Caching Schemes for DCOP Search Algorithms*

William Yeoh
Computer Science
USC
Los Angeles, CA 90089
wyeoh@usc.edu

Pradeep Varakantham
Robotics Institute
CMU
Pittsburgh, PA 15232
pradeepv@cs.cmu.edu

Sven Koenig
Computer Science
USC
Los Angeles, CA 90089
skoenig@usc.edu

ABSTRACT

Distributed Constraint Optimization (DCOP) is useful for solving agent-coordination problems. Any-space DCOP search algorithms require only a small amount of memory but can be sped up by caching information. However, their current caching schemes do not exploit the cached information when deciding which information to preempt from the cache when a new piece of information needs to be cached. Our contributions are three-fold: (1) We frame the problem as an optimization problem. (2) We introduce three new caching schemes (MaxPriority, MaxEffort and MaxUtility) that exploit the cached information in a DCOP-specific way. (3) We evaluate how the resulting speed up depends on the search strategy of the DCOP search algorithm. Our experimental results show that, on all tested DCOP problem classes, our MaxEffort and MaxUtility schemes speed up ADOPT (which uses best-first search) more than the other tested caching schemes, while our MaxPriority scheme speeds up BnB-ADOPT (which uses depth-first branch-and-bound search) at least as much as the other tested caching schemes.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed AI; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms; Experimentation

Keywords

ADOPT; BnB-ADOPT; Caching; DCOP; Distributed Constraint Optimization; Distributed Search Algorithms

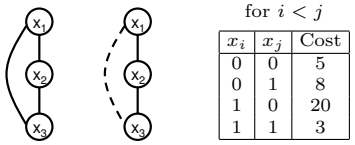
*This research has been partly supported by an NSF award to Sven Koenig under contract IIS-0350584. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

Cite as: Caching Schemes for DCOP Search Algorithms, William Yeoh, Pradeep Varakantham and Sven Koenig, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. XXX-XXX.

1. INTRODUCTION

Distributed Constraint Optimization (DCOP) problems are problems where agents need to coordinate with each other which values they should take on to minimize the sum of the resulting constraint costs. Many agent-coordination problems can be expressed as DCOP problems, including the scheduling of meetings [7], the allocation of targets to sensors in sensor networks [6] and the synchronization of traffic lights [5]. Researchers have developed a variety of complete algorithms to find cost-minimal solutions for DCOP problems, including ADOPT [11], OptAPO [8], DPOP [12], NCBB [2], AFB [3] and BnB-ADOPT [14]. Most of these algorithms make extreme assumptions about the amount of memory that each agent has available. DCOP search algorithms (like ADOPT) require only a polynomial amount of memory per agent. DCOP inference algorithms (like DPOP), on the other hand, require an exponential amount of memory per agent, which results in significantly smaller runtimes. Any-space versions of these algorithms bridge the two extremes by trading off memory requirements and runtime. Any-space DCOP search algorithms cache information units when additional memory is available. Any-space ADOPT [10] and any-space NCBB [1], for example, require only a polynomial amount of memory per agent and can vary their memory requirements by linear factors. Any-space DCOP inference algorithms use cycle cutsets to limit the amount of required memory. MB-DPOP [13], for example, unfortunately still requires an exponential amount of memory per agent and can vary its memory requirements only by exponential factors and thus only in a much more coarse-grained way.

We therefore build on DCOP search algorithms in this paper and investigate their caching schemes, in particular which information unit to preempt from the cache when a new information unit needs to be cached. We frame this problem as an optimization problem. We calculate the utility $U(I) = P(I)E(I)$ of an information unit I based on its *likelihood of future use* $P(I)$ and the *invested search effort* $E(I)$, which is the search effort that has been spent to acquire it and thus will likely have to be spent again if it is preempted from the cache. Each agent then greedily maximizes the sum of the utilities of all cached information units by preempting an information unit with the smallest utility. We show how the agents can calculate estimates $\hat{P}(I)$ and $\hat{E}(I)$ of $P(I)$ and $E(I)$, respectively, by exploiting the cached information units in a DCOP-specific way. We evaluate our ideas by classifying caching schemes into three categories: Caching schemes of Category 1 preempt



for $i < j$

x_i	x_j	Cost
0	0	5
0	1	8
1	0	20
1	1	3

(c)

Figure 1: Example DCOP Problem

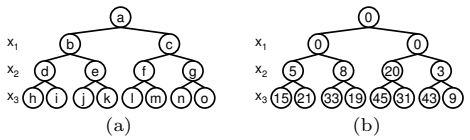


Figure 2: Search Tree

an information unit with the smallest $P(I)$. Examples are the Least-Recently-Used scheme of any-space ADOPT and the First-In-First-Out scheme of any-space NCBB. Caching schemes of Category 2 preempt an information unit with the smallest $E(I)$. Finally, caching schemes of Category 3 combine both principles by preempting an information unit with the smallest $U(I) = P(I)E(I)$. We introduce a new caching scheme for each category. The*MaxPriority* scheme of Category 1 uses $\hat{P}(I)$ to estimate $P(I)$, the *MaxEffort* scheme of Category 2 uses $\hat{E}(I)$ to estimate $\hat{P}(I)$, and the *MaxUtility* scheme of Category 3 uses $\hat{U}(I) = \hat{P}(I)\hat{E}(I)$ to estimate $U(I) = P(I)E(I)$. We compare these caching schemes experimentally against standard caching schemes from the operating systems literature, namely the First-In-First-Out, Last-In-First-Out, Least-Recently-Used and Least-Frequently Used schemes. We do this in the context of ADOPT and BnB-ADOPT, two DCOP search algorithms with identical memory requirements and communication frameworks but different search strategies. The MaxPriority scheme outperforms all other caching schemes from Category 1 and the MaxEffort scheme outperforms all other caching schemes from Category 2, demonstrating the benefits of our estimates $\hat{P}(I)$ and $\hat{E}(I)$ individually. Overall, our experimental results show that our MaxEffort and MaxUtility schemes speed up ADOPT (which uses best-first search) more than the other tested caching schemes, while our MaxPriority scheme speeds up BnB-ADOPT (which uses depth-first branch-and-bound search) at least as much as the other tested caching schemes.

