IDB-ADOPT : A Depth-First Search DCOP Algorithm

William Yeoh[†], Sven Koenig[†], Ariel Felner[‡]

[†]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{wyeoh, skoenig}@usc.edu

[‡]Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva, 85104, Israel
felner@bgu.ac.il

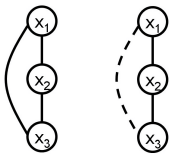
Abstract. Many agent coordination problems can be modeled as distributed constraint optimization problems (DCOPs). ADOPT is an asynchronous and distributed search algorithm that is able to solve DCOPs optimally. In this paper, we introduce Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches. Each depth-first search is provided with a bound, initially a large integer, and returns the first solution whose cost is smaller than or equal to the bound. The bound is then reduced to the cost of this solution minus one and the process repeats. If there is no solution whose cost is smaller than or equal to the bound, it returns a cost-minimal solution. Thus, IDB-ADOPT is an anytime algorithm that solves DCOPs with integer costs optimally. Our experimental results for graph coloring problems show that IDB-ADOPT runs faster (that is, needs fewer cycles) than ADOPT on large DCOPs, with savings of up to one order of magnitude.

1 Introduction

Many agent coordination problems can be modeled as distributed constraint optimization problems (DCOPs), including the scheduling of meetings [8], the coordination of unmanned aerial vehicles [14], and the allocation of targets to sensors in sensor networks [7, 10, 13]. Unfortunately, solving DCOPs optimally is NP-hard. A variety of search algorithms have therefore been developed to solve DCOPs as fast as possible to scale up to real-world domains [9, 10, 12, 3]. ADOPT (Asynchronous Distributed Constraint Optimization) [10] is one of the pioneering DCOP algorithms and currently probably the most extended one [11, 4, 2]. It is an asynchronous and distributed best-first search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. Researchers have recently scaled up ADOPT by one order of magnitude by providing it with informed heuristics that focus its search [1]. However, its runtime is still large for realistic DCOPs and it therefore needs to get scaled up further. In particular, in the original ADOPT, each vertex can switch back and forth between different values and then has to redo many searches since the results

from the previous searches have already been purged from memory due to its memory limitations. In this paper, we address this problem of ADOPT by introducing Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches, where vertices do not switch back and forth between different values. IDB-ADOPT is motivated by insights from heuristic search that depth-first searches can outperform best-first searches in combinatorial domains with search trees whose depths are bounded [15], and DCOPs are such domains. Each depth-first search of IDB-ADOPT is provided with a bound, initially a large integer, and returns the first solution whose cost is smaller than or equal to the bound. The bound is then reduced to the cost of this solution minus one. If there is no solution whose cost is smaller than or equal to the bound, it returns a cost-minimal solution. Thus, IDB-ADOPT is an anytime algorithm that solves DCOPs with integer costs optimally. Our experimental results for graph coloring problems show that IDB-ADOPT runs faster than ADOPT on large DCOPs, with savings of up to one order of magnitude.

2 DCOPs



x_1	x_2	Cost
0	0	5
0	1	8
1	0	20
1	1	3

x_1	x_3	Cost
0	0	5
0	1	8
1	0	20
1	1	3

x_2	x_3	Cost
0	0	5
0	1	8
1	0	20
1	1	3

Fig. 1. DCOP Example

Distributed constraint optimization problems (DCOPs) model agent coordination problems as constraint optimization problems on *constraint graphs*. Each vertex of a constraint graph represents an agent (sometimes referred to as variable) and can *take on* a value from a given set (its domain). Edges denote constraints. The cost of a constraint depends on the values of the vertex endpoints of the corresponding edge, given by a table. We assume in this paper that all costs are non-negative integers. An assignment of values to all vertices is a (complete) *solution*. The *cost* of the solution is the sum of the costs of the constraints. One wants to find a cost-minimal solution. As an example, Figure 1 (left) shows a simple DCOP with three vertices, x_1, x_2 and x_3, that each can take on the values zero or one. There are three constraints, whose costs are specified by the tables. For example, there is one constraint between vertices x_1 and x_2. If both vertices take on value zero, then the cost of the constraint is five. The cost of the solution $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$ is nine and cost-minimal.

3 ADOPT

We now give a slightly simplified description of ADOPT that makes the search principle behind it easy to understand and is sufficient for our purposes. The reader is referred to the original paper [10] for a full step-by-step description of DCOPs, ADOPT and the message passing mechanism used by ADOPT. ADOPT basically operates as follows: In a preprocessing step, ADOPT transforms the constraint graph into a *constraint tree* with the property that constraints exist only between a vertex and its ancestors and/or descendants. To simplify our description further, we assume that every vertex has at most one child in the constraint tree. In other words, the constraint tree is a chain, which is the case for our example DCOP. Figure 1 (center) shows one possible constraint tree for our example DCOP. ADOPT then performs a search as will be described next.

