A Memory Bounded Hybrid Approach to Distributed Constraint Optimization

James Atlas, Matt Warner, and Keith Decker

Computer and Information Sciences
University of Delaware
Newark, DE 19716
atlas, warner, decker@cis.udel.edu

Abstract. Distributed Constraint Optimization (DCOP) is a general framework that can model complex problems in multi-agent systems. Current algorithms that solve general DCOP instances rely on two main techniques, search and dynamic programming. While search algorithms such as ADOPT maintain polynomial memory requirements for participating agents, they incur a large performance penalty to establish initial bounds and to backtrack to previous solutions. Dynamic programming algorithms such as DPOP can be more efficient than search algorithms, but are not usually memory bounded and do not take advantage of search space pruning.
We introduce a new hybrid algorithm using a memory bounded version of DPOP to establish bounds in ADOPT. For problems with memory requirements below the bound we observe similar performance to DPOP. For problems with memory requirements above the given bound we observe significant improvements in performance compared to previous ADOPT techniques and the most recent memory bounded version of DPOP (MB-DPOP).

1 Introduction

Many historical problems in the AI community can be transformed into Constraint Satisfaction Problems (CSP). With the advent of distributed AI, multi-agent systems became a popular way to model the complex interactions and coordination required to solve distributed problems. CSPs were originally extended to distributed agent environments in [17]. Early domains for distributed constraint satisfaction problems (DisCSP) included job shop scheduling [6] and resource allocation [10]. Many domains for agent systems, especially teamwork coordination, distributed scheduling, and sensor networks, involve overly constrained problems that are difficult or impossible to satisfy for every constraint.

Recent approaches to solving problems in these domains rely on optimization techniques that map constraints into multi-valued utility functions. Instead of finding an assignment that satisfies all constraints, these approaches find an assignment that produces a high level of global utility. This extension to the original DisCSP approach has become popular in multi-agent systems, and has been labeled the Distributed Constraint Optimization Problem (DCOP) [6].

Current algorithms that solve complete DCOPs use two main approaches: search and dynamic programming. Search based algorithms that originated from DisCSP typically use some form of backtracking [18] or bounds propagation, as in ADOPT [11]. These algorithms tend to have polynomial memory requirements for each agent, but incur a large performance penalty to establish initial bounds and to backtrack to previous solutions. Some previous work has been done in the area of preprocessing techniques for ADOPT in [1]. Dynamic programming based algorithms include DPOP and its extensions [14, 16]. The original version of DPOP is not memory bounded, and the newest extension, MB-DPOP, uses iteration to achieve memory bounding but does not take advantage of search space pruning. Some other algorithms exist, such as OptAPO [8], but involve partial centralization and/or have been shown to scale worse than ADOPT and DPOP [16].

We introduce a new hybrid algorithm, ADOPT with bounded dynamic programming (ADOPT-BDP), that can take advantage of the strengths of both ADOPT and DPOP. Our algorithm employs a memory bounded version of DPOP to generate lower and upper bounds for each agent. The bounds are then used to guide the search thresholds in ADOPT. The result is an algorithm that can take advantage of a bounded amount of memory at each agent. For many problem types and configurations we achieve better performance than either DPOP or ADOPT.

We begin with a formal introduction of DCOP and a brief summary of both DPOP and ADOPT. We then introduce our version of ADOPT with bounded dynamic programming. We show how the intermediate results from the dynamic programming can be used as lower and upper bounds and threshold values at each agent. We compare performance in several domains, and results include comparisons to the original DPOP and ADOPT algorithms, to the earlier techniques presented in [1], and to the recent memory bounded version of DPOP, MB-DPOP [16]. We conclude with a discussion of results, related work, and future work.

2 DCOP

DCOP has been formalized in slightly different ways in recent literature, so we will adopt the definition as presented in [14]. A Distributed Constraint Optimization Problem with n variables and m constraints consists of the tuple $< X, D, U >$ where:

- $X = \{x_1,..,x_n\}$ is a set of variables, each one assigned to a unique agent
- $D = \{d_1,..,d_n\}$ is a set of finite domains for each variable
- $U = \{u_1,..,u_m\}$ is a set of utility functions such that each function involves a subset of variables in X and defines a utility for each combination of values among these variables

An optimal solution to a DCOP instance consists of an assignment of values in D to X such that the sum of utilities in U is maximal. Problem domains that require minimum cost instead of maximum utility can map costs into negative utilities. The utility functions represent soft constraints but can also represent hard constraints by using arbitrarily large negative values.

