Improving BnB-ADOPT$^+$-AC

Patricia Gutierrez Pedro Meseguer
IIIA - CSIC
Universitat Autonoma de Barcelona
08193 Bellaterra, Spain
{patricia|pedro}@iiia.csic.es

ABSTRACT

Several multiagent tasks can be formulated and solved as DCOPs. BnB-ADOPT$^+$-AC is one of the most efficient algorithms for optimal DCOP solving. It is based on BnB-ADOPT, removing redundant messages and maintaining soft arc consistency during search. In this paper, we present several improvements for this algorithm, namely (i) a better implementation, (ii) processing exactly simultaneous deletions, and (iii) searching on arc consistent cost functions. We present empirical results showing the benefits of these improvements on several benchmarks.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms

Algorithms

Keywords

distributed constraint optimization, soft arc consistency

1. INTRODUCTION

Distributed Constraint Optimization Problems (DCOPs) provide a useful framework for modeling many multiagent coordination tasks. Some of them are meeting scheduling [9], sensor network [6], traffic control [7], coalition structure generation [15], among others. DCOPs involve a number of distributed agents handling variables with finite domains and cost functions over positive integers. Agents exchange messages to coordinate and find a complete variable assignment with minimal cost.

Several distributed search algorithms have been proposed to optimally solve DCOPs: ADOPT [13], DPOP [14], NCBB [2], OptAPO [10], among others. In this paper we consider BnB-ADOPT [16], which uses depth-first branch-and-bound search. In particular, we work on the BnB-ADOPT$^+$-AC version [4], which combines BnB-ADOPT$^+$ with soft arc consistency (AC) in DCOP resolution. Soft arc consistency allows to calculate lower bounds that are useful to identify sub-optimal values, when the individual cost of a value surpasses a suitable upper bound. In BnB-ADOPT$^+$-AC suboptimal values are removed dynamically during search, with two consequences. First, the search space of the problem becomes smaller, so its traversal can be done more efficiently. Secondly, as result of these deletions more informed lower bounds might appear, leading to further deletions. Although this process requires some extra computation and information exchange over the version not including AC, its overall effect is very beneficial for the global performance, leading to a substantial decrement in the amount of communication and computation required for optimal DCOP solving [4].

In this paper we present several improvements for BnB-ADOPT$^+$-AC, namely (i) a better implementation (ii) searching on arc consistent cost functions and (iii) processing exactly simultaneous deletions. We present an empirical investigation on two benchmarks, first comparing BnB-ADOPT$^+$ with the DP2 heuristic [1], with AC and with the combination AC-DP2. Interestingly, results show that combining AC with DP2 (something theoretically possible) produces better results than any of them in isolation. Second, the proposed modifications are empirically evaluated. Their combination always obtains the best results in both communication and computation for all problems tested. For some cases, savings reach up to one order of magnitude.

The paper is structured as follows. In Section 2 we summarize some concepts needed in the rest of the paper. In Section 3 we present a better implementation of BnB-ADOPT$^+$-AC which substancially reduces computation. In Section 4 we describe the issue of simultaneous deletions, providing solutions involving extra messages. In Section 5 we explain preprocess and search process using AC cost functions. We experimentally evaluate these points in Section 6. Finally, we conclude the paper in Section 7.

2. PRELIMINARIES

2.1 DCOP

A *Distributed Constraint Optimization Problem (DCOP)* is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is the set of finite domains of $\mathcal{X}$, such that x_1 takes values in D_1,...,x_n takes values in D_n; $\mathcal{C}$ is a set of cost functions used to evaluate the costs of value assignments; $\mathcal{A} = \{a_1, \ldots, a_p\}$ is a set of agents and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent. We use binary $C_{ij} : D_i \times D_j \mapsto N \cup \{0, \infty\}$ and unary $C_i : D_i \mapsto N \cup \{0, \infty\}$ cost functions. The cost of a complete assignment, in which every variable has assigned a value, is the sum

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), June, 4–8, 2012, Valencia, Spain.
Copyright © 2012, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

of all binary and unary cost functions evaluated on those values. We make the common assumption that one agent handles only one variable, and thus we use the terms variable and agent interchangeably. Agents communicate through messages, which are never lost and are delivered in the order that they were sent. Message delay is random but finite.

A *DFS pseudo-tree* is a graph structure used to represent a DCOP instance, where nodes in the graph correspond to variables and edges connect pairs of variables appearing in the same binary cost function. There is a subset of edges called *tree edges* that form a rooted tree and are chosen following a depth-first (DFS) traversal of the graph. All other remaining edges are called *backedges*. Tree edges connect parent-child nodes, while backedges connect a node with its pseudo-parents and pseudo-children. Two variables sharing cost functions are in the same branch of the DFS tree. Several distributed algorithms exploit a pseudo-tree arrangement of their variables [13, 16], which allow agents positioned in different branches of the DFS to perform search in parallel.

2.2 BnB-ADOPT

BnB-ADOPT [16] is a distributed search algorithm that optimally solves DCOP. It is a depth-first branch-and-bound version of ADOPT [13] showing better performance.