3.1 Values of ADOPT

During the search of ADOPT, every vertex of the constraint graph maintains some values. Every vertex maintains the value from its domain that it currently takes on (called its *current value*), initially the best value (the best value is defined below). Every vertex also maintains the values of its (connected) ancestors in the constraint tree (called its *current context*). These values correspond to a partial solution of the DCOP. Every vertex maintains, for each possible value that it can take on, *lower bounds* on the cost of the cost-minimal solution of the DCOP that is consistent with this value and its current context. These lower bounds are initialized with the sum of the costs of the constraints between the (connected) ancestors, which can be calculated since the current context is known. One can obtain larger initial lower bounds to speed up the search by adding informed pre-computed values (called heuristics), if available, to the the sum of the costs of the constraints between the (connected) ancestors. We call the lower bound of the current value of a vertex its *current lower bound*. We call the smallest lower bound over all values that a vertex can take on its *best lower bound* and the corresponding value its *best value*. Every vertex also maintains an *upper bound* on the cost of the cost-minimal solution of the DCOP that is consistent with its current context. The upper bound is simply the cost-minimal solution found so far during the search that is consistent with the current context, initially infinity. Finally, every vertex also maintains a *threshold* (whose role will be explained below), initially zero. For every vertex, ADOPT maintains the following threshold invariant: The threshold of a vertex is guaranteed to be between its best lower bound and its upper bound. To keep the threshold invariant satisfied, the vertex changes the value of its threshold as follows: If the threshold is smaller than the best lower bound of the vertex then the threshold is increased to the best lower bound. Similarly, if the threshold is larger than the upper bound then the threshold is decreased to the upper bound. These situations occur when the best lower bound increases above the threshold or the upper bound decreases below the threshold.

3.2 Operation of ADOPT

Each vertex operates as follows: If its current lower bound is smaller than or equal to the threshold, then the vertex keeps its current value. Otherwise, it changes its current

value by taking on its best value and then informs its (connected) descendants in the constraint tree about its new value. Its descendants then perform similar computations to help the vertex decrease its upper bound and increase its current lower bound. ADOPT terminates when the threshold of the root vertex of the constraint tree is equal to its upper bound.

3.3 Thresholds of ADOPT

The threshold of a vertex is of special importance in the remainder of this paper. We now explain how it influences the values taken on by the vertex. As already explained above, if the current lower bound of a vertex is smaller than or equal to the threshold, then the vertex keeps its current value. Otherwise, it changes its current value by taking on its best value. At this point in time, there are two possible cases:

- **Case 1:** If there are still values whose lower bounds are smaller then its threshold, then the vertex takes on its best value and keeps it until the lower bound of that value increases above the threshold. The vertex repeats this procedure until all lower bounds are larger than or equal to the threshold and Case 2 is reached. Note that, during Case 1, the vertex takes on each value only once, unless its ancestors switch values, and keeps this value as long as the lower bound of the value is smaller than or equal to the threshold even if some other value has a smaller lower bound. This results in a depth-first search behavior.
- **Case 2:** If all lower bounds are larger than or equal to the threshold, then the vertex increases the threshold to the best lower bound (to satisfy the threshold invariant), if necessary, and then takes on its best value until the lower bound of that value increases. The vertex then repeats this procedure. Note that, once Case 2 is reached, the vertex cannot go back to Case 1 unless its ancestors switch values. During Case 2, the vertex always takes on its best value. This results in a best-first search behavior. The vertex can switch back and forth between values and, in the process, take on the same value several times.

ADOPT initializes the threshold of the root vertex of the constraint tree to zero. The root vertex therefore starts with Case 2, and ADOPT performs a best-first search. The threshold is important when a vertex switches back to a value that it had taken on earlier already during the best-first search. In this case, it has already increased the lower bound of this value, otherwise it would not have switched from the value to a different one earlier. It now has to redo this search to restore the lower bounds of its descendants at the point in time when it last switched from this value to another value. These lower bounds have been purged from memory since each vertex uses only a bounded amount of memory. ADOPT restores these lower bounds with a depth-first search (inside the best-first search) to be efficient. A best-first search is not needed for this purpose since ADOPT only repeats a previous search. Case 1 performs this depth-first search automatically.

