Forward Bounding on a Pseudo Tree for DCOPs

Omer Litov and Amnon Meisels

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract. Distributed constraint optimization problems (DCOP) are a popular way of representing and solving multi-agent coordination problems. Complete search algorithms for solving DCOPs fall into two groups: algorithms that use a pseudo tree (*ADOPT*[10], *bnb-ADOPT*[12], *NCBB*[1]), and algorithms that do not use one (*AFB*[4], *ConcFB*[11]). Each of these groups has different limitations. The algorithms that use a pseudo tree are dependent upon the existence of a good (minimal depth) pseudo tree. However, highly constrained problems tend to have bad trees. On the other hand, non pseudo-tree algorithms can fail to take advantage of the possibility of dividing the problem into smaller sub-problems, which can boost the performance of the algorithm. A new algorithm, *PT-SFB*, is presented which incorporates a hybrid approach of the two groups, in an attempt to take the best from each one. Every agent identifies when it is possible to divide the remaining problem (given a partial assignment) into smaller sub-problems that can be solved independently. When the remaining problem cannot be divided at a given point, a sequential assignment will take place, which outperforms pseudo-tree algorithms in these cases. In all cases the proposed algorithm performs forward-bounding. An extensive experimental evaluation is presented, comparing *PT-SFB* to several state-of-the-art algorithms.

1 Introduction

The distributed constraint optimization problem is a well known framework for representing and solving multi-agent coordination problems, such as DCOPs. A distributed constraints optimization problem (DCOP) consists of a set of agents that need to take on a value, which may be constrained with value assignments of other agents. The goal of the agents is to find a full assignment, with a minimal sum of costs. Complete algorithms for solving a DCOPs can be divided into two main categories: algorithms that use a pseudo tree, and algorithms that do not use one. A pseudo tree is a DFS tree of the constraints graph, where each agent is constrained with an ancestor or a descendent, but not with an agent from a different sub-tree. A pseudo tree thus divides the problem into separate sub-problems, which can be solved simultaneously, independent of one another. Such a division improves performance considerably. On the other hand, finding the best DFS (minimal depth) is NP-hard, and on heavily constrained problems, a good tree may not exist. Even when a good DFS tree is found, it may still

have parts that consist of long chains (which do not divide the problem into sub-problems).

The algorithms that use a pseudo tree are designed for "good" trees, and tend to perform poorly when the problems are highly constrained. The second group of algorithms does not use a pseudo-tree and is not effected by the existence of a "good" tree. Such algorithms may outperform pseudo tree based algorithms on highly constrained problems (cf. [5, 11]). On the other hand, such algorithms fail to take advantage of the possibility of dividing the problem into smaller sub-problems, and solve them simultaneously and independently. Another disadvantage of non-tree algorithms compared to tree-based algorithms, is that it requires communication links between all agents, and not only with constrained agents.

An algorithm that can combine the advantages of both types of algorithms may be more efficient. The present study proposes the *Pseudo Tree Synchronous Forward Bound* (*PT-SFB*) algorithm, which implements a variation of Synchronous Forward Bounding on a pseudo tree. The algorithm has 3 major properties:

1. When possible, it will divide the problem into sub-problems.
2. On parts of the tree which are essentially a chain, it will act like SFB, producing satisfactory results even when the pseudo tree is less than ideal.
3. It enforces a strict bound update, which increases the pruning of the problem.

An algorithm that combine the two approches for constraints *satisfaction* (DCSP) was proposed in [3]. It has a very different structure from *PT-SFB*, and it is based on a variation of *AFC* [9] with no-goods, which makes it inapplicable to DCOP.

Additionally, *PT-SFB* enables a compromise between the communication link requirements in tree-based and in non-tree algorithms, since an agent is unaware of agents in a different sub-tree. An extensive experimental comparison between *PT-SFB* and several other state of the art algorithms is presented and demonstrates that the proposed algorithm outperforms former algorithms on most of the problems tested. The *PT-SFB* algorithm provides a framework for extending other algorithms, such as *pseudo-tree-ConcFB*, which have the potential to dominate every state of the art algorithm on a large variety of problem densities.

2 DCOP

A distributed constraint optimization problem (DCOP) is composed of a set of agents $A_1, \ldots, A_n$, each holding a set of constrained variables. Each variable X_i has a domain D_i - a set of possible values. Constraints (or relations) exist between variables. Each constraint involves some variables (possibly belonging to different agents) and defines a non-negative cost for every possible value combination of these variables. A binary constraint is a constraint involving only two variables. An assignment is a pair including a variable, and a value from that variable's

domain. A partial assignment (PA) is a set of assignments, in which each variable appears at most once. The cost of a partial assignment is computed using all constraints that involve only variables that appear in the partial assignment. Each such constraint defines some cost for the value assignments detailed in the partial assignment. These costs are calculated, and the sum is noted as the cost of the partial assignment. A full assignment is a partial assignment that includes all the variables. A solution to the DCOP is a full assignment of minimal cost.