2. DCOP PROBLEMS

A DCOP problem is defined by a finite set of agents $V = \{x_1, x_2, ..., x_n\}$; a set of finite domains $D = \{D_1, D_2, ..., D_n\}$, where domain D_i is the set of possible values for agent $x_i \in V$; and a set of binary constraints $F = \{f_1, f_2, ..., f_m\}$, where each constraint $f_i : D_{i_1} \times D_{i_2} \rightarrow \mathbb{R}^+ \cup \infty$ specifies its non-negative constraint cost as a function of the values of the two different agents $x_{i_1}, x_{i_2} \in V$ that share the constraint. Each agent is responsible for assigning itself (= taking on) values from its domain. The agents coordinate these value assignments by exchanging messages. A solution is an agent-value assignment for a subset of agents. Its cost is the sum of the constraint costs of all constraints shared by agents with known values. A solution is called complete if and only if it is an agent-value assignment for all agents. The objective is to determine a cost-minimal complete solution.

DCOP problems are commonly visualized as *constraint graphs* whose vertices are the agents and whose edges are the constraints. Most DCOP search algorithms operate on *pseudo-trees*. A pseudo-tree is a spanning tree of the constraint graph with the property that no two vertices in different subtrees of the spanning tree are connected by an edge in the constraint graph. Figure 1(a) shows the constraint graph of an example DCOP problem with three agents that can each take on the values 0 or 1, Figure 1(b) shows one possible pseudo-tree (the dotted line is an edge of the constraint graph that is not part of the pseudo-tree), and Figure 1(c) shows the constraint costs.

3. ADOPT AND BnB-ADOPT

We now provide a brief overview of ADOPT and BnB-ADOPT, both of which transform the constraint graph to a pseudo-tree in a pre-processing step and then search for a cost-minimal complete solution. They have identical memory requirements and communication frameworks. However, ADOPT uses best-first search while BnB-ADOPT uses depth-first branch-and-bound search. Complete descriptions can be found in [11, 14].

3.1 Search Trees

The operation of ADOPT and BnB-ADOPT can be visualized with AND/OR search trees [9] due to their ability to capture independent subproblems of a DCOP problem. The constraint graph of our example DCOP problem is fully connected and thus has no independent subproblems. We therefore use regular search trees and terminology from A* [4] instead. Each level of a search tree corresponds to an agent. For our example DCOP problem, the level of depth 1 corresponds to agent x_1. A left branch that enters a level means that the corresponding agent takes on the value 0. A right branch means that the corresponding agent takes on the value 1. A node in the search tree thus corresponds to a solution. For our example DCOP problem, the solution of node e is $(x_1 = 0, x_2 = 1)$, where we use the identifiers shown in Figure 2(a) to refer to nodes. The numbers inside the nodes in Figure 2(b) are the costs of the solutions of the nodes and are thus equal to the f-values of the nodes for an A* search with zero heuristics.

3.2 Search Strategy of ADOPT

The execution trace shown in Figure 3 visualizes the search strategy of ADOPT on the example DCOP problem. We assume for our explanation that the agents operate sequentially and communication is instantaneous because these simplifying assumptions allow us to explain the search strategy of ADOPT easily. The grey nodes of the search trees are the currently expanded node and its ancestors. The root node is always grey. Each agent takes on the value in the solution that corresponds to the grey node in its level and maintains upper and lower bounds for all children of the grey node in the level above it. For our example DCOP problem, agent x_2 takes on value 1 in Step 4 and maintains upper and lower bounds for nodes f and g. The numbers in the nodes are the lower bounds. Crosses show lower bounds that are not maintained due to memory limitations. Each agent initializes the lower bounds of the nodes that it maintains with the f-values and then repeat-

