graph coloring (a) and random (b) problems.

These variants were tested on even larger random problems with 15 variables, a domain size 5 and 10 neighbors per agent. Constraint costs are uniformly sampled in the range of 0 to 10000. 50 instances were generated for each set of parameter configuration and the reported results were averaged over all runs. Fig. 37 presents the difference in performance of the five ConcFB variants. One can see that dynamic ordering improves the NCCC count of FB by a factor of three. Adding additional concurrent search processes improves NCCC performance by a factor of two (both with lexicographical ordering and with dynamic reordering), and another 10 percent is gained by the dynamic splitting.

Similar trends are observed for network load, presented in Fig. 37(b). One can see that the total number of messages sent by FB decreases when dynamic ordering is introduced by roughly the same factor as NCCCs. This indicates that the fail first dynamic ordering heuristic improves the algorithm's performance through increased pruning and not necessarily through increased concurrency. When comparing the total number of messages of the multiple search processes variants and the single search process one, one can see that there is little improvement, despite the improved NCCC count (up to a factor of 2). This implies that concurrent search ConcFB improves NCCC count by increasing concurrency of computation, but does not reduce the total amount of computation performed.

One can see that the dynamic reordering heuristic (Section 5.1.2) introduces a significant improvement to the naive Synchronous Forward Bounding algorithm. Fig. 38 presents a comparison of SFB with dynamic reordering (FB-DR) and SBB, ConcFB, BnB-ADOPT+ and AFB. As before, the experimental setup included random problems with 12 agents, a domain size of 5 and varying number of agents.

FB-DR provides a significant improvement over all state of the art algorithms. In comparison to ConcFB, FB-DR generates roughly the same number of messages but does significantly more NCLOs.

This can be understood by realizing that both ConcFB and FB-DR share the same powerful pruning abilities and thus cover a similar amount of the search space. The introduction of SP splitting by ConcFB results in greater concurrency and fewer NCLOs in comparison to FB-DR (this is in accordance to the difference in SBB and ConcSBB discussed in Section 6.3).

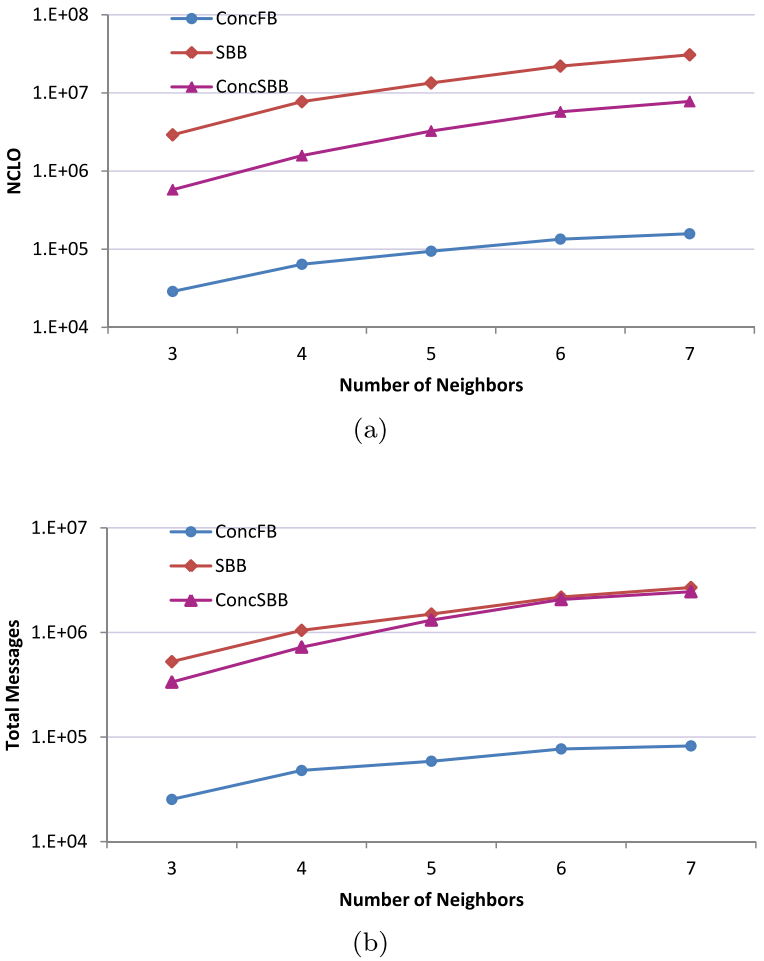


Fig. 35. Experimental results for random problems with 12 agents, domain size of 5 and varying number of neighbors.

