Saving Messages in ADOPT-based Algorithms [*]

Patricia Gutierrez and Pedro Meseguer

IIIA, Institut d'Investigació en Intel.ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{patricia|pedro}@iiia.csic.es

Abstract. ADOPT and BnB-ADOPT are two related algorithms essential for distributed constraint optimization. They exchange a large number of messages, which is a major drawback for their practical usage. Aiming at increasing their efficiency, we present results showing that some of their messages are redundant so they can be removed without compromising their optimality and termination properties. Removing most of those redundant messages we obtain ADOPT$^+$ and BnB-ADOPT$^+$, which in practice, cause substantial reductions on communication costs with respect to the original algorithms.

1 Introduction

Distributed Constraint Optimization Problems (DCOP) can be found in many real domains for modeling multiagent coordination problems. DCOPs include a finite number of agents, with the usual assumption that each agent holds one variable with a finite and discrete domain. Variables are related by binary cost functions. The cost of a variable assigning a value is the sum of cost functions evaluated on that assignment. The goal is to find a complete assignment of minimum cost by message passing (for details on DCOP definition see [1]).

Considering distributed search for DCOP solving, the first proposed complete algorithm was ADOPT [1]. Later on, the closely related BnB-ADOPT [2] was presented. This algorithm changes the nature of the search from ADOPT best-first search to a depth-first search strategy, obtaining a better performance. Both algorithms are complete, compute the optimum cost and terminate [1, 2]. ADOPT and BnB-ADOPT have a similar communication strategy, using the same set of messages (with small differences in their processes). They also share the same data structures and semantic for store and update their lower and upper bound tables.

ADOPT and –to a lesser extent– BnB-ADOPT exchange a large number of messages. Often, this is a major drawback for their practical application, despite their good theoretical properties (completeness, optimality, termination). Aiming at decreasing the number of exchanged messages without compromising optimality and termination, this paper provides two contributions. First, we present a new version of ADOPT that uses fewer messages than the original algorithm. Some of the proposed modifications come from common knowledge inside the DCOP community, or have been inspired by the

[*] This work is partially supported by the project TIN2009-13591-C02-02.

BnB-ADOPT algorithm, so we do not claim to be their unique authors. Secondly, we provide two results for detecting redundant messages in our version of ADOPT and BnB-ADOPT. These results allow us to generate new versions of these algorithms, called ADOPT$^+$ and BnB-ADOPT$^+$, which save messages with respect to the original algorithms. Experimentally, we have seen that these new versions cause very significant savings in communication costs (the number of exchanged messages is divided by a factor often larger than 2) on several widely used DCOP benchmarks.

This paper is structured as follows. First we describe our ADOPT version in section 2, which saves some messages with respect to the original ADOPT. We present the communication structure of our ADOPT version, that is closely related with the one of BnB-ADOPT, in section 3. Then, we introduce the main contribution of this paper in section 4, that consist of the results that allow us to detect redundant messages. These results are illustrated with some examples in section 5. Using these results but aiming also at efficiency, we propose new versions of these algorithms, called ADOPT$^+$ and BnB-ADOPT$^+$, in section 6. We report experimental results on the benefits of these new versions with respect to the older ones in section 7. Finally, we conclude in section 8.

2 Reengineering ADOPT

In this section we introduce our ADOPT version. It has some differences with the original ADOPT [1]. We have included these changes for efficiency purposes, but they do not compromise the optimality and termination of ADOPT. Specifically, our version differs from the original ADOPT in the following points:

1. ADOPT sends THRESHOLD messages to children and VALUE messages to children and pseudochildren. Since every time ADOPT sends a THRESHOLD message it also sends a VALUE message, we include all the information in a single VALUE message. The treatment of THRESHOLD and VALUE information is exactly the same, only that it is not splitted in two separate messages. With this simple modification the number of messages is reduced significantly. Obviously, these changes has no effect on optimality and termination of ADOPT. From now on, we consider VALUE and COST messages only.
2. In ADOPT, each agent reads one message of the input queue, processes it and performs backtrack. Executing the backtrack procedure the agent decides if it must change its value. In any case, it sends the corresponding VALUE messages to its children and pseudochildren and a COST message to its parent. This is done for each message until the queue is empty. We modify the algorithm in the following way: on each iteration, the agent reads and processes all messages from the input queue without performing backtrack, and when the queue is empty, it performs backtrack sending the corresponding VALUE and COST messages.
   With this modification the algorithm is equally able to find the optimum and terminates. Since all messages are processed, an agent $self$ will update its data structures ($context$, lb, ub, th) in the same way and order as before, but it will not assign its value or generate messages until the queue is empty. So, there are messages that

would have been sent by $self$ in the original ADOPT that will not be send with this modification. Let us assume an omitted message msg_1 and a final message msg_2 of the same type as msg_1 sent by $self$. It may happen:

(a) If the omitted msg_1 is a VALUE, not sending it will cause no harm, because if this VALUE is processed, the receiver would updates in its context only $self$ assignment, which will be overwritten anyway when msg_2 arrives.

(b) If the omitted msg_1 is a COST, the same thing happens: the last message msg_2 will overwrite the lb and ub tables that might have been updated by msg_1. Also, if the omitted msg_1 would have cause that a new not linked variable would be added to the receiver context, we can assure that this variable will also be added with msg_2, since both messages have the same context variables.

However, when an agent receives VALUE or COST messages, information may be reinitialized if contexts are not compatible. It could be the case that the omitted msg_1 would have caused this effect in the original execution. If this is not caused also by msg_2, then the reinitialization was useless, since the considered obsolete information is now required and will be recalculated. So we can avoid this useless reinitialization. Therefore, these changes has no effect on optimality and termination of ADOPT. This way of processing the input queue reduces a great deal the number of exchanged messages, which is very beneficial for executing this algorithm on difficult instances. [1]

3. Finally, we include a timestamp for every assignment that travels in VALUE messages and in COST messages contexts (one timestamp per value). This timestamp permits to determine which of two assignments is more recent. We allow the receiver context to be updated also by COST messages if they contain more recent assignments. As consequence, an agent $self$ is able to process more updated COSTs instead of discarding them. In the original ADOPT algorithm, COSTs with more recent information are discarded and $self$ would need to wait for delayed VALUE messages until its context is updated with the most recent information. Now, the accepted COST contains the same information as the ones being discarded, so we could have considered them before. Therefore, these changes has no effect on optimality and termination of ADOPT.

In the following, we assume an ADOPT version that includes these changes with respect to the original algorithm [1].

3 Communication Structure

In this section, we summarize the communication structure of our version of ADOPT and BnB-ADOPT. We assume that the reader has some familiarity with ADOPT and BnB-ADOPT code (for a more complete description, the reader should consult the original sources [1, 2]). ADOPT and BnB-ADOPT arrange agents in a DFS tree. An

[1] In his/her review, an anonymous reviewer pointed out that this idea has already been implemented in Jay Modi's code, and in DCOPolis code of ADOPT (http://dcopolis.sourceforge.net).

agent *self* knows its parent, its children and pseudochildren. Also, $self$ holds a context, which is a set of assignations involving $self$ ancestors that will be updated with message exchange.

Our ADOPT version uses the following messages:

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

An agent of our ADOPT version executes the following loop: it reads and processes all incoming messages, and changes value if the lower bound of the current value surpasses the threshold. This strategy allows the agent to change its value whenever it detects a better local assignment. Then, it sends the following messages: a VALUE per child, a VALUE per pseudochild and a COST to its parent. Every time a VALUE or COST message is sent the receiver context is updated. Every time a COST message is sent the receiver lower bound and upper bound are updated. Agents maintain a bounded interval consisting in a lower and upper bound that will be refined during execution. When this interval shrinks to zero (lower bound equals the upper bound) the cost of the optimal solution has been found.