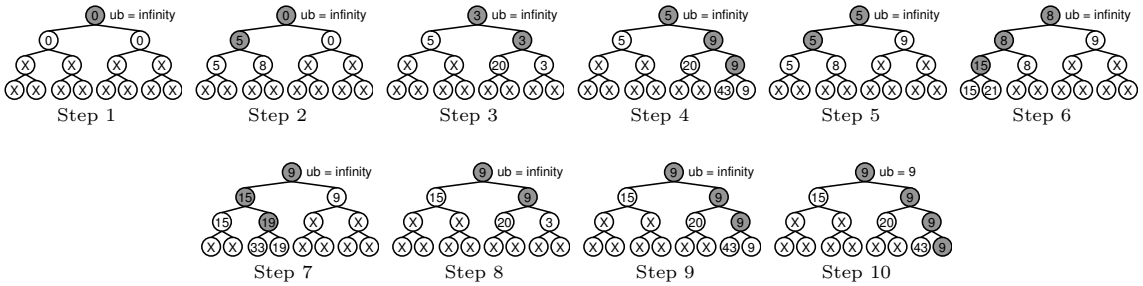


Figure 3: Execution Trace of ADOPT with the Default Amount of Memory

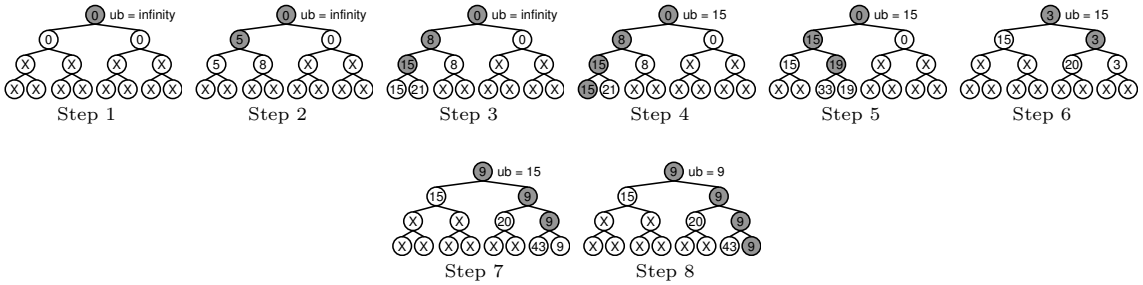


Figure 4: Execution Trace of BnB-ADOPT with the Default Amount of Memory

edly sets them to the minimum of the lower bounds of the children of the nodes. The lower bound of the root node is updated in the same way and is thus a lower bound on the cost of a cost-minimal complete solution. We show only the upper bound *ub* of the root node, which is repeatedly set to the smallest cost of all complete solutions found so far. The upper bound of the root node is thus an upper bound on the cost of a cost-minimal complete solution.

ADOPT expands nodes in a depth-first search order. After it has expanded a node, it always expands the child of the node with the smallest lower bound and backtracks when the lower bounds of all unexpanded children of the node are larger than the lower bound of the root node. The resulting order of expansions is identical to a *best-first search* order if one considers only nodes that ADOPT expands for the first time. ADOPT terminates when the upper and lower bounds of the root node are equal, indicating that it has found a cost-minimal complete solution. ADOPT is forced to re-expand nodes due to its best-first search strategy since agents have to purge some of their upper and lower bounds due to their memory limitations. For our example DCOP problem, ADOPT first expands node g in Step 4 and then re-expands it in Step 9. Overall, it expands nodes in the order a, b, c, g, b^*, d, e, c^*, g^* and o. It re-expands the nodes with asterisks.

3.3 Search Strategy of BnB-ADOPT

The execution trace shown in Figure 4 visualizes the search strategy of BnB-ADOPT on the example DCOP problem under the same assumptions as for ADOPT. BnB-ADOPT operates in the same way as ADOPT except that it expands nodes in a *depth-first branch-and-bound search* order. After it has expanded a node, it always expands the child of the node with the smallest lower bound and backtracks when the lower bounds of all unexpanded children of the node are larger than the upper bound of the root node. Overall, it expands nodes in the order a, b, d, h, e, c, g and o. It expands some nodes that ADOPT does not expand, such as node h, but never re-expands nodes.

4. CACHING

Agents can use available memory to cache upper and lower bounds and then reuse them when they need them again, which avoids search effort. For example, the execution trace shown in Figure 5 visualizes the search strategy of ADOPT, as before, except that all agents now have a sufficient amount of memory to maintain all upper and lower bounds. ADOPT terminates three steps earlier since it no longer needs to re-expand nodes.

4.1 Cache Design

Each agent of ADOPT and BnB-ADOPT maintains upper and lower bounds for all children of the grey node in the level of the search tree above it. The solution of the grey node in the level above it is called the *current context* of the agent. (We say that the agent visits a context if this context is the current context of the agent.) Thus, the current context of an agent is an agent-value assignment for all of its ancestors in the pseudo-tree. Each agent thus maintains upper and lower bounds for each combination of its current context and all values from its domain. For our example DCOP problem, agent x_3 in Step 4 maintains a lower bound of 43 for its current context $(x_1 = 1, x_2 = 1)$ and its value 0 and a lower bound of 9 for its current context and its value 1. The agents exchange VALUE and COST messages to determine their current contexts and update the upper and

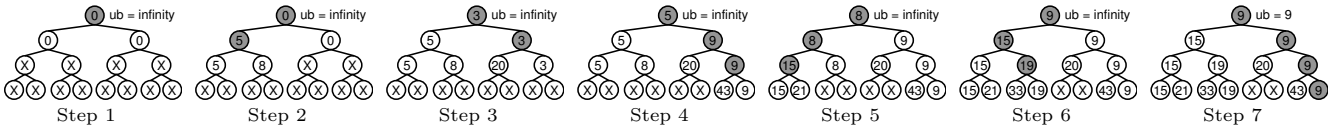


Figure 5: Execution Trace of ADOPT with the Maximum Amount of Memory

lower bounds that they maintain.

We bundle the current context of an agent and all upper and lower bounds that it maintains into an *information unit*. Formally, an information unit I of agent $x_i \in V$ is given by the tuple $\langle LB^I, UB^I, X^I \rangle$, where X^I is a context of the agent and LB^I and UB^I are tuples of lower bounds $LB^I(d)$ and upper bounds $UB^I(d)$, one for each value $d \in D_i$ from its domain. Each agent of ADOPT and BnB-ADOPT caches only one information unit, namely for its current context. We now generalize ADOPT and BnB-ADOPT by allowing agents to cache more than one information unit. An agent always uses the information unit with its current context. We assume that an agent can cache a given number of information units. If the agent receives upper and lower bounds via COST messages for a context that is equal to the context of a cached information unit, then it updates the bounds of that information unit. If the agent receives upper and lower bounds for a context that is not equal to the context of any cached information unit and the cache is not yet full, then it creates a new information unit and caches the context and bounds in that information unit. If the agent receives upper and lower bounds for a context that is not equal to the context of any cached information unit and the cache is full, then it ignores the bounds, with one exception: If the agent receives upper and lower bounds for a context and then switches its current context to that context, then it needs to cache them since an agent always has to cache the information unit with its current context. A caching scheme then decides which information unit to preempt from the cache.