3 State of the Art Complete DCOP Algorithms

Current algorithms that solve complete DCOPs use two main approaches: search and dynamic programming. Our algorithm uses a hybrid approach that involves both search and dynamic programming based on ADOPT and DPOP. Our algorithm also uses the traditional arrangement of agents into a pseudotree to solve the problem. We refer the reader to [14] or [11] for more detail on pseudotrees and how they are created.

3.1 ADOPT

The original ADOPT algorithm operates asynchronously on a pseudotree of agents [11]. Each agent represents a single variable in the DCOP and chooses values concurrently. Three main types of messages are exchanged between agents:

- VALUE messages sent down to all neighbors lower in the pseudotree
- COST messages sent up only to the parent
- THRESHOLD messages sent down only to children

Each agent keeps track of lower and upper bounds and threshold values as well as contextual information including the state of other variable values when the bounds/thresholds were recorded. The threshold values determine when an agent changes its variable to a different value. By carefully recording bounds on children, when a parent changes its value it can send a threshold to its child to help the child quickly recover a previous search. The algorithm begins termination when the root agent's threshold reaches its upper bound.

Since ADOPT is an opportunistic best-first search, the average case complexity will depend heavily on the problem structure. The worst case complexity is determined by the total messages sent, and is O($|d|^n$) where $|d|$ is the maximum domain size, and n is the depth of the pseudotree. ADOPT is also completely asynchronous, so additional message overhead exists when a context is out of date and the message is discarded.

3.2 DPOP

The original DPOP algorithm operates in three main phases. The first phase generates a traditional pseudotree from the DCOP instance using a distributed algorithm. The second phase joins utility hypercubes (multi-dimensional matrices) from children and the local agent and propagates them towards the root. The third phase chooses an assignment for each domain in a top down fashion beginning with the root agent. Each phase is semi-synchronous in that each parent must wait for messages from its children in phase 2, and children wait for messages from parents in phase 3. However, agents in seperate branches of the tree may process concurrently. Given a strict linear ordered tree DPOP is synchronous, but given a fully distributed binary tree up to half of the agents may be concurrently processing at any given time.

The complexity of DPOP depends on the size of the largest computation and utility message during phase two. It has been shown that this size directly corresponds to the induced width of the pseudotree generated in phase one [14]. DPOP uses polynomial time heuristics to generate the pseudotree since finding the minimum induced width pseudotree is NP-hard. The induced width of the pseudotree is always less than or equal to the depth of the pseudotree. The memory complexity of DPOP is O($|d|^m$) where $|d|$ is the maximum domain size and m is the induced width of the pseudotree. The number of messages produced is linear.

The original version of DPOP is extremely fast and efficient for loosely constrained problems, and can scale to thousands of sparsely constrained agents. Unfortunately for many problems the original version of DPOP is intractable because of constraint density. In such problems the exponential memory requirement precludes any solution. A memory bounded version of DPOP [16] (MB-DPOP) bounds the memory requirement, but in doing so increases the number of messages from linear to exponential, making a direct tradeoff between memory usage and messages/cycles.

4 A Hybrid Approach: ADOPT-BDP

We now introduce our hybrid algorithm, ADOPT-BDP, that takes advantage of the strengths of both ADOPT and DPOP. Our algorithm extends the ADOPT algorithm as shown in Algorithm 1. We do not show the procedure for pseudotree creation as they are the same as employed in previous DCOP algorithms [11, 14]. We have included only our modifications to the ADOPT algorithm and not its entirety.

The pseudocode shown in Algorithm 1 follows a typical sequence for our hybrid algorithm:

1. An agent initiates a DPOP utility propagation by sending a DPRequest to its children.
2. This DPRequest propagates to agents at the pseudotree leaves who begin propagating utility upwards.
3. Each agent stores the propagated utility in its $boundsCache$.
4. The propagation stops at the agent who initiated the DPRequest.
5. Traditional ADOPT backtracking commences.
6. When an agent receives a new value assignment for its context it initializes its bounds with values from its $boundsCache$.

4.1 DPOP with memory bounds

The DPOP utility propagation that is part of our hybrid algorithm is memory bounded. The original DPOP algorithm can use up to an exponential amount of memory during its utility propagation phase. It was shown in [13] that the DPOP algorithm can be modified to produce an approximate result using memory bounds. We have used the techniques from [13] to split hypercubes requiring too much memory into upper and lower bounds hypercubes. These two hypercubes can be reduced in size to meet the memory bounds by removing the dimension representing the highest agent in the tree. When a dimension is removed from a lower bounds hypercube we choose the minimum value over the remaining dimensions for each value in the removed dimension. Conversely we choose the maximum value for the upper bounds hypercube. If the memory bound was not reached during utility propagation, the upper and lower bounds are equal.