In this paper, we will assume each agent is assigned a single variable, and use the term "agent" and "variable" interchangeably. We will assume that constraints are at most binary and the delay in delivering a message is finite. These assumptions are commonly used in DCOP algorithms[10, 13].

3 PT-SFB – Pseudo Tree Synchronous Forward Bounding

3.1 Simplified PT-SFB

The algorithm *Pseudo-tree Synchronous Forward-bounding (PT-SFB)* starts by distributively constructing a DFS tree of the constraints graph, also referred to as a pseudo tree. Several distributed algorithms for forming DFS trees exist [2, 7]. Since a DFS tree has no cross edges, any two agents that are part of separate sub-trees (i.e., do not have an ancestor\descendant relationship) cannot be constrained. This property means that two separate sub-trees can be solved independently of one another, since there are no constraints between them. The use of pseudo trees in DCOP was first proposed in the algorithm *ADOPT* [10]. Agent j is called an ancestor of agent i in the tree if j is the parent of agent i, or if j is recursively the parent of an ancestor of agent i. If agent j is an ancestor of agent i, than agent i is called a descendant of agent j. We define pseudo-children of i (marked $PC(i)$) as in [10] as all descendants of i which are have a direct constraint with agent i. The full use of the pseudo tree in *PT-SFB* will be described later.

The *PT-SFB* algorithm uses 6 types of messages:

1. **CPA** - a message containing a partial assignment, sent from an agent to its child.
2. **Backtrack** - sent from an agent to its parent when the agent has exhausted its search space.
3. **LB_Request** - sent from an agent to its descendants, requesting a lower bound estimate for a given partial assignment.
4. **LB_Report** - a lower bound estimate, sent as a response to a LB_Request.
5. **UB** - a message sent from an agent to specific ancestors, containing the cost of a solution found to the agent's sub-problem.
6. **Reduce_UB** - sent from an agent to its child, reducing the upper bound of the child in order to increase pruning.

After constructing the pseudo tree, the algorithm assigns values sequentially to every agent, starting with the root of the tree, down to the leaves. The assignments are done only by agents holding a **CPA**, which is a message received

from the agent's parent in the pseudo tree. A **CPA** contains the Current Partial Assignment of all ancestors of the current agent. In addition a **CPA** message contains an Upper Bound (UB) on the solution cost, and an accumulated cost of constraints between all assignments it contains. When an agent i extends the CPA with an assignment to its variable, it sends a **LB_Request** message, containing a copy of the CPA to all of its descendants in the pseudo tree. This is the method of performing Synchronous Forward-Bounding (SFB). An agent j receiving a **LB_Request** calculates a lower bound on the added cost caused by extending the CPA with an assignment to its variable, by computing $\min_{d \in D_j} \sum_{k \in CPA} Cost(X_k = val(k), X_j = d)$, where $val(k)$ is the value assignment of X_k in the received CPA. This lower bound is sent back in a **LB_Report** message. The agent i which holds the original **CPA** can now check whether the sum of all of the reported costs and the CPA cost exceeds the known UB. If so, it will try to assign a different value to its variable, or send a **Backtrack** message to its parent if no value assignment is possible. If the sum of the **LB_Report** message and the CPA cost does not exceed the UB, the CPA can safely be sent forward to the agent's children on the pseudo-tree.

An agent sends forward a CPA to its children in different ways, according to the number of children it has. If the agent only has one child, then the **CPA** sent to its child is clear. In this situation, the agent sends to its child a **CPA** with the UB currently known to it. The cost on this CPA is equal to the cost the sending agent received from its parent, plus the added cost of assigning a value to its variable. In the case of a single child, the assigning agent waits for the response to its **LB_Request** before sending its CPA forward. This is a realization of the *synchronous* forward-bounding (SFB) mechanism. If, on the other hand, the agent has more than one child it effectively divides the problem into a separate sub-problem for each one of its children on the pseudo-tree. In order to do so, it sends them a **CPA** with cost=0, and an upper bound that is computed separately for each child, as follows

$$UB_j = UB_i - CPA\_Cost - \sum_{k \neq j \in children(i)} Sub\_Tree\_LB(k) \qquad (1)$$

where i is the sender, j is the child, and $Sub\_Tree\_LB(k)$ is the current best known lower bound on the cost of the sub tree rooted at each child k (different than j). The best known $Sub\_Tree\_LB(k)$ starts as the sum of all **LB_Reports** received from members of that sub tree. When that sub tree finishes its search, the value of $Sub\_Tree\_LB(k)$ will be changed to the cost of the best solution of that sub tree. The next subsections describe the method for obtaining a value for $Sub\_Tree\_LB(k)$ before all **LB_Reports** have been received. The new values of *cost* and UB enable agents in different sub trees to solve their sub problems independently. As can be seen from the above description, when an assigning agent has more than a single child on the pseudo-tree it performs a different form of forward-bounding. Not synchronous, but asynchronous among the different sub-trees rooted at the assigning agent. In other words, it only waits for responses to its requests for lower-bounds from the given sub-tree to arrive before sending its CPA forward.