Agents in BnB-ADOPT are first arranged in a DFS pseudo-tree, so they know their parent, pseudoparents, children and pseudo-children. Agents i and j sharing cost function C_{ij} maintain a local copy of the cost function. Every agent x_i maintain a *context* which contains its knowledge about the current value assignments of its ancestors. The context is updated through message exchange. For every domain value d and the current context, x_i maintains a lower and upper bound $LB(d)$ and $UB(d)$, and the bounds LB and UB, calculated in the following way:

$$\delta(d) = \sum_{(x_j,d_j)\in context} C_{ij}(d, d_j)$$

$$LB(d) = \delta(d) + \sum_{x_c \in children} lb_c(d) \qquad LB = \min_{d\in D_i}\{LB(d)\}$$

$$UB(d) = \delta(d) + \sum_{x_c \in children} ub_c(d) \qquad UB = \min_{d\in D_i}\{UB(d)\}$$

where $lb_c(d)$ and $ub_c(d)$ store the LB and UB values of children x_c for domain value d.

The goal of every agent is to explore and ultimately choose the value that minimizes LB. For pruning, agents store a threshold TH, which is an estimated upper bound calculated with the cost of the best solution found so far. The use of TH allows agents to change value more efficiently.

Some communication is needed to calculate the global cost of agents assignments and coordinate search towards the optimal solution. Three types of messages are used: VALUE, COST, and TERMINATE, with the following meaning:

- VALUE($i; j; val; th$): agent i informs child or pseudochild j that it has taken value val with threshold th;
- COST($k; j; context; lb; ub$): agent k informs parent j that with *context* its bounds are lb and ub;
- TERMINATE($i; j$): agent i informs child j that i terminates.

Each agent executes the following loop: it reads and processes all incoming messages, decides about its value assignment and sends a VALUE message to each child or pseudochild and a COST message to its parent.

BnB-ADOPT$^+$ is a version of BnB-ADOPT which removes most of the redundant messages largely improving its efficiency, specially in communication. See [5] for details.

2.3 Soft Arc Consistency

Search can be improved by enforcing soft arc consistency, as a result some sub-optimal values can be identified and removed, making the search space smaller and therefore, speeding up the search process. Let (i, a) be agent x_i taking value a, $\top$ the lowest unacceptable cost and C_ϕ a zero-ary cost function that represents a lower bound of any complete assignment, we consider soft local consistencies defined in [8], as follows.

- *Node Consistency**: (i, a) is node consistent* (NC*) if $C_\phi + C_i(a) < \top$; x_i is NC* if all its values are NC* and there is $a \in D_i$ s.t. $C_i(a) = 0$; a problem is NC* if every variable is NC*.
- *Arc consistency**: (i, a) is arc consistency (AC) wrt. cost function C_{ij} if there is $b \in D_j$ s.t. $C_{ij}(a, b) = 0$; b is a *support* of a; x_i is AC if all its values are AC wrt. every binary cost function involving x_i; a problem is AC* if every variable is AC and NC*.

AC* can be reached modifying the original problem to obtain supports to NC* values and removing not NC* values. Supports are obtained on every value by (1) projecting the minimum cost from its binary cost functions to its unary costs, and (2) projecting the minimum unary cost into C_ϕ. Projection (1) requires the decrement of a minimum cost λ from the binary cost functions, and the increment of λ in the unary costs. Projection (2) requires the decrement of a minimum cost λ from the unary cost functions, and the increment of λ to C_ϕ. Every time a not NC* value is removed on agent x_i the AC* property must be rechecked on neighboring agents. The systematic application of these operations maintains the optimum in the resulting problem. These problems –the original and the AC* modified problem– are refered as *equivalent* [8].

NC* has become standard in the soft local consistency community, to the point that higher local consistencies using it are named without asterisk [3]. Following this trend, in the rest of the paper AC* is written AC.

2.4 BnB-ADOPT$^+$-AC

In [4] this algorithm is called BnB-ADOPT$^+$-AC*. Since, nowadays AC* is written AC [3], we follow this terminology and refer to this algorithm as BnB-ADOPT$^+$-AC (without asterisk).

BnB-ADOPT$^+$-AC [4] is an algorithm that combines distributed branch-and-bound search with soft arc consistency. Its search process is based on BnB-ADOPT$^+$, maintaining the same data and communication structure. The main difference is that agents are able to detect and delete suboptimal values. A value can be found sub-optimal as result of enforcing AC on a copy of the original cost functions.

The inclusion of AC in BnB-ADOPT$^+$ have caused a number of modifications in the original algorithm, both in the structure of the exchanged messages and in the computation done. Regarding messages:

- COST messages include the variable *subtreeContr* that aggregates the costs of unary projections to C_ϕ made on every agent of the DFS subtree;
- VALUE messages include $\top$, which is constantly refined with the best solution found so far, and C_ϕ. Both are calculated at the *root* agent of the DFS;
- a new DEL message is introduced to inform deletions to neighbors; with message $\mathrm{DEL}(i;j;d)$, i informs neighbor j that it has deleted value d. When received, neighbors recheck the AC property on their values, which may lead to further deletions.

Regarding memory, the domain of neighbors have to be represented in each agent, so memory requirements increase in $O(nd)$. Regarding computation, values are tested for deletion and cost functions are projected (binary into unary, unary into C_ϕ). A value d is proved sub-optimal and can be deleted unconditionally from the domain of x_i if $C_i(d) + C_\phi > \top$. Only the agent owner of a variable can delete values in its domain. AC enforcing is done in a preprocessing step and also during search every time a value is deleted.