Algorithm 1 ADOPT with Bounded Dynamic Programming

```
1: initialize()
2: if $agent$ is DP source then
3:     send DPRequest to children
4: $waitingForDP \leftarrow true$

5: whenReceived(DPRequest $r$)
6: if $agent$ is leaf then
7:     $propagateUtil(r)$
8:     $waitingForDP \leftarrow false$
9: else
10:    add $r$ to $waitList$
11:    $waitingForDP \leftarrow true$
12:    send $r$ to children

13: propagateUtil(DPRequest $r$)
14: $u \leftarrow$ combine $utilPropagations$ (bounded)
15: add $u$ to $boundsCache$
16: reduce $u$ by domain of $agent$
17: if $r$ did not originate at $agent$ then
18:    send UtilMessage($u$) to parent
19: remove $r$ from $waitList$
20: if $waitList$ is empty then
21:    $waitForDP \leftarrow false$
22:    $backtrack()$

23: whenReceived(UtilMessage $u$)
24: add $u$ to $utilPropagations$ for $u.r$
25: if all child messages for $u.r \in utilPropagations$ then
26:    $propagateUtil(u.r)$

27: backtrack()
28: if not $waitingForDP$ then
29:    normal backtrack()
30:    if $agent$ is DP source AND context change then
31:        send DPRequest to children
32:        $waitingForDP \leftarrow true$

33: updateContexts($V = v$)
34: $\ldots$
35: if $(V = v)$ not compatible with $context$ then
36:    update $context$
37:    set $lb$, $ub$, and $threshold$ from $boundsCache$
38: $\ldots$
```

4.2 Integration with ADOPT

When an ADOPT agent receives a VALUE message, it updates its contexts with the new value assignment. In normal ADOPT when a variable in the context changes value, each child lower bound is set to zero and the child upper bound is set to the maximum possible value (i.e. positive infinity). In our modified update method (line 32), we check the $boundsCache$ for a result from the DPOP component with a matching context. If found, we can immediately set the bounds and thresholds to better values. If the DPOP component completed its propagation without reaching the memory bound, then our cached lower and upper bounds are equal. The agent does not need to explore any child assignments for this context because it has perfect information.

Since each agent in the ADOPT algorithm operates asynchronously, we wanted to be careful that our agents would not throw away work done by the DPOP component. To accomplish this, we implemented a synchronous mode for the utility propagation. When an agent issues a DPRequest, each agent encountered while this request propagates to the leaves enters a blocking mode. This blocking mode prevents the agent from backtracking while it is waiting for the utility propagation from its children. An agent resumes asynchronous backtracking when all DPRequests have been matched to subsequent utility propagations and its $waitList$ is empty.

In this paper we have limited the source for DPRequests to the root node, which produces a phased execution of utility propagation and ADOPT. If the memory bound does not change during execution, there are only two phases, the initial bounded utility propagation and then ADOPT. For the any-bound solution presented in this paper, the phases repeat these two initial phases until the bounded utility propagation reaches the maximum allowable memory bound. Using multiple nodes as sources for DPRequests allows ADOPT to continue execution in the upper portions of the search tree while utility propagation is performed in the lower portions. Results for multiple node settings are not presented in this paper and are the topic of future research.

4.3 Correctness and Complexity Analysis

The correctness of our hybrid algorithm can easily be seen by observing that we never overestimate a lower bound or underestimate an upper bound. During the DPOP utility propagation when we reduce a hypercube because of memory limitations we lose information about a dimension. We choose the minimum value from this dimension for our lower bound, so we guarantee that our lower bound is less than or equal to any possible assignment for the reduced dimension. We choose the maximum value for this dimension for our upper bound, and similarly guarantee that it is greater than or equal to any possible assignment for the reduced dimension. Thus since our bounds are correct for all agents, ADOPT can perform its best first search within these bounds and is guaranteed to find the optimal solution.

The worst case complexity of the hybrid algorithm is generally the same as ADOPT, which is O($|d|^n$) because of the possible total number of messages. The only case where the worst case complexity is the same as DPOP, O($|d|^m$), is when the given memory bound is large enough for complete DPOP utility propagation. However, in practice the performance is much better than the worst case because the worst case assumes the utility propagation provides little benefit. While the benefit from combining search and dynamic programming varies according to the problem domain and setup, it is very clear from our results that even a small amount of dynamic programming can provide large speedups to searches.

4.4 Any-bound solution

Several static values for memory bounds perform well for most problem types. If a problem type is known in advance, it would be best to use offline statistical methods to learn an optimal setting for the memory bound for each problem type. Since this is not possible for all settings, we have developed a simple iterative increasing approach, shown in Algorithm 2, to set the value for MB. The approach begins by setting MB to a small value. The first utility propagation runs using this value. The total time for propagation, t, is measured. If the algorithm does not find the solution after another t units of time, the value for MB increases until it reaches the maximum allowable memory bound.