Upper Bounds are updated both up the tree and down the tree. Let us examine the different methods of UB propagation between agents, according to their number of children. When an agent has exactly one child, its upper bound UB is the same as that of its child's UB, which means that all agents within a chain in the pseudo tree share the same UB. Define $Latest\_Divider_i$ as the latest ancestor of agent i that has more than one child. In other words, the $Latest\_Divider_i$ is the closest ancestor of i, who divided the problem into sub-problems. All agents who are ancestors of i **and** descendants of $Latest\_Divider_i$ are part of a chain in the pseudo-tree containing agent i.

When a leaf agent extends a **CPA** with an assignment to its variable, in effect it finds a new UB. That UB is relevant to all agents in its chain. The agent sends **UB** messages up to all members of its chain, up to the $Latest\_Divider_i$. Agents in i's chain will simply update their UB, but the $Latest\_Divider_i$ needs to sum the best UB values received from each one of its children. The $Latest\_Divider_i$ will then send its new UB to all members of its own chain, up to its $Latest\_Divider$. This process is carried out until the root is able to compute the UB of the entire problem.

Each agent holds two variables for holding the value of the upper bound: $Received\_UB$, which is the upper bound received from the agent's parent, and UB which is the minimum between $Received\_UB$ and the upper bound computation described above. When an agent with more than one child receives a **Backtrack** message from one of its children, it first updates the Sub_Tree_LB of that child to be the latest UB received from that sub tree. This holds because the latest UB received from a sub-tree is the best solution found, and since the child has backtracked, no better solution exists.

This change affects the UB of its other children, according to equation 1 above. The agent then sends **Reduce_UB** messages to its other children, with the difference between the new computed UB_j and the latest UB_j sent to agent (son) j. An agent receiving a **Reduce_UB** message updates its $Received\_UB$ and checks if it is lower than its current UB. If it is, it updates its UB and sends **Reduce_UB** messages to its own children. This tight two way bound update greatly increases the pruning of the search tree.

Finally, when an agent receives a **Backtrack** message from one of its children, it does not wait for similar messages from its other children. Instead, it sends forward a **CPA** message with the next assignment to its variable. This means that agents do not have to wait until other sub-trees finish their computations, which increases concurrency. As a result, when the other children backtrack they will potentially have a better UB, since the Sub_Tree_LB of the first agent is now more accurate.

3.2 Improved PT-SFB

The simplified version of the *PT-SFB* algorithm described above is extended in several ways which greatly increase its efficiency. The first improvement is a method to reduce the number of **LB_Request** and **LB_Report** messages.

Let us examine a situation where agent j, which is a descendant of the current agent i, is not constrained with i. Obviously the **LB_Report** massage sent by j to i is exactly the same as the **LB_Report** sent by j to i's parent. So instead of sending **LB_Request** messages to all descendants, an agent only sends **LB_Request** messages to descendants it is constrained with. The **LB_Reports** of all other descendants are received from the agent parent together with the **CPA** message. In order to do so, the agent maintains two separate data structures, one for the **LB_Reports** received from its parent, and a second one, called $LB\_Adjustments$ only for constrained descendants. The $LB\_Adjustment$ only contains the difference between the **LB_Reports** received from the agent's parent, and the **LB_Report** received directly from the constrained descendant. The two separate data structures offer an additional benefit for computing Sub_Tree_LB, since even when an agent hasn't yet received **LB_Reports** from the given sub-tree, the Sub_Tree_LB can be computed based on the **LB_Reports** received from the agent parent. This allows a tighter UB computation for the children of that agent. We mark $Sub\_Tree\_LB(j, val)$, the calculation described above, to distinguish it from $Sub\_Tree\_LB(j)$ which refers to just the sum of $LB\_Reports$ received from the agent's parent.

