ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees

Pragnesh Jay Modi MODI@ISI.EDU
Wei-Min Shen SHEN@ISI.EDU
Milind Tambe TAMBE@USC.EDU
Information Sciences Institute and Computer Science Department
University of Southern California
Marina del Rey, CA 90292, USA

Makoto Yokoo YOKOO@CSLAB.KECL.NTT.CO.JP
NTT Communication Science Labs
2-4 Hikaridai, Seika-cho
Soraku-gun, Kyoto 619-0237 Japan

Abstract

The Distributed Constraint Optimization Problem (DCOP) is able to model a wide variety of distributed reasoning tasks that arise in multiagent systems. Unfortunately, existing methods for DCOP are not able to provide theoretical guarantees on global solution quality while allowing agents to operate asynchronously. We show how this failure can be remedied by allowing agents to make local decisions based on conservative cost estimates rather than relying on global certainty as previous approaches have done. This novel approach results in a polynomial-space algorithm for DCOP named *Adopt* that is guaranteed to find the globally optimal solution while allowing agents to execute asynchronously and in parallel. Detailed experimental results show that on benchmark problems Adopt obtains speedups of several orders of magnitude over other approaches. Adopt can also perform *bounded-error approximation* – it has the ability to quickly find approximate solutions and, unlike heuristic search methods, still maintain a theoretical guarantee on solution quality.

1. Introduction

A large class of multiagent coordination problems such as distributed planning, distributed scheduling, distributed resource allocation and others, can be modelled via the Distributed Constraint Optimization Problem (DCOP) [10], [11], [16], [23]. Satellite constellations [2], disaster rescue [12], multiagent teamwork [28], human/agent organizations [4], intelligent forces [3], distributed and reconfigurable robots [25] and sensor networks [27] are some examples of multiagent applications where distributed reasoning problems arise. DCOP provides a useful framework for investigating how agents can coordinate their decision-making in such domains.

A DCOP includes a set of variables, each variable is assigned to an agent who has control of its value, and agents must coordinate their choice of values so that a global objective function is optimized. The global objective function is modeled as a set of constraints, and each agent knows about the constraints in which its variables are involved. In this paper, we model the global objective function as a set of *valued* constraints, that is, constraints that are described as functions that return a range of values, rather than predicates that return only true or false. DCOP significantly generalizes the Distributed Constraint Satisfaction Problem (DisCSP) framework [18] [26] [29] in which problem solutions are characterized with a designation of "satisfactory or unsatisfactory" and so do not model problems where solutions have degrees of quality or cost.

DCOP demands techniques that go beyond existing methods for finding distributed satisfactory solutions and their simple extensions for optimization. We argue that a DCOP method for the types of real-world applications previously mentioned must meet three key requirements. First, since the domains are inherently distributed, we require a method where agents can optimize a global function in a distributed fashion using only local communication (communication with neighboring agents). Methods where all agents must communicate with a single central agent are unacceptable. Second, we require a method that is able to find solutions quickly by allowing agents to operate asynchronously. A synchronous method where an agent sits idle while waiting for a particular message from a particular agent is unacceptable because it is wasting time when it could potentially be doing useful work. For example, Figure 1 shows groups of loosely connected agent subcommunities which could potentially execute search in parallel rather than sitting idle. Finally, provable quality guarantees on system performance are needed. For example, mission failure by a satellite constellation performing space exploration can result in extraordinary monetary and scientific losses. Thus, we require a method that not only efficiently finds provably optimal solutions whenever possible but also allows principled solution-quality/computation-time tradeoffs when time is limited.

A solution strategy that is able to provide quality guarantees, while at the same time meeting the requirements of distribution and asynchrony, is currently missing from the research literature. In previous work Yokoo, Durfee, Ishida, and Kuwabara have developed the Asynchronous Backtracking (ABT) algorithm for DisCSP [29] [30] but this algorithm is limited to satisfaction-based problems. Simple extensions to ABT for optimization have relied on converting an optimization problem into a sequence of DisCSPs using iterative thresholding [11]. This approach has applied only to limited types of optimization problems (e.g. Hierarchical DisCSPs, Maximal DisCSPs), but has failed to apply to more general DCOP problems, even rather natural ones such as minimizing the total number of constraint violations (MaxCSP). Another existing algorithm that can provide quality guarantees for optimization problems, the Synchronous Branch and Bound (SynchBB) algorithm [10] discussed later, is prohibitively slow since it requires synchronous, sequential communication. Other fast, asynchronous solutions, such as variants of local search [10] [31], cannot provide guarantees on the quality of the solutions they find.

As we can see from the above, one of the main obstacles for solving DCOP is combining quality guarantees with asynchrony. Previous approaches have failed to provide quality guarantees in DCOP using a distributed, asynchronous model because it is difficult to ensure a systematic backtrack search when agents are asynchronously changing their variable values. We argue that the main reason behind these failures is that previous approaches insist on backtracking *only* when they conclude, with certainty, that the current solution will not lead to the optimal solution. For example, SynchBB [10] is an algorithm for DCOP where an agent concludes with certainty that the current partial solution will not lead to

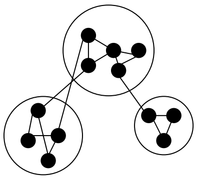

Figure 1: Loosely connected subcommunities of problem solvers

a globally optimal solution by comparing cost with a global upper bound. This approach to DCOP fails to be asynchronous and parallel because computing a global upper bound requires that all costs in the constraint network be accumulated within a single agent before decisions can be made. An alternative approach to DCOP relies on repeated application of a DisCSP algorithm like ABT. An agent executing the ABT algorithm concludes with certainty that the current partial solution being explored will not lead to a global satisfactory solution whenever it locally detects an unsatisfiable constraint. This approach fails to generalize to DCOP because it relies on the limited representation of DisCSP, where only one constraint needs to be broken for a candidate solution to be globally inconsistent.

