An Any-space Algorithm for Distributed Constraint Optimization*

Anton Chechetka and **Katia Sycara**
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{antonc, katia}@cs.cmu.edu

Abstract

The Distributed Constraint Optimization Problem (DCOP) is a powerful formalism for multiagent coordination problems, including planning and scheduling. This paper presents modifications to a polynomial-space branch-and-bound based algorithm, called NCBB, for solving DCOP, that make the algorithm any-space. This enables a continuous tradeoff between $O(bp)$ space, $O(bp^{H+1})$ time complexity and $O(p^w + bp)$ space and $O(bHp^{w+1})$ time, where p is variables domain size, H is Depth-First Search (DFS) traversal depth of constraint graph, b is the branching factor of DFS tree, and w is the context width of the tree. This flexibility allows one to apply NCBB to areas, where the limited amount of available memory prevents one from using more efficient exponential space algorithms. Sensor networks is an example of such domain. We demonstrate both theoretically and empirically that caching does not lead to an increase in complexity even under assumption that all cache lookups fail. We also show experimentally that the use of cahing leads to significant speedups of problem solving.

Introduction

The Distributed Constraint Optimization Problem (DCOP) is a popular problem formulation for distributed reasoning, where multiple agents need to optimize a cost function expressed as a sum over a set of distributed valued constraints (Hirayama & Yokoo 1997). Distributed planning and scheduling is a major application area for DCOP: meeting scheduling in large organizations (Maheswaran *et al.* 2004; Garrido & Sycara 1996; Liu & Sycara 1994), multiagent plan coordination (Cox, Durfee, & Bartold 2005), job shop scheduling (Liu & Sycara 1995), and sensors allocation to targets for tracking in distributed sensor networks (Scerri *et al.* 2003) are among the examples.

Because DCOP is not an algorithm, but just a problem formulation, one needs efficient solution techniques to successfully apply DCOP to said and other domains. As a response to this need, several algorithms have appeared recently, such as ADOPT (Modi *et al.* 2005), DPOP (Petcu & Faltings 2005), OptAPO (Mailler & Lesser 2004) and NCBB (Chechetka & Sycara 2006). One remaining difficulty, however, is that to choose an appropriate algorithm for a given task is a problem in itself: algorithms scale differently depending on communication cost and constraints number (Chechetka & Sycara 2006; Davin & Modi 2005). Also, the memory requirements differ quite drastically, from polynomial (ADOPT, NCBB) to exponential (DPOP). This may rule out using high-performance DPOP on nodes with scarce memory, such as distributed sensor net nodes (Mainwaring *et al.* 2002) and embedded systems.

We adopt an any-space caching scheme of (Darwiche 2001) with minor modifications and show both theoretically and empirically that the overhead concerned with caching is insignificant, so the overall performance of the algorithm is improved in an overwhelming majority of cases. We demonstrate on a distributed sensor network dataset that the improvements can be significant, up to a factor of 2.

Any-space property of NCBB with caching allows one to apply this algorithm over a wide range of hardware platforms and extract performance gains from the memory available in excess of its minimal requirements. This is an advantage over state of the art polynomial-memory algorithms that do not have this feature.They limit themselves to the minimally necessary memory, even if more space is available. Denote p the domain size of the variables, H the depth of Depth-First Search (DFS) traversal of constraint graph, b the branching factor of DFS tree, and w the context width of the tree ($w \leq H$ by definition; more detailed definitions are presented in later sections). NCBB with caching allows the user to manually tweak its behavior and asymptotical time complexity from $O(bp^{H+1})$ (no caching) to $O(bHp^{w+1})$ (full caching). The latter option is especially preferable when context width w of the tree is significantly smaller than H. Corresponding space requirements are $O(bp)$ to $O(p^w + bp)$. One can see that it is an example of a time-space tradeoff.

DCOP

A Distributed Constraint Optimization problem consists of

- a set of n variables $V = \{x_1, \ldots, x_n\}$
- a set of discrete finite domains for each of the variables $D = \{D_1, \ldots, D_n\}$

*This research was supported by DARPA grant HR0011-05-1-0020.

- a set of m constraints $f = \{f_1, \ldots, f_m\}$ where each constraint is a function $f_i : D_{i_1}, \ldots, D_{i_j} \rightarrow N \cup \infty$
- a set of k agents $A = \{a_1, \ldots, a_k\}$
- a distribution mapping $Q : V \rightarrow A$. It assigns each variable to exactly one agent. This agent is said to *own* the variable. The variable owner has exclusive control over the variable value and knows all the constraints that involve that variable. An agent is assumed to not know about the constraints not involving its variable.