4.2 Caching Problem

We frame the problem of which information unit to preempt from the cache as an optimization problem. We calculate the utility

$$U(I) := P(I)E(I)$$

of an information unit I based on its likelihood of future use $P(I)$ and the invested search effort $E(I)$. Each agent then greedily maximizes the sum of the utilities of all cached information units by preempting an information unit with the smallest utility.

4.2.1 Likelihood of Future Use: P(I)

The likelihood of future use $P(I)$ measures the probability that the context of information unit I will again become the current context. The likelihood of future use $P(I)$ corresponds to the probability that the node in the search tree whose solution corresponds to the context of information unit I will be re-expanded. It is important to use the likelihood of future use to measure the utility of an information unit because it is pointless to cache an information unit whose context will never again become the current context. It is affected by two factors:

- **Asynchronous Execution:** The agents of ADOPT and BnB-ADOPT can visit intermediate contexts for a short period of time when their current context changes because they operate asynchronously and communication can be delayed. Assume, for example, that the current context of agent x_3 is $(x_1 = 0, x_2 = 0)$ and its next context would be $(x_1 = 1, x_2 = 1)$ if the agents operated sequentially and communication were instantaneous. The VALUE message from agent x_1 about its new value 1 can arrive at agent x_3 before the VALUE message from agent x_2 about its new value 1. Then, agent x_3 visits context $(x_1 = 0, x_2 = 1)$ for a short period of time. The more agent-value assignments a context has in common with the current context of the agent, the larger the likelihood of future use is.
- **Search Strategy:** If the agents of BnB-ADOPT operated sequentially, then they would not re-visit contexts due to its depth-first branch-and-bound search strategy. On the other hand, the agents of ADOPT would still re-visit contexts due to its best-first search strategy.

4.2.2 Invested Search Effort: E(I)

Existing caching schemes for DCOP search algorithms use only the likelihood of future use to measure the utility of an information unit. It is also important to use the invested search effort (= search effort that has been spent to acquire the information unit and thus will likely have to be spent again if it is preempted from the cache but its context gets revisited) to measure the utility of an information unit because it might be better to preempt an information unit from the cache whose likelihood of reuse is 100 percent but whose invested search effort is almost zero than an information unit whose likelihood of future use is only 50 percent but whose invested search effort is large. The invested search effort $E(I)$ corresponds to the number of nodes expanded and re-expanded in the subtree of the search tree rooted at the node whose solution corresponds to the context of information unit I.

4.3 Caching Schemes

We classify caching schemes into three categories:

- **Category 1**: Caching schemes that preempt an information unit I with the smallest likelihood of future use $P(I)$. We introduce a new MaxPriority scheme for this category.
- **Category 2**: Caching schemes that preempt an information unit I with the smallest invested search effort

$E(I)$. We introduce a new MaxEffort scheme for this category.

- **Category 3**: Caching schemes that preempt an information unit I with the smallest utility $U(I) = P(I)E(I)$. We introduce a new MaxUtility scheme for this category.

4.3.1 Benchmark Schemes

We use page replacement schemes for virtual memory management from the operating systems literature as benchmark schemes.

In particular, we use First-In-First-Out (FIFO) and Least-Recently-Used (LRU) as benchmark schemes of Category 1, which are similar to existing caching schemes for DCOP search algorithms. For example, any-space NCBB uses a version of the FIFO scheme and any-space ADOPT uses a version of the LRU scheme. The FIFO scheme preempts the information unit that has been in the cache for the longest time, and the LRU scheme preempts the information unit that has not been used or updated for the longest time. Both caching schemes use the intuition that an information unit that has been cached, used or updated recently will likely be used again.

Similarly, we use Last-In-First-Out (LIFO) and Least-Frequently-Used (LFU) as benchmark schemes of Category 2. The LIFO scheme preempts the information unit that has been in the cache for the shortest time, and the LFU scheme preempts the information unit that has been used or updated the least number of times. Both caching schemes use the intuition that a large search effort has been invested in an information unit that has been in the cache for a long time (which assumes similar update frequencies for all information units) or that has been used or updated frequently (which assumes a similar ratio of use and update frequencies for all information units).

4.3.2 MaxPriority Scheme

The MaxPriority scheme attempts to preempt the information unit I with the smallest likelihood of future use $P(I)$. The likelihood of future use of an information unit is affected by both the asynchronous execution and the search strategy of a DCOP search algorithm. It is currently unknown how to best estimate the likelihood of future use due to the search strategy. The MaxPriority scheme thus estimates only the likelihood of future use due to the asynchronous execution. The more agent-value assignments the context of the information unit has in common with the current context of the agent, the larger the likelihood of future use due to the asynchronous execution is.