When performing projections in two constrained agents i and j, changes on C_{ij} should be done carefully since i and j operate asynchronously and after a while they might end up with different copies of C_{ij}. In [4] authors explain how to maintain a *Legal Representation of Cost Functions*. To maintain a legal representation, i has to simulate the action of j on its C_{ij} copy, and vice versa. This can be done in the following way. There is a pre-established ordering for projections. In preprocess, agents always project first on higher agents and afterwards on lower agents (where higher/lower is referred to the position of agents in the DFS tree). When a deletion occurs in one agent, the agent projects binary costs over neighbors and sends DEL messages to neighbors. When a DEL message is received on a neighbor, the neighbor projects binary costs over $self$. By this, it is assured that the same cost is not counted twice when performing projections which may lead to delete optimal values. For details, see [4].

3. IMPROVING BNB-ADOPT$^+$-AC IMPLEMENTATION

In BnB-ADOPT$^+$-AC, agents check their domain every time there is a potential opportunity for deletion. For example, every time an agent processes a COST or a VALUE message, it checks if some values can be deleted from its domain. Since C_ϕ and $\top$ are informed in VALUE messages and the contribution of every children to C_ϕ is informed in COST messages, every time this information is updated a new opportunity for value deletion might appear. Now, we propose to check for deletions after the agent has completely processed the input queue. Such as it happens when agents decide to change value, agents will first gather all information from incoming messages before checking its domain. Also, instead of sending one DEL message for every deleted value, we send a list of all deleted values in the same DEL message. These modifications reduce computational cost and assure that agents will send at most one DEL message to each neighbour per cycle.

In BnB-ADOPT$^+$-AC unary projections to C_ϕ are done every time there is a possibility to increment the agents local contribution to C_ϕ. This can happen when a value is deleted or when AC is reinforced. Now, we propose to perform this operation after the agent has completely processed the input queue. We do this for the following reason. Every time there is a unary projection, the unary costs of the agent are decremented. In the centralized case, this decrement is quickly compensated with an increment in the global C_ϕ, but in a distributed setting this compensation is not immediate, it takes some time. First the agent contribution must travel in COST messages to the *root* agent, where is aggregated with other contributions. Afterwards the aggregated C_ϕ is informed in VALUE messages to lower neighbours. Since this process might take several cycles, we delay the decrement of unary costs on an agent until the next COST message is sent.

Delaying checking for deletions and projections of unary contributions to C_ϕ reduce considerably the computational effort made by BnB-ADOPT$^+$-AC, although in some cases this causes some extra messages (since these operations are not done as early as they could be). For empirical results on this see Section 6.

4. SIMULTANEOUS DELETIONS

If deletions are non-simultaneous in BnB-ADOPT$^+$-AC [4](that is, if two deletions never occur at the same time on neighboring agents), it is easy to see that projections are always done in the same order on every agent, so cost functions on both agents eventually remain equivalent. However, in the case that deletions occur at the same time on neighbouring agents, something different happens.

Consider the example in Figure 1. First row correspond to actions taking place inside agent i and second row actions taking place inside agent j. Every column show simultaneous operations, occurring at the same time on i and j. Agents i and j only store the unary costs of their own domain. Black domain values and costs are the actual values and costs stored in an agent. Gray domain values and costs are what an agent believes of the neighbor agent. Lines represent binary costs C_{ij} with cost one. Initially, $C_\phi = 0$, $C_i(b) = 1$, $C_j(b) = 1$ and the rest of unary costs are zero.

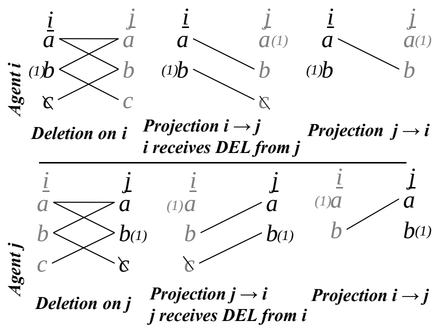

Figure 1: Agents i and j, and the process of two simultaneous deletions. Possible values for each agent are a, b, c, unary costs appear between parenthesis. In black, what an agent knows of itself. In grey, what an agent believes of the other agent. Lines indicate pairs of values with cost 1, no lines indicate cost 0.

On the first column, two simultaneous deletions take place. On second column, both agents make a projection over the neighbor. When projecting over the neighbor, binary costs are reduced from C_{ij} and the agent assumes that an increment in the unary costs of neighbor will eventually occur when the DEL message arrives to the neighbor. But notice that, because these operations occur at the same time, the order of resulting projections is opposite on agent i and j. This would not be the case if one deletion would have preceded the other. Then both agents would have kept the same ordering in projections (for example, a projection first from i to j and after from j to i) and they would have obtained $C_\phi = 1$. Notice that in the example C_ϕ remains zero.