BnB-ADOPT uses the following messages:

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

A BnB-ADOPT agent executes the following loop: it reads and processes all incoming messages, and changes value if the lower bound of the current value surpasses the upper bound. In that case the value is proven to be suboptimal and it can be discarded as long as the context is not changed. Then, the agent sends the following messages: a VALUE per child, a VALUE per pseudochild and a COST to its parent (for more details, see [2]). Every time a VALUE or COST message is sent the receiver context is updated. Every time a COST message is sent the receiver lower bound and upper bound are updated. As ADOPT, agents maintain a bounded interval that will be refined during the execution. When this interval shrinks to zero (lower bound equals the upper bound) the cost of the optimal solution has been found.

ADOPT and BnB-ADOPT differ in the semantic of threshold values. BnB-ADOPT calculates thresholds in such a way that they represent an estimated upper bound for every agent subtree, and are used for pruning. On the other hand, ADOPT manages thresholds as lower bounds for every agent subtree, that allow agents to efficiently reconstruct partial solutions.

As explained in section 2, our ADOPT version associates with each assignment (either travelling in VALUE or COST messages) a timestamp. This permits COST messages to update the context of receiver, if some value is more recent than the value in the receiver context. On this respect, our ADOPT version and BnB-ADOPT act in the same way (in BnB-ADOPT timestamps are called counters, referred as ID in [2]).

4 Redundant Messages

In this section we present the results on redundant messages. They are valid for our ADOPT version and for BnB-ADOPT. In the following i, j and k are agents, all three executing either our ADOPT version or BnB-ADOPT. Agent i, holding variable x_i, *takes value* v when the assignment $x_i \leftarrow v$ is made and i informs of it to its children, pseudochildren and parent. The *state* of i is defined by (1) its value, (2) its context (as the set of values of agents located before i in its branch, timestamps are not considered part of the context), and (3) for each possible value v and each $j \in children(i)$, the lower and upper bounds $lb(v,j)$/$ub(v,j)$. A message msg sent from i to j is *redundant* if at some future time t, the collective effect of other messages arriving j between msg and t would cause the same effect, so msg could have been avoided. A message msg sent from i to j containing the assignment $x_i \leftarrow v$ with timestamp t *updates* $context_j[i]$ with timestamp t' if and only if $t > t'$.

Lemma 1. *If i takes value v_1 with timestamp t_1, and the next value it takes is v_2 (possibly equal to v_1) with timestamp t_2, there is no message with timestamp t for i st. $t_1 < t < t_2$.*

Proof. There is no VALUE sent from i with timestamp between t_1 and t_2, since v_1 and v_2 are consecutive values. About COSTs, they build their contexts from the information contained in VALUEs. Since no VALUE can include a timestamp between t_1 and t_2, no COST will contain it for i. □

Theorem 1. *If i sends to j two consecutive VALUEs with the same val, the second message is redundant.*

Proof. Let V$_1$ and V$_2$ be two consecutive VALUEs sent from i to j with the same value val with timestamps t_1 and $t_2, t_1 < t_2$. Between V$_1$ and V$_2$ any messages may arrive to j. When V$_1$ reaches j, it may happen:

1. V$_1$ does not update $context_j[i]$ (V$_1$ is discarded). When V$_2$ arrives: (a) V$_2$ does not update $context_j[i]$ (V$_2$ is discarded). Future messages will be processed as if V$_2$ would have not been received, so V$_2$ is redundant. (b) V$_2$ updates $context_j[i]$, that has timestamp t. There are two options: (i) $t_2 > t > t_1$ and (ii) $t_2 > t = t_1$. Option (i) is impossible because Lemma 1. Option (ii) is possible, but since $t = t_1$ the value contained in V$_2$ is already in $context_j[i]$. About future messages, every message accepted with timestamp t_2 of $context_j[i]$ would also be accepted if timestamp of $context_j[i]$ were t_1. Since there are no messages with timestamp between t_1 and t_2 for i, we conclude that V$_2$ is redundant.
2. V$_1$ updates $context_j[i] \leftarrow val$, timestamp t_1. When V$_2$ arrives: (a) V$_2$ does not update $context_j[i]$: as case (1.a). (b) V$_2$ updates $context_j[i]$: since V$_1$ updated $context_j$ and Lemma 1, the timestamp of $context_j[i]$ must be t_1. Updating with