The goal of the agents is to collectively find an assignment for all the variables $B^* = \{d_1, \ldots, d_n | d_i \in D_i\}$ such that the global cost is minimized. The global cost is defined as a sum of all the constraints

$$F(B) = \sum_{i=1}^{m} f_i(B)$$

It has been shown (Modi 2003) that DCOP is NP-complete.

In this article we consider a restricted version of DCOP. Like the majority of the literature (Mailler & Lesser 2004; Modi *et al.* 2005), we assume all the constraints to be binary, that is to depend on exactly two variables, although our approach can in principle be extended to k-ary constraints. We also assume that each agent owns exactly one variable. The terms *agent* and *variable* will be used interchangeably.

We will use the term *constraint graph* to refer to a graph where variables are nodes and two nodes have an edge between them if and only if there is a constraint involving these variables. We assume that two agents have a direct communicational link if and only if they share a constraint. The link is lossless and deliveres the messages in the same order that they were sent.

NCBB Algorithm

NCBB, or no-commitment branch and bound search, is a polynomial-space algorithm for solving DCOP. It is a conventional branch and bound search (Lawler & Wood 1966) with modifications to make it efficient for multiagent settings. The main improvements are incremental computation and communication of lower bounds on solution cost and concurrent search in non-intersecting areas of the search space by different agents. We refer the interested reader to (Chechetka & Sycara 2006) for comprehensive description of communicaton scheme, data structures organization, and details of lower bound propagation and pruning. Here we concentrate on a caching scheme for NCBB.

Variables Prioritization

Before the algorithm can be executed, the agents need to be prioritized in a depth-first search (DFS) tree, in which each agent has a single parent (except for the root agent that has no parents) and multiple children. Formally, each agent x_i's neighbors in the constraint graph are assigned to the following sets:

- an ordered set $ancestors_i$ such that $ancestors_i[k]$ is higher in the tree than $ancestors_i[k+1]$ and $ancestors_i[last]$ is the parent of x_i.

```
function NCBBMainLoop()
1:  if (¬ IAmRoot) updateContext();
2:  while (¬ "search finished" received)
3:     search();
4:     if (¬ IAmRoot) updateContext();
5:     else break;
6:     end if;
7:  end while;
8:   recallSolution();
9:  for (∀ x ∈ children)
10:     send "search finished" to x;
11:  end for;
end NCBBMainLoop;
```

Figure 1: NCBB main loop

- a set $children_i$ of immediate x_i's children in DFS ordering
- for each child $c_j \in children_i$ a set $dscnd_i[c_j]$ of descendants belonging to the same subtree of the DFS tree as c_j. Note that $c_j \in dscnd_i[c_j]$.

A property of these sets is

$$(\bigcup_{c_j \in children_i} dscnd_i[c_j]) \cup ancestors_i \equiv neighbors_i$$

It is important that constraints are only allowed between agents that are in an ancestor-descendant relationship in the DFS tree. One should also note that an agent does not have information about all its ancestors or descendants in the tree, but only about those with whom it shares a constraint.

Any constraint graph can be ordered into some DFS tree using distributed algorithm from (Lynch 1996). Pseudo-tree ordering (Chechetka & Sycara 2005) can also be used after minor modifications to the search algorithm. This allows to apply NCBB to problems without centralizing the information about all the constraints.

Auxiliary Definitions and Properties

Definition 1 *Agent cost for agent x and assignment B*

$$AgentCost(x, B) \equiv \sum_{y \in ancestors_x} f_{x,y}(B(x), B(y))$$

where $B(x)$ is the value of variable x under assignment B, $f_{x,y}$ is the constraint depending on x and y.

Definition 2 *Subtree cost for subtree rooted at agent x and assignment B*

$$SubtreeCost(x, B) \equiv \sum_{y \in x \cup DFSdesc(x)} AgentCost(y, B)$$

where $DFSdesc(x)$ is a set of all x's descendants in the DFS tree, including the agents that are not x's neighbors in the constraint graph.