Algorithm 2 Any-bound Algorithm

```
 1: $MAX \leftarrow$ max memory allowed
 2: $MB \leftarrow$ small amount of memory $< MAX$
 3: $t \leftarrow$ total time taken by last util propagation

 4: **backtrack()**
 5: **if** not $waitingForDP$ **then**
 6:     normal backtrack()
 7:     **if** $agent$ is DP source AND context change **then**
 8:         ...
 9:     **else if** $MB < MAX$ AND time $> t$ **then**
10:         $MB \leftarrow max(MB \cdot |d|, MAX)$
11:         send DPRequest to children
12:         $waitingForDP \leftarrow true$
```

5 Results

We tested our new hybrid algorithm in 3 different domains: distributed sensor networks (DSN), graph coloring problems (GCP), and meeting scheduling problems (MSP). For the initial tests we used the same set of problem data used in [1], which can be obtained from [12]. Additionally we generated a set of increasingly difficult meeting scheduling problems that require increasing amounts of memory for DPOP.

Results are reported using four measurements: total number of messages (Messages), total message byte size (Size), number of cycles (Cycles), and total runtime (Time) in milliseconds. Similar to experiments in previously published DCOP algorithms [11, 1, 14, 16], all algorithms were run in a deterministic fashion by giving each agent an opportunity to process messages once per cycle. Note that the total number of messages or cycles should not be directly compared between algorithms. DPOP-based algorithms are semi-synchronous and send low numbers of large-size messages and ADOPT-based algorithms are asynchronous and send high numbers of small-size messages. Similarly, ADOPT-based algorithms do less computation and send more messages per cycle, while DPOP-based algorithms do more computation and send less messages per cycle.

We compare our algorithm, ADOPT-BDP, with ADOPT-DP2 [1], and MB-DPOP [16]. We do not include the original ADOPT because ADOPT-DP2 is significantly better for all cases. We also do not include the original DPOP because MB-DPOP with a large enough memory bound is exactly the same. For ADOPT-BDP and MB-DPOP we set a maximum memory bound. For the any-bound version of ADOPT-BDP, represented by a + next to the bound, we start the memory bound at this value and allow it to increase up to the maximum available in the system. All memory bounds are shown as the maximum total size of values in a hypercube. Each value is stored internally as a 4-byte integer, so this allows up to 1000 values in a hypercube for ADOPT-BDP(4KB).

	DSN				GCP			
Algorithm	Msgs	Size	Cycles	Time	Msgs	Size	Cycles	Time
ADOPT-DP2	19797	485008	121	382	200330	5101232	6010	2390
ADOPT-BDP(4KB)	137	9746	27	27	1056	30762	39	36
ADOPT-BDP(40KB)	137	9746	27	28	19	4108	9	18
ADOPT-BDP(4KB+)	137	9746	27	29	130	8831	14	23
MB-DPOP(4KB)	137	9472	27	23	23	4187	11	17
MB-DPOP(40KB)	137	9472	27	23	19	4070	9	16

Table 1. Results from the graph coloring problem (GCP) and distributed sensor network (DSN) domains. The total message size (Size) is in bytes and the total runtime (Time) is in milliseconds. All numbers are averages over 98 tests for the GCP and 59 tests for the DSN. Each set of tests had the following graph properties: GCP: Induced Width (IW) = 3 to 7, number of variables (V) = 9 to 12, number of constraint edges (E) = 18 to 30, size of domains (|d|) = 3. DSN: IW = 2 to 2, V = 40 to 100, E = 32 to 121, |d| = 5.

5.1 DSN and GCP

The DSN and GCP tests use the model from [11, 1] and can be obtained from [12]. They are trivial for DPOP based algorithms since they are sparse problems and require only a small amount of memory to perform dynamic programming. We show results for these tests in Table 1. Neither test case is particularly difficult and are not as interesting as later results because they are solved by ADOPT-BDP and MB-DPOP in under 40 milliseconds using less than 40KB of memory. However, these tests highlight the usefulness of including dynamic programming in our hybrid approach, as seen in the improvement over ADOPT-DP2, but do not benefit from search space pruning because the search space is very small.