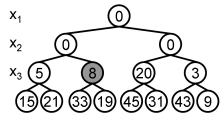


Fig. 2. Search Tree

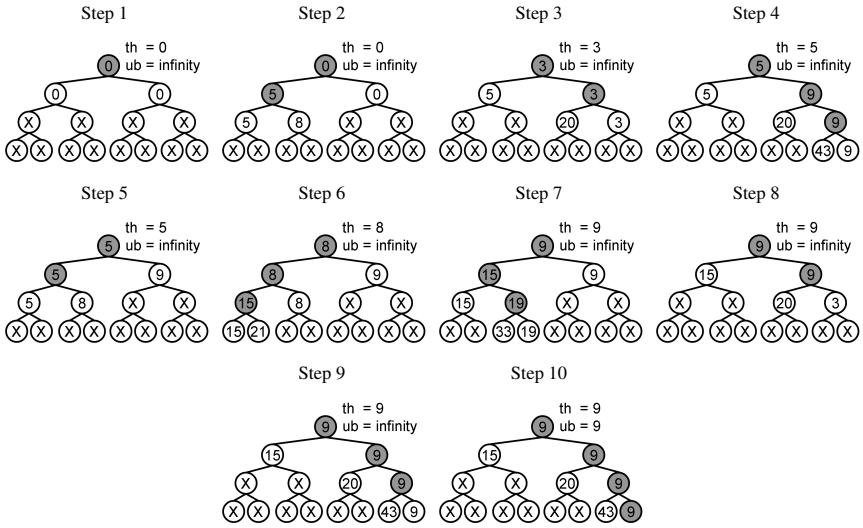


Fig. 3. Illustration of ADOPT

4 Illustration of ADOPT

We now visualize the searches that ADOPT performs for our example DCOP. We use search trees for this purpose, as shown in Figure 2. The constraint tree of our example DCOP orders the vertices completely since it is a chain. The search tree reflects this ordering. Its first (second, third) layer corresponds to vertex x_1 (x_2, x_3), respectively. Left branches means that the corresponding vertex takes on the value zero, and right branches mean that it takes on the value one. Thus, each node in the search tree represents the values of all ancestors of the corresponding vertex in the constraint tree. The shaded node in the search tree, for example, corresponds to $x_1 := 0$ and $x_2 := 1$. It is annotated with the initial lower bound on the cost of the cost-minimal solution of the DCOP that is consistent with these values for the zero heuristics (no heuristics were added), in this case the cost of the constraint between vertices x_1 and x_2 (= 8). This value is the initial lower bound of vertex x_2 for value one if its ancestor x_1 has value zero. Thus, the lower bounds of a vertex label the children of the vertex in the search tree. To simplify our description, we assume that ADOPT performs a synchronous rather than an asynchronous search and propagates information with infinite speed. Figure 3 shows

the resulting search. Consider the root vertex x_1. ADOPT initializes its threshold (th) with zero, its upper bound (ub) with infinity, and its lower bounds with the initial lower bounds shown earlier. Both of its lower bounds are equal to zero. Thus, it breaks ties and takes on value zero (Step 1). The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 0$. The best value of vertex x_2 is zero and its best lower bound is five. Thus, vertex x_1 can update its lower bound for value zero from zero to five (Step 2). The best value of vertex x_1 is now one and its best lower bound (shown inside the root node of the search tree) is zero. Vertex x_1 then switches to value one. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 1$. The best value of vertex x_2 is one and its best lower bound is three. Thus, vertex x_1 can update its lower bound for value one from zero to three. This violates the threshold invariant, and thus the threshold is also increased to three. Its best value, however, remains unchanged. Vertex x_2 thus takes on value one. The lower bounds of vertex x_3 are initialized with the initial lower bounds shown earlier for $x_1 := 1$ and $x_2 := 1$. The best value of vertex x_3 is one and its best lower bound is nine. Thus, vertex x_2 can update its lower bound for value one from three to nine. Then, vertex x_1 can update its lower bound for value one from three to nine (Step 4). The best value of vertex x_1 is now zero and its best lower bound is five. Vertex x_1 thus updates its threshold to five and then switches to value zero. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 0$ and the previous lower bounds are purged from memory. (If the ancestors of a vertex switch their values, then the vertex changes its node in the search tree to a different node in its layer. Since each vertex has only a bounded amount of memory, it can store information only for its current node in the search tree. Thus, it has to delete its current lower bounds, as shown in the figure with the X's, and replace them with the initial lower bounds for the new values of its ancestors.) The search continues and eventually reaches the node with $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$ in Step 10. This is a solution with cost nine. Thus, vertex x_1 updates its upper bound to nine, the termination condition is satisfied, and ADOPT terminates with the cost-minimal solution $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$.