Definition 3 *For incomplete assignments B', define*

$$AgentCost^*(x, B') \equiv \min_{B \supset B'} AgentCost(x, B)$$

$$SubtreeCost^*(x, B') \equiv \min_{B \supset B'} SubtreeCost(x, B)$$

function **search**()
20: add all children to $idle$ set;
21: $\forall d_i \in D_i$: costSoFar(d_i) $\leftarrow \sum_{y \in ancestors_i} f(d_i, val(y))$;
22: $\forall d_i \in D_i \; \forall y \in$ children: visited(y,d_i) $\leftarrow$ **false**;
23: pending = $\emptyset$;
24: **while** (idle $\cup$ pending $\neq \emptyset$)
25: **for** ($\forall y \in$ idle)
26: choose my_value(y) : visited(y, my_value(y)) = false;
27: visited(y, my_value(y)) $\leftarrow$ true;
28: send my_value(y) to $\forall z \in$ dscnd(y);
29: send "start search, bound - costSoFar(my_value(y))" to y;
30: idle = idle $\setminus$ y; pending = pending $\cup$ y;
31: **end for;**
32: receive cost from y $\in$ pending;
33: pending = pending $\setminus$ y;
34: costSoFar(my_value(y)) $\leftarrow$ costSoFar(my_value(y)) + cost;
35: **if** ($\exists d_i \in D_i$: (costSoFar(d_i) $<$ bound) $\wedge$
36: visited(y, d_i) = false)
37: idle $\leftarrow$ (idle $\cup$ y);
38: **end if;**
39: **end while;**
40: result = $\min_{d_i \in D_i}$ costSoFar(d_i);
41: send result to parent; **return** result;
end search;

Figure 2: search() function for agent x_i

Because the variables in different subtrees do not share any constraints, we can write a dynamic programming recursive relation

$$\begin{aligned} &SubtreeCost^*(x_i, B'(all\_ancestors_i)) = \\ &\min_{d_i \in D_i} (AgentCost(X_i, B' \cup (x_i = d_i)) + \\ &+ \sum_{y \in children_i} SubtreeCost^*(y, B' \cup (x_i = d_i))), \end{aligned} \quad (1)$$

where $all\_ancestors_i$ denotes all ancestors of x_i in the DFS tree, not only those with direct links to x_i.

Main Loop

Figure 1 lists the outline of NCBB's main loop. An agent x_i listens to value announcements form $ancestors_i$ until x_i's parent orders it to start searching. `updateContext()` updates x_i's knowledge about $ancestors_i$ values. It returns $true$ iff the last message instructed x_i to start search. An instruction to start search includes the new upper bound on $SubtreeCost^*(x_i, B'(all\_ancestors_i))$, which is also recorded in a `bound` variable.

After receiving an instruction to search, agent x_i tries to determine its value d_i that achieves $SubtreeCost^*(x_i, B'(all\_ancestors_i))$.

According to the recursive relation (1), it executes the `search()` function, outlined in Figure 2. It tries every possible value in D_i for every subtree rooted in x_i's children. Then it looks for $SubtreeCost^*(x_i, B'(all\_ancestors_i))$ in `costSoFar` map and returns this value to x_i's parent. By construction, after all $|D_i| \times |children_i|$ queries have been answered, `costSoFar` is a map

$$\begin{aligned} &d_i \rightarrow \\ &(AgentCost(X_i, B' \cup (x_i = d_i)) + \\ &+ \sum_{y \in children_i} SubtreeCost^*(y, B' \cup (x_i = d_i))). \end{aligned}$$

As in regular branch and bound algorithm, search space may be pruned by removing from consideration d_i, for which

$$costSoFar(d_i) > bound$$

This algorithm is complete and requires only polynomial memory, more precisely, $O(|D_i| \times |neighbors_i|)$ for agent x_i. It is an advantage over exponential memory algorithms, such as DPOP. The recursive nature of the algorithm leads one to the time complexity estimate of $O(bp^{H+1})$ (Theorem 1), where p is the size of variables' domains (assuming equal domain sizes), b is the branching factor of the DFS tree, and H is the depth of the DFS tree. However, the fact that an agent forgets everything about its previous search result as soon as it gets a new search instruction leads one to an idea that extra available memory may be used to store the results of previous searches and reuse them. As was demonstrated in (Darwiche 2001), this approach results in a continuous time-space tradeoff that can be controlled by the user to maximize the performance of the algorithm for a given problem and hardware.

Caching in NCBB

Motivating Example

To show why it might be beneficial to store the results of the previous searches, consider a simple example: a problem with structure depicted on Figure 3(a). Suppose all the variables have binary domains and variable A is the root of the DFS tree.

Let us trace the execution of NCBB on this problem. First A chooses one of its values (supposes it chooses 0) to try on B's subtree (note that because A is the root of the tree, it does not wait for any agents to communicate their values to it). Upon receiving the messages from A, B consequently tries both of its values on C's subtree. For each of the two B's values C reports the minimal cost

$$SubtreeCost^*(C, (A = 0, B = b))$$

to B. B then reports the minimal cost over its subtree to A. Next A orders B to calculate

$$SubtreeCost^*(B, (A = 1))$$

and B again queries C to get

$$SubtreeCost^*(C, (A = 1, B = b)).$$

To calculate these optimal costs, C in turn queries D. Note, however, that for the given problem

$$\begin{aligned} &SubtreeCost^*(C, (A = 0, B = b)) = \\ &= SubtreeCost^*(C, (A = 1, B = b)) \end{aligned}$$

because no variable in C's subtree has a constraint with A. Therefore, if C stored the results of previous searches, it would be able to answer new B's queries right away, without querying D with the same question multiple times . Because sending a message is usually a lot more expensive than doing a cache lookup, storing the past results would result in faster problem solving.