To solve this challenging problem, we propose a new distributed constraint optimization algorithm, called *Adopt* (Asynchronous Distributed OPTimization)[1]. Adopt, to the best of our knowledge, is the first algorithm for DCOP that can find the optimal solution, or a solution within a user-specified distance from the optimal, using only localized asynchronous communication and polynomial space at each agent. Communication is local in the sense that an agent does not send messages to every other agent, but only to neighboring agents. The main idea behind Adopt is to obtain asynchrony by allowing each agent to change variable value whenever it detects there is a *possibility* that some other solution may be better than the one currently under investigation. This search strategy allows asynchronous computation because an agent does not need global information to make its local decisions – it can go ahead and begin making decisions with only local information. Because this search strategy allows partial solutions to be abandoned before suboptimality is proved, partial solutions may need to be revisited. The second key idea in Adopt is to efficiently reconstruct previously considered partial solutions (using only polynomial space) through the use of *backtrack threshold* – an allowance on solution cost that prevents backtracking. We will show in this paper that these two key ideas together yield efficient asynchronous search for optimal solutions. Finally, the third key idea in Adopt is to provide a termination detection mechanism built into the algorithm – agents terminate whenever they find a complete solution whose cost is under their current backtrack threshold. Previous asynchronous search algorithms have typically required a termination detection algorithm to be invoked separately, which can be problematic since it requires additional message passing.

Adopt's ability to provide quality guarantees and built-in termination detection naturally leads to a practical technique for bounded-error approximation. A bounded-error approximation algorithm is guaranteed to deliver a solution whose quality is within a user-specified distance from the optimal, and usually in much less time than is required to deliver the optimal solution. Finding the optimal solution to a DCOP can be very costly for some problems where sufficient resources (e.g. time) may not be available. Therefore, bounded-error approximation is a crucial capability needed for making effective solution-

1. Additional details may also be found in the first author's PhD thesis [22]. This article is an extension of an earlier conference paper [20]. Additional exposition, examples and experiments are presented here.

quality/computation-time tradeoffs in the real world. Approaches that use incomplete search to find solutions quickly have thus far lacked the capability of providing a theoretical guarantee on solution quality.

Our evaluation results show that Adopt obtains several orders of magnitude speed-up over SynchBB, the only existing complete algorithm for DCOP. The speedups are shown to be partly due to the novel search strategy and partly due to the asynchrony and parallelism allowed by the search strategy. Also, although distributed constraint optimization is intractable in the worst case, our experiments demonstrate that some classes of problems exhibit special properties in which optimal algorithms can perform very well. In particular, Adopt is able to guarantee optimality at low cost for large problems when the constraint network is sparse – a typical feature of distributed sensor networks [19]. We also present empirical results demonstrating an important feature of the algorithm, namely, the ability to perform bounded-error approximation. We present experimental results demonstrating that time-to-solution decreases as the given error-bound is allowed to increase.

2. Problem Definition

A Distributed Constraint Optimization Problem (DCOP) consists of n variables $V = \{x_1, x_2, ...x_n\}$, each assigned to an agent, where the values of the variables are taken from finite, discrete domains $D_1, D_2, ..., D_n$, respectively. Only the agent who is assigned a variable has control of its value and knowledge of its domain. The goal for the agents is to choose values for variables such that a given global *objective function* is minimized. The objective function is described as the summation over a set of *cost functions*. A cost function for a pair of variables x_i, x_j is defined as $f_{ij} : D_i \times D_j \rightarrow N$. The cost functions in DCOP are the analogue of constraints from DisCSP and are sometimes referred to as "valued" or "soft" constraints. For convenience in this paper, we will refer to cost functions simply as constraints. Figure 2.a shows an example DCOP with four agents where each has a single variable with domain $\{0, 1\}$. Two agents x_i, x_j are *neighbors* if they have a constraint between them. In Figure 2.a, x_1 and x_3 are neighbors but x_1 and x_4 are not. All four constraints are identical in this example but this is not required.

The objective is to find an assignment $\mathcal{A}^*$ of values to variables such that the aggregate cost F is minimized. Stated formally, we wish to find $\mathcal{A}$ $(= \mathcal{A}^*)$ such that $F(\mathcal{A})$ is minimized, where the objective function F is defined as

$$F(\mathcal{A}) = \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j) \quad , where\ x_i \leftarrow d_i, \\ x_j \leftarrow d_j\ in\ \mathcal{A}$$

In Figure 2.a, $F(\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0)\}) = 4$ and $F(\{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}) = 0$. In this example, $\mathcal{A}^* = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}$.

In this paper, we will assume each agent is assigned a single variable and use the terms "agent" and "variable" interchangeably. We will assume that constraints are at most binary, i.e., involve no more than two variables. These assumptions are discussed further in the next section. We assume that message transfer may have random but finite delay and messages are received in the order in which they are sent between any pair of agents. Messages sent from different agents to a single agent may be received in any order.

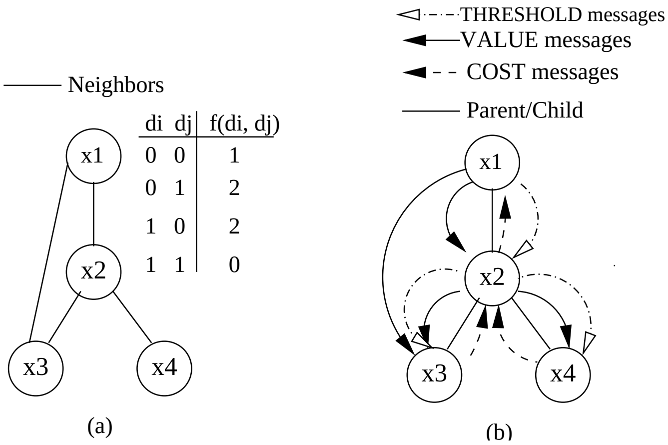

Figure 2: (a) Constraint graph. (b) Adopt communication graph.

2.1 Scope of the Representation

The scope of the DCOP representation and its ability to represent distributed optimization problems can be understood along the three important dimensions discussed next.

Aggregation operator. The properties of the summation operator which is used to aggregate costs from the component constraints imposes some restrictions on the types of global objective functions that can be represented. In particular, the techniques we will present apply only to aggregation operators that are associative, commutative, and monotonic. This class of optimization functions is described formally by Schiex, Fargier and Verfaillie as Valued CSPs [24]. Monotonicity requires that the cost of a solution can only increase as more costs are aggregated. For example, summation over the natural numbers is monotonic but summation over the integers is not. While the monotonicity assumption does limit the applicability of our approach, it also encompasses a broad and useful class of optimization problem and so is not an overly severe limitation.