Split depth	n	$n-1$	$n-2$	$n-3$	$n-4$	$n-5$	$n-6$
Max SPs	1161	22 529	134 534	243 467	296 046	308 274	316 837

Fig. 36. The maximal number of concurrent SPs in ConcSBB as a function of split depth request with 12 agents, a domain size 5 and 4 neighbors per agent.

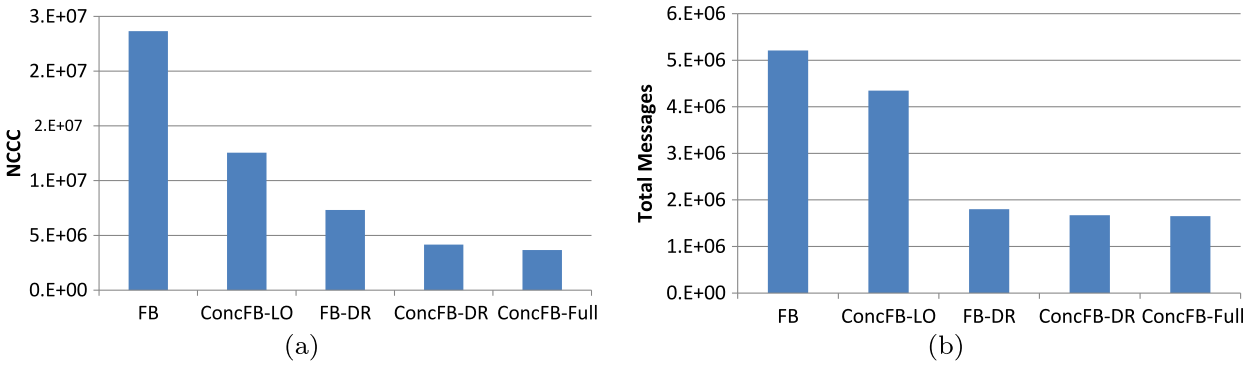
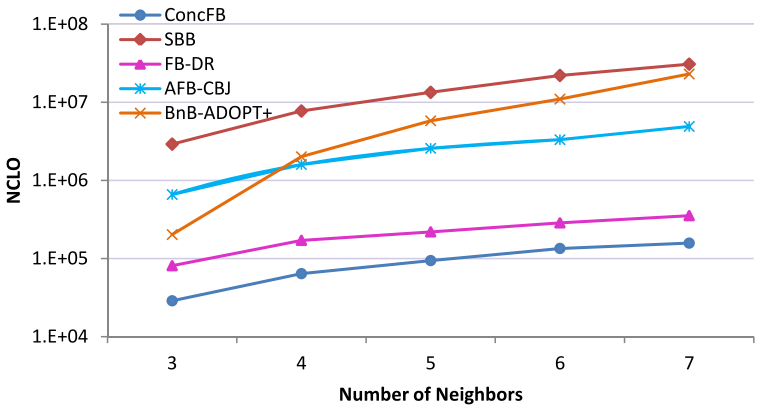
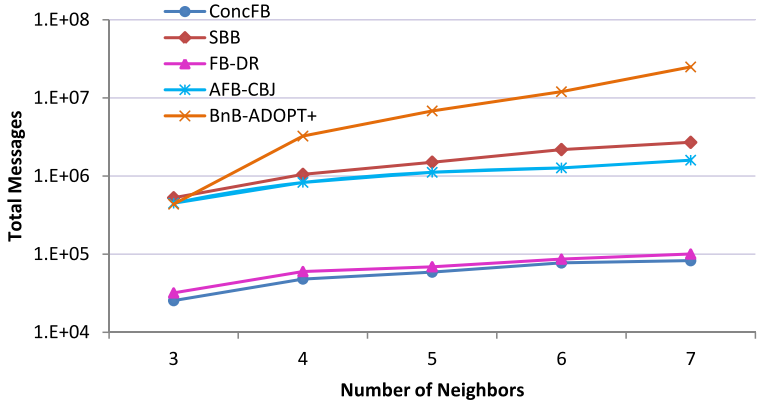


Fig. 37. ConcFb analysis.