It is interesting to see that the behavior of ADOPT is similar to that of Korf's recursive best-first search (RBFS) [6], which ADOPT generalizes to the asynchronous and distributed case. For example, ADOPT does not need centralized control and is able to take advantage of parallel computations in case it operates on constraint trees that are not chains. ADOPT and RBFS operate under the same memory limitations. They both perform best-first searches and use depth-first searches to redo previous best-first searches in order to restore information already purged from memory. Vertex x_1 in the example switches back and forth between values zero and one, and then has to redo the previous searches. For example, the best value of vertex x_1 is one and its best lower bound is nine in Step 7. Vertex x_1 thus switches to value one. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 1$ (namely, 20 and 3), but were already larger at the end of the previous search with $x_1 := 1$ in Step 4 (namely, 20 and 9). ADOPT uses a depth-first search to restore them in Steps 9-10, which is similar to what RBFS does in this situation.

```
procedure IDB-Adopt()
{01} threshold := a large integer;
{02} loop
{03}    set the threshold of the root vertex to threshold;
{04}    run the original ADOPT algorithm;
{05}    if (solution quality found > threshold)
{06}       return solution found;
{07}    end if;
{08}    threshold := solution quality found - 1;
{09} end loop;
```

Fig. 4. IDB-ADOPT

5 IDB-ADOPT

A best-first search without memory limitations visits only the necessary nodes in the search tree to find the optimal solution [5]. However, ADOPT performs a best-first search where each vertex has only a bounded amount of memory and thus has to redo many searches. To remedy this situations, we make the following observation about ADOPT: If the cost of the cost-minimal solution is less than or equal to the threshold of the root vertex, then ADOPT performs a depth-first search and terminates after finding the first solution whose cost is less than or equal to the threshold.

> **Explanation:** When the initial threshold of the root vertex is smaller than the initial upper bound of the root vertex but larger than or equal to the cost of the cost-minimal solution (which implies that it is also larger than or equal to the best lower bound of the root vertex), ADOPT performs a depth-first search. The upper bound of the root vertex is the cost of a cost-minimal solution found so far. If the upper bound is larger than the threshold, then the depth-first search continues. Once the upper bound is smaller than or equal to the threshold, then the threshold gets set to the upper bound and the termination condition of ADOPT is satisfied. Thus, once the depth-first search finds a solution with a cost that is smaller than or equal to the threshold, ADOPT terminates with that solution.

Our objective is to make ADOPT faster by only modifying it slightly based on the above observation. We introduce Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing a best-first search to performing a series of depth-first searches. It assumes that the constraint costs are non-negative integers. Thus, if there is no solution of integer cost x or smaller, then the cost-minimal solution must have a cost of $x+1$ or larger. Figure 4 shows the pseudo code of IDB-ADOPT, which uses ADOPT to implement the depth-first searches. IDB-ADOPT sets the threshold of the root vertex to a large integer, that is, an integer larger than or equal to the cost of a cost-minimal solution. Such an integer can easily be obtained by summing the largest possible cost of each constraint over all constraints. IDB-ADOPT then runs ADOPT. According to the above observation, ADOPT terminates with a solution whose cost is less than or equal to the threshold if

such a solution exists, which is the case since the threshold is larger than or equal to the cost of a cost-minimal solution. IDB-ADOPT then sets the threshold of the root vertex to the cost of the solution minus one and runs ADOPT again. This process continues until ADOPT terminates with a solution whose cost is larger than the threshold. This solution is a cost-minimal solution.

> **Explanation:** We use x to refer to the threshold of the root vertex of the constraint tree at the beginning of the last search of ADOPT. Note that the previous (second-to-last) search of ADOPT has already found a cost-minimal solution of cost $x+1$. The last search of ADOPT only verifies that the solution is indeed cost-minimal. It behaves as follows: ADOPT performs a depth-first search until all lower bounds of the root vertex of the constraint tree are larger than x. At this point in time, at least one of its lower bounds is smaller than or equal to the cost of a cost-minimal solution and thus equal to $x+1$. ADOPT either has not found a solution of cost $x+1$ yet or has found such a solution already. In the first case, the root vertex takes on its best value whose lower bound is, as argued above, $x+1$ and performs a depth-first search until it either finds a solution with that cost or increases the lower bound of that value and then repeats the process with a different value whose lower bound is $x+1$. (In this case, it redoes one search for each value that it revisits. However, it cannot revisit any value more than once since it will find a solution with cost $x+1$ during one of the searches and thus will not take on values whose lower bounds are larger than $x+1$. Notice that this property is not guaranteed for initial thresholds of the root node of the constraint tree that are smaller than x, including the zero value used by ADOPT.) Finally, it finds a solution with cost $x+1$ since one exists. Its upper bound is then set to $x+1$. In the second case, its upper bound is already equal to $x+1$. Either way, its best lower bound is now equal to its upper bound and its threshold is always between the two. Thus, its threshold is now equal to its upper bound and the termination condition of ADOPT is satisfied. ADOPT then terminates with a solution with cost $x+1$, which must be a cost-minimal solution since the best lower bound of the root vertex of the constraint tree is equal to its upper bound.