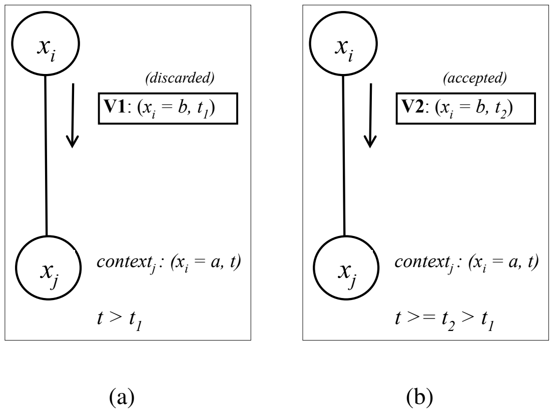
V$_2$ does not change $context_j[i]$ but its timestamp is put to t_2. Since there are no messages with timestamp between t_1 and t_2 (Lemma 1), any future message that could update $context_j$ with t_2 would also update it with t_1. So V$_2$ is redundant.

We have not considered the threshold contained in VALUE messages because both algorithms are complete, compute the optimum cost and terminate without the use of thresholds (they are included to increase efficiency). For the original ADOPT algorithm, the threshold is present in the implementation of its termination condition, although it is not essential to use the threshold to detect termination. The ADOPT termination condition is $LB = UB$, when the size of the bound interval shrinks to zero. Then, the cost of the optimal solution has been determined and agents can safely terminate. Actually, the threshold is always maintained between the LB and UB (using the **`MaintainThresoldInvariant`** method, [1]). So when $LB = UB$, the threshold equals UB. □

Theorem 2. *If k sends to j two consecutive COSTs with the same content (context, lower/upper bound) and k has not detected a context change, the second message is redundant.*

Proof. Let C$_1$ and C$_2$ be two consecutive COSTs sent from k to j with the same content, and $context_k$ has not changed between sending them. Any message may arrive to j between C_1 and C_2. Upon reception, the more recent values of C$_1$ (and later of C$_2$) are copied in $context_j$ (by **PriorityMerge** [2]). Copying C$_2$ more recent values in $context_j$ is not essential. Let us assume that these values are not copied. Then, some messages that would have been ignored between C_1 and C_2 will now be accepted. Since there is no context change between C$_1$ and C$_2$, these messages will necessarily include contexts compatible with k context, so they will update timestamps only, generating COSTs with the same bounds. At some point, j will receive all the more recent values of C$_2$ (necessarily before any context change). After this, j will behave as if it would have copied C$_2$ more recent values. So if those values are not copied, this will not cause any harm. Because of that, our proof concentrates on bounds. When C$_1$ arrives, it may happen:

1. C$_1$ is not compatible with $context_j$, its bounds are discarded. When C$_2$ arrives: (a) C$_2$ is not compatible with $context_j$, its bounds are discarded. So C$_2$ is redundant. (b) C$_2$ is compatible with $context_j$, its bounds are included in j. Since C$_1$ was not compatible, there is at least one agent above j that changed its value, received by j between C$_1$ and C$_2$. There are one or several VALUEs on its/their way towards k or k descendants. Upon reception, one or several COSTs will be generated. The last of them will be sent from k to j with more updated bounds. C$_2$ could have been avoided because a more updated COST will arrive to j. So C$_2$ is redundant.
2. C$_1$ is compatible with $context_j$, its bounds are included. When C$_2$ arrives: (a) C$_2$ is not compatible with $context_j$, its bounds are discarded. So C$_2$ is redundant. (b) C$_2$ is compatible with $context_j$, it bounds are included but this causes no change in j bounds, unless bounds are reinitialized. In this case there is at least one agent above j that changed its value, same as case (1.b). So C$_2$ is redundant. □