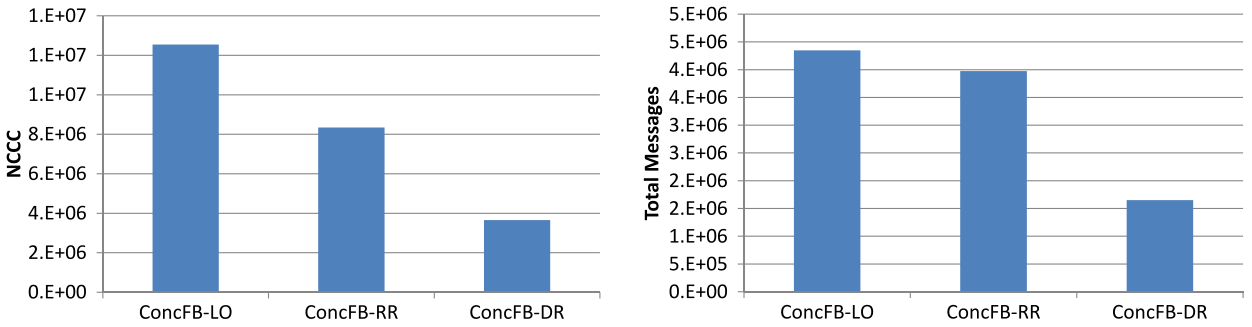


(a)



(b)

Fig. 38. Performance of SFB with dynamic reordering (FB-DR) on problems with 12 agents, domain size of 5 and varying number of neighbors.



(a) (b)

Fig. 39. Performance of ConcFB with different ordering scheme on problems with 15 agents, domain size of 5 and 10 neighbors.

We further analyzed the empirical impact of the chosen dynamic ordering heuristic with respect to the static lexicographical ordering and random dynamic ordering. The evaluation included ConcFB with static lexicographical ordering (ConcFB-LO), the dynamic fail first ordering heuristic described in Section 5.1.2 (ConcFB-DR) and dynamic random ordering (ConcFB-RR). 50 instances of random problems with 15 agents, a domain size of 5 and 10 neighbors to each agent were examined and the results of this evaluation are presented in Fig. 39.

The results demonstrates the impact of dynamic ordering heuristics. As demonstrated in Fig. 16(b) the introduction of random ordering balances the load on agents which enables higher concurrency and reduces the number of NCCCs.

Nonetheless, the portion of the search space covered with random ordering remains roughly the same as that covered in the lexicographical ordering version of ConcFB and hence the number of messages passed between agents is similar.

The proposed dynamic ordering attempts to further improve on the random ordering scheme by guiding the search to more promising parts of the search space. This enables ConcFB-DR to better prune the search and results in a lower number of NCCCs and reduced network load.

7. Conclusion

The present paper presents ConcFB – a Concurrent Forward Bounding algorithm for solving DCOPs. ConcFB combines two powerful techniques: Concurrent search and synchronized Forward Bounding. The combined approach results in an algorithm which is characterized by a high degree of concurrency and is capable of rapid pruning of the search space.

Although any DCOP algorithm may be applied to the individual search processes, the present work focuses on SFB. SFB has two important features that make it suitable for concurrent search: powerful pruning abilities and synchronous progress. In particular, the latter feature guarantees that any split request received by the agent is relevant to the system's overall state. This is not necessarily the case when dealing with asynchronous algorithms. For example, in Asynchronous Forward Bounding and in BnB-ADOPT a split request may be triggered by an agent holding an invalid view of the current assignment. In other words, the partial assignment held by the agent requesting a split can be outdated. This raises significant challenges in the application of Concurrent Search to asynchronous algorithms which is left open for future research.