Thus, IDB-ADOPT is, like ADOPT, an asynchronous and distributed search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. IDB-ADOPT checks whether the cost of the solution found by ADOPT is larger than the threshold. If so, it returns this solution, which is a cost-minimal solution. Otherwise, it runs ADOPT again (from scratch) with a new threshold. Since this threshold is reduced from one ADOPT search to the next, ADOPT finds solutions of smaller and smaller costs until it eventually finds the cost-minimal solution. Thus, IDB-ADOPT can be used as an anytime algorithm [16].

6 Illustration of IDB-ADOPT

Figure 5 shows the searches of IDB-ADOPT for our example DCOP. Consider the root vertex x_1. IDB-ADOPT initializes its threshold with 60 (the sum of the largest possible

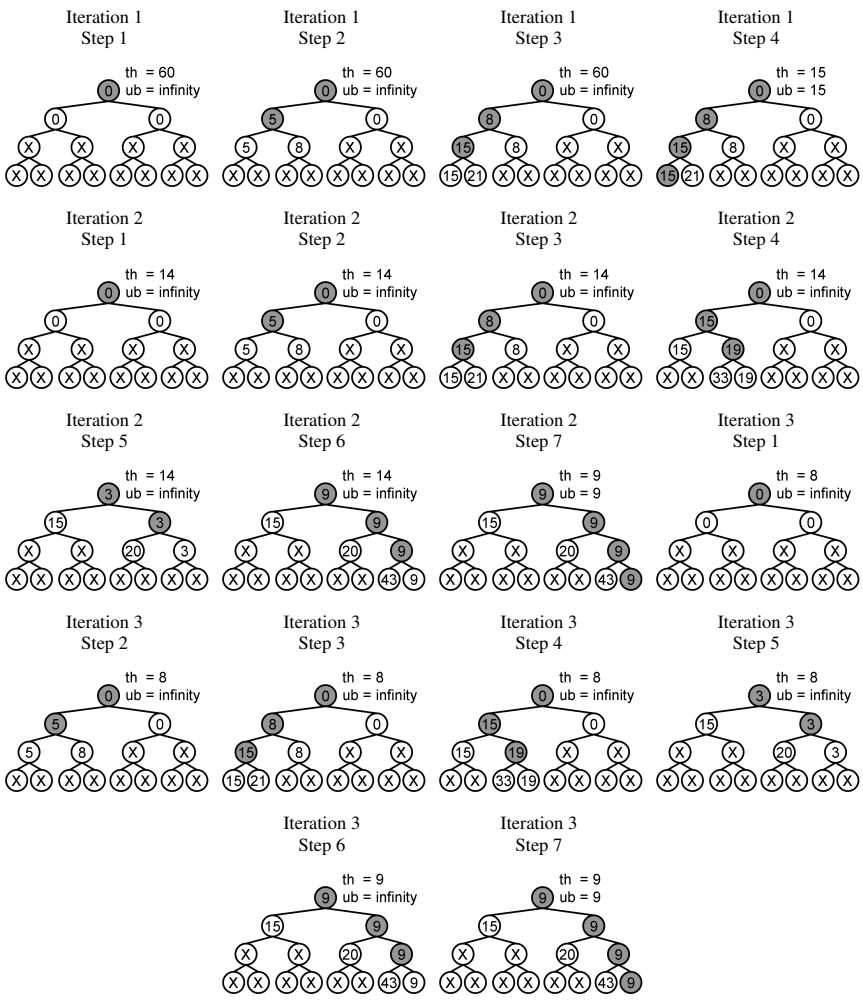


Fig. 5. Illustration of IDB-ADOPT

cost of each constraint over all constraints, which is guaranteed to be larger than or equal to the cost of a cost-minimal solution), its upper bound with infinity, and its lower bounds with the initial lower bounds shown earlier. IDB-ADOPT then starts the first ADOPT search. Both of its lower bounds are equal to zero. Thus, it breaks ties and takes on value zero (Iteration 1, Step 1). Now consider vertex x_2. Its lower bounds are initialized with the initial lower bounds shown earlier for $x_1 := 0$. The best value of vertex x_2 is zero and its best lower bound is five. Thus, vertex x_1 can update its lower bound for value zero from zero to five. The best value of vertex x_1 is now one and its best lower bound is zero. However, vertex x_1 does *not* change its value since the lower bound of its current value remains below the threshold. Vertex x_2 thus takes on value