Both agents projected at the same time a binary cost of 1 to the unary cost of its neighbor, but this operation has not actually taken place on the neighbor, so this cost has been lost from the problem. Notice that costs are not counted twice and no illegal deletions are produced, but we have lost a cost of 1 from the problem, which diminish deletion opportunities. In addition, on the last column the resulting cost functions on i and j are not equivalent. [1]

We can avoid this undesirable behavior assuring synchronous deletions. It is impossible that two agents know they are performing deletions at the same time, but it is possible that they communicate beforehand and agree on the order to follow. If one deletion always precedes the other, projections on neighboring agents maintain the same order. This assures that cost functions remain equivalent and no costs are lost from the problem.

4.1 Synchronizing Deletions

To maintain synchronous deletions, two main changes must be done in BnB-ADOPT$^+$-AC:

- Two new messages are introduced to synchronize deletions: SYNC$_1$ and SYNC$_2$
- Agents have a *locked* property. While an agent is *locked* it is able to read and process messages, but it will not change its value or send messages to neighbors, except for synchronization messages SYNC$_1$ and SYNC$_2$. An agent is *locked* because it is waiting to delete a value, or because a deletion is occurring in one or several neighbors. A *locked* agent changes to *unlocked* when it is no longer locked with any of its neighbors.

On Figure 2 a pseudocode of the synchronous deletion process for BnB-ADOPT$^+$-AC is shown. The rest of the algorithm is shown in Figure 3. The pseudocode is based on the implementation proposed in [4]. Modifications are described below.

Synchronizing deletion contains the following steps:

1. After completely processing the input queue, the `backtrack` method is invoked and the agent checks its domain looking for sub-optimal values [line 47].

[1]One may wonder if the approach of [4] is correct and complete. The answer is yes because search is done using a copy of the original cost functions which is only modified to reflect value deletions. There is another copy of cost functions used for AC enforcement, on which cost projections are done, but this copy is not used for search. This is further elaborated in section 5.

```
1  procedure CheckDomainForDeletions()
2    for each $v \in D_{self}$ do
3      if $C_{self}(v) + C_\phi > \top$ or $(\sum_{ch \in children} lb(ch, v) > \top$
4      and $childrenContexts(ch, v) = \{self\})$ then
5        $valuesToDelete.add(v)$;
6    if $valuesToDelete.size > 0$ then
7      for each $k \in neighbors(self)$ do
8        if $\neg hasStopped(k)$ then
9          sendMsg:$(DEL, self, k, valuesToDelete)$;
10         $locked(k) = true$;
11     UpdateLockStatus();

12 procedure ProcessDelete($msg$)
13   if $locked(msg.sender)$ and $self < sender$ then
14     $processPending(msg.sender) = msg$;
15   else
16     $D_{sender} \leftarrow D_{sender} - \{msg.valuesToDelete\}$;
17     BinaryProjection($self, sender$);
18     sendMsg:$(SYNC_1, self, msg.sender)$;
19     if $\neg hasStopped(msg.sender)$ then
20       $locked(msg.sender) = true$;
21       UpdateLockStatus();

22 procedure ProcessSYNC$_1$($msg$)
23   $locked(msg.sender) = false$;
24   UpdateLockStatus();
25   if $\neg locked$ then
26     $D_{self} \leftarrow D_{self} - valuesToDelete$;
27     $valuesToDelete \leftarrow \emptyset$
28     for each $k \in neighbours$ do
29       BinaryProjection($k, self$); sendMsg:$(SYNC_2, self, k)$;
30     for each $msg \in processPending$ do
31       ProcessDelete(msg);
32       $processPending.remove(msg)$;

33 procedure ProcessSYNC$_2$($msg$)
34   $locked(msg.sender) = false$;
35   UpdateLockStatus();

36 procedure UpdateLockStatus()
37   $locked = false$;
38   for each $k \in neigbors(self)$ do if $locked(k)$ then $locked = true$;

39 procedure ProcessStop($msg$)
40   if $msg.sender == parent$ then $receivedTerminate \leftarrow true$;
41   $locked(msg.sender) = false$;
42   UpdateLockStatus();
43   $hasStopped(msg.seder) = true$;

44 procedure Backtrack()
45   if $locked$ then return;
46   UpdateLBUB();
47   CheckDomainForDeletions();
48   if $locked$ then return;
49   if $LB(value) \geq \min(TH, UB)$ then
50     $value \leftarrow argmin_{v \in D_{self}}\{LB(v)\}$;
51   UnaryProjectionOverCo();
52   if $value$ has changed then
53     SendValueToLowerNeighbors();
54   else
55     SendValueToChildrenToUpdateTH();
56   if $(receivedTerminate$ or $self == root)$ and $LB == UB$ and
57   $LB(value) == UB(value)$ then
58     SendStopMessageToLowerNeighbors();
59   SendCostToParent();
```

Figure 2: Pseudocode for Syncronizing Deletions.