The present work also demonstrates the benefits of applying dynamic reordering to DCOP algorithms. Some work on reordering pseudo tree based algorithms was presented by [24]. However, there is no work specifying how to adapt these to asynchronous algorithms which use a time stamping mechanism such as AFB and BnB-ADOPT. This stems from the mechanism's reliance on the agents total order to infer the relation between two different time stamps – a point we intend to pursue in future work.

To conclude, the benefits of the proposed concurrent search approach are threefold:

1. The agents applying forward bounding on disjoint parts of the problem share upper bound information across multiple search processes.
2. Each search process is independent of all others and can apply different search heuristics.
3. Additional search processes can easily be added to maintain a preferred level of concurrency.

Three enhancements to the basic ConcFB algorithm are presented. The first enhancement introduces a random ordering heuristic for agents which significantly balances the work load. The second one introduces a fail first heuristic for DCOPs which results in a significant improvement of the algorithm's pruning abilities. Finally, dynamic splitting – the ability to spawn new search processes – is added and shown to further improve ConcFB's performance.

ConcFB is proved to terminate with an optimal solution and its performance is extensively evaluated. An extensive set of experiments on both structured and unstructured problems evaluates ConcFB's performance against other state of the art DCOP algorithms. Additionally, a new concurrent algorithm – ConcSBB – was implemented to provide further insights on the workings of a concurrent search algorithm with multiple Search Processes. Measuring performance in terms of NCLOs, NC-Steps, network load and CPU time, ConcFB is shown to outperform BnB-ADOPT, BnB-ADOPT+, AFB-CBJ, ODPOP and ConcSBB.

A second set of experiments which quantifies the gain of different enhancements to the ConcFB algorithm is also presented. It is shown that the combination of a Fail First heuristic with multiple search processes and a dynamic splitting scheme can improve the algorithm's performance by up to an order of magnitude.

References

[1] Syed Muhammad Ali, Sven Koenig, Milind Tambe, Preprocessing techniques for accelerating the DCOP algorithm ADOPT, in: 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05), Utrecht, The Netherlands, July 2005, pp. 1041–1048.
[2] Ismel Brito, Amnon Meisels, Pedro Meseguer, Roie Zivan, Distributed constraint satisfaction with partially known constraints, Constraints 14 (2008) 199–234.
[3] David A. Burke, Kenneth N. Brown, Using relaxations to improve search in distributed constraint optimisation, Artificial Intelligence Review 28 (2007) 35–50.
[4] Anton Chechetka, Katia P. Sycara, No-commitment branch and bound search for distributed constraint optimization, in: 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06), Hakodate, Japan, May 2006, pp. 1427–1429.
[5] Rina Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (3) (1990) 273–312.
[6] Rina Dechter, Constraint Processing, Elsevier Morgan Kaufmann, ISBN 978-1-55860-890-0, 2003.
[7] Rina Dechter, Robert Mateescu, AND/OR search spaces for graphical models, Artificial Intelligence 171 (2–3) (2007) 73–106.
[8] Eugene C. Freuder, Michael J. Quinn, Taking advantage of stable sets of variables in constraint satisfaction problems, in: 9th International Joint Conference on Artificial Intelligence (IJCAI'85), Los Angeles, USA, 1985, pp. 1076–1078.
[9] Amir Gershman, Amnon Meisels, Roie Zivan, Asynchronous forward bounding, Journal of Artificial Intelligence Research 34 (2009) 25–46.
[10] Tal Grinshpoun, Amnon Meisels, Completeness and performance of the APO algorithm, Journal of Artificial Intelligence Research 33 (2008) 223–258.
[11] Patricia Gutierrez, Pedro Meseguer, Saving redundant messages in BnB-ADOPT, in: 24th AAAI Conference on Artificial Intelligence (AAAI'10), July 2010, pp. 1259–1260.
[12] Katsutoshi Hirayama, Makoto Yokoo, Distributed partial constraint satisfaction problem, in: 3rd International Conference on Principles and Practice of Constraint Programming (CP'97), Linz, Austria, 1997, pp. 222–236.