The MaxPriority scheme, however, uses additional knowledge of the operation of ADOPT and BnB-ADOPT in the form of the lower bounds, namely that each agent takes on the value with the smallest lower bound in the information unit with its current context. We now discuss how the MaxPriority scheme estimates the likelihood of future use of an information unit I of agent x: Let $x^1 \dots x^k$ be the ancestors of agent x in the pseudo-tree, ordered in increasing order of their depth in the pseudo-tree. Consider any ancestor x^l and assume that estimates of the lower bounds in the information unit of ancestor x^l with the current context of the ancestor are available. Let $I(x^l)$ be the index of the lower bound of the value of ancestor x^l in the context of information unit I in decreasing order of all lower bounds, with one exception: $I(x^l)$ is infinity if the value of ancestor x^l in the context of information unit I is equal to the value of ancestor x^l in the current context of agent x. For example, assume that ancestor x^l can take on four values, namely 0, 1, 2 and 3. Assume that the following estimates of the lower bounds in information unit I' of the ancestor with the current context of the ancestor are available: $LB^{I'}(0) = 8$, $LB^{I'}(1) = 12$, $LB^{I'}(2) = 10$ and $LB^{I'}(3) = 6$. If the value of the ancestor in the current context of agent x is 0, then it is most likely that the ancestor still takes on this value. Since each agent takes on the value with the smallest lower bound in the information unit with its current context, the value that the ancestor currently takes on is in increasing order of likelihood: 1, 2, 3 and 0. The lower bounds are in decreasing order: $LB^{I'}(1) = 12$ (index 0), $LB^{I'}(2) = 10$ (index 1), $LB^{I'}(0) = 8$ (index 2) and $LB^{I'}(3) = 6$ (index 3). Thus, the index of the lower bound of value 3 is 3, the index of the lower bound of value 2 is 1, and the index of the lower bound of value 1 is 0. The index of the lower bound of value 0 is changed to infinity because it is the the value of the ancestor in the current context of agent x. Thus, the larger the index is, the more likely it is that the ancestor currently takes on this value. Now consider the tuple $(I(x^1), \dots, I(x^k))$ for each information unit I cached by agent x. The MaxPriority scheme preempts the information unit whose tuple is lexicographically smallest among these information units. More generally, it uses the index of the tuple of an information unit I in the increasing lexicographic order of the tuples of all information units cached by the agent as estimate $\hat{P}(I)$ of $P(I)$. $\hat{P}(I)$ is not meant to approximate $P(I)$ but be proportional to it.

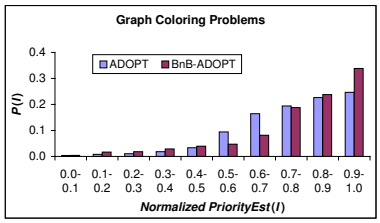


Figure 6: Correlation of $\hat{P}(I)$ and $P(I)$

To evaluate how well $\hat{P}(I)$ and $P(I)$ are correlated for ADOPT and BnB-ADOPT, we use the same experimental formulation and setup as [7, 14] and conduct experiments on graph coloring problems with 10 vertices, density 2 and domain cardinality 5. The caches of all agents are sufficiently large to store all information units. For each information unit I of each agent x directly before the agent changes its current context, we divide $\hat{P}(I)$ by the number of cached information units at that point in time minus one to normalize it into the interval from 0 to 1, shown as *Normalized PriorityEst*(I) in Figure 6, and then classify it into one of 10 buckets that cover the interval evenly. We then calculate the frequency for each bucket of the event that the contexts of its information units became the current contexts of their agents after the context switch, shown as $P(I)$ in Figure 6. The Pearson's coefficient shows indeed a strong correlation of $\hat{P}(I)$ and $P(I)$ with $\rho > 0.85$.

4.3.3 MaxEffort Scheme

The MaxEffort scheme attempts to preempt the informa-

tion unit I with the smallest invested search effort $E(I)$. It estimates the invested search effort of an information unit by using knowledge of the operation of ADOPT and BnB-ADOPT in the form of the upper and lower bounds, namely that the lower bounds will increase over time and the upper bounds will decrease over time. Thus, the difference between the upper and lower bounds of an information unit decreases as more search effort is invested in it. We now discuss how the MaxEffort scheme estimates the invested search effort of an information unit I of agent x_i: The MaxEffort scheme calculates the average difference $AD(I)$ between the upper bounds $UB^I(d)$ and lower bounds $LB^I(d)$ of the information unit over all values $d \in D_i$ from the domain of agent x_i: $AD(I) := \frac{\sum_{d \in D_i}(UB^I(d) - LB^I(d))}{|D_i|}$. The MaxEffort scheme preempts the information unit whose average difference is largest among all cached information units. More generally, it uses $\hat{E}(I) := AD(I') - AD(I)$ as estimate $\hat{E}(I)$ of $E(I)$, where I' is the information unit of agent x_i with the largest average difference. $\hat{E}(I)$ is not meant to approximate $E(I)$ but be proportional to it.

To evaluate how well $\hat{E}(I)$ and $E(I)$ are correlated for ADOPT and BnB-ADOPT, we use the same experimental formulation and setup as described for the MaxPriority scheme. For the information unit I with the current context of each agent x directly after the agent changed its current context, we divide $\hat{E}(I)$ by the largest such estimate over all information units cached by agent x at that point in time to normalize it into the interval from 0 to 1, shown as *Normalized EffortEst*(I) in Figure 7, and then classify it into one of 10 buckets that cover the interval evenly. We then calculate the average for each bucket of the number of cycles [11] that the contexts of its information units had already been the current contexts of their agents before the context switches and divide each average by the largest average over all buckets to normalize it into the interval from 0 to 1, shown as $E(I)$ in Figure 7. The Pearson's coefficient shows indeed a strong correlation of $\hat{E}(I)$ and $E(I)$ with $\rho > 0.85$.

4.3.4 MaxUtility Scheme

The MaxUtility scheme attempts to preempt the information unit I with the smallest value of $U(I) = P(I)E(I)$. It uses $\hat{U}(I) := \hat{P}(I)\hat{E}(I)$ as estimate $\hat{U}(I)$ of $U(I)$. It calculates $\hat{P}(I)$ like the MaxPriority scheme and $\hat{E}(I)$ like the MaxEffort scheme.