2. A value v is proved sub-optimal under certain conditions [lines 3-4]: when its unary cost plus C_ϕ exceeds $\top$ or when the sum of its lower bounds exceeds $\top$ and this bounds were sent with a context that contains only the *self* agent (the bounds do not depend on any other agent, for more detail see [4]). When agent i realizes that it can delete values from its domain, instead of immediately erasing them, it marks them as pend-

```
60  procedure AC-Preprocess()
61    for each $i \in neigbors(self)$ do
62      if $i < self$ then
63        BinaryProjection($self, i$);
64        BinaryProjection($i, self$);
65      else
66        BinaryProjection($i, self$);
67        BinaryProjection($self, i$);
68    CheckDomainForDeletions();
69    UnaryProjectionOverCo();

70  procedure BinaryProjection($i, j$)
71    for each $a \in D_i$ do
72      $v \leftarrow argmin_{b \in D_j}\{C_{ij}(a, b)\}$;
73      $\alpha \leftarrow C_{ij}(a, v)$;
74    for each $b \in D_j$ do
75      $C_{ij}(a, b) \leftarrow C_{ij}(a, b) - \alpha$;
76      if $i = self$ then $C_i(a) \leftarrow C_i(a) + \alpha$;

77  procedure Start()
78    for each $d \in D_i, ch \in children(self)$ do
79        InitChild($ch, d$);
80    InitSelf();
81    Backtrack();
82    loop forever
83      if (message queue is not empty) then
84        while (message queue is not empty) do
85          pop $msg$ off message queue
84          Process($msg$);
86        Backtrack();

87  procedure InitSelf()
88    $value \leftarrow argmin_{v \in D_{self}}\{LB(v)\}$;
89    $TH = \infty$;

90  procedure InitChild($ch, d$)
91    $lb(ch, d) = 0$;
92    $ub(ch, d) = \infty$;

93  procedure ProcessVALUE($msg$)
94    add ($msg.sender, msg.value$) to $context$
95    CheckContextWithChildren();
96    if ($msg.sender == parent$) then
97      $TH = msg.threshold$;
98    if $C_\phi < msg.C_\phi$ then $C_\phi = msg.C_\phi$;
99    if $\top < msg.\top$ then $\top = msg.\top$;

100 procedure ProcessCOST($msg$)
101   $d \leftarrow$ value of $self$ in $msg.context$
102   PriorityMerge($context, msg.context$);
103   CheckContextWithChildren();
104   if ($context$ compatible with $msg.context$)
105     $childrenContexts(msg.sender, d) = msg.context$;
107     $lb(msg.sender, d) = max\{msg.lb, lb(msg.sender, d)\}$;
108     $ub(msg.sender, d) = min\{msg.ub, ub(msg.sender, d)\}$;
109   $subtreeContr(msg.sender) = msg.subtreeContr$;

110 procedure CheckContextWithChildren()
111   for each $d \in D_i, ch \in children(self)$ do
112     if ($childrenContexts(ch, d)$ incompatible with $context$) then
113       InitChild($ch, d$);
```

Figure 3: Pseudocode for Preprocess and Process Phase.

ing to delete and sends DEL messages to neighbours $k_1, k_2, ..k_i$. Afterwards, i is *locked* with neighbours $k_1, k_2, ..k_i$, so i can process incoming messages but it can not change its value or send VALUE, COST or DEL messages [lines 6-11].

3. When neighbor k receives a DEL message from i it can be the case that k is already *locked* with i, this means that simultaneous deletions are taking place. In this case, the higher agent is the one that processes the DEL message first, otherwise the message remains as process pending [lines 13-14] and will be processed afterwards when the agent is *unlocked* [lines 30-32]. To process the DEL message, k deletes the values of i from its domain copy of i, and performs a projection $P_{i\rightarrow k}$ from i to k. After this, it sends a message SYNC$_1$ to i to inform that the deletion has been processed, and change its status to *locked* with i [lines 16-21].

4. Only after receiving SYNC$_1$ message from *all* its neighbours i is unlocked. At this point, all neighbours k have done a projection $P_{i\rightarrow k}$ from i to k. Then, i deletes the values from its domain, makes projections $P_{i\rightarrow k}$ on every neighbor k, and send a SYNC$_2$ messages to neighbours [lines 23-29].

5. When neighbour k receives a SYNC$_2$ message from i, it unlocks from i [lines 34-35].

6. A special case should be consider on termination. When an agent terminates execution it informs its lower neighbours [lines 40-43]. Once an agent has stopped, it will no longer be considered in the synchronizing process because it will not be able to respond, causing other agents to freeze forever. Therefore, before sending DEL messages to an agent or updating the *locked* status with an agent, it is first checked that the agent has not stopped execution [line 8, line 19].

5. SEARCH ON AC COST FUNCTIONS

One of the main goals of AC is to construct strong lower bounds. Zero-ary cost C_ϕ is a lower bound of the optimal solution. Unary cost $C_i(v) + C_\phi$ is a lower bound of domain value v. Lower bounds are useful to identify sub-optimal values when $C_i(v) + C_\phi > \top$. In addition, they can provide a heuristics for value selection which may improve search efficiency.

In [1] authors propose a preprocessing technique for ADOPT algorithm, and show that by calculating lower bounds for every domain value they are able to speed up ADOPT search. In [11] authors transform the original problem into an equivalent one projecting costs in a preprocessing step, then ADOPT is executed in the equivalent problem with performance improvements. In these two approaches authors aggregate lower bounds and use them during ADOPT search, but no deletions are performed.

We think deletions are a key point when enforcing soft arc consistency, since they lead to refinements in the lower bounds and further deletions. We perform deletions during AC preprocessing and also during search. Aiming at maintaining such deletions, we would like to use the same cost functions to perform search and to enforce AC: this would provide us a good value ordering (because costs are updated in the AC cost functions), combined with the deletions caused by AC enforcing.