Arity of component constraints. Our restriction to binary constraints can also impose difficulties on representing some problems. For example, a requirement stating that "2 out of 3 agents" are needed for a task is most naturally represented as a ternary constraint over all three agents rather than as an aggregation of pairwise binary constraints. But we also note that algorithms for DisCSP were first developed assuming binary constraints and later successfully generalized to n-ary constraints. Thus, we take a similar approach for DCOP and first assume binary constraints in this paper and propose extensions for n-ary constraints in future work. Furthermore, we argue that it is desirable to limit the arity of constraints whenever possible since agents' decisions can become very tightly coupled otherwise, which makes problems significantly more difficult to solve in terms of computation and communication.

Number of variables per agent. Our assumption of one variable per agent can be problematic in domains where agents have complex local subproblems that are more appropriately modeled using multiple variables. Yokoo et al. [29] describe some methods for dealing with multiple variables per agent in DisCSP. For example, one can convert a constraint reasoning problem involving multiple variables into a problem with only one variable by defining a new variable whose domain is the cross product of the domains of each of the original variables. Another method is to create multiple virtual agents within a single real agent and assign one local variable to each virtual agent. Both of these approaches allow the use of the techniques presented in this paper to apply when agents have multiple local variables.

2.2 Sensor Network as DCOP

One application that motivates us is a distributed sensor network domain that has gained the attention of a number of multiagent researchers in recent years[15]. The domain consists of multiple fixed directional sensors, each controlled by an autonomous agent, and multiple targets in their sensing range. Figure 3.a shows 9 sensors in a grid configuration. Three sensors must be pointed at a target to track it and no sensor may track more than one target. Agents are able to send messages to each other using low-bandwidth radio-frequency communication. However communication range is limited so agents may only communicate with nearby agents. Assuming each target is within the sensing range of only the four nearest sensors, we see that only two of the four targets can be feasibly tracked. The agents must coordinate in order to track the targets with highest weights as denoted by the number next to each target.

One representation of the sensor domain using DCOP is shown in Figure 3.b. Only the variables and constraints for agents A1 and A2 are shown. This mapping to DCOP attempts to represents the domain in the sense that if agents find values for variables that minimize the sum cost over all the constraints, they will minimize the sum weight of ignored targets. We chose to represent the problem using only unary and binary constraints and avoid higher arity constraints.

The mapping proceeds as follows. We create a variable T_j^i for each target j and for each agent i who could possibly sense that target. The domain of variable T_j^i is the set of agent triples that could track target j. For example, the domain of T_1^i is the set $\{A1A2A3, A1A2A4, A1A3A4, A2A3A4, Ignore\}$. The $Ignore$ value indicates that zero agents are allocated to the target. An equality constraint between two variables T_j^i and T_j^k belonging to agents i and k requires them to agree on which 3 agents will track the target j. A mutual exclusion constraint between two variables T_j^i and T_k^i within agent i requires that the agent not participate in tracking two targets. The cost of violating these two constraints is set to a very high value (Inf) essentially making them "hard" constraints. Finally, an Ignore constraint defined for each target gives the cost for choosing the $Ignore$ value for that target.

3. Basic Ideas

The Adopt algorithm brings forth three new ideas not seen in previous distributed constraint reasoning algorithms: a) a novel asynchronous search strategy where solutions may be abandoned before they are proven suboptimal, b) efficient reconstruction of those abandoned solutions, and c) built-in termination detection. Each idea is discussed next.

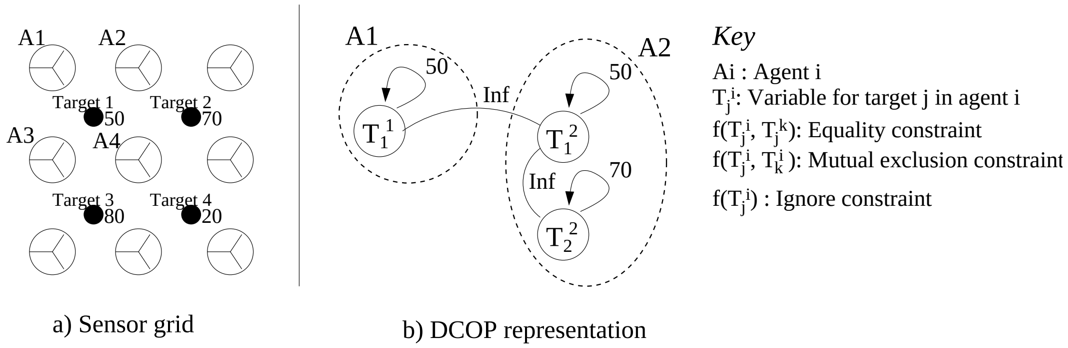

Figure 3: Mapping of sensor network to DCOP

3.1 Opportunistic Best-First Search

Agents are prioritized into a tree structure in which each agent has a single *parent* and multiple *children*. Using this priority ordering, Adopt performs a distributed backtrack search using an "opportunistic" best-first search strategy, i.e., each agent keeps on choosing the best value based on the current available information. Stated differently, each agent always chooses the variable value with smallest lower bound. This search strategy is in contrast to previous distributed "branch and bound" type search algorithms for DCOP (e.g. SynchBB [10]) which require agents to have access to a global upper bound. Adopt's new search strategy is significant because lower bounds are more suitable for asynchronous search – a lower bound can be computed without necessarily having accumulated global cost information. In Adopt, an initial lower bound is immediately computed based only on local cost. The lower bound is then iteratively refined as new cost information is asynchronously received from other agents. Note that because this search strategy allows agents to abandon partial solutions before they have proved the solution is definitely suboptimal, they may be forced to re-explore previously considered solutions. The next idea in Adopt addresses this issue.

3.2 Backtrack Thresholds: Efficiently Reconstructing Abandoned Solutions

To allow agents to efficiently reconstruct a previously explored solution, which is a frequent action due to Adopt's search strategy, Adopt uses the second idea of using a stored lower bound as a *backtrack threshold*. This technique increases efficiency, but requires only polynomial space in the worst case, which is much better than the exponential space that would be required to simply memorize partial solutions in case they need to be revisited. The basic idea behind backtrack thresholds is that when an agent knows from previous search experience that *lb* is a lower bound for its subtree, it should inform the agents in the subtree not to bother searching for a solution whose cost is less than *lb*. In this way, a parent agent determines the value of the backtrack threshold and sends the threshold to its children. Then, the children use the backtrack threshold as an *allowance* on solution cost – a child agent will not change its variable value so long as cost is less than the backtrack threshold given to it by its parent. Since the backtrack threshold is calculated using a previously known lower bound, it is ensured to be less than or equal to the cost of the optimal solution. This ensures that the optimal solution will not be missed.