	Messages				Size			
Algorithm	A	B	C	D	A	B	C	D
ADOPT-DP2	574697	170152	843375	259414	14264392	4337852	20625288	6300347
ADOPT-BDP(40KB+)	12759	5528	9099	4144	455224	235978	363331	213342
ADOPT-BDP(4KB)	60210	23288	80559	18417	1514479	616857	1992710	470906
ADOPT-BDP(40KB)	27747	5814	10131	4144	747693	225052	374771	213342
ADOPT-BDP(400KB)	4467	2252	7011	4431	358609	609835	587176	380091
ADOPT-BDP(4MB)	44	50	140	142	575550	2012400	1200000	518656
MBDPOP(4KB)	9776	20965	15284	9488	1155686	2822918	1852018	756398
MBDPOP(40KB)	571	1862	1088	908	589198	2269986	1224274	538244
MBDPOP(400KB)	82	202	216	161	576548	2016706	1201770	518884
MBDPOP(4MB)	44	50	140	142	575462	2012300	1199720	518372

	Cycles				Time			
Algorithm	A	B	C	D	A	B	C	D
ADOPT-DP2	8905	2425	4409	1360	11389	3560	17316	5235
ADOPT-BDP(40KB+)	224	90	66	39	453	275	319	204
ADOPT-BDP(4KB)	950	343	439	116	1684	783	1814	454
ADOPT-BDP(40KB)	449	94	71	39	750	265	335	205
ADOPT-BDP(400KB)	88	44	55	41	305	505	471	326
ADOPT-BDP(4MB)	21	14	19	19	506	1727	1022	430
MBDPOP(4KB)	4883	5364	6766	3868	968	2216	1643	868
MBDPOP(40KB)	282	511	442	352	431	1545	893	429
MBDPOP(400KB)	39	61	54	27	437	1487	889	397
MBDPOP(4MB)	21	14	19	19	508	1732	1017	430

Table 2. Results from the meeting scheduling problem (MSP) domain. Case A has 23 variables (V), 43 constraint edges (E), and induced width (IW) of 5; Case B has V=26, E=47, and IW=5; Case C has V=71, E=122, and IW=5; Case D has V=72, E=123, and IW=5.

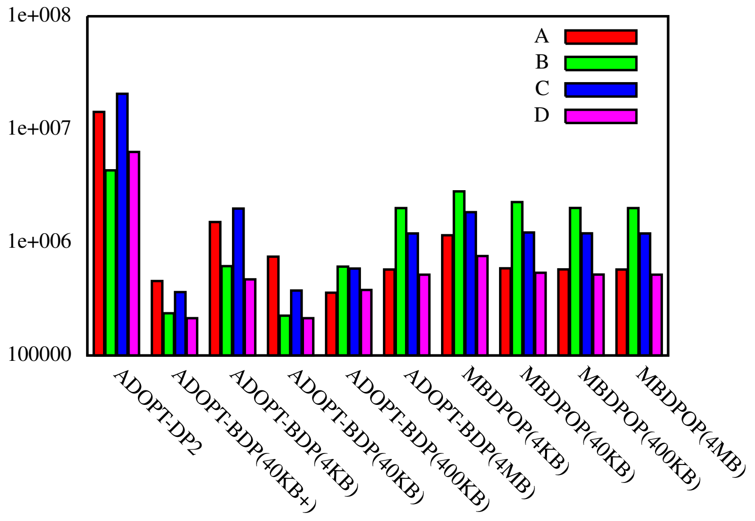

Fig. 1. MSP: Total Message Size (in bytes, logarithmic)

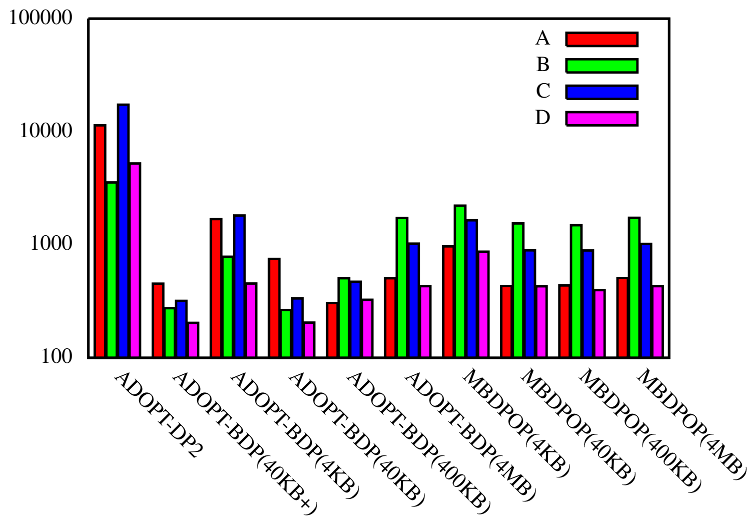

Fig. 2. MSP: Runtime (in milliseconds, logarithmic)

5.2 MSP

Results for the MSP cases from [1] are shown in Figures 1 and 2 and Table 2. We used the problem sets generated using the PEAV model from [7]. There are four sets of test cases, labelled A, B, C, and D in the figure. Each case contained 30 similar problems and domain sizes of 9. The results shown are averages over these 30 problems.