x_i
(discarded)
V1: ($x_i = b$, t_1)
x_j
$context_j$: ($x_i = a$, t)
$t > t_1$
(a)
x_i
(accepted)
V2: ($x_i = b$, t_2)
x_j
$context_j$: ($x_i = a$, t)
$t >= t_2 > t_1$
(b)

Fig. 1. Case V_1 discarded, and V_2 accepted

5 Examples

5.1 Example 1

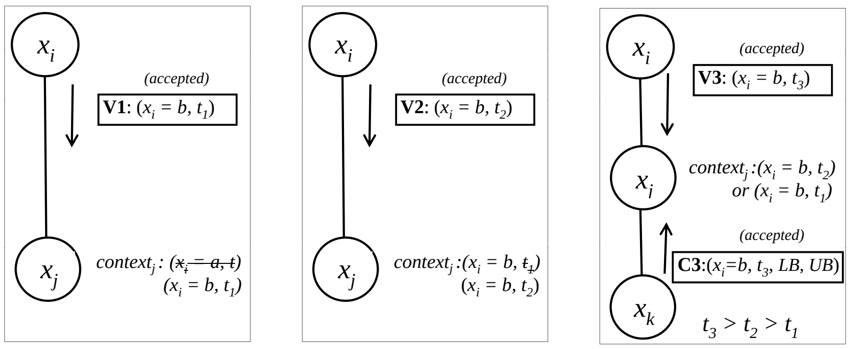
We present an example to illustrate Theorem 1. Consider agents i and j, holding variables x_i and x_j respectively. Agent i sends two consecutive VALUE messages V$_1$ and V$_2$ to agent j. As explained in the previous section, if V$_2$ is discarded then V$_2$ is redundant since it does not change agent j state and it has no effect in future message processing. Now, we will consider two possible scenarios in which message V$_2$ is accepted.

Case V_1 discarded, and V_2 accepted:

Consider agent i sending message V$_1$ to agent j informing the assignment $x_i = b$ with timestamp t_1 (Figure 1.a). Agent j has in its context the assignment $x_i = a$ with timestamp $t \geq t_1$, so V$_1$ is discarded. Then, message V$_2$ is sent to agent j with timestamp t_2 (Figure 1.b). If V$_2$ is accepted, then timestamp t_2 has to be more updated that t ($t_2 > t$). Then, we get $t_2 > t \geq t_1$. The case $t_2 > t > t_1$ is impossible, because from Lemma 1, there is no message with timestamp between t_1 and t_2 for i. The case $t_2 > t = t_1$ is possible, but the value contained in V$_2$ is already in j context (because $t = t_1$). In this case, we explain in the proof of Theorem 1, why V$_2$ is redundant with respect future messages.

Case V_1 accepted, and V_2 accepted:

Let us consider that agent i sends message V_1 to agent j informing the assignment $x_i = b$ with timestamp t_1, and is accepted (Figure 2.a). Then $context_j[i]$ is updated and its timestamp is set to t_1. Between V_1 and V_2 many messages may arrive to agent j with different timestamps, and $context_j$ might be updated. But if V_2 is accepted, we can assure that timestamp in $context_j[i]$ must be t_1 when message V_2 arrives (otherwise V_2 would not have been accepted). Upon V_2 reception, $context_j[i]$ remains the same (because V_1 and V_2 contains the same assignment) and only timestamp t_2 is updated



(a) (b) (c)