Unfortunately, using the same cost functions for search and for AC enforcing is not an easy task. Consider the following case in Figure 4. Suppose an agent x has child ch, descendant d and ancestor a. There is a back-edge between a and d (Figure 4 (left)). Suppose a deletion takes place on descendant d and some costs c are moved from C_{da} to C_a (Figure 4 (center)). As result, an increment of costs occurs in a and a decrement occurs in ch subtree (this is because when calculating costs, agents aggregate their back-edges costs). The problem arises when COST messages arrive to

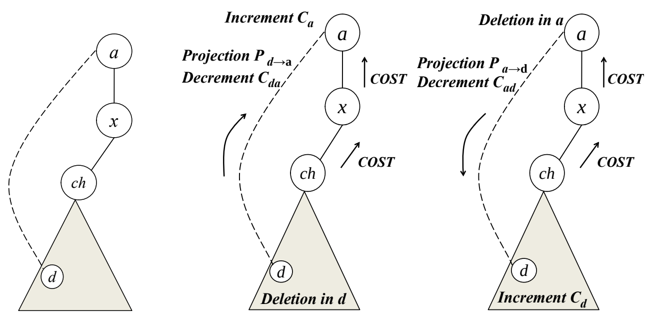

Figure 4: Performing projections during search process. Left: a problem instance. Center: a deletion takes place on a descendant. Right: a deletion takes place on an ancestor

x or a. If these messages where sent before deletion, they will contain in their LB, UB the cost c, but this cost is no longer in ch subtree, so the message must be ignored. Furthermore, the lb and ub tables in x and a should be reinitialized, because they may contain aggregated costs involving c, which will be counted twice if a COST message is sent from this agents without reinitialization.

Similarly, some problems occur if a deletion takes place on ancestor a and a cost c is moved from C_{ad} to C_d (Figure 4 (right)). There will be a cost increment in ch subtree and a decrement in a. If tables lb and ub are not reinitialized in x and a, the new COST messages from ch subtree containing c might not be accepted because the algorithm assumes that LB and UB improve monotonically, unless there is a context change.

Therefore, after deletions lb and ub tables must be reinitialized on neighbouring agent and even on other agents in the DFS branch. These reinitializations after deletions might lead to a serious degradation in performance, which makes deletion useless for efficiency gains.

To avoid reinitializations, we propose the following approach:

- Two copies of the cost functions are used: C_{search} and C_{AC}. Initially, they are identical.
- In preprocess, projections are performed on C_{AC}. Since we proposed a mechanism to synchronize deletions, it is assured that all projections are done in the same order on every agent, so no costs are lost from the problem and after preprocessing C_{search} and C_{AC} represent equivalent problems (one is AC, the other is not neccesarily AC).
- After preprocess, we make $C_{search} = C_{AC}$
- During search, costs are calculated using C_{search} aggregating binary, unary and zero-ary costs. Any new projection performed during search is done only on C_{AC}.
- During search, any value deletion computed using C_{AC} is applied on C_{search}. Observe that this is a legal operation.

Notice that syncronous deletions are needed during preprocessing to search in AC cost functions.

6. EXPERIMENTAL RESULTS

We evaluate experimentally the changes proposed in BnB-ADOPT$^+$-AC on two benchmarks: random DCOPs and structured Meeting Scheduling. Experiments are executed on a discrete event simulator and performance is evaluated in terms of the total number of messages exchanged among agents (#Total Msgs) and the number of non concurrent constraint checks (#NCCC) [12]. In every cycle of the simulator, agents read incoming messages from the message queue, process them and send outgoing messages. The simulation stops when all agents have stopped and no messages are exchanged.

Binary random DCOP are characterized by $< n; d; p_1 >$ where n is the number of variables, d is the domain size and p_1 is the network connectivity. A random instance contains $p_1 * n(n-1)/2$ cost functions. We tested random DCOP instances with: $< n = 10; d = 10; p_1 = 0.3, \dots, 0.8 >$. Costs are selected from an uniform cost distribution. To introduce some variability among tuple costs, two types of binary cost functions are used, small and large. Small cost functions extract costs from the set $\{0, \dots, 10\}$ while large ones extract costs from the set $\{0, \dots, 1000\}$. The proportion of large cost functions is 1/4 of the total number of cost functions. Results appear in Table 1, averaged over 50 instances.

Meeting scheduling instances are obtained from the public DCOP repository [17]. In this formulation, variable represent meetings, domain represent time sots assigned for a meeting, and costs functions are set between meetings that share participants. We present cases A (23 variables), B (26 variables), C (71 variables) and D (72 variables). Results appear in Table 2, averaged over 30 instances.

We compare with several extensions of BnB-ADOPT. This algorithm has proved to be clearly more efficient than ADOPT and as efficient as NCBB for DCOP solving [16]. Comparison with other complete algorithm such as DPOP or OptAPO is truly difficult to measure –and scapes to the purpose of this paper. DPOP uses a linear number of messages but their size can be exponential, while BnB-ADOPT uses a linear size and exponential number of messages. OptAPO is a partially centralized algorithm while BnB-ADOPT is fully decentralized.