The second improvement helps reduce computations. We maintain another data structure called $Local\_Cost$, which is defined as follows,

$$Local\_Cost_i(j, d) = \sum_{k \in ancestors(j) \cup \{j\}} Cost(X_k = val(k), X_i = d) \qquad (2)$$

where $val(k)$ is the value assignment of agent k in the latest **LB_Report** sent from k to i, and $d \in D_i$. In other words, $Local\_Cost_i(j, d)$ is the sum of the cost of all constraints between agent i with value assignment d, and its ancestors, from the root down to agent j, with the latest known value assignments. When agent i receives a **LB_Request** message from agent j, for each $d \in D_i$ it will update

$$Local\_Cost_i(j, d) = Local\_Cost_i(ECA_i(j), d) + Cost(X_j = val(j), X_i = d)$$

where $ECA_i(j)$ (Earlier Constrained ancestor) is the latest ancestor of j which is constrained with agent i. The value of the **LB_Report** massage sent by agent i to agent j is $\min_{d \in D_i} Local\_Cost_i(j, d)$. This method provides several benefits. First, when an agent receives a **LB_Report** message it only needs to check its constraints with a single agent, instead of every assigned agent. Secondly, it means that **LB_Request** messages now contain only a single value assignment. Finally, when an agent receives a **CPA** message it needs no computations, since its added cost for every value d is already calculated and stored at $Local\_Cost_i(parent(i), d)$ (marked for simplicity as $Local\_Cost_i(d)$).

The third improvement relates to the structure of the pseudo tree. The desired pseudo tree is as shallow as possible, since a shallow tree can be divided into smaller independent sup-problems. In contrast to a deep tree, which consists of bigger and fewer sub-problems. Since finding the best pseudo tree is NP-hard,

the proposed algorithm uses a simple heuristic. It simultaneously starts a distributed DFS algorithm at each agent, which results in a number of DFS trees, each one rooted at a different agent. Then it selects the tree with the smallest height.

The final improvement to the simple version of the PT-SFB algorithm is a simple value ordering heuristic. It selects first to assign the value with the lowest $Local\_Cost$.

3.3 Pseudo Code

As mentioned, the algorithm starts with constructing a pseudo tree. When the constructing of the tree is finished the root of the tree calls the "when receive **CPA**" procedure with an empty partial assignment, and a high enough UB. The UB cannot be infinity, in order for the child UB computation to work, so $UB = max\_cost^n$ can be used.

When receiving a **CPA** message, the agent saves all data received with the message for later use. The agent then calculates Sub_Tree_LB (line 3-4) for all of its children ($C(i)$), as described above. Then it calculates a lower bound on assigning each one of its values (line 5-6). For each child j it calls $Assign\ Next\ Val(null, j)$ in order to continue the search independently in every child. For a leaf in the pseudo tree, $assignNextVal$ is called with no child and no prior value.

Procedure 1 when received (**CPA**, $cost$, UB, $LBReports$)

```
1: save all received variables
2: UpperBound ← ReceivedUB
3: for all j ∈ C(i) do
4:     SubTreeLB(j) ← Σ_{k∈descendants(j)} LBReports
5: for all val ∈ D_i do
6:     Domain_LB_i(d) ← cost + Local_Cost_i(d) + Σ_{j∈C(i)} Sub_Tree_LB_i(j)
7: for all j ∈ C(i) do
8:     call assignNextVal(null, j)
9: if |C(i)| = 0 then
10:    call assignNextVal(null, null)
```

In $assign\ next\ val(pVal, j)$, the next $val \in D_i$, after $pVal$ (the prior value of agent i, searched by j) is selected (line 1). A good heuristic for value order was mentioned, but in any heuristic it is preferable that the order remain constant between the val searched by different children, in order to improve UB computation. If no unsearched values for child j exist, or no remaining val such that $Domain\_LB(val) < UB$, $null$ is returned. When all of the children have finished searching all relevant values, a **Backtrack** message is sent to i's parent. If a relevant value is found, a **LB Request** is sent to all descendants of j, which are also Pseudo Children of agent i (constrained descendants of i) (line 5). When

all **LB Report** messages has arrived from that sub-tree, the $LB\_Adjustment$, $Sub\_Tree\_LB\_Adjustment$ and $Domain\_LB$ are computed (line 6-11). The lower bound change effects the UB computation of other children which are searching on the same val, according to equation 1. They are informed by a **Reduce UB** message (line 12-14). Finally a check is made to determine if the chosen assignment can still provide a solution with a smaller cost than UB. If it can, a call to the procedure *continue Assignment* is made in order to send the **CPA** forward, if it cannot, the next value is chosen by calling $assign\ next\ val(val, j)$.