Fig. 2. Case V_1 accepted, and V_2 accepted

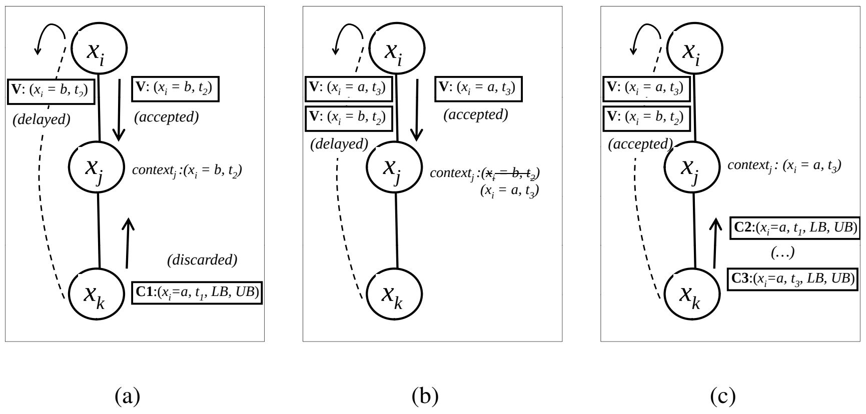
(Figure 2.b). Since there is no message with timestamp between t_1 and t_2 (Lemma 1) is easy to see that any VALUE message V_3 or COST message C_3 that would update $context_j[i]$ having timestamp t_2, will also update it if $context_j[i]$ would have timestamp t_1 (Figure 2.c). So updating t_2 in $context_j[i]$ makes no difference in future message processing. Therefore, V_2 is redundant.

5.2 Example 2

We present a second example to illustrate Theorem 2. Consider agents k and j, holding variables x_k and x_j respectively. Agent k sends two consecutive COST messages C_1 and C_2 to agent j. As explained in the previous section, if C_2 is discarded then C_2 is redundant since bounds LB and UB are not updated and its reception has no effect in future message processing. Now, we will consider two possible scenarios in which message C_2 is accepted.

Case C_1 discarded, and C_2 accepted:

Consider first a higher agent i connected with agents j and k. Variable x_i changes its value from a to b and agent i sends the corresponding VALUE messages. The VALUE message sent to agent j is received and $context_j[i]$ is updated, but the VALUE message sent to agent k is delayed (Figure3.a)). Now, agent k sends a COST message C_1 to agent j and this message is discarded because contexts are incompatible on variable x_i, since agent k has not received the last VALUE message from agent i yet (Figure 3.a)). Between message C_1 and C_2 other message may arrive, so let us assume that agent i changes its value again to a and sends the corresponding VALUE messages to agents j and k, but message to agent k is delayed once again (Figure 3.b). With these messages delayed, there is no context change in agent k and message C_2 is sent. Message C_2 is accepted (since context are now compatible) and it will update agent j bounds (Figure 3.c). However, as there are still 2 delayed VALUE messages from agent i to agent k, we can assure that when they arrive to agent k there will be a context change, so a new



x_i
V: (x_i = b, t_2)
(delayed)
V: (x_i = b, t_2)
(accepted)
x_j
context$_j$:(x_i = b, t_2)
(discarded)
x_k
C1:(x_i=a, t_1, LB, UB)
V: (x_i = a, t_3)
V: (x_i = b, t_2)
(delayed)
V: (x_i = a, t_3)
(accepted)
context$_j$:(~~x_i = b, t_2~~) (x_i = a, t_3)
(accepted)
context$_j$: (x_i = a, t_3)
C2:(x_i=a, t_1, LB, UB)
(...)
C3:(x_i=a, t_3, LB, UB)
(a)
(b)
(c)

Fig. 3. Case C_1 discarded, and C_2 accepted

COST message C_3 will be generated with more updated information (Figure 3.c). As we can see, message C_2 can be ignored since a message C_3 will eventually arrive to agent j and update its bounds with more recent information.