For every case in Meeting Scheduling and every network connectivity in random DCOPs we show results for:

1. First row: BnB-ADOPT$^+$ including a preprocessing step to calculate DP2 heuristic [1]. The preprocessing requires a single pass of messages from leafs to the *root* of the DFS tree calculating lower bounds for every value to focus search. This version improves over the basic BnB-ADOPT$^+$.
2. Second row: BnB-ADOPT$^+$-AC as described in [4].
3. Third row: BnB-ADOPT$^+$-AC with the DP2 preprocess.
4. Fourth row: BnB-ADOPT$^+$-AC with DP2 preprocess and the implementation improvements described in Section 3.
5. Fifth row: BnB-ADOPT$^+$-AC with DP2 preprocess, implementation improvements described in Section 3, synchronous deletions and searching on AC cost functions as described in Sections 4 and 5.

p_1	Algoritm BnB-ADOPT$^+$ with	#Total Msgs	#NCCC
	DP2	5,237	57,233
	AC	2,753	95,278
	AC-DP2	1,455	55,837
	AC-DP2-Opt	1,709	21,917
0.3	AC-DP2-Opt-Sync	**1,145**	**14,900**
	DP2	74,412	991,071
	AC	29,318	1,241,569
	AC-DP2	21,292	929,218
	AC-DP2-Opt	29,176	369,061
0.4	AC-DP2-Opt-Sync	**12,711**	**151,028**
	DP2	114,615	2,041,320
	AC	68,746	4,278,971
	AC-DP2	48,168	3,719,577
	AC-DP2-Opt	52,276	1,086,782
0.5	AC-DP2-Opt-Sync	**13,492**	**152,007**
	DP2	393,487	6,290,061
	AC	283,767	25,342,026
	AC-DP2	148,158	12,629,504
	AC-DP2-Opt	155,435	2,819,398
0.6	AC-DP2-Opt-Sync	**44,037**	**766,606**
	DP2	1,128,513	23,430,101
	AC	1,256,489	137,506,730
	AC-DP2	734,539	76,132,133
	AC-DP2-Opt	842,227	16,878,749
0.7	AC-DP2-Opt-Sync	**173,850**	**3,080,989**
	DP2	1,207,525	28,551,056
	AC	1,885,804	226,355,134
	AC-DP2	782,946	90,405,429
	AC-DP2-Opt	907,013	20,900,258
0.8	AC-DP2-Opt-Sync	**217,847**	**4,382,452**

Table 1: Experimental results on random DCOPs (10 variables) averaged over 50 instances.

When using DP2, communication and computational effort (#Total Msgs, #NCCCs) of DP2 preprocessing are included in the results. When searching on AC cost functions (fifth row), it is necessary to execute first the AC preprocess and afterwards the DP2 preprocess, so the bounds aggregated by DP2 correspond to the AC cost functions that will be used during search.

For random instances (Table 1), on small and medium connectivities, less messages are needed enforcing simple AC that using BnB-ADOPT$^+$ with DP2 (first and second row), on the other hand more NCCCs are needed since enforcing AC requires more computational effort. When combining BnB-ADOPT$^+$-AC with DP2 (third row), we observe a consistent decrement in messages and NCCCs. This confirms empirically that these techniques aiming to different objectives –the first to provide heuristic values during search, the second to erase sub-optimal values– can be enhanced when combined.

Optimizing the implementation of BnB-ADOPT$^+$-AC - (fourth row) combined with DP2 we obtain important reductions in NCCCs. We observe a moderate increment (10%-20%) in the number of messages. This effect is due to the slight delay in deletions and projections on C_ϕ (as mentioned in Section 3). However the decrement in computational effort is so important that globally we consider the modifications introduced as an improvement.

p_1	Algoritm BnB-ADOPT$^+$ with	#Total Msgs	#NCCC
	DP2	59,529	310,984
	AC	156,448	7,875,894
	AC-DP2	45,830	1,176,493
	AC-DP2-Opt	55,873	350,132
A	AC-DP2-Opt-Sync	**39,947**	**263,345**
	DP2	20,802	73,900
	AC	82,234	2,601,983
	AC-DP2	18,643	384,778
	AC-DP2-Opt	19,172	94,092
B	AC-DP2-Opt-Sync	**6,859**	**41,999**
	DP2	43,916	129,500
	AC	444,730	13,549,666
	AC-DP2	38,051	584,284
	AC-DP2-Opt	42,745	119,395
C	AC-DP2-Opt-Sync	**13,946**	**37,770**
	DP2	26,448	55,073
	AC	304,214	6,157,253
	AC-DP2	26,271	329,370
	AC-DP2-Opt	29,155	70,428
D	AC-DP2-Opt-Sync	**17,712**	**53,405**

Table 2: Experimental results on Meeting Scheduling instances (23, 26, 71 and 72 variables) averaged over 30 instances.

Including simultaneous deletions and searching in AC cost functions (fifth row) shows clear benefits in the number of exchanged messages and NCCC. Savings are higher on medium and higher connected problems, reaching up to one order of magnitude in some cases. This makes sense because on high connected problems, removing a sub-optimal value means avoiding context changes and reinitializations in many connected agents which are lower in the DFS tree.