Using backtrack thresholds to reconstruct previously explored solutions becomes more difficult when an agent has multiple children. In particular, an agent must be able to subdivide backtrack threshold correctly among its multiple children but this is a challenging task because the agent cannot remember how cost was accumulated from its children in the past, at least without requiring exponential space in the worst case. We address this difficulty by allowing the agent to subdivide the threshold arbitrarily and then correct this subdivision over time as cost feedback is received from the children. This is accomplished through a set of program invariants (described in more detail in the next section) that are maintained at each agent. Each agent maintains an **AllocationInvariant** which states that its local cost plus the sum of the thresholds allocated to its children must equal its own backtrack threshold. A **ChildThresholdInvariant** states that no child should be given allowance less than its lower bound. By always maintaining these invariants as cost feedback is received from its children, the parent continually re-balances the subdivision of backtrack threshold among its children until ultimately the correct threshold is given to each child.

3.3 Built-in Termination Detection

Finally, the third key idea is the use of bound intervals for tracking the progress towards the optimal solution, thereby providing a built-in termination detection mechanism. A bound interval consists of both a lower bound and an upper bound on the optimal solution cost. When the size of the bound interval shrinks to zero, i.e., the lower bound equals the upper bound, the cost of the optimal solution has been determined and agents can safely terminate when a solution of this cost is obtained. Most previous distributed search algorithms have required a separate termination detection algorithm. In contrast, the bound intervals in Adopt provide a natural termination detection criterion integrated within the algorithm. This is important because (as we will see in section 6) bound intervals can be used to perform bounded-error approximation. As soon as the bound interval shrinks to a user-specified size, agents can terminate early while guaranteeing they have found a solution whose cost is within the given distance of the optimal solution. This means that agents can find an approximate solution faster than the optimal one but still provide a theoretical guarantee on global solution quality.

4. Asynchronous Search for DCOP

We present the details of the Adopt algorithm for solving DCOP. The procedures shown in Figure 4 and 5 are executed concurrently by each agent. Illustrative examples are also presented in this section.

4.1 Details of Algorithm

As mentioned, agents first are prioritized in a Depth-First Search (DFS) tree which defines parent/child relationships between the agents. The DFS tree ordering is equivalent to the pseudo-tree arrangements described by Freuder and Quinn [8]. For a given constraint graph, a DFS tree is *valid* if there are no constraints between agents in different subtrees of the DFS tree. Constraints are only allowed between an agent and its ancestors or descendants. There are many possible DFS trees for a given constraint graph, and every connected constraint graph can be ordered into some DFS tree. Figure 2.b shows a DFS tree formed from the constraint graph in Figure 2.a – x_1 is the root, x_1 is the parent of x_2, and x_2 is the parent of both x_3 and x_4. We assume parent and children are neighbors. Distributed algorithms for forming DFS trees are presented in[9] [17]. We will assume the DFS ordering is done in a preprocessing step so every agent knows its parent and children.

Variable value assignments (VALUE messages) are sent down the DFS tree while cost feedback (COST messages) percolate back up the DFS tree. It may be useful to view COST messages as a generalization of NOGOOD message from DisCSP algorithms. THRESHOLD messages are used to reduce redundant search and sent only from parent to child. The communication in Adopt is shown in Figure 2.b. VALUE messages are sent down constraint edges – an agent x_i sends VALUE messages only to neighbors lower in the DFS tree and receives VALUE messages only from neighbors higher in the DFS tree. A COST message is sent only from child to parent. A COST message sent from x_i to its parent contains the cost calculated at x_i plus any costs reported to x_i from its children. A THRESHOLD message contains a single number representing a backtrack threshold, initially zero.

Procedures from Adopt are shown in Figure 4 and 5. x_i represents the agent's local variable and d_i represents its current value. Each agent maintains a record of higher priority neighbors' current variable assignments:

- **Definition:** A *context* is a partial solution of the form $\{(x_j,d_j), (x_k,d_k)...\}$. A variable can appear in a context no more than once. Two contexts are *compatible* if they do not disagree on any variable assignment. $CurrentContext$ is a context which holds x_i's view of the assignments of higher neighbors.

A COST message contains three fields: *context*, *lb* and *ub*. The *context* field of a COST message sent from x_l to its parent x_i contains x_l's $CurrentContext$. This field is necessary because calculated costs are dependent on the values of higher variables, so an agent must attach the context under which costs were calculated to every COST message. This is similar to the *context attachment* mechanism in ABT [29]. When x_i receives a COST message from child x_l, and d is the value of x_i in the *context* field, then x_i stores *lb*, indexed by d and x_l, as $lb(d, x_l)$ (line 32). Similarly, the *ub* field is stored as $ub(d, x_l)$ and the *context* field is stored as $context(d, x_l)$ (line 33-34). Before any COST messages are received or whenever contexts become incompatible, i.e., $CurrentContext$ becomes incompatible with $context(d, x_l)$, then $lb(d, x_l)$ is (re)initialized to zero and $ub(d, x_l)$ is (re)initialized to a maximum value Inf (line 3-4, 18-19, 29-30).

x_i calculates cost as local cost plus any cost feedback received from its children. Procedures for calculation of cost ($LB(d)$,$UB(d)$,LB,UB) are not shown in Figure 4 but are given by the following definitions. The *local cost* δ for a particular value choice $d_i \in D_i$, is the sum of costs from constraints between x_i and higher neighbors:

- **Definition:** $\delta(d_i) = \sum_{(x_j,d_j)\in CurrentContext} f_{ij}(d_i, d_j)$ is the *local cost* at x_i, when x_i chooses d_i.

For example, in Figure 2.a, suppose x_3 received messages that x_1 and x_2 currently have assigned the value 0. Then x_3's $CurrentContext$ would be $\{(x_1, 0), (x_2, 0)\}$. If x_3 chooses

0 for itself, it would incur a cost of 1 from $f_{1,3}(0,0)$ (its constraint with x_1) and a cost of 1 from $f_{2,3}(0,0)$ (its constraint with x_2). So x_3's local cost, $\delta(0) = 1 + 1 = 2$.