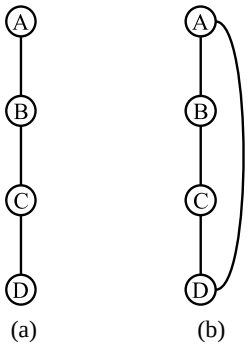


Figure 3: Caching motivating examples

Distinguishing Between Ancestors Assignments

The example presented in the previous section confirms that caching search results might lead to efficiency gains, but it also poses an important question: how does an agent x_i determine if it has the search result corresponding to the given assignment to $all\_ancestors_i$ variables? This is especially problematic in NCBB, because an agent x_i does not have any information about not just assignments, but even existance of variables other than its direct neighbors. Another simple example (Figure 3(b)) demonstrates that knowing only direct neighbors' values is not enough to make such a decision:

$$SubtreeCost^*(C, (A = 0, B = b)) \neq \\ \neq SubtreeCost^*(C, (A = 1, B = b))$$

because there is a constraint $A-D$. On the other hand, agent C does not observe the difference in A's assignment. This means that using caching will require the agents to give up some of the privacy and announce their assignments to third parties (by a third party for an agent x_i here we understand any agent that does not share a constraint with x_i).

The question of which variables from $all\_ancestors_i$ influence the value $SubtreeCost^*(x_i, B'(all\_ancestors_i))$ has been answered in an extensive literature on Bayesian Networks (Jordan 2004; Jensen 1996): if assignments B_1 and B_2 to all variables in $all\_ancestors_i$ agree on all the values of variables from set $context_i$ such that

Definition 4

$$context_i \equiv \{y \in all\_ancestors_i \ \ s.t. \\ \exists z \in \{x_i \cup DFSDesc(x_i)\} : f_{y,z} \in f\}$$

then $SubtreeCost^*(x_i, B_1) = SubtreeCost^*(x_i, B_2)$.

Indeed, let us write $SubtreeCost^*(x_i, B)$ (B is an assignment to all variables in $all\_ancestors_i$) explicitly:

$$\min_{B \supset B'} \sum_{y \in x_i \cup DFSDesc(x_i)} \sum_{z \in ancestors_y} f_{y,z}(B(y), B(z))$$

The sum being minimized depends only on values of x_i, variables in $DFSDesc(x_i)$ and $context_i$. Because $context_i \subseteq all\_ancestors_i$ and

$$\{x_i \cup DFSDesc(x_i)\} \cap all\_ancestors_i = \emptyset,$$

we get

$$SubtreeCost^*(x_i, B') = SubtreeCost^*(x_i, B'|_{context_i})$$

where $B'|_{context_i}$ means denotes the restriction of B' to variables in $context_i$.

A corollary of this fact is that the memory complexity of NCBB with full caching is $O(p^w + bn)$, where p is the domain size of the variables (assuming equal domain sizes for all variables), and

$$w = \max_{i \in 1..n} |context_i|$$

is called the *context width* of the DFS tree.

```
function discoverContext()
50:  for ∀ y ∈ children
51:     receive context_y from y;
52:  end for;
53:  context_i ← ⋃_{y∈children} context_y \ i;
54:  send context_i to ancestors_i[last];
end discoverContext;

function searchCaching()
60:  result = lookupCache(val(context_i))
61:  if result ≡ ∅;
62:     result = search();
63:     cacheInsert(val(context_i) → result)
64:  else;
65:     send result to parent;
66:  end if;
end discoverContext;

in search() function (Fig. 2) replace line 29: with:
29: send "start search, bound - costSoFar(my_value(y)),
                                  val(context_y)" to y;
```

Figure 4: Modifications to NCBB (to be executed by agent x_i) that allow caching

Implementation in NCBB

Given that only variables in $context_i$ influence $SubtreeCost^*(x_i, B)$, we present the modifications to the original NCBB algorithm that allow the agents to cache the results of previous searches.

First, every agent x_i needs to become aware of what variables constitute $context_i$. This is achieved by a bottom-up propagation phase before execution of the main loop. Each agent executes `discoverContext()` function listed in Figure 4.

Second, instead of `search()` function in the main loop, agents execute `searchCaching()`, which first tries to locate the corresponding result in the cache, and calls `search()` if the lookup fails. The value of $context_i$ is communicated by i's parent in a "start search" message.

Finally, in `search()` function line (29:) is changed to include the value of $context_y$ in a message to child y so that the child can perform a cache lookup.