Procedure 2 Assign Next Val($pVal$, j)

1: $val \leftarrow next\_Val(j)$
2: **if** $val = null$ **and** j is the last child for which $val = null$ **then**
3: send(**Backtrack**) to i's parent
4: **else**
5: send(**LB_Request**, val) to all $k \in PC(i) \cap Descendants(j)$
6: when all **LB_Report** messages from that sub-tree arrives, proceed:
7: **for all** $k \in PC(i) \cap Descendants(j)$ **do**
8: $LB\_Adjustment(k) \leftarrow LB\_Report(k) - parent\_LB\_Report(k)$
9: $Sub\_Tree\_LB\_Adjustment(j) \leftarrow \sum_{k \in PC(i) \cap Descendants(j)} LB\_Adjustment(k)$
10: $srchVal(j) \leftarrow val$
11: $Domain\_LB(val) \leftarrow Domain\_LB(val) + Sub\_Tree\_LB\_Adjustment(j)$
12: **for all** $k \neq j \in C(i)$ **do**
13: **if** $srchVal(k) = val$ **then**
14: send(**Reduce_UB**, $Sub\_Tree\_LB\_Adjustment$(j)) to k
15: **if** $Domain\_LB(val) < UB$ **then**
16: call $continueAssignment(val, j)$
17: **else**
18: call $assignNextVal(val, j)$

Procedure 3 when received (**LB Request**, val)

1: $k \leftarrow ECA_i(sender)$
2: **for all** $d \in D_i$ **do**
3: $Local\_Cost_i(sender, d) \leftarrow Local\_Cost_i(k, d) + Cost(X_{sender} = val, X_i = d)$
4: send(**LB Report**, $\min_{d \in D_i} Local\_Cost_i(sender, d)$) to $sender$

When a leaf agent calls the **continue Assignment** procedure (line 2-6), it means that it has reached a solution to the sub problem it is in. It computes the UB, and send it to all members of its chain, up to the $Latest\_Divider_i$, before attempting a new value. A non-leaf agent, starts by combining the **LB Reports** that came with the **CPA** from its parent, with the reports it received directly from its descendants. An agent with one child (a chain) simply adds his local cost and send the **CPA** forward (line 9-11). On the other hand, an

Procedure 4 continue Assignment$(val,\ j)$

1: $newCPA \leftarrow$ copy$(Received_CPA)$ + $(X_i = val)$
2: **if** $|C(i)| = 0$ **then**
3: $UB \leftarrow Received\_Cost + Local\_Cost_i(val)$
4: **for** every ancestor k up to $Latest\_Divider_i$ **do**
5: send (**UB**, $newCPA$, UB) to k
6: call $assignNextVal(val, null)$
7: **else**
8: $new\_LB\_Reports \leftarrow$ combine$(LB_Reports,\ LB_Adjustment)$
9: **if** $|C(i) = 1|$ **then**
10: $newCost \leftarrow Received\_Cost + Local\_Cost_i(val)$
11: send(**CPA**, $newCPA, newCost, UB, new\_LB\_Reports$)
12: **else**
13: $newCost \leftarrow 0$
14: $newUB \leftarrow UB - Received\_Cost - Local\_Cost_i - \sum_{k \neq j \in C(i)} Sub\_Tree\_LB(k, val)$
15: $Sub\_Tree\_UB(j, val) \leftarrow newUB$
16: send(**CPA**, $newCPA, newCost, newUB, new\_LB\_Reports$)

agent which has multiple children computes a new UB for that child, where $Sub\_Tree\_LB(k, val)$ is the best known lower bound for every $k \in C(i)$ for that specific val, as described above. This effectively divides the problem into smaller sub-problems. The data structure Sub_Tree_UB saves the best known UB for each of the children of i, for every $d \in D_i$, for later use.

Procedure 5 when received (**UB**, CPA, $newUB$)

1: **if** $|C(i)| = 1$ **then**
2: $UB \leftarrow newUB$
3: **else**
4: combine CPA with best CPA received from other children for the same value
5: $child \leftarrow$ the child which is the ancestor of $sender$
6: $val \leftarrow srchVal(child)$
7: $Sub\_Tree\_UB(child, val) \leftarrow newUB$
8: $totalUB \leftarrow Received\_Cost + Local\_Cost_i(val) + \sum_{j \in C(i)} Sub\_Tree\_UB(j, val)$
9: **if** $totalUB < UB$ **then**
10: $UB \leftarrow totalUB$
11: **for** every ancestor k up to $Latest\_Divider_i$ **do**
12: send (**UB**, $newCPA$, UB) to k

When an agent with one child receives an **UB** message, it simply needs to update its UB. When an agent with more then one child receives an **UB** message, it combines the received CPA with the one it received from its parent, and with the other CPA it received from its other children (line 4), and save it. In order to perform line 4, the agent holds a data structure $BestCPA(val, child)$

which holds for every $val \in D_i$ the lowest cost CPA received from each one of his children. $BestCPA$ is deleted when agent i receives a new **CPA** from his parent. Combining all of the CPA is straight-forward, since every agent can only be assigned in one of these CPAs. It then identifies from which sub-tree the message came (line 5), and for which $val \in D_i$ it is relevant and then calculate the best total upper bound found so far for val (line 8), and compare it to the old UB. A reduction in a Sub_Tree_UB does not necessary mean that the calculated $totalUB$ is smaller then the old UB, since at the beginning the sum of all Sub_Tree_UB is actually greater then the real UB. If the new $totalUB$ is better then the old UB, it is updated, and **UB** messages is sent to all members of the chain containing i, up to $Latest\_Divider_i$.