When x_i receives a COST message, it adds $lb(d, x_l)$ to its local cost $\delta(d)$ to calculate a *lower bound for value d*, denoted $LB(d)$:

- **Definition:** $\forall d \in D_i, LB(d) = \delta(d) + \sum_{x_l \in Children} lb(d, x_l)$ is a *lower bound* for the subtree rooted at x_i, when x_i chooses d.

Similarly, x_i adds $ub(d, x_l)$ to its local cost $\delta(d)$ to calculate an *upper bound for value d*, denoted $UB(d)$:

- **Definition:** $\forall d \in D_i, UB(d) = \delta(d) + \sum_{x_l \in Children} ub(d, x_l)$ is a *upper bound* for the subtree rooted at x_i, when x_i chooses d.

The minimum lower bound over all value choices for x_i is the *lower bound for variable* x_i, denoted LB:

- **Definition:** $LB = \min_{d \in D_i} LB(d)$ is a *lower bound* for the subtree rooted at x_i.

Similarly, the minimum upper bound over all value choices for x_i is the *upper bound for variable* x_i, denoted UB:

- **Definition:** $UB = \min_{d \in D_i} UB(d)$ is an *upper bound* for the subtree rooted at x_i.

x_i sends LB and UB to its parent as the lb and ub fields of a COST message (line 52). Intuitively, $LB = k$ indicates that it is not possible for the sum of the local costs at each agent in the subtree rooted at x_i to be less than k, given that all higher agents have chosen the values in $CurrentContext$. Similarly, $UB = k$ indicates that the optimal cost in the subtree rooted at x_i will be no greater than k, given that all higher agents have chosen the values in $CurrentContext$. Note that a leaf agent has no subtree so $\delta(d) = LB(d) = UB(d)$ for all value choices d and thus, LB is always equal to UB at a leaf. If x_i is not a leaf but has not yet received any COST messages from its children, UB is equal to maximum value Inf and LB is equal to the minimum local cost $\delta(d)$ over all value choices $d \in D_i$.

x_i's backtrack threshold is stored in the *threshold* variable, initialized to zero (line 1). Its value is updated in three ways. First, its value can be increased whenever x_i determines that the cost of the optimal solution within its subtree must be greater than the current value of *threshold*. Second, if x_i determines that the cost of the optimal solution within its subtree must necessarily be less than the current value of $threshold$, it decreases $threshold$. These two updates are performed by comparing $threshold$ to LB and UB (lines 53-56, figure 5). The updating of $threshold$ is summarized by the following invariant.

- **ThresholdInvariant:** $LB \leq threshold \leq UB$. The threshold on cost for the subtree rooted at x_i cannot be less than its lower bound or greater than its upper bound.

A parent is also able to set a child's *threshold* value by sending it a THRESHOLD message. This is the third way in which an agent's $threshold$ value is updated. The reason for this is that in some cases, the parent is able to determine a bound on the optimal cost of a solution within an agent's subtree, but the agent itself may not know this bound. The THRESHOLD message is a way for the parent to inform the agent about this bound.

A parent agent is able to correctly set the *threshold* value of its children by subdividing its own *threshold* value among its children and then using the following two equations to re-balance over time as cost feedback is received from the children. Let $t(d, x_l)$ denote the threshold on cost allocated by parent x_i to child x_l, given x_i chooses value d. Then, the values of $t(d, x_l)$ are subject to the following two invariants.

- **AllocationInvariant:** For current value $d_i \in D_i$, $threshold = \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$. The threshold on cost for x_i must equal the local cost of choosing d plus the sum of the thresholds allocated to x_i's children.

- **ChildThresholdInvariant:** $\forall d \in D_i, \forall x_l \in Children, lb(d, x_l) \leq t(d, x_l) \leq ub(d, x_l)$. The threshold allocated to child x_l by parent x_i cannot be less than the lower bound or greater than the upper bound reported by x_l to x_i.

By adhering to these invariants, an agent is able to use its own *threshold* to determine bounds on the cost of the optimal solution within its childrens' subtrees.

The *threshold* value is used to determine when to change variable value. Whenever $LB(d_i)$ exceeds *threshold*, x_i changes its variable value to one with smaller lower bound (line 40-41). (Such a value necessarily exists since otherwise ThresholdInvariant would be violated.) Note that x_i cannot prove that its current value is definitely suboptimal because it is possible that *threshold* is less than the cost of the optimal solution. However, it changes value to one with smaller cost anyway – thereby realizing the best-first search strategy described in Section 3.1.

4.2 Example of Algorithm Execution

Figure 6 shows an example of algorithm execution for the DCOP shown in figure 2. Line numbers mentioned in the description refer to figures 4 and 5. This example is meant to illustrate the search process and the exchange of VALUE and COST messages. COST messages are labelled in the figures as [LB,UB,CurrentContext]. For simplicity, not every message sent by every agent is shown. In particular, THRESHOLD messages are omitted from the description. (A later example will illustrate how backtrack thresholds are handled.)

All agents begin by concurrently choosing a value for their variable (line 5). For this example, let us assume they all choose value 0. Each agent sends its value to all lower priority neighbors (figure 6.a). Since the algorithm is asynchronous, there are many possible execution paths from here – we describe one possible execution path.