Complexity Estimates

Let us estimate the time complexity of NCBB with and without caching. First, if no caching occurs then

Theorem 1 *If all variables have domains of size p, the DFS tree has a branching factor b and depth H, then NCBB without caching has time complexity of $O(bp^{H+1})$.*

Proof: Agent x_i receives a search request for each possible assignment to $all\_ancestors_i$. If $|all\_ancestors_i| = h$ and every variable has domain size n, then

$$requests(x_i) = p^h.$$

For each search request x_i sends n requests to each of its b children. Because the agents perform computations concurrently, to obtain the time complexity of the algorithm we need to sum the time over the longest root-leaf path in the DFS tree:

$$T \sim \sum_{h=0}^{H}(p^h \times pb) = \sum_{h=0}^{H}(bp^{h+1}) = O(bp^{H+1})\square$$

If the amount of memory available is enough to cache all the intermediate results, we get a significant reduction in time complexity:

Theorem 2 *If all variables have domains of size p, the DFS tree has a branching factor b, depth H and context width w, cache lookups and inserts take constant time, and every agent has enough memory to store all the search results then NCBB with caching has time complexity of $O(bHp^{w+1})$.*

Proof: Because intermediate search results are cached, an agent x_i receives $p^{|context_p|}$ search requests, where x_p is x_i's parent. For each request it either finds the corresponding value in cache and returns it (this takes constant time c) or processes bp search requests to its children. Therefore, the worst-case time complexity is

$$T \sim \sum_{h=0}^{H}(p^{|context_{i_{h-1}}|} \times (pb + c)) \leq \sum_{h=0}^{H}(bp^{w+1} + cp^w) = O(bHp^{w+1})$$

where i_{h-1} is the index of $(h-1)^{th}$ variable along the root-leaf path. □

It might be the case that an agent does not have sufficient memory to store optimal subtree cost values for each possible instantiation of $context$. This is especially a concern for applications such sensor nets. Following (Darwiche 2001), we use the notion of *cache factor*:

Definition 5 *Cache factor cf_i of the node x_i is the relation*

$$cf_i = \frac{capacity_i}{\#context_i}$$

where $capacity_i$ is the number of enties of form

$$val(context_i) \rightarrow SubtreeCost^*(x_i, val(context_i))$$

that x_i's cache can accomodate, and $\#context_i$ is the number of all possible instantiations of x_i's context.

If all variables have domains of size p, then $\#context_i = p^{|context_i|}$.

We now show that even for $cf = 0$, that is when all cache lookups fail, the asymptotical time complexity of NCBB does not increase as compared to no caching case:

Theorem 3 *If all variables have domains of size p, the DFS tree has a branching factor b and depth H, and cache lookups and inserts take constant time, then NCBB with caching has time complexity of $O(bp^{H+1})$.*

Proof: Again, as in theorem 1, agent x_i that has depth h in the DFS tree receives p^h search requests. For each request in addition to $O(bp)$ time on requests to its children it needs to spend a constant amount c of time on cache lookup and insert. Summing over the longest root-leaf path we get

$$T \sim \sum_{h=0}^{H}(p^h \times (pb + c)) = \sum_{h=0}^{H}(bp^{h+1} + cp^h) = O(bp^{H+1}(1 + \tfrac{c}{bp})) = O(bp^{H+1})\square$$

Because Theorem 3 states that even in the worst case of all cache lookups failing the complexity of NCBB with and without caching is asymptotically the same, an optimal strategy would be to store as many intermediate results as memory allows. This behavior makes NCBB with caching an any-space algorithm and gives user the ability to manually trade off solution time for the memory available to the algorithm.

Implementing this behavior does not need any modifications to the NCBB algorithm itself. Instead the implementation of `cacheInsert()` function has to be changed so that inserting a new element into cache may fail. One of the simplest implementation, suggested in (Darwiche 2001), is to cache the new results until the memory is exhausted and do not cache anything after that. However, we found the following heuristic to yield better results: store the most recent search results. When the cache is exhausted, before inserting the new result remove the least recent result from it. The experimental results that we report all were obtained using this heuristic.

Evaluation

Evaluation Metric

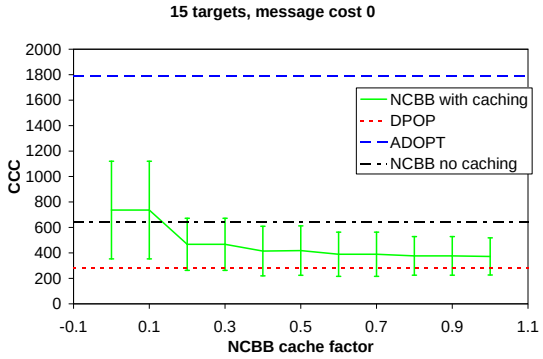
The main metric for evaluation was concurrent constraint checks (CCC) (Meisels *et al.* 2002). It combines computational and communicational cost of the problem solving process and takes parallelism into account. The unit of cost is time needed to perform one constraint check. The cost of a message is defined in terms of constraint checks and is assumed to be independent of message size. We also assumed that the cost of a cache lookup equals to 1 constraint check.