4.4 Analysis of the Caching Schemes

We measure the memory complexity in the number of floating point numbers. The memory size of a context is $O(|V|)$. The memory size of an information unit is $O(maxDom + |V|)$ since it stores a context and one upper and lower bound for each possible value, where $maxDom := \max_{D_i \in D} |D_i|$ is the largest domain cardinality over all agents. An agent can try to cache $O(maxDom^{|V|})$ information units at the same time, namely one for each of the $O(maxDom^{|V|})$ possible contexts. Thus, the maximum cache size and thus memory complexity of caching per agent is $O((maxDom+|V|)maxDom^{|V|}) = O(maxDom^{|V|})$ since an agent maintains only one cache.

We measure the message complexity in the number of floating point numbers as well. The message complexity of ADOPT and BnB-ADOPT is $O(|V|)$. None of the caching schemes increase this message complexity, except for the MaxPriority and MaxUtility schemes. An agent that uses these caching schemes needs to know, for all of its ancestors, the indices of the lower bounds for all of their values in their information units with their current contexts. VALUE messages therefore need to include these indices for the sending agent, and COST messages need to include these indices for all ancestors of the sending agent. There are $O(|V|)$ ancestors, each one of which has $O(maxDom)$ values and thus indices. The message complexity therefore increases to $O(|V| + maxDom|V|) = O(maxDom|V|)$.

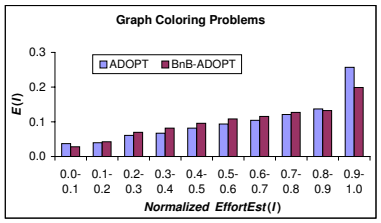


Figure 7: Correlation of $\hat{E}(I)$ and $E(I)$

5. EXPERIMENTAL RESULTS

We now compare the caching schemes experimentally. We use the same experimental formulation and setup as [7, 14] and conduct experiments with three DCOP problem classes, namely *graph coloring problems* with 10 vertices, density 2 and domain cardinality 5; *sensor network problems* with 12 targets and domain cardinality 5; and *meeting scheduling problems* with 10 meetings and domain cardinality 5. We average the experimental results over 50 DCOP problem instances. We vary the DCOP search algorithm, caching scheme and cache size of each agent. All agents always have the same cache factor [1]. The cache factor of an agent is the ratio of (the number of information units that fit into its cache - 1) and (the number of its possible contexts - 1). Agents that can cache only one information unit thus have cache factor zero, and agents that can cache all information units have cache factor one or larger. We measure the resulting runtime of ADOPT and BnB-ADOPT in time slices, called cycles, where smaller numbers of cycles indicate smaller runtimes [11]. In each cycle, all agents receive and process all of their incoming messages and send all outgoing messages. A new cycle starts immediately after the last agent sends its outgoing messages in the current cycle.

Table 1 shows the runtime on graph coloring problems, sensor network problems and meeting scheduling problems. The runtime decreases for each combination of DCOP search algorithm and caching scheme as the cache factor increases, as expected. The smallest runtime for each cache factor is shown in italics. The runtime of all caching schemes is identical for each DCOP search algorithm if the cache factor is zero (because all information units need to be preempted) or one (because no information units need to be preempted). The speed up from caching is much larger for ADOPT than BnB-ADOPT. The caching schemes of Category 2 (LIFO, LFU, MaxEffort) result in a smaller runtime than the ones of Category 1 (FIFO, LRU, MaxPriority) for ADOPT and vice versa for BnB-ADOPT. However, the relationships within each category are similar for ADOPT and BnB-ADOPT. For Category 1, the resulting runtime is non-decreasing in the order: MaxPriority scheme, LRU scheme

	ADOPT											BnB-ADOPT										
Cache Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
FIFO	*165107*	143354	120950	92474	67138	49658	32557	19820	9868	6686	*4836*	*2538*	2381	2366	2364	2363	2359	2357	2357	2355	2348	*2347*
LRU	*165107*	142868	120459	90613	63210	46094	26880	17942	8679	6405	*4836*	*2538*	2376	*2365*	2364	2362	2360	2357	2355	2351	2348	*2347*
MaxPriority	*165107*	126769	90063	64192	36916	25728	14445	10649	6882	5890	*4836*	*2538*	*2373*	*2365*	*2362*	*2360*	*2357*	2355	2351	2349	2348	*2347*
LIFO	*165107*	96220	57872	39833	24579	18579	11945	9674	6911	6035	*4836*	*2538*	2462	2432	2409	2392	2383	2365	2355	2350	*2347*	*2347*
LFU	*165107*	91355	56192	38592	23648	17958	11536	9371	6767	5973	*4836*	*2538*	2474	2445	2417	2399	2386	2366	2355	2350	*2347*	*2347*
MaxE↑ort	*165107*	70660	41190	*28321*	17757	*13842*	*9666*	*8240*	*6310*	*5723*	*4836*	*2538*	2426	2389	2375	2365	2358	*2353*	*2350*	*2348*	*2347*	*2347*
MaxUtility	*165107*	*70599*	*40911*	28407	*17482*	14015	9691	8342	6324	5788	*4836*	*2538*	2426	2389	2374	2365	2358	*2353*	*2350*	*2348*	*2347*	*2347*