x_2 will receive x_1's VALUE message. In line 15, it will record this value into its $CurrentContext$. In line 21, it will enter the **backTrack** procedure. x_2 computes $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 1+0+0 = 1$ and $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 2+0+0 = 2$. Since $LB(0) < LB(1)$, we have $LB = LB(0) = 1$. x_2 will also compute $UB(0) = \delta(0) + ub(0, x_3) + ub(0, x_4) = 1+Inf+Inf = Inf$ and $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 2+Inf+Inf = Inf$. Thus, $UB = Inf$. In line 38, *threshold* is compared to UB. *threshold* was set to 1 (in order to be equal to LB) in the maintainAllocationInvariant procedure call from line 20. Since $threshold = 1$ is not equal $UB = Inf$, the test fails. The test in line

```
Initialize
    threshold ← 0; CurrentContext ← {}
    forall d ∈ D_i, x_l ∈ Children do
        lb(d, x_l) ← 0; t(d, x_l) ← 0
        ub(d, x_l) ← Inf; context(d, x_l) ← {}; enddo
    d_i ← d that minimizes LB(d)
    backTrack

when received (THRESHOLD, t, context)
    if context compatible with CurrentContext:
        threshold ← t
        maintainThresholdInvariant
        backTrack; endif

when received (TERMINATE, context)
    record TERMINATE received from parent
    CurrentContext ← context
    backTrack

when received (VALUE, (x_j,d_j))
    if TERMINATE not received from parent:
        add (x_j,d_j) to CurrentContext
        forall d ∈ D_i, x_l ∈ Children do
            if context(d, x_l) incompatible with CurrentContext:
                lb(d, x_l) ← 0; t(d, x_l) ← 0
                ub(d, x_l) ← Inf; context(d, x_l) ← {}; endif; enddo
        maintainThresholdInvariant
        backTrack; endif

when received (COST, x_k, context, lb, ub)
    d ← value of x_i in context
    remove (x_i,d) from context
    if TERMINATE not received from parent:
        forall (x_j,d_j) ∈ context and x_j is not my neighbor do
            add (x_j,d_j) to CurrentContext;enddo
        forall d' ∈ D_i, x_l ∈ Children do
            if context(d', x_l) incompatible with CurrentContext:
                lb(d', x_l) ← 0; t(d', x_l) ← 0
                ub(d', x_l) ← Inf; context(d', x_l) ← {};endif;enddo;endif
    if context compatible with CurrentContext:
        lb(d, x_k) ← lb
        ub(d, x_k) ← ub
        context(d, x_k) ← context
        maintainChildThresholdInvariant
        maintainThresholdInvariant; endif
    backTrack

procedure backTrack
    if threshold == UB:
        d_i ← d that minimizes UB(d)
    else if LB(d_i) > threshold:
        d_i ← d that minimizes LB(d);endif
    SEND (VALUE, (x_i, d_i))
            to each lower priority neighbor
    maintainAllocationInvariant
    if threshold == UB:
        if TERMINATE received from parent
        or x_i is root:
            SEND (TERMINATE,
            CurrentContext ∪ {(x_i, d_i)})
                to each child
            Terminate execution; endif;endif
    SEND (COST, x_i, CurrentContext, LB, UB)
        to parent
```

Figure 4: Procedures for receiving messages (Adopt algorithm). Definitions of terms $LB(d)$,$UB(d)$,LB, UB are given in the text.

`procedure maintainThresholdInvariant`
if $threshold < LB$
  $threshold \leftarrow LB$; **endif**
if $threshold > UB$
  $threshold \leftarrow UB$; **endif**

%note: procedure assumes ThresholdInvariant is satisfied
`procedure maintainAllocationInvariant`
while $threshold > \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ **do**
  choose $x_l \in Children$ where $ub(d_i, x_l) > t(d_i, x_l)$
  increment $t(d_i, x_l)$; **enddo**
while $threshold < \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ **do**
  choose $x_l \in Children$ where $t(d_i, x_l) > lb(d_i, x_l)$
  decrement $t(d_i, x_l)$; **enddo**
SEND (**THRESHOLD**, $t(d_i, x_l)$, $CurrentContext$)
    to each child x_l

`procedure maintainChildThresholdInvariant`
forall $d \in D_i, x_l \in Children$ **do**
  while $lb(d, x_l) > t(d, x_l)$ **do**
    increment $t(d, x_l)$; **enddo;endo**
forall $d \in D_i, x_l \in Children$ **do**
  while $t(d, x_l) > ub(d, x_l)$ **do**
    decrement $t(d, x_l)$; **enddo;enddo**

Figure 5: Procedures for updating backtrack thresholds

40 also fails since $LB(0) = 1$ is not greater that $threshold = 1$. Thus, x_2 will stick with its current value $x_2 = 0$. In line 52, x_2 sends the corresponding COST message to x_1 (figure 6.b).

Concurrently with x_2's execution, x_3 will go through a similar execution. x_3 will evaluate its constraints with higher agents and compute $LB(0) = \delta(0) = f_{1,3}(0,0) + f_{2,3}(0,0) = 1 + 1 = 2$. A change of value to $x_3 = 1$ would incur a cost of $LB(1) = \delta(1) = f_{1,3}(0,1) + f_{2,3}(0,1) = 2 + 2 = 4$, so instead x_3 will stick with $x_3 = 0$. x_3 will send a COST message with $LB = UB = 2$, with associated context $\{(x_1, 0), (x_2, 0)\}$, to its parent x_2. x_4 executes similarly (figure 6.b).

Next, x_1 receives x_2's COST message. In line 31, x_1 will test the received context $\{(x_1, 0)\}$ against $CurrentContext$ for compatibility. Since x_1's $CurrentContext$ is empty, the test will pass. (Note that the root never receives VALUE messages, so its $CurrentContext$ is always empty.) The received costs will be stored in lines 32-33 as $lb(0, x_2) = 1$ and $ub(0, x_2) = Inf$. In line 37, execution enters the **backTrack** procedure. x_1 computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 = 0$ and $LB(0) = \delta(0) + lb(0, x_2) = 0 + 1 = 1$. Since $LB(1) < LB(0)$, we have $LB = LB(1) = 0$. Similarly, $UB = Inf$. Since $threshold = 0$ is not equal $UB = Inf$, the test in line 38 fails. The test in line 40 succeeds and x_1 will choose its value d that minimizes $LB(d)$. Thus, x_1 switches value to $x_1 = 1$. It will again send VALUE messages to its linked descendants (figure 6.c).

Next, let us assume that the COST messages sent to x_2 in figure 6.b are delayed. Instead, x_2 receives x_1's VALUE message from figure 6.c. In line 15, x_2 will update its