Case C_1 accepted, and C_2 accepted:

Consider that agent k sends message C_1 to agent j and it is accepted because contexts are compatible. In this case bounds LB and UB are updated in agent j. If message C_2 arrives right after C_1, it is easy to realize that is redundant, since it would copy the same information in agent j. However, between C_1 and C_2 some messages may arrive, and as result of this the bounds informed by C_1 might be reinitialized. This could only happen if a higher agent changes its value. So if a higher agent i changes its value from a to b, when the corresponding VALUE message arrives to agent j the bounds are reinitialized. After this agent i must change again its value to a if we want a scenario where C_2 message could be accepted. In this case, as there is no context change in agent k, we can assure that there is one or more delayed VALUE messages sent to agent k (same case as Figure 3.b)). When those delayed messages arrive to agent k there will be a context change, generating a new COST message C_3 with more updated information (same case as Figure 3.c). So message C_2 can be ignored since a message C_3 will eventually arrive to agent j and update its bounds.

6 New Versions

Temporarily, we define ADOPT$^+$ as our ADOPT version (see section 2) with the following changes: (1) the second of two consecutive VALUEs with the same i, j and val is not sent, (2) the second of two consecutive COSTs with the same k, j, $context$, lb and ub when k detects no context change is not sent. In the same sense, we temporarily define BnB-ADOPT$^+$ as BnB-ADOPT with the following changes: (1) the second of two consecutive VALUEs with the same i, j and val is not sent, (2) the second of two consecutive COSTs with the same k, j, $context$, lb and ub when k detects no context change is not sent.

Theorem 3. *ADOPT$^+$ (respectively BnB-ADOPT$^+$) terminates with the cost of a cost-minimal solution.*

Proof. By Theorems 1 and 2, messages not sent by ADOPT$^+$ (respec. BnB-ADOPT$^+$) are redundant so they can be eliminated. ADOPT (respec. BnB-ADOPT) terminates with the cost of a cost-minimal solution [1] (respec. [2]), so ADOPT$^+$ (respec. BnB-ADOPT$^+$) also terminates with the cost of a cost-minimal solution. □

But the new algorithm is not efficient because we have ignored thresholds. Looking for an adequate threshold management, we define ADOPT$^+$ (respec. BnB-ADOPT$^+$) as our ADOPT version (respec. BnB-ADOPT) algorithm with the following changes:

1. Agent i remembers for each neighbor agent j the last message sent.
2. A COST from j to i includes a boolean $ThReq$, set to true when (i) the threshold contained in the last VALUE message received could not be copied into j threshold (ADOPT case) or (ii) j threshold was reinitialized (ADOPT and BnB-ADOPT cases).
3. If j has to send i a COST equal to (ignoring timestamps) the last COST sent, the new COST is sent if and only if j has detected a context change between them.
4. If i has to send j a VALUE equal to (ignoring timestamps) the last VALUE sent, the new VALUE is sent if and only if the last COST that i received from j had $ThReq = true$.

7 Experimental Results

Performance is evaluated in terms of communication cost (messages exchanged) and computation effort (non-concurrent constraint checks). We consider also the cycles as the number of iteration the simulator must perform until the solution is found.

We tested our algorithms on binary random DCOPs. In addition, BnB-ADOPT was also tested on meeting scheduling and sensor networks. Binary random DCOP are characterized by $\langle n, d, p_1 \rangle$, where n is the number of variables, d is the domain size and p_1 is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.2, ..., 0.8 \rangle$. Costs are selected randomly from the set {0,..., 100}. Results appear in Table 1.a and Table 2.a, averaged over 50 instances.

In the meeting scheduling formulation, variables represent meetings, domain represent the time slot assigned for each meeting, and there are constraints between meetings that share participants [3]. We present 4 cases with different hierarchical scenarios. Results of the execution appear in Table 2.b, averaged over 30 instances.

In the sensor network formulation, variables represent targets, domain represent the time slots in which they might be tracked, and there are constraints between adjacent targets [3]. We present 4 cases with different topologies scenarios. Results of the execution appear in Table 2.c, averaged over 30 instances.