Concurrent constraint checks can be easily added as a benchmark for a simulated distributed system by adding a special field to every message being sent. This method works both for single-processor simulations and for real distributed systems. To account for dependencies between different agents and the time that some agents wait for messages from others, every message includes a field with the local value of the metric at a sender agent. Upon receiving the message, the receiver updates its local metric value according to

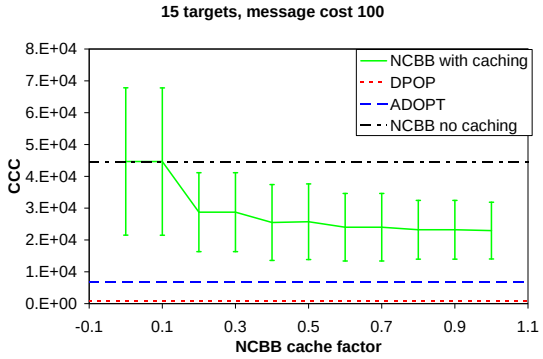
$$v_{receiver} = \max\{v_{receiver}, v_{sender} + cost_{message}\},$$

where v is a "tick counter" for a given agent. v receives an increment of 1 when its owner performs a constraint evaluation. The cost to find a solution is measured as

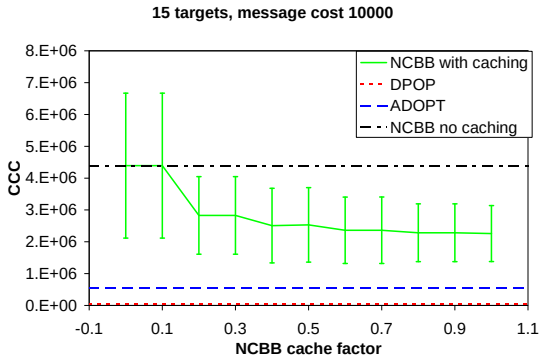
$$\max_x v_x$$



(a)



(b)



(c)

Figure 5: Target allocation results, 15 targets. X is the agents' cache factor, Y is the number of Concurrent Constraint Checks needed to find a solution. Horizontal lines represent algorithms that do not have the ability to vary cache size.

which is an approximation of the time it would take the system to solve the problem if a constraint evaluation took 1 time unit, and message delivery took $cost_{message}$ time units.

Datasets and Algorithm Parameters

We have evaluated NCBB on the distributed sensor network dataset from the University of Southern California (Pearce 2005) and on a randomly generated set of distributed graph coloring problems.

In sensor network domain 54 sensors were used to track multiple targets. Every target is visible to 4 sensors and to track a target successfully 3 sensors are needed. A sensor can only track one target. The problem is to minimize the costs of targets that are not tracked successfully. The details of mapping target allocation to DCOP are presented in (Modi *et al.* 2001). Every possible target-sensor combination in this formulation corresponds to a variable ("possible" means that the target is within the sensing range of the sensor). Every valid combination of sensors that may be used for tracking a given target corresponds to a value for every variable related to this target. The problems in the repository are already converted to standard DCOP formulation.

The tracking problem does not involve any dynamics of the targets: the configuration of targets and sensors is assumed to be fixed. One approach for applications with evolving sensors and targets configuration is to re-solve a static problem from scratch on every time step. An algorithm that would reuse the solution of the previous time step to solve the problem of the next time step faster is an important direction for future work.

The random graph coloring dataset consisted of problems with 16 variables and 32 constraints. Every variable has a domain of size 3, and randomly generated constraints map pairs of variables values to integers in the range from 0..100. The distribution of constraint costs was uniform.

In the experiments we varied global cache factor, that is the value cf, that was the same for all the agents in the problem.

Results

Target Tracking The experimental results for target tracking are shown in Figure 5. On all plots X axis denotes cache factor (the ratio between the number of entries that fit in the available memory and the total number of context instantiations). Y axis denotes concurrent constraint checks needed to solve the problem.

There are also horizontal lines on the plots, that correspond to the performance of algorithms that do not have the ability to tweak the amount of memory being used. That is why their value does not depend on cache factor - only NCBB with caching uses cache factor to determine how much memory it can use. These results are included for the reader to be able to compare the efficiency of NCBB and other available algorithms and to see the impact of caching more clearly.

Figure 5 presents the performance results on the target tracking dataset for 3 values of message cost: 0, 100 CCC, and 10000 CCC. For comparison we also present the performance of ADOPT and DPOP under same conditions. One can conclude that for all communcation costs caching leads to significant speedups of the problem solving process. Pure computational cost (that is the results for message cost 0) of NCBB with caching is significantly better than that of ADOPT and very close to DPOP (Figure 5(a)). For expensive communication (Figure 5(b), 5(c)) ADOPT and DPOP are significantly faster than NCBB on this type of problems. Here and in the rest of the section "faster" means "spend less concurrent constraint checks to find a solution".