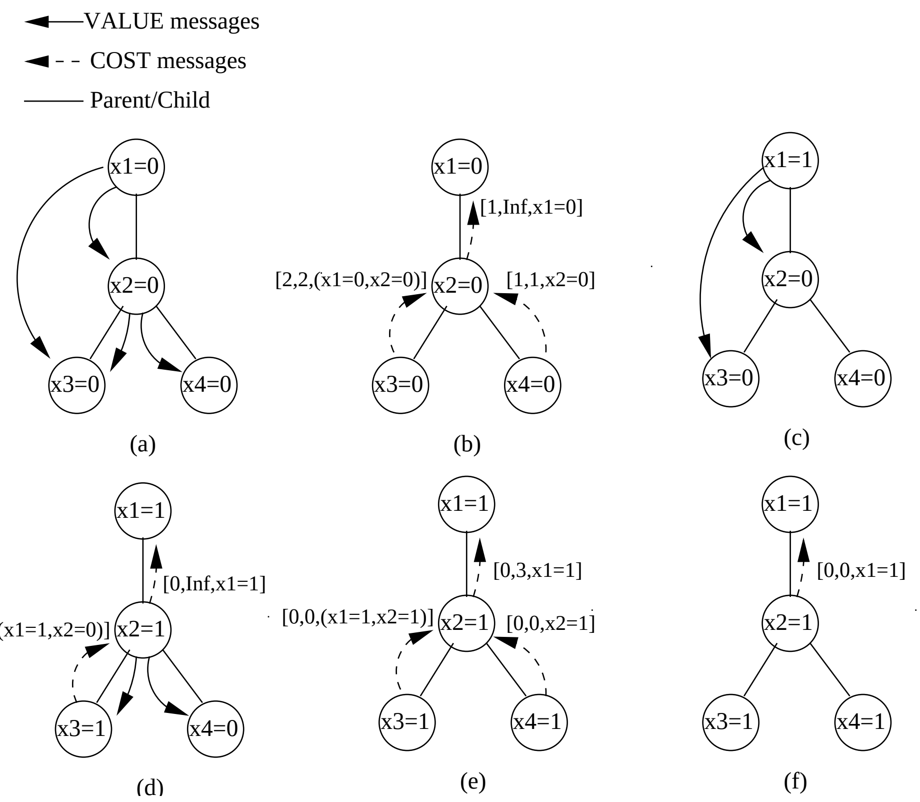

Figure 6: Example Adopt execution for the DCOP shown in figure 2

$CurrentContext$ to $\{(x_1, 1)\}$. For brevity, the remaining portion of this procedure is not described.

Next, x_2 finally receives the COST message sent to it from x_3 in figure 6.b. x_2 will test the received context against $CurrentContext$ and find that they are incompatible because one contains $(x_1, 0)$ while the other contains $(x_1, 1)$ (line 31). Thus, the costs in that COST message will not be stored due to the context change. However, the COST message from x_4 will be stored in lines 32-33 as $lb(0, x_3) = 1$ and $ub(0, x_3) = 1$. In line 37, x_2 then proceeds to the **backTrack** procedure where it will choose its best value. The best value is now $x_2 = 1$ since $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 2 + 0 + 1 = 3$. Figure 6.d shows the change in both x_2 and x_3 values after receiving x_1's VALUE message from figure 6.c. x_2 and x_3 send the new COST messages with the new context where $x_1 = 1$. x_2 also sends VALUE messages to x_3 and x_4 informing them of its new value.

Next, figure 6.e shows the new COST message that is sent by x_2 to x_1 after receiving the COST messages sent from x_3 and x_4 in figure 6.d. Notice that x_2 computes LB as $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and UB as $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 0 + 2 + 1 = 3$. Figure 6.e also shows the new COST message sent by x_3 after receiving x_2's new value of $x_2 = 1$. Similarly, x_4 will change variable value and send a COST message with $LB = 0$ and $UB = 0$. In this way, we see the agents have ultimately settled on the optimal configuration with all values equal to 1 (total cost $= 0$).

Finally in figure 6.f, x_2 receives the COST messages from figure 6.e, computes a new bound interval $LB = 0, UB = 0$ and sends this information to x_1. Upon receipt of this message, x_1 will compute $UB = UB(0) = \delta(0) + ub(0, x_2) = 0 + 0 = 0$. Note that x_1's *threshold* value is also equal to zero. *threshold* was initialized to zero in line 1 and can only be increased if i) a THRESHOLD message is received (line 8), or b) the ThresholdInvariant is violated (line 54, figure 5). The root never receives THRESHOLD messages, so case (i) never occurred. Since x_1's LB was never greater than zero in this example, *threshold* could never have been less than LB, so case (ii) never occurred. Thus, *threshold* was never increased and remains equal to zero. So, we have the test $threshold == UB$ in line 45 evaluate to true. In line 48, it will send a TERMINATE message to x_2, and then x_1 will terminate in line 51. x_2 will receive the TERMINATE message in line 11, evaluate $threshold == UB(= 0)$ to be true in line 45 and then terminate in line 51. The other agents will terminate in a similar manner.

4.3 Example of Backtrack Thresholds

We illustrate how backtrack thresholds are computed, updated and balanced between children. The key difficulty is due to context changes. An agent only stores cost information for the current context. When the context changes, the stored cost information must be deleted (in order to maintain polynomial space). If a previous context is later returned to, the agent no longer has the previous context's detailed cost information available. However, the agent had reported the total sum of costs to its parent, who has that information stored. So, although the precise information about how the costs were accumulated from the children is lost, the total sum is available from the parent. It is precisely this sum that the parent sends to the agent via the THRESHOLD message. The child then heuristically re-subdivides, or allocates, the threshold among its own children. Since this allocation may be incorrect, it then corrects for over-estimates over time as cost feedback is (re)received from the children.

Figure 7 shows a portion of a DFS tree. The constraints are not shown. Line numbers mentioned in the description refer to figure 4 and figure 5. x_p has parent x_q, which is the root, and two children x_i and x_j. For simplicity, assume $D_p = \{d_p\}$ and $\delta(d_p) = 1$, i.e, x_p has only one value in its domain and this value has a local cost of 1.

Suppose x_p receives COST messages containing lower bounds of 4 and 6 from its two children (figure 7.a). The costs reported to x_p are stored as $lb(d_p, x_i) = 4$ and $lb(d_p, x_j) = 6$ (line 32) and associated context as $context(d_p, x_i) = context(d_p, x_j) = \{(x_q, d_q)\}$. LB is computed as $LB = LB(d_p) = \delta(d_p) + lb(d_p, x_i) + lb(d_p, x_j) = 1 + 4 + 6 = 11$. In figure 7.b, the corresponding COST message is sent to parent x_q. After the COST message is sent, suppose a context change occurs at x_p through the receipt of a VALUE message $x_q = d'_q$. In line 18-19, x_p will reset $lb(d_p, x_i)$, $lb(d_p, x_j)$, $t(d_p, x_i)$ and $t(d_p, x_j)$ to zero.

Next, x_q receives the information sent by x_p. x_q will set $lb(d_q, x_p) = 11$ (line 32), and enter the **maintainChildThresholdInvariant** procedure (line 35). Let us assume that $t(d_q, x_p)$ is still zero from initialization. Then, the test in line 65 succeeds since $lb(d_q, x_p) = 11 > t(d_q, x_p) = 0$ and x_q detects that the ChildThresholdInvariant is being violated. In order to correct this, x_q increases $t(d_q, x_p)$ to 11 in line 66.