Procedure 6 when received (**Backtrack**)

1: $val \leftarrow srchVal(sender)$
2: **if** $|C(i)| > 1$ **then**
3: $\quad diff \leftarrow Sub\_Tree\_UB(sender, val) - Sub\_Tree\_LB(sender, val)$
4: $\quad Sub\_Tree\_LB(sender, val) \leftarrow Sub\_Tree\_UB(sender, val)$
5: $\quad$ **for all** $j \in C(i)$ such that $j \neq sender$ **and** $srchVal(j) = val$ **do**
6: $\quad\quad$ send (**Reduce UB**, $diff$) to j
7: call $assignNextVal(val, sender)$

When an agent with several children receives a **Backtrack** message, it can conclude that the best UB it received from that sub tree is the very best possible, so it can also be considered as the Sub_Tree_LB. It saves that new Sub_Tree_LB, and send all children searching on the same val a **Reduce UB** message, with the difference between the new and the old Sub_Tree_LB.

Procedure 7 when received (**Reduce UB**, $diff$)

1: $Received\_UB \leftarrow Received\_UB - diff$
2: **if** $Received\_UB < UB$ **then**
3: $\quad$ **if** $|C(i)| > 1$ **then**
4: $\quad\quad newDiff \leftarrow UB - ReceivedUB$
5: $\quad$ **else**
6: $\quad\quad newDiff \leftarrow last\_sent\_UB - Received\_UB$
7: $\quad UB \leftarrow Received\_UB$
8: $\quad$ **for all** $j \in C(i)$ **do**
9: $\quad\quad$ send (**Reduce UB**, $newDiff$) to j

An agent receiving a **Reduce UB** message, updates the $Received\_UB$ from its parent, and then check if it is better than his current UB. Then the agent calculates the difference between the last computed UB for its children and the new one. If the agent has more then one child, the reduction of the UB is simply

the difference between the new and the old UB. If the agent has only one child, then it shares with its child both UB and *Received_UB*. Intuitively, the agent should just send forward the same $diff$, but if the agent did not send an old **Reduce UB** (because the *Received_UB* was higher then UB), it should send to its child the difference between the last UB sent, and the current *Received_UB*.

4 Correctness of PT-SFB

In order to prove the correctness of *PT-SFB*, two claims must be established. First, that the algorithm terminates and second that when the algorithm terminates the cost of the complete assignment found by the algorithm is optimal. To prove termination, it is enough to show that the same partial assignment will not be searched twice. We can prove that recursively: Let's assume that agent i does not receive the same CPA from its parent twice. The assumption is trivial for the root. For a given CPA received from an agent's parent, it may send to its children several CPA, but each one will have a different assignment to the agent's variable, so they are all different from one another. Since an agent cannot receive the same CPA twice, and for a given CPA it will not send two identical CPAs to its child, the proof of termination is complete.

Next we prove that on termination, the *cost* of the assignment found by the algorithm is optimal. We assume by contradiction that an assignment with $cost' < cost$ exists. Since the algorithm did not find that assignment, it must have been pruned along the way. An assignment can only be pruned in the procedure **assign Next Val**, in line 1 or in line 15, and only if $Domain\_LB(val) \geq UB$ in the pruning agent. We will define the following lemmas:

Lemma 1. *For a given CPA from agent i's parent, any complete assignment of the sub-problem of i with the assignment $(X_i = val)$ will cost at least $Domain\_LB(val)$*

To prove lemma 1, let's remember that $Domain\_LB_i(d)$ is defined as:
$Domain\_LB_i(d) = cost + Local\_Cost_i(d) + \sum_{k \in descendants(i)} LBReport(k)$
(in "when receive **CPA**" line 4,6). Since any descendant of i will add to the cost of the complete assignment at least its $LBReport$, the cost of any complete assignment will be at least $Domain\_LB_i(d)$.