We show MB-DPOP and ADOPT-BDP with memory bound settings of 4KB, 40KB, 400KB, and 4MB. Since the maximum induced width is 5 for these problems, the maximum memory required at an agent for full utility propagation is $9^{5+1} = 531441$ values (2.1MB). Thus the 4MB bound produces a full utility propagation. We also show the any-bound version of ADOPT-BDP starting with a 40KB memory bound which it can increase up to 4MB (shown as ADOPT-BDP(40KB+)). We observe that ADOPT-BDP is more efficient at solving the problems in all four sets of test cases than MB-DPOP for various bound settings. The any-bound version of ADOPT-BDP is the best performer over all four of the test cases.

5.3 MSP (scaling)

To test the scalability of our hybrid approach to large problem sets, we generated 40 test problems using the same PEAV model for the MSP as before. For these tests we slowly increased each parameter at similar rates, with the range of values covered for V from 10 to 342, E from 9 to 622, and IW from 1 to 13. We set a runtime threshold of 20 minute and ran 4 tests on each algorithm for each parameter setting. If a test did not complete for a given setting of V, then it does not have a data point for that setting. Results are shown in Figures 5 and 4.

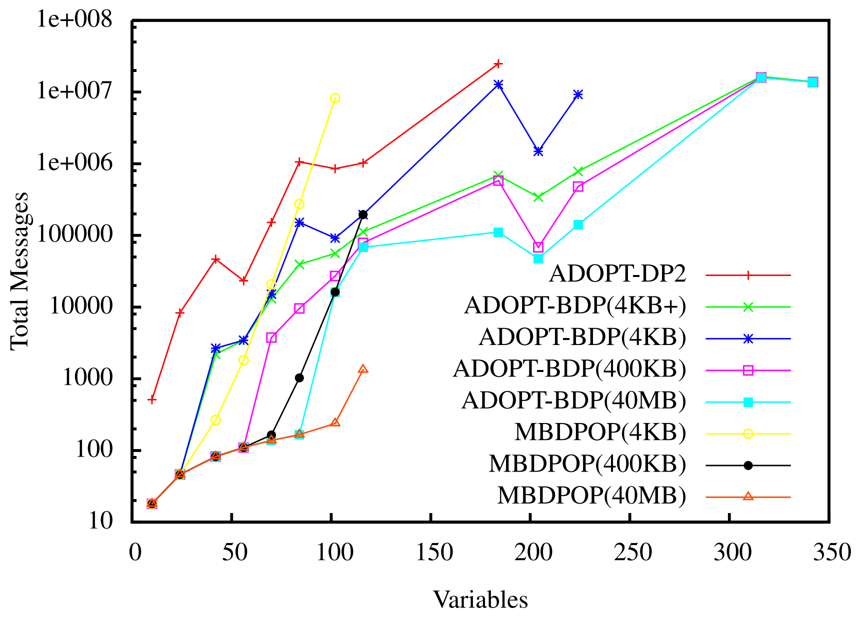

Fig. 3. MSP (scaling difficulty): Total Messages (logarithmic)

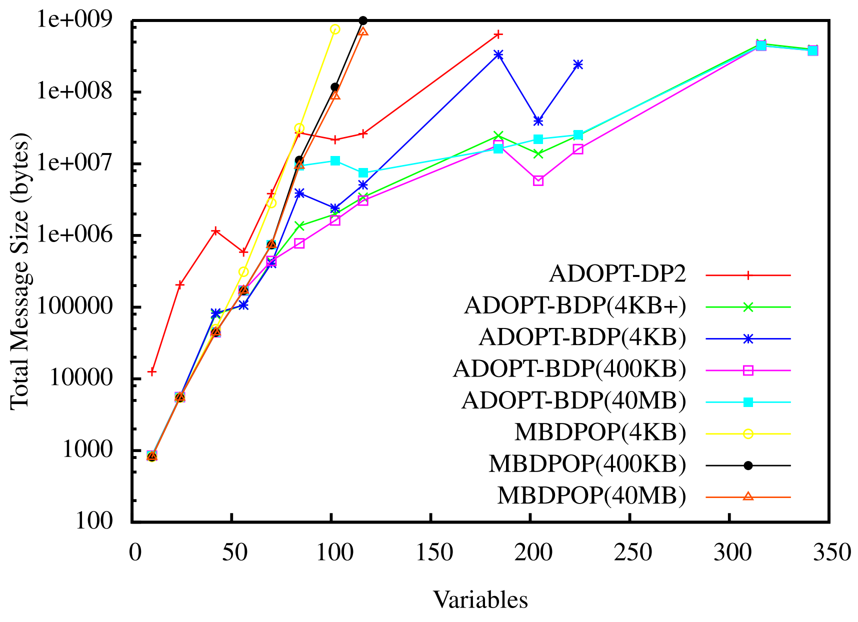

Fig. 4. MSP (scaling difficulty): Total Message Size (in bytes, logarithmic)