zero. The lower bounds of vertex x_3 are initialized with the initial lower bounds shown earlier for $x_1 := 0$ and $x_2 := 0$. The best value of vertex x_3 is zero and its best lower bound is fifteen (Iteration 1, Step 3). Thus, vertex x_2 can update its lower bound for value zero from five to fifteen. The best value of vertex x_2 is now one and its best lower bound is eight. However, vertex x_2 does not change its value since the lower bound of its current value remains below the threshold (Iteration 1, Step 2). Thus, the search has reached the node with $x_1 := 0$, $x_2 := 0$ and $x_3 := 0$ in Iteration 1, Step 4. This is a solution with cost fifteen. Thus, vertex x_1 updates first its upper bound to fifteen and then also its threshold to fifteen. The termination condition is satisfied, and the ADOPT search terminates with the solution $x_1 := 0$, $x_2 := 0$ and $x_3 := 0$. Consider again the root vertex x_1. IDB-ADOPT then initializes its threshold with fourteen, its upper bound with infinity, and its lower bounds with the initial lower bounds shown earlier. IDB-ADOPT then starts the second ADOPT search, and so on. Three observations are important here: First, each ADOPT search performs a depth-first search and backtracks only when the lower bound of a value is larger than the threshold. For example, vertex x_2 switches from value zero to value one in Iteration 2, Steps 3-4 because the lower bound of its value zero has become larger than the threshold. No vertex switches back during an ADOPT search to a previous value unless its ancestors have switched values. Second, different ADOPT searches do repeat some of the effort. All three ADOPT searches, for example, consider the case where $x_1 := 0$ and $x_2 := 0$. Finally, the second ADOPT search already found the cost-minimal solution but the third ADOPT search is needed to verify that it is indeed cost-minimal. Note that our example DCOP is too small for IDB-ADOPT to run faster than ADOPT, which we will explain below.

7 ADOPT versus IDB-ADOPT

ADOPT and IDB-ADOPT compare as follows: IDB-ADOPT performs repeated ADOPT searches that produce better and better solutions. Each ADOPT search performed by IDB-ADOPT has the property that vertices do not switch back and forth between different values, unless their connected ancestors have switched values, and thus does not incur the overhead of redoing previous searches. In fact, IDB-ADOPT redoes no search within an ADOPT search (except for the last one). It achieves this efficiency by performing depth-first searches rather than best-first searches. However, depth-first searches are sources of a different inefficiency since they explore partial solutions that best-first searches do not explore. Thus, there is a trade-off between using a best-first search and having to explore partial solutions repeatedly, and using a depth-first search and having to explore additional (unimportant) partial solutions. We expect a best-first search to do better if the heuristics (that are used to initialize the lower bounds) are good and it thus does not have to redo many searches. We expect a depth-first search to do better if the heuristics are misleading, for example, if they are uninformed or the DCOPs are large. Our experimental results for graph coloring problems indeed show that IDB-ADOPT runs faster than ADOPT on large DCOPs. Note, however, that it was our objective to modify ADOPT only slightly. Indeed, IDB-ADOPT modifies ADOPT by putting a control loop on top of ADOPT that is only 9 lines long and calls it repeatedly with different thresholds for the root vertex of the constraint tree. This slight

modification, however, does not implement the principle of a depth-first search fully. In fact, IDB-ADOPT needed to perform only a single complete branch-and-bound depth-first search and return the cost-minimal solution found. Instead, IDB-ADOPT performs repeated depth-first searches, each of which repeats parts of the previous depth-first searches, which results in additional (unnecessary) overhead because IDB-ADOPT partially redoes searches from one ADOPT search to the next. Every vertex takes on all of its values that are smaller than or equal to the threshold, unless the ADOPT search terminated before that. Since the threshold of the previous ADOPT search was larger, the vertex has taken on these values already during the previous ADOPT search, unless the previous ADOPT search terminated before that. (The previous ADOPT search terminated earlier than the current one since the threshold of the previous ADOPT search was smaller than the one of the current ADOPT search.) Thus, the current ADOPT search can prune more than the previous ADOPT search but needs to search beyond the solution found by the previous ADOPT search. Our experimental results show that IDB-ADOPT still runs faster than ADOPT on large DCOPs in spite of this overhead. An asynchronous and distributed branch-and-bound depth-first search algorithm would run even faster than IDB-ADOPT. It would share with ADOPT and IDB-ADOPT that it only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. It is future work to develop such an algorithm.