Number of Cycles for Graph Coloring Problems

	ADOPT											BnB-ADOPT										
Cache Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
FIFO	*47087*	39669	21310	13586	8932	6075	4209	3523	2946	2793	*2451*	*1249*	1202	1178	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*
LRU	*47087*	39746	21414	12107	6556	4937	3828	3437	2902	2725	*2451*	*1249*	1204	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*
MaxPriority	*47087*	28289	16000	10524	6298	4850	3632	3231	2786	*2626*	*2451*	*1249*	*1198*	1178	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*	*1177*
LIFO	*47087*	28289	16000	10524	6298	4850	3632	3231	2786	*2626*	*2451*	*1249*	1228	1212	1202	1195	1189	1183	1179	*1177*	*1177*	*1177*
LFU	*47087*	27589	15767	10435	6270	4865	3605	3252	*2760*	2627	*2451*	*1249*	1229	1212	1202	1195	1189	1183	1179	*1177*	*1177*	*1177*
MaxE↑ort	*47087*	*17522*	*9364*	*6606*	*4567*	*4058*	*3375*	*3206*	2785	*2626*	*2451*	*1249*	1221	1204	1195	1186	1181	1179	1178	*1177*	*1177*	*1177*
MaxUtility	*47087*	18054	9731	6994	4761	4237	3524	3305	2874	2755	*2451*	*1249*	1220	1204	1195	1186	1181	1179	1178	*1177*	*1177*	*1177*

Number of Cycles for Sensor Network Problems

	ADOPT											BnB-ADOPT										
Cache Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
FIFO	*31080*	27707	21596	14824	8858	6229	4332	3720	3056	2863	*2486*	*1240*	1172	*1128*	1129	1125	1123	1120	1118	1117	*1116*	*1116*
LRU	*31080*	27903	20237	13555	7224	5423	4012	3556	2939	2727	*2486*	*1240*	1172	*1128*	1127	1124	1122	1120	1117	1117	*1116*	*1116*
MaxPriority	*31080*	21231	14826	10609	6286	5125	3929	3407	2870	2656	*2486*	*1240*	*1164*	*1128*	*1123*	*1120*	*1119*	1118	1117	1117	*1116*	*1116*
LIFO	*31080*	21231	14826	10609	6286	5125	3929	3407	2870	2656	*2486*	*1240*	1213	1197	1170	1151	1140	1128	1120	1117	*1116*	*1116*
LFU	*31080*	21234	14789	10620	6301	5075	3920	3410	2862	*2655*	*2486*	*1240*	1213	1197	1170	1151	1140	1128	1120	1117	*1116*	*1116*
MaxE↑ort	*31080*	*16727*	*9526*	*6860*	*4769*	*4039*	*3325*	*3141*	*2790*	2674	*2486*	*1240*	1196	1166	1144	1131	1123	*1117*	*1116*	*1116*	*1116*	*1116*
MaxUtility	*31080*	16807	9783	7062	4949	4182	3471	3265	2880	2754	*2486*	*1240*	1197	1166	1144	1131	1123	*1117*	*1116*	*1116*	*1116*	*1116*

Number of Cycles for Meeting Scheduling Problems

Table 1: Runtimes

and FIFO scheme. Thus, the MaxPriority scheme is a good caching scheme for Category 1. For Category 2, it is non-decreasing in the order: MaxEffort scheme, LIFO scheme and LFU scheme. Thus, the MaxEffort scheme is a good caching scheme for Category 2. The resulting runtime is about the same for the MaxUtility and MaxEffort schemes. Overall, the MaxEffort and MaxUtility schemes speed up ADOPT more than the other tested caching schemes, while our MaxPriority scheme speeds up BnB-ADOPT at least as much as the other tested caching schemes.

We now explain some of these observations. Table 2 shows the number of unique and repeated contexts per agent on graph coloring problems. (The trends are similar for the other two DCOP problem classes.) Assume that an agent records its context whenever its current context changes. The number of unique contexts is the number of different contexts recorded. The number of repeated contexts is the total number of contexts recorded minus the number of unique contexts. The sum of both numbers is correlated with the runtime of a DCOP search algorithm. The number of unique contexts depends mainly on the search strategy of the DCOP search algorithm. The number of repeated contexts decreases for each combination of DCOP search algorithm and caching scheme as the cache factor increases. However, the decrease is by up to one order of magnitude larger for ADOPT than BnB-ADOPT, which is also reflected in the resulting runtimes. If the agents of BnB-ADOPT operated sequentially, then they would not re-visit contexts. Thus, caching would not speed up BnB-ADOPT. Since the agents operate asynchronously, caching can speed up BnB-ADOPT slightly because agents can visit intermediate contexts for a short period of time when their current context changes. The search effort invested in the information units with the intermediate contexts is thus small. The search effort invested in the other information units is often large but they do not need to be cached because their contexts do not need to be re-visited. On the other hand, if the agents of ADOPT operated sequentially, then they would still re-visit contexts due to its best-first search strategy. Caching thus speeds up ADOPT a lot. The search effort invested in the information units with these contexts varies. It is therefore important to select a good caching scheme carefully, as the resulting runtimes show. Caching schemes of Category 1 are not well suited for ADOPT, as the resulting runtimes show, since their estimates of the likelihood of future use take into account only that the agents re-visit contexts due to the asynchronous execution of ADOPT but not that they also re-visit contexts due to its best-first search strategy. Caching schemes of Category 2 are not well suited for BnB-ADOPT, as the resulting runtimes show, since the contexts of the information units with the largest invested search effort do not need need to be re-visited. Ideally, the MaxUtility scheme should result in smaller runtimes than the other caching schemes since an ideal caching scheme minimizes the search effort of a DCOP search algorithm by accurately estimating both the likelihood of future use $P(I)$ of an information unit I and the invested search effort $E(I)$. However, while our estimate $\hat{P}(I)$ is correlated with $P(I)$ and our estimate $\hat{E}(I)$ is correlated with $E(I)$, these correlations are not linear, as shown in Figures 6 and 7. Thus, the information unit with the smallest value of $\hat{P}(I)$ (which gets preempted by the MaxPriority scheme) or $\hat{E}(I)$ (which gets preempted by the MaxEffort scheme) is often also the information unit with the smallest value of $P(I)$ or $E(I)$, respectively. However, the information unit with the smallest value of $\hat{U}(I) = \hat{P}(I)\hat{E}(I)$ (which gets preempted by the MaxUtility scheme) is often not the information unit with the smallest value of $U(I) = P(I)E(I)$. It is therefore important to investigate these correlations further and determine how one can use them to improve on the MaxUtility scheme.