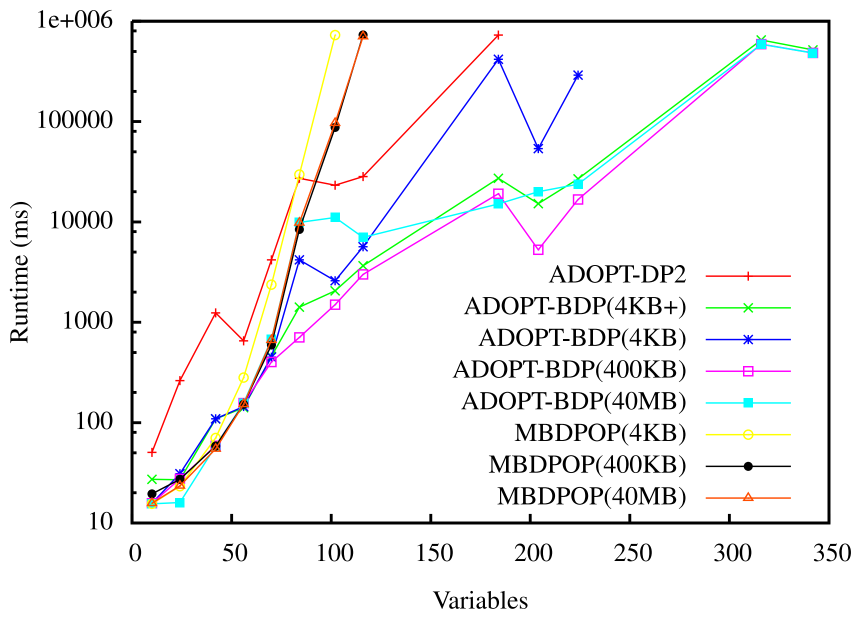

Fig. 5. MSP (scaling difficulty): Runtime (in milliseconds, logarithmic)

The scalability of our approach clearly indicates the advantages of combining search and dynamic programming. By using the asynchronous search space pruning capabilities of ADOPT, we are able to solve much larger problems than MB-DPOP, and notice improved performance when the memory requirements for a problem approach the amount of available memory. In particular, significant improvements begin in both overall message size and runtime in Figures 5 and 4 at the tests with 84 variables (IW of 6; max memory = 40MB). We also observe that our hybrid algorithm scales better than the previous best ADOPT technique, DP2, and offers similar performance to MB-DPOP for low induced width problems. MB-DPOP cannot complete the problems at 184 variables with induced width of 9 or more difficult problems within the 20 minute time limit.

6 Related Work

ADOPT and DPOP are covered earlier in this paper. We included in our comparisons related preprocessing techniques for ADOPT, namely the DP2 procedure from [1]. We also included comparisons with MB-DPOP. We did not compare results to all existing complete algorithms for DCOPs. OptAPO has been shown to have similar overall complexity to ADOPT in [4], but makes a tradeoff for less messages with more computation spent in each cycle. NCBB, presented in [3] offers a somewhat similar branch and bound algorithm to ADOPT, and for many cases performs better than ADOPT, but shares a similar scaling problem. Its performance closely resembles that of ADOPT using the DP2 preprocessing heuristic. NCBB has also been extended in [2] to cache bounds for previous search contexts, allowing quick retrieval of a previous search state. This technique improved the performance of NCBB, and may be useful to implement in our ADOPT-BDP algorithm.

Several incomplete algorithms exist for solving DCOPs. They often provide better performance, especially on difficult problems, but do not guarantee to find the optimal solution. A description and comparison of two popular local search algorithms, DBA and DSA, can be found in [19]. A hybrid local/global search algorithm using DPOP is presented in [15]. All of these algorithms show good results for time versus optimality, but do not converge to the optimal solution over time. We did not provide comparisons with these algorithms because they are suboptimal. However, since our algorithm is based on ADOPT, we can use the any-time and bounded error approximation techniques that have previously been applied to ADOPT in [11].