Lemma 2. *For a given CPA from agent i's parent, any complete assignment of the sub-problem of i with a cost greater the UB cannot be a part of a complete assignment of the entire problem with a cost smaller then the total UB.*

We will prove lemma 2 by induction, by showing that for an assignment CPA_i with $cost_i > UB_i$ to i's sub problem, any complete assignment $CPA_j \supset CPA_i$ to i's parent's (j) sub-problem, will have $cost_j > UB_j$. Lets first look at what an agent's UB consists of. An agent can receive its UB from its parent in the form of a **CPA** message or in the form of **Reduce UB** messages or from its descendants in the form of **UB** messages. If the agent UB is a result of a **UB** message from its descendants, then a better solution has already found, and thus the new solution will result in a higher cost. If the UB was received from

i's parent j, we have two scenarios. If they are part of a chain, then $cost_i = cost_j$, and $UB_i = UB_j$. Otherwise, by manipulating equation 1 we can find that:

$$UB_j = UB_i + CPA\_Cost + \sum_{k \neq i \in children(j)} Sub\_Tree\_LB(k) \quad (3)$$

and from that we can see that any solution CPA_i with $cost_i > UB_i$ to i's sub-problem, will mean that any complete assignment $CPA_j \supset CPA_i$ to j's sub-problem, will have $cost_j > UB_j$.

From lemma 1 and lemma 2, we can see that any pruned assignment can not have a better cost than the one found by the algorithm.

5 Experimental Evaluation

To evaluate the performance of *PT-SFB* it is compared to *bnb-ADOPT+* [6] (improved version of [12]) and *NCBB* [1], which are pseudo tree based algorithms, and to *ConcFB* [11] as the best non-tree based algorithm. The performance of all algorithms was evaluated on randomly generated problems with the *Agent Zero* simulator. The experiments were run on DCOP problems with 12 agents and a domain size of 8. The problems had a variety of constraints density, in the range of $p1 = 0.2$ to 0.6. Constraints density p_1 is the probability of any two agents to be constrained. For any two constrained agents, a cost in the range of 0 to 100 was randomly generated for every combination of values. In every setting, 30 problem instances were randomly generated and solved by every algorithm. Results are averages over all problem instances with the same parameters values. All pseudo tree algorithms used the tree building heuristic described above, so as to compare clearly the performance of the algorithms themselves. All algorithms were tested with the best heuristics and modifications presented in their papers. There are three commonly used measures for evaluating the performance of DCOP algorithms [8]. The number of non-concurrent constraints-checks (NCCCs) that evaluates distributed run-time. A different measure for evaluating non-concurrent run-time is the number of non-concurrent steps of computation (NSCS), which better represents the run time when computation is much faster than message sending. Finally, the total Message Count evaluates the network load of the algorithms.

Figure 1 presents the run-time results of all algorithms, against the constraints density of the randomly generated DCOPs. *BnB-ADOPT+* does not complete its run on problems with density over 0.5 within 15 minutes. This is why it is not presented for higher densities. Clearly, *PT-SFB* outperforms all other state-of-the-art algorithms by a large margin, for all tested densities. When comparing Non-Concurrent Steps of Computation in Figure 2, we can see that *PT-SFB* largely outperforms *bnb-ADOPT+*, and that it keeps an almost constant factor improvement over *NCBB*. *ConcFB* is much less effected by the change in density, since it does not dependent on a pseudo tree to be built. This makes it less efficient than *PT-SFB* for low density problems (where good pseudo trees are usually found), but more efficient in higher density. In terms

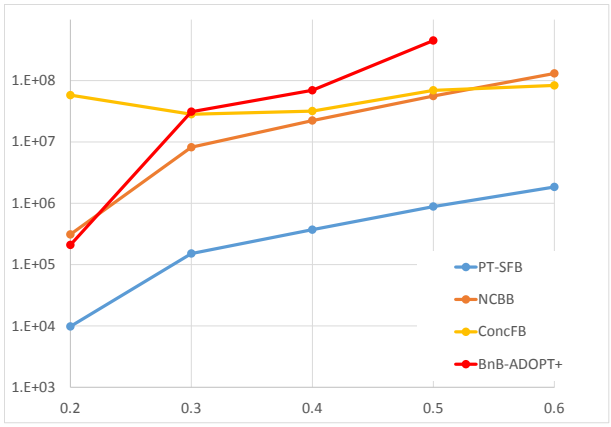

Fig. 1. Number of non-concurrent constraints checks - NCCCs

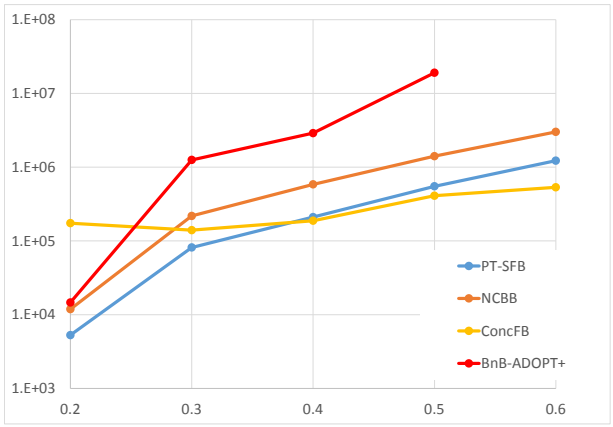

Fig. 2. Number of non-concurrent steps of computation - NCSCs

of network load, as seen in Figure 3, the difference between *PT-SFB* and the other pseudo tree algorithms remains. However, *PT-SFB* performs better than *ConcFB* even on denser problems, although it may be less efficient on problems which are denser than those that were tested.