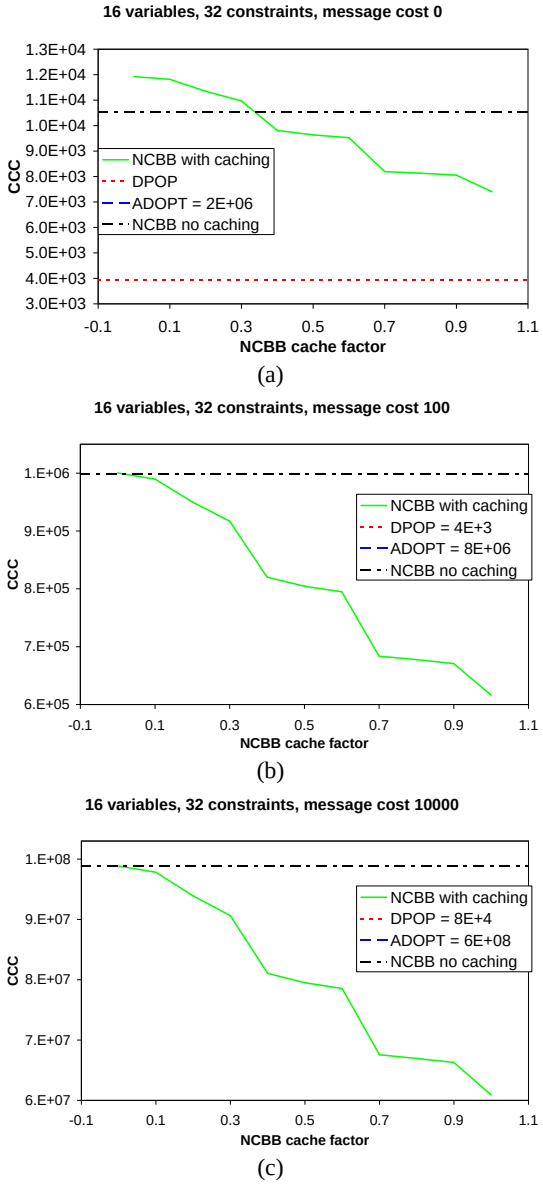


Figure 6: Random graph coloring, 16 agents, 32 constraints. X is cache factor, Y is the number of CCC needed to find a solution. Horizontal lines represent algorithms that cannot vary cache size. When a horizontal line is too high or too low to be plotted, its position is stated in the legend.

Graph Coloring We have also investigated the impact of caching on solving randomly generated graph coloring problems (Figure 6). Here one can observe the performance gain of about 40% due to caching. Note that because the performance of NCBB, ADOPT and DPOP differ dramatically on this dataset, it was not possible to plot all the results on the same Y scale. Because of that we have put into the legend information about locations of lines that did not fit on the plots (ADOPT for all three cases, because its cost is too high, and DPOP on figure 6(b), 6(c), because its cost is too low to be displayed).

A major difference with target tracking is that on random graph coloring the performance gain is roughly linear in cache factor, while on target tracking problems varying cache factor in the range from 0.5 to 1 has little effect on performance. We believe that this stark difference is due to the different amount of search space pruning that the algorithm is able to achieve on problems with different structure: on target allocation problems NCBB prunes much smaller portion of the search space than on random graph coloring problems.

One can see that for both sets of problems and for almost all conditions NCBB with caching has better performance than without caching, which is in line with theoretical estimates in Theorems 1 and 3. The only exceptions are cases of free communication, which is an unrealistic assumption and was included in the experiments to isolate the computational component of the solution complexity from communicational one.

Performance Analysis On both problem sets DPOP performs better than NCBB with caching, and one might think that using DPOP is always preferable to NCBB. However, DPOP requires each agent to have enough memory to store all instantiations of context and corresponding cost. The required memory is exponential in context size. If even one of the agents does not have enough memory, DPOP cannot be applied. However, while there may not be enough memory to run DPOP, there may be more memory available on the agents, than is required by polynomial time algorithms. In this intermediate situation using NCBB with caching provides maximal benefit in terms of performance. Although the efficiency of DPOP is rarely achieved by NCBB, controllable space-time tradeoff provides a way to exploit the extra available memory and obtain a performance improvement over purely polynomial-space algorithms.