On random DCOPs, ADOPT$^+$ showed clear benefits on communication costs with respect to our ADOPT version. It divided the number of exchanged messages by a factor from 1.1 to almost 3, maintaining the number of cycles practically constant.

(a) Random DCOPs.

p_1	#Messages	#NCCC	#Cycles
	524	2,877	**29**
0.2	**468**	**2,827**	**29**
	1,898,519	39,870,169	**82,565**
0.3	**1,003,300**	**36,330,792**	82,604
	39,456,612	1,032,565,939	**1,461,551**
0.4	**16,810,366**	**891,171,565**	1,461,910
	410,414,194	12,187,384,497	12,826,157
0.5	**145,182,982**	**10,259,021,210**	**12,811,734**

Table 1. Results of our ADOPT version (first row) compared to ADOPT$^+$ (second row)

(a) Random DCOPs

p_1	#Messages	#NCCC	#Cycles
	1,393,339	11,002,964	**53,065**
0.4	**657,714**	**10,827,544**	53,074
	11,055,317	85,306,147	360,297
0.5	**4,570,180**	**84,383,065**	**360,167**
	68,116,304	508,186,224	**1,987,584**
0.6	**24,809,153**	**499,214,418**	1,987,915
	184,735,389	1,366,404,208	4,740,277
0.7	**59,900,198**	**1,339,303,291**	**4,740,040**
	293,922,594	2,153,776,854	**6,873,799**
0.8	**86,233,163**	**2,112,858,127**	6,873,805

(b) Meeting Scheduling

	#Messages	#NCCC	#Cycles
	96,493	697,774	**4,427**
A	**35,767**	**690,786**	**4,427**
	182,652	879,417	**7,150**
B	**69,453**	**801,384**	7,150
	34,374	167,058	**1,278**
C	**13,862**	**157,995**	1,278
	47,729	155,833	**1,733**
D	**20,386**	**141,816**	1,733

(c) Sensor Network

	#Messages	#NCCC	#Cycles
	7,040	15,097	**226**
A	**1,074**	**14,514**	**226**
	10,258	23,597	**320**
B	**1,859**	**22,659**	**320**
	19,563	118,795	**981**
C	**6,236**	**116,434**	**981**
	56,398	169,748	**1,660**
D	**17,484**	**167,658**	**1,660**

Table 2. Results of BnB-ADOPT (first row) compared to BnB-ADOPT$^+$ (second row)

In the same sense, experiments with random DCOPs show that our algorithm BnB-ADOPT$^+$ reduces the number of messages by a factor from 2 to 3 when connectivity increases with respect to BnB-ADOPT. For meeting scheduling, messages are reduced by a factor of at least 2, and for sensor networks, by a factor between 3 and 6. We have achieved important savings for all problems tested. BnB-ADOPT$^+$ was able of processing only half of messages (or less) and reach the optimal solution maintaining the number of cycles practically constant. This can be sustained in the empirical fact that there is a large reduction in messages, a slight reduction in the non-concurrent constraint checks, and the number of cycles remains constant.

8 Conclusion

We have presented two contributions to increase the performance of distributed constrained optimization algorithms. First, we describe our version of the ADOPT algorithm, which saves some messages with respect to the original algorithm. Secondly, we present theoretical results to detect redundant messages in our ADOPT version and in BnB-ADOPT. Using these results we generate two algorithms, called ADOPT$^+$ and BnB-ADOPT$^+$, which caused substantial savings with respect to our ADOPT version and BnB-ADOPT, when tested on commonly used benchmarks of the DCOP community.

Acknowledgements

Authors sincerely thank reviewers for their useful criticisms.

References

1. P. J. Modi, W.M. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, (161):149–180, 2005.
2. W. Yeoh, A. Felner, and S. Koenig. Bnb-adopt: An asynchronous branch-and-bound DCOP algorithm. *Proc. of AAMAS-08*, pages 591–598, 2008.
3. Z. Yin. USC dcop repository. Meeting scheduling and sensor net datasets, 2008.