6. CONCLUSIONS

Any-space DCOP search algorithms require only a small amount of memory but can be sped up by caching infor-

	ADOPT											BnB-ADOPT										
Cache Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
FIFO	53566	48369	41745	33433	24676	19098	12823	8438	4476	3253	2377	419	412	410	409	409	409	409	409	409	409	409
LRU	53566	48228	41529	32611	23062	17520	10647	7616	4044	3169	2377	419	412	409	409	409	409	409	409	409	409	409
MaxPriority	53566	43532	32253	24083	14499	10632	6301	4924	3343	2973	2377	419	412	409	410	409	409	409	409	409	409	409
LIFO	53566	34833	22598	16478	10759	8536	5696	4784	3461	3081	2377	419	416	413	412	411	410	410	409	409	409	409
LFU	53566	33609	22199	16150	10402	8270	5522	4661	3382	3035	2377	419	414	413	412	411	410	410	409	409	409	409
MaxE↑ort	53566	27953	17669	12843	8402	6807	4835	4214	3197	2908	2377	419	413	411	411	411	410	410	410	409	409	409
MaxUtility	53566	28048	17704	12977	8364	6949	4910	4301	3243	2989	2377	419	413	411	411	411	410	410	410	409	409	409

Number of Repeated Contexts for Graph Coloring Problems

	ADOPT											BnB-ADOPT										
Cache Factor	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
FIFO	271	279	282	284	284	285	284	284	277	269	254	198	202	202	202	202	202	202	202	202	202	202
LRU	271	279	282	284	284	286	283	282	274	268	254	198	202	202	202	202	202	202	202	202	202	202
MaxPriority	271	281	284	286	284	283	278	275	267	263	254	198	202	202	202	202	202	202	202	202	202	202
LIFO	271	273	274	275	272	272	268	267	263	262	254	198	200	200	201	201	201	202	202	202	202	202
LFU	271	273	274	275	273	273	268	267	263	262	254	198	199	199	200	200	201	201	202	202	202	202
MaxE↑ort	271	276	279	281	280	279	274	273	266	263	254	198	200	200	201	202	202	202	202	202	202	202
MaxUtility	271	276	280	281	281	279	275	273	267	264	254	198	200	201	201	202	202	202	202	202	202	202

Number of Unique Contexts for Graph Coloring Problems

Table 2: Repeated and Unique Contexts

mation. However, their current caching schemes did not exploit the cached information units when deciding which information unit to preempt from the cache when a new information unit needs to be cached. We framed the problem of which information unit to preempt from the cache as an optimization problem, where each agent greedily maximizes the sum of the utilities of all cached information units by preempting an information unit with the smallest utility. Existing caching schemes used only the likelihood of future use $P(I)$ to measure the utility of an information unit I. We suggested to also use the invested search effort $E(I)$. We showed how caching schemes can calculate estimates $\hat{P}(I)$ and $\hat{E}(I)$ of $P(I)$ and $E(I)$, respectively, by exploiting the cached information units in a DCOP-specific way. We introduced three new caching schemes. The MaxPriority scheme uses the lower bounds to calculate $\hat{P}(I)$. The MaxEffort scheme uses the upper and lower bounds to calculate $\hat{E}(I)$. Finally, the MaxUtility scheme combines the MaxPriority and MaxEffort schemes to calculate $\hat{U}(I) = \hat{P}(I)\hat{E}(I)$.

Our experimental results show that, on all tested DCOP problem classes, our MaxEffort and MaxUtility schemes speed up ADOPT more than the other tested caching schemes, while our MaxPriority scheme speeds up BnB-ADOPT at least as much as the other tested caching schemes. The speed up from caching is much larger for ADOPT than BnB-ADOPT since ADOPT re-expands nodes of the search tree due to its best-first search strategy if agents had to preempt information units due to their memory limitations. It is future work to improve our estimates of the utilities of information units and apply our caching schemes to additional DCOP search algorithms, which should be possible since they also maintain upper and lower bounds on the cost of a cost-minimal complete solution.

7. REFERENCES

[1] A. Chechetka and K. Sycara. An any-space algorithm for distributed constraint optimization. In *Proceedings of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, pages 33–40, 2006.

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

[3] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward-bounding for distributed constraints optimization. In *Proceedings of ECAI*, pages 103–107, 2006.

[4] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC4(2):100–107, 1968.

[5] R. Junges and A. Bazzan. Evaluating the performance of DCOP algorithms in a real world, dynamic problem. In *Proceedings of AAMAS*, pages 599–606, 2008.

[6] V. Lesser, C. Ortiz, and M. Tambe, editors. *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer, 2003.

[7] R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *Proceedings of AAMAS*, pages 310–317, 2004.

[8] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of AAMAS*, pages 438–445, 2004.

[9] R. Marinescu and R. Dechter. AND/OR branch-and-bound for graphical models. In *Proceedings of IJCAI*, pages 224–229, 2005.

[10] T. Matsui, H. Matsuo, and A. Iwata. Efficient methods for asynchronous distributed constraint optimization algorithm. In *Proceedings of Artificial Intelligence and Applications*, pages 727–732, 2005.

[11] 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.

[12] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of IJCAI*, pages 1413–1420, 2005.

[13] A. Petcu and B. Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of IJCAI*, pages 1452–1457, 2007.

[14] W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. In *Proceedings of AAMAS*, pages 591–598, 2008.