Figure 4 presents the average height of the pseudo tree created with and without the improvement of the heuristic mentioned. It illustrates the increased difficulty of the pseudo tree based algorithms on densely constrained problem instances.

6 Conclusion

A new DCOP algorithm is proposed, that combines two separate approaches, pseudo tree and forward bounding. The combined approach is aimed at exploiting the best aspects of both approaches. Like other pseudo tree algorithms,

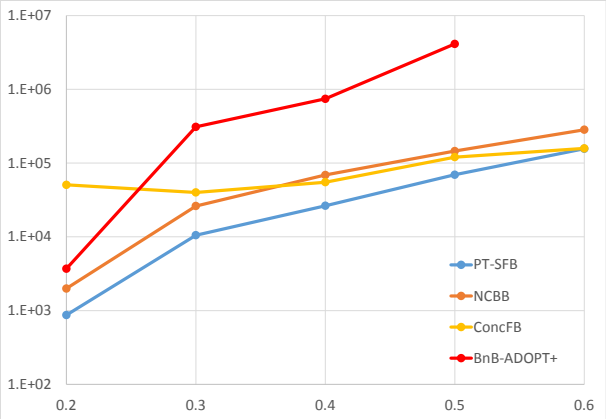

Fig. 3. Total number of messages sent

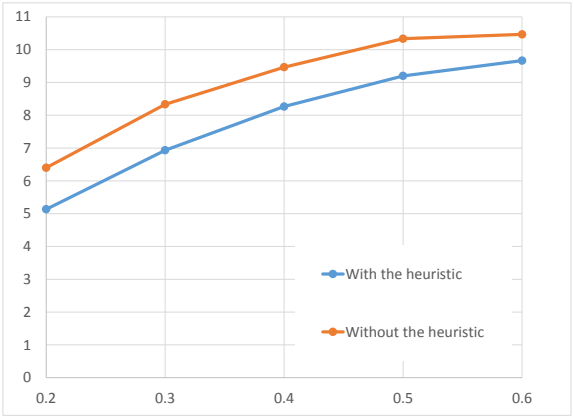

Fig. 4. Average height of pseudo tree

PT-SFB divides the problem into smaller sub-problems and solves them simultaneously and independently. Thanks to the Forward Bounding technique, together with a tight bound maintenance, *PT-SFB* prunes large sections of the search space. The algorithm treats differently chains in the tree, which helps it avoid excess computation on less then ideal pseudo trees.

The extensive experimental evaluation on randomly generated DCOPs demonstrates that *PT-SFB* outperforms other tree based algorithms. *PT-SFB* outperforms *ConcFB*, the top forward bounding algorithm, on low to medium density DCOP instances, but on high density problems it falls short on some of the measures.

Further improvement of the *PT-SFB* algorithm may be achieved by enforcing better *LB* maintenance. The same hybrid idea can also be implemented on more complex algorithms, like *PT-ConcFB*, which have the potential to outperform every state-of-the-art algorithm on every density setting.

References

1. Anton Chechetka and 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 2006), Hakodate, Japan, May 8-12, 2006*, pages 1427–1429, 2006.
2. Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49(6):297–301, 1994.
3. Redouane Ezzahir, Christian Bessiere, Mohamed Wahbi, Imade Benelallam, and Houssine Bouyakhf. Asynchronous inter-level forward-checking for discsps. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 304–318, 2009.
4. A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward-bounding for distributed constraints optimization. In *Proc. ECAI-06*, pages 103–107, Lago di Garda, August 2006.
5. A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward bounding. *J. of Artificial Intelligence Research*, 34:25–46, 2009.
6. Patricia Gutierrez and Pedro Meseguer. Bnb-adopt$^+$ with several soft arc consistency levels. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, pages 67–72, 2010.
7. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
8. A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.
9. Amnon Meisels and Roie Zivan. Asynchronous forward-checking for discsps. *Constraints*, 12(1):131–150, 2007.
10. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. ADOPT: asynchronous distributed constraints optimization with quality guarantees. *Artificial Intelligence*, 161:1-2:149–180, January 2005.
11. Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artif. Intell.*, 193:186–216, 2012.
12. William Yeoh, Ariel Felner, and Sven Koenig. Bnb-adopt: An asynchronous branch-and-bound DCOP algorithm. *J. Artif. Intell. Res. (JAIR)*, 38:85–133, 2010.
13. M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.