In Meeting Scheduling problems (Table 2), we also observe important benefits. In these problem a simple preprocess to calculate DP2 heuristic is better than maintaining AC (first and second row). Moreover, we noticed during experimentation that the lower bound obtained in the *root* agent by DP2 preprocessing is actually very close to the optimal cost. This lead us to think that these instances, although contain a higher number of variables and cost functions than the random instances, are relatively easy to solve in the sense that only by DP2 preprocess we obtain a good estimation of the optimal cost, before starting the search. Observe that even in this case, our improved BnB-ADOPT$^+$-AC version (fifth row) is able to reduce messages and NCCC in all instances, in some cases by a factor of 2 or 3.

From these results we can extract some conclusions. First, it is beneficial to combined soft AC techniques with DP2 heuristic, the joint effort of both techniques is effectively summed-up and the result is an improvement in performance. Second, the combination of the proposed modifications causes substantial savings in both communication and computation effort with respect to existing versions of the considered algorithm. Third, maintaining soft AC to delete sub-optimal values provides more savings when the problem is connected and is hard to solve (the problem requires many messages and computation and the cost of the optimal solution can not be inferred accurately from a single pass preprocessing technique such as DP2).

7. CONCLUSIONS

In this paper we improve the algorithm BnB-ADOPT$^+$-AC, originally presented in [4], in several ways. First, we propose some modifications in the implementation of the algorithm, where checking for deletions and projections to C_ϕ are delayed until the agent executes the `Backtrack` procedure. Experimentally we show that this alternative reduces significantly the number of constraint checks, although the number of messages is slightly increased. This is due to the fact that an agent does not refine the C_ϕ or identify sub-optimal values as early as it could. However the decrement in computational effort is so important that we globally consider this change as an improvement.

Secondly, we address the issue of simultaneous deletions in the asynchronous setting of BnB-ADOPT$^+$-AC. When neighboring agents perform deletions at the same time, the order of projections in both agents is opposite and as a result some costs might be lost from the cost functions where AC is enforced. During search, BnB-ADOPT$^+$-AC uses original cost functions while AC is enforced in a copy of these cost functions, so the reported issue on simultaneous deletions does not affect optimality or termination. However by losing costs from the problem we lose information which could lead to identify sub-optimal values. To avoid this, we provide a synchronization mechanism to assure that projections are always done in the same order on every agent. This synchronization mechanism assures that no costs are lost but it requires some extra synchronization messages.

Finally, we propose to search on AC cost functions obtained after a preprocessing step since lower bounds calculated for every value can provide a heuristic for value selection. To do this, we need to assure synchronous deletions so $C_{original}$ and C_{AC} are equivalent after preprocessing (no costs are lost). Deletions are able to improve search and the inclusion of synchronization messages to guarantee that no costs are lost in the preprocessing, is compensated with message savings during search.

The aggregation of these three modifications produces a complete algorithm with communication and computation efforts substantially smaller than previous versions of BnB-ADOPT$^+$ (including either AC [4], DP2 heuristics [1] or a combination of both). In most cases, messages and NCCCs are reduced by at least a factor of 2, reaching up to one order of magnitude for some cases. These results allow us to consider the proposed approach as an important step foward towards more efficient algorithms for optimal DCOP solving.

It remains as future work to apply and evaluate the impact of the proposed techniques in other distributed search algorithms.

8. ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their comments, they helped us to make a better paper. Patricia Gutierrez has an FPI scholarship BES-2008-006653. She and Pedro Meseguer are partially supported by the project TIN2009-13591-C02-02.

9. REFERENCES

[1] S. Ali, S. Koenig, and M. Tambee. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. *Proc. of AAMAS*, 2005.

[2] A. Chechetka and K. P. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proc. of AAMAS*, pages 1427–1429, 2006.

[3] M. Cooper, S. de Givry, M. Sanchez, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174:449–478, 2010.

[4] P. Gutierrez and P. Meseguer. BnB-ADOPT$^+$ with several soft arc consistency levels. *Proc. of ECAI*, pages 67–72, 2010.

[5] P. Gutierrez and P. Meseguer. Saving messages in BnB-ADOPT. *Proc. of AAAI*, 2010.

[6] M. Jain, M. Taylor, M. Tambe, and M. Yokoo. DCOPs meet the realworld: Exploring unknown reward matrices with applications to mobile sensor networks. *Proc. of IJCAI*, 2009.

[7] R. Junges and A. L. C. Bazzan. Evaluating the performance of DCOP algorithms in a real world dynamic problem. *Proc. of AAMAS*, 2008.

[8] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. *Proc. of IJCAI*, 2003.

[9] R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. *Proc. of AAMAS*, 2004.

[10] R. Mailler and V. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 25:529–576, 2006.

[11] T. Matsui, M. Silaghi, K. Hirayama, M. Yokoo, and H. Matsuo. Directed soft arc consistency in pseudo trees. *Proc. of AAMAS*, 2009.

[12] A. Meisels, E. Kaplansky, Igor, Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. *Proc. of DCR*, pages 86–93, 2002.

[13] P. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.

[14] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proc. IJCAI-05*, pages 266–271, 2005.

[15] S. Ueda, A. Iwasaki, and M. Yokoo. Coalition structure generation based on distributed constraint optimization. *Proc. of 24th AAAI*, pages 197–203, 2010.

[16] W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: Asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

[17] Z. Yin. USC DCOP repository. *http://teamcore.usc.edu/dcop*, 2008.