[13] Robert Junges, Ana L.C. Bazzan, Evaluating the performance of DCOP algorithms in a real world, dynamic problem, in: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08), Estoril, Portugal, May 2008, ISBN 978-0-9817381-1-6, 2008, pp. 599–606.
[14] Viliam Lisý, Roie Zivan, Katia P. Sycara, Michal Pechoucek, Deception in networks of mobile sensing agents, in: 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10), Toronto, Canada, 2010, pp. 1031–1038.
[15] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann, ISBN 1-55860-348-4, 1996.
[16] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, Pradeep Varakantham, Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling, in: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), New York, NY, USA, 2004, pp. 310–317.
[17] Radu Marinescu, Rina Dechter, AND/OR branch-and-bound search for combinatorial optimization in graphical models, Artificial Intelligence 173 (16–17) (2009) 1457–1491.
[18] Robert Mateescu, Rina Dechter, AND/OR cutset conditioning, in: 19th International Joint Conference on Artificial Intelligence (IJCAI'05), San Francisco, CA, USA, 2005, pp. 230–235.
[19] Amnon Meisels, Distributed Search by Constrained Agents: Algorithms, Performance, Communication, Springer-Verlag, ISBN 1848000391, 2007.
[20] Amnon Meisels, Igor Razgon, Distributed forward-checking with conflict-based backjumping and dynamic ordering, in: Workshop on Cooperative Solvers in Constraint Programming (CoSolv'02), Ithaca, NY, USA, 2002.
[21] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, Makoto Yokoo, ADOPT: asynchronous distributed constraints optimization with quality guarantees, Artificial Intelligence 161 (1–2) (2005) 149–180.
[22] Adrian Petcu, Boi Faltings, A scalable method for multiagent constraint optimization, in: 19th International Joint Conference on Artificial Intelligence (IJCAI'05), Edinburgh, Scotland, UK, August 2005, pp. 266–271.
[23] Adrian Petcu, Boi Faltings, ODPOP: An algorithm for open/distributed constraint optimization, in: 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (AAAI'06), Boston, MA, USA, July 2006, pp. 703–708.
[24] Marius-Calin Silaghi, Makoto Yokoo, Dynamic DFS tree in ADOPT-ing, in: 22nd AAAI Conference on Artificial Intelligence (AAAI'07), Vancouver, British, Columbia, Canada, July 2007, pp. 763–769.
[25] Ruben Stranders, Alessandro Farinelli, Alex Rogers, Nick R. Jennings, Decentralised coordination of continuously valued control parameters using the max-sum algorithm, in: Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'09), Budapest, Hungary, May 2009, pp. 601–608.
[26] William Yeoh, Ariel Felner, Sven Koenig, BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm, Journal of Artificial Intelligence Research 38 (2010) 85–133.
[27] Makoto Yokoo, Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems, Springer-Verlag, ISBN 3-540-67596-5, 2000.
[28] Makoto Yokoo, Algorithms for distributed constraint satisfaction problems: A review, Autonomous Agents and Multi-Agent Systems 3 (2000) 198–212.
[29] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, Kazuhiro Kuwabara, Distributed constraint satisfaction problem: Formalization and algorithms, IEEE Transactions on Data and Knowledge Engineering 10 (1998) 673–685.
[30] Roie Zivan, Amnon Meisels, Dynamic ordering for asynchronous backtracking on DisCSPs, in: 11th International Conference on Principles and Practice of Constraint Programming (CP'05), Sitges (Barcelona), Spain, October 2005, pp. 32–46.
[31] Roie Zivan, Amnon Meisels, Concurrent search for distributed CSPs, Artificial Intelligence 170 (4–5) (2006) 440–461.
[32] Roie Zivan, Amnon Meisels, Message delay and DisCSP search algorithms, Annals of Mathematics and Artificial Intelligence 46 (2006) 415–439.
[33] Roie Zivan, Amnon Meisels, Dynamic ordering for asynchronous backtracking on DisCSPs, Constraints 11 (2006) 179–197.
[34] Roie Zivan, Moshe Zazone, Amnon Meisels, Min-domain ordering for asynchronous backtracking, in: 13th International Conference on Principles and Practice of Constraint Programming (CP'07), Rhode Island, USA, September 2007, pp. 758–772.