8 Experiments

We evaluated IDB-ADOPT against ADOPT with uninformed heuristics (zero heuristics) and the currently best-known informed heuristics (dp2 heuristics) [1] on graph-coloring problems. Their number of vertices varied from 5 to 10. Their constraint costs were in the range from one to an upper bound that varied from 3 over 10, 25, 50, 100 to 10000. We randomly generated 500 graph-coloring problems with three values per vertex and an average link density of four for each configuration of these two parameters.

In Experiment 1, we measured the average number of cycles needed by IDB-ADOPT and ADOPT for finding optimal solutions for graph-coloring problems with constraint costs ranging from 1 to 10000, as shown in Figure 6. (The number of cycles is a measure of the runtime that takes into account that the vertices can process information in parallel [10]. A smaller number of cycles implies a smaller runtime.) Heuristics speed up both IDB-ADOPT and ADOPT but the number of cycles needed by IDB-ADOPT with uninformed heuristics is already smaller than the one needed by ADOPT with informed heuristics. The speed up of informed IDB-ADOPT over informed ADOPT tends to increase with the number of vertices, as shown in Figure 7. IDB-ADOPT is 88.7 percent faster than ADOPT when the number of vertices is 10. That is, IDB-ADOPT speeds up ADOPT by a factor of about 9 in this case, which is about one order of magnitude.

In Experiment 2, we measured the speed up of informed IDB-ADOPT over informed ADOPT for finding optimal solutions for graph-coloring problems with 10 vertices. The speed up tends to increase with the range of constraint costs, as shown in Figure 8. IDB-ADOPT is 36.0 percent slower than ADOPT when the constraint costs range from 1 to 3, but 88.7 percent faster than ADOPT when the constraint costs range

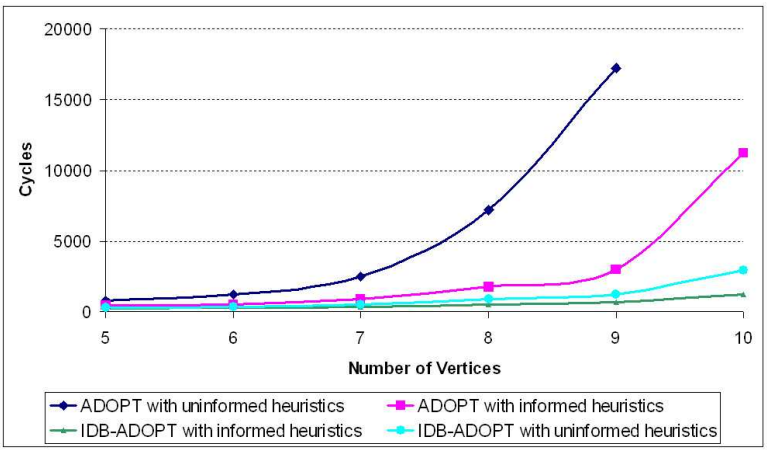


Fig. 6. Experiment 1

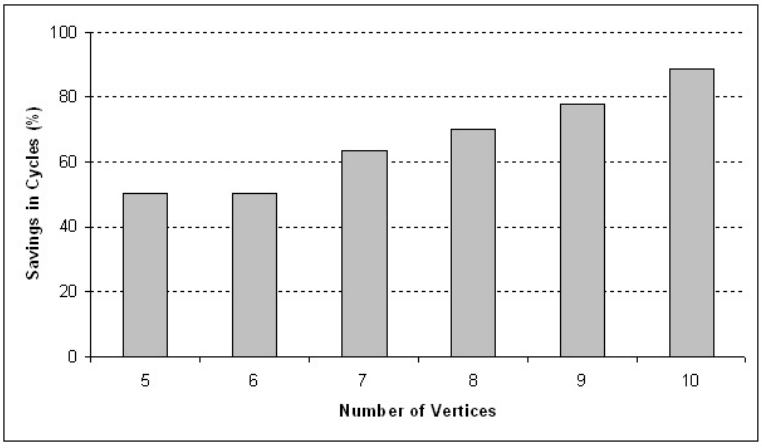


Fig. 7. Speed Ups for Experiment 1

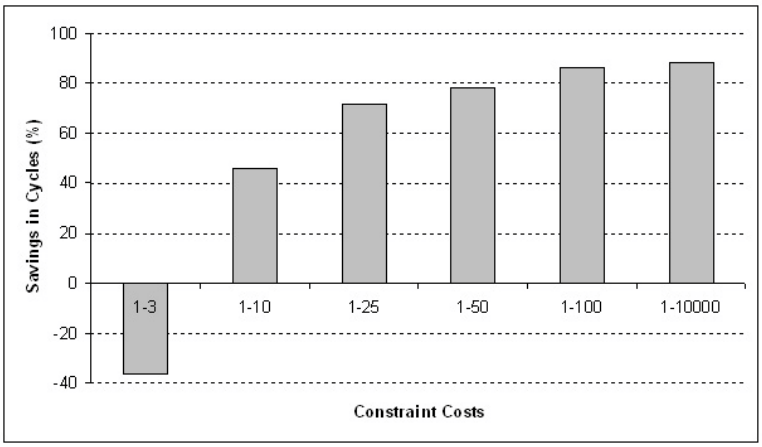


Fig. 8. Speed Ups for Experiment 2

from 1 to 10000 and seems to converge to about this value. The larger the ranges of constraint costs, the more complex the DCOPs and the more misleading the heuristics tend to be, which explains the results.