The difference in performance between NCBB with full caching and DPOP may seem surprising, especially given the same asymptotical complexity of the two algorithms. However, if one considers the communication pattern of the algorithms, the reason for such a difference is clear. NCBB sends $O(bHp^{\omega+1})$ *concurrent*[1] fixed length messages, while DPOP sends $O(bH)$ *concurrent* messages of size $O(p^{\omega+1})$. Because the CCC metric does not take message size into account, communicational cost of NCBB is $O(p^{\omega+1})$ times greater than that of DPOP under this metric. Therefore, for large messsage costs, when communicational complexity dominates, DPOP is $O(p^{\omega+1})$ times faster than NCBB. One can also argue that in reality the communicational protocols are designed so that information is sent in packets, and if the message is too large, it is split into several packets, so the cost of sending a large message is roughly proportional to its size. This will need to be taken into account when designing more accurate performance metrics than CCC.

[1]Here *concurrent* means that the messages that can be delivered in parallel, i.e. sent simultaneously and do not have a common source or destination, are counted as one message, by analogy with concurrent constraint checks

Conclusions and Future Work

We have presented an algorithm for solving DCOP with controllable time-space and demonstrated experimentally the performance improvements over the original version. However, several important issues need to be addressed in the future work. First, to make informed decision about the trade-off point one needs a theoretical complexity estimate not only for extreme cases of no caching and full caching, but also for the continuum in between. Note that the estimates from (Darwiche 2001) do not apply in our case because of the asynchronous nature of computation by multiple agents. Second, the algorithm needs to be generalized to efficiently handle $k-$ary constraints to avoid overheads of converting "natural" problem formulations with $k-$ary constraints to binary DCOP.

References

Chechetka, A., and Sycara, K. 2005. A decentralized variable ordering method for distributed constraint optimization. In *Proceedings of the Forth International Joint Conference on Autonomous Agents and Multi-Agent Systems poster session*.

Chechetka, A., and Sycara, K. 2006. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems*.

Cox, J. S.; Durfee, E. H.; and Bartold, T. 2005. A distributed framework for solving the multiagent plan coordination problem. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*.

Darwiche, A. 2001. Recursive conditioning. *Artif. Intell.* 126(1-2):5–41.

Davin, J., and Modi, P. J. 2005. Impact of problem centralization in distributed constraint optimization algorithms. In *Proceedings of Forth International Joint Conference On Autonomous Agents and Multi-Agent Systems*.

Garrido, L., and Sycara, K. 1996. Multi-agent meeting scheduling: preliminary results. In *1996 International Conference on Multi-Agent Systems (ICMAS '96)*, 95 – 102.

Hirayama, K., and Yokoo, M. 1997. Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, 222–236.

Jensen, F. V. 1996. *An Introduction to Bayesian Networks*. London, UK: UCL Press.

Jordan, M. I. 2004. Graphical models. *Statistical Science (Special Issue on Bayesian Statistics)* 19:140–155.

Lawler, E. L., and Wood, D. E. 1966. Branch-and-bound methods: A survey. *Operations Research* 14(4):699–719.

Liu, J. S., and Sycara, K. 1994. Distributed meeting scheduling. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Society*.

Liu, J. S., and Sycara, K. 1995. Exploiting problem structure for distributed constraint optimization. In *Proceedings of the First International Conference on Multi-Agent Systems*.

Lynch, N. A. 1996. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Maheswaran, R. T.; Tambe, M.; Bowring, E.; Pearce, J. P.; and Varakantham, P. 2004. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*.

Mailler, R., and Lesser, V. 2004. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems*.

Mainwaring, A.; Culler, D.; Polastre, J.; Szewczyk, R.; and Anderson, J. 2002. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*.

Meisels, A.; Kaplansky, E.; Razgon, I.; and Zivan, R. 2002. Comparing performance of distributed constraints processing algorithms. In *Proceedings of Workshop on Distributed Constraint Reasoning, First International Joint Conference on Autonomous Agents and Multiagent Systems*.

Modi, P. J.; Jung, H.; Tambe, M.; Shen, W.-M.; and Kulkarni, S. 2001. A dynamic distributed constraint satisfaction approach to resource allocation. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*.

Modi, P. J.; Shen, W.-M.; Tambe, M.; and Yokoo, M. 2005. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence* 16(1–2):149–180.

Modi, P. J. 2003. *Distributed constraint optimization for multiagent systems*. Ph.D. Dissertation. Adviser - Wei-Min Shen and Adviser - Milind Tambe.

Pearce, J. P. 2005. USC dcop repository.

Petcu, A., and Faltings, B. 2005. An efficient constraint optimization method for large multiagent systems. In *Proceedings of the Large Scale Multi-Agent Systems Workshop, Forth International Joint Conference on Autonomous Agents and Multi-Agent Systems*.

Scerri, P.; Modi, P. J.; Shen, W.-M.; and Tambe, M. 2003. Are multiagent algorithms relevant for real hardware? a case study of distributed constraint algorithms. In *ACM Symposium on Applied Computing*.