Some centralized constraint processing techniques also combine search and dynamic programming. AND/OR Branch-and-Bound (AOBB) search is combined with static and dynamic mini-bucket elimination in [9]. Backtracking with Tree-Decomposition (BTD) also combines backtracking search with stored structural goods/nogoods to achieve a similar effect for CSPs in [5]. Both AOBB and BTD have extensions for valued-CSPs for optimization problems. Like AOBB, BTD is not an asynchronous algorithm. BTD also does not use a bottom-up heuristic to guide the search; instead it fully explores partial assignments and stores the results at the separators (which avoids redundent exploration on subsequent searches). Many additional improvements have been made to the original AOBB and BTD algorithms in the centralized context. It is unclear how AOBB and BTD and their improvements would extend to a distributed setting.

7 Conclusions and Future Work

We have introduced a hybrid approach to solving distributed constraint optimization problems. Our approach is based on two common approaches to solving DCOPs: search and dynamic programming. By extending current algorithms that use each approach, we are able to achieve performance that reflects the strengths of both approaches. We also introduce an iterative increasing procedure that chooses an appropriate memory bound less than or equal to the available memory.

We compare the performance of our hybrid algorithm to other complete algorithms for DCOPs. Because of the diversity of implementations, the total message byte size and runtime measurements best reflect the overall complexity of the algorithms involved. Using our any-bound approach, we conclude that our algorithm is better than previous ADOPT-based approaches, and for problems that can be solved in memory using DPOP, our algorithm performs similarly to DPOP. Furthermore, we show that our any-bound algorithm is similar to MB-DPOP for small problems, but scales better on hard problems with a much lower curve than MB-DPOP.

There are several possibilities to further enhance our hybrid approach. First, caching mechanisms may prove useful to prevent discarding bounds when the variable context is changed in ADOPT. Second, in this paper we have limited the source for DPRequests to the root node, but using additional source agents for DPRequests may prove beneficial. Third, we intend to explore alternative mechanisms for memory bounding such as symbolic value representation and value-bounded sparse hypercubes. Fourth, our initial hybrid algorithm relied on the integration of two existing distributed algorithms, providing an inherently distributed algorithm, but limiting techniques to concepts present in both algorithms. Several hybridization techniques have been introduced in the area of centralized algorithms, and extending these techniques for a distributed setting is an area for future research.

References

1. S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1041–1048, New York, NY, USA, 2005. ACM Press.
2. A. Chechetka and K. Sycara. An any-space algorithm for distributed constraint optimization. In *Proceedings of AAAI Spring Symposium on Distributed Plan and Schedule Management*, March 2006.
3. A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1427–1429, New York, NY, USA, 2006. ACM Press.
4. J. Davin and P. J. Modi. Impact of problem centralization in distributed constraint optimization algorithms. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1057–1063, New York, NY, USA, 2005. ACM Press.
5. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1):43–75, 2003.
6. J. Liu and K. P. Sycara. Exploiting problem structure for distributed constraint optimization. In *Proceedings of the First International Conference on Multi–Agent Systems*, pages 246–254, San Francisco, CA, 1995. MIT Press.
7. R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 310–317, Washington, DC, USA, 2004. IEEE Computer Society.
8. R. Mailler and V. Lesser. Asynchronous Partial Overlay: A New Algorithm for Solving Distributed Constraint Satisfaction Problems. *Journal of Artificial Intelligence Research*, 25:529–576, April 2006.
9. R. Marinescu and R. Dechter. AND/OR branch-and-bound for graphical models. In *IJCAI*, pages 224–229, 2005.
10. P. J. Modi, H. Jung, M. Tambe, W.-M. Shen, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. *Lecture Notes in Computer Science*, 2239:685–, 2001.
11. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS 03*, 2003.
12. J. P. Pearce. USC DCOP repository, 2005.
13. A. Petcu and B. Faltings. A-DPOP: Approximations in distributed optimization. In *Poster in Principles and Practice of Constraint Programming CP 2005*, pages 802–806, Sitges, Spain, October 2005.

14. A. Petcu and B. Faltings. DPOP: A scalable method for multiagent constraint optimization. In *IJCAI 05*, pages 266–271, Edinburgh, Scotland, Aug 2005.
15. A. Petcu and B. Faltings. LS-DPOP: A propagation/local search hybrid for distributed optimization. In *CP 2005- LSCS'05: Second International Workshop on Local Search Techniques in Constraint Satisfaction*, Sitges, Spain, October 2005.
16. A. Petcu and B. Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *IJCAI-07*, Hyderabad, India, Jan 2007.
17. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.
18. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.
19. W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 185–192, New York, NY, USA, 2003. ACM Press.