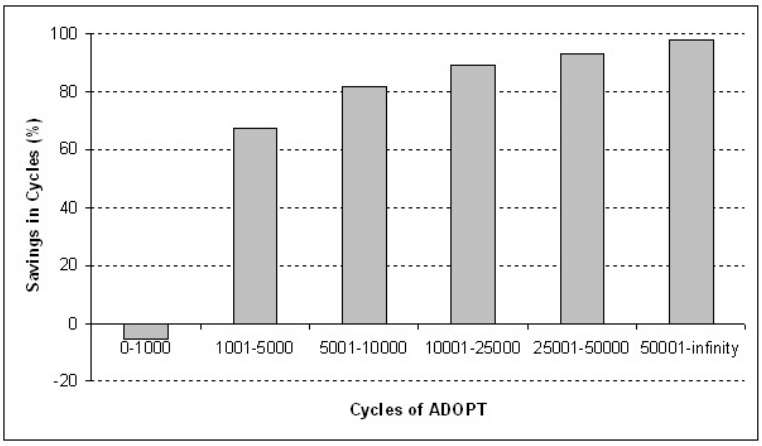


Fig. 9. Speed Ups for Experiment 3

In Experiment 3, we measured the speed up of informed IDB-ADOPT over informed ADOPT for finding optimal solution for graph-coloring problems with 10 vertices and constraint costs ranging from 1 to 10000. We classified them into buckets depending on how many cycles ADOPT needed to solve them: 0-1000, 1001-5000, 5001-10000, 10001-25000, 25001-50000 and 50001-∞ cycles. The speed up tends to in-

crease with the number of cycles ADOPT needed, as shown in Figure 9. IDB-ADOPT is 5.8 percent slower than ADOPT in the bucket 0-1000, but 97.8 percent faster than ADOPT in the bucket 50001-∞ and seems to converge to about this value. Again, the more cycles ADOPT needs, the more complex the DCOPs and the more misleading the heuristics tend to be, which explains the results.

9 Conclusions

In this paper, we introduced Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches. IDB-ADOPT is, like ADOPT, an asynchronous and distributed search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. Our experimental results for graph coloring problems showed that IDB-ADOPT has smaller cycle counts than ADOPT on large DCOPs, with savings of up to one order of magnitude. In addition, IDB-ADOPT produces suboptimal solutions quickly and then improves them. It can thus be used as an anytime algorithm.

Acknowledgments

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 author 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.

References

1. S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the dcop algorithm adopt. In *AAMAS*, pages 1041–1048, 2005.
2. E. Bowring, M. Tambe, and M. Yokoo. Multiply-constrained distributed constraint optimization. In *AAMAS*, pages 1413–1420, 2006.
3. A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS*, pages 1427–1429, 2006.
4. J. Davin and J. Modi. Hierarchical variable ordering for multiagent agreement problems. In *AAMAS*, pages 1433–1435, 2006.
5. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, 32(3):505–536, 1985.
6. R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
7. V. Lesser, C. Ortiz, and M. Tambe, editors. *Distributed sensor networks: A multiagent perspective*. Kluwer, 2003.
8. 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 *AAMAS*, pages 310–317, 2004.
9. R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, pages 438–445, 2004.

10. P. Modi, W. Shen, M. Tambe, and M. Yokoo. ADOPT: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
11. F. Pecora, J. Modi, and P. Scerri. Reasoning about and dynamically posting n-ary constraints in adopt. In *DCR*, 2006.
12. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, pages 1413–1420, 2005.
13. P. Scerri, J. Modi, M. Tambe, and W. Shen. Are multiagent algorithms relevant for real hardware? A case study of distributed constraint algorithms. In *ACM Symposium on Applied Computing*, pages 38–44, 2003.
14. N. Schurr, S. Okamoto, R. Maheswaran, P. Scerri, and M. Tambe. Evolution of a teamwork model. In R. Sun, editor, *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 307–327. Cambridge University Press, 2005.
15. W. Zhang and R. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241–292, 1995.
16. S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Department, University of California at Berkeley, Berkeley (California), 1993.