Next, in figure 7.c, x_q revisits the value d_q and sends the corresponding VALUE message $x_q = d_q$. Note that this solution context has already been explored in the past, but x_p has retained no information about it. However, the parent x_q has retained the sum of the costs, so x_q sends the THRESHOLD message with $t(d_q, x_p) = 11$.

Next, x_p receives the THRESHOLD message. In line 8, the value is stored in the *threshold* variable. Execution proceeds to the **backTrack** procedure where **maintainAllocationInvariant** is invoked in line 44. Notice that the test in line 57 of **maintainAllocationInvariant** evaluates to true since $threshold = 11 > \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1+0+0$. Thus, in lines 57-59, x_p increases the thresholds for its children until the invariant is satisfied. Suppose that the split is $t(d_p, x_i) = 10$ and $t(d_p, x_j) = 0$. This is an arbitrary subdivision that satisfies the AllocationInvariant – there are many other values of $t(d_p, x_i)$ and $t(d_p, x_j)$ that could be used. In line 63, these values are sent via a THRESHOLD message (figure 7.d).

By giving x_i a threshold of 10, x_p risks sub-optimality by overestimating the threshold in that subtree. This is because the best known lower bound in x_i's subtree was only 4. We now show how this arbitrary allocation of threshold can be corrected over time. Agents continue execution until, in figure 7.e, x_p receives a COST message from its right child x_j indicating that the lower bound in that subtree is 6. x_j is guaranteed to send such a message because there can be no solution in that subtree of cost less than 6, as evidenced by the COST message previously sent by x_j in figure 7.a. x_p will set $lb(d_p, x_j) = 6$ (line 32) and enter the **MaintainChildThresholdInvariant** procedure in line 35. Note that the test in line 65 will succeed since $lb(d_p, x_j) = 6 > t(d_p, x_j) = 5$ and the ChildThresholdInvariant is being violated. In order to correct this, x_p increases $t(d_p, x_j)$ to 6 in line 66. Execution returns to line 35 and continues to line 44, where the **maintainAllocationInvariant** is invoked. The test in line 60 of this procedure will succeed since $threshold = 11 < \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1 + 10 + 6 = 17$ and so the AllocationInvariant is being violated. In lines 60-62, x_p lowers $t(d_p, x_i)$ to 4 to satisfy the invariant. In line 63, x_p sends the new (correct) threshold values to its children (figure 7.f).

In this way, a parent agent continually re-balances the threshold given to its independent subtrees in order to avoid overestimating the cost in each subtree and, as discussed in section 3.2, allowing more efficient search.

5. Algorithm Correctness and Complexity

We use three theorems to show that Adopt is guaranteed to terminate with the globally optimal solution. In Theorem 1, we show that the bounds computed by each agent are always correct. In Theorem 2, we show the algorithm will always terminate. Finally, in Theorem 3 we show that the optimal solution is obtained upon termination.

Property 1 is used in the proof of Theorem 1. Let $OPT(x_i, context)$ denote the cost of the optimal solution in the subtree rooted at x_i, given that higher priority variables have values in *context*.

Property 1: $\forall x_i \in V$,

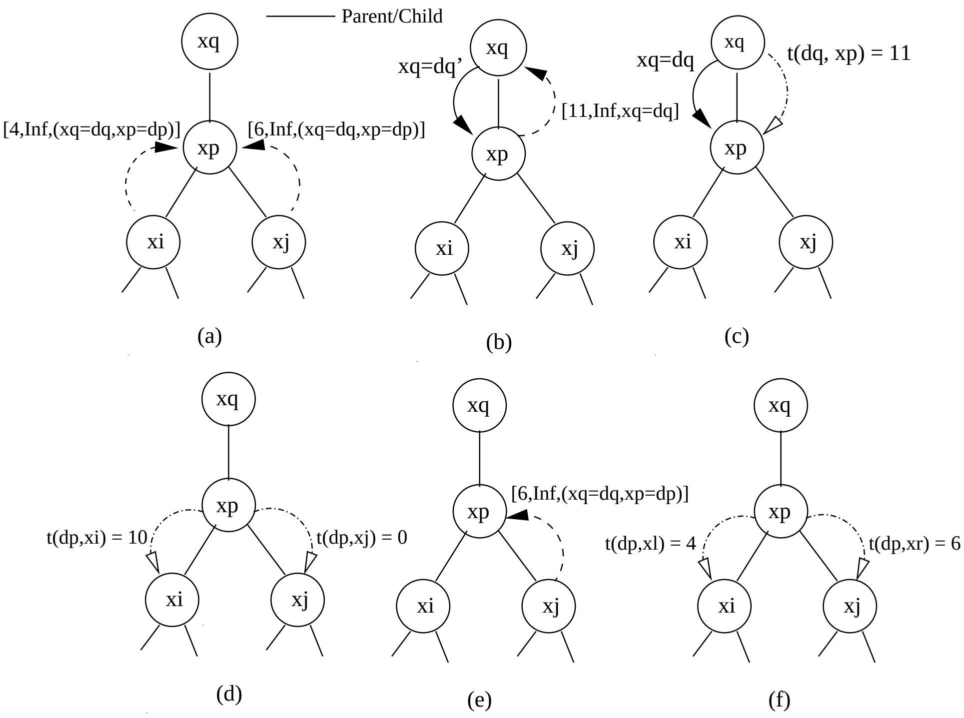

Figure 7: Example of backtrack thresholds in Adopt

$$OPT\quad(x_i, CurrentContext) \stackrel{def}{=} \min_{d \in D_i} \delta(d) +$$
$$\textstyle\sum_{x_l \in Children} OPT(x_l, CurrentContext \cup (x_i, d))$$

For example if x_i is a leaf, then $OPT(x_i, context)$ is equal to $\delta(d)$, where d is the value that minimizes δ. This inductive property states that the cost of the optimal solution in the subtree rooted at x_i is obtained when x_i chooses value d that minimizes the sum of its local cost plus the cost of the optimal solution in its subtree.

Theorem 1 shows that the lower bound LB computed by an agent is *never greater* than the cost of the optimal solution within its subtree, and the upper bound UB is *never less* than the cost of the optimal solution within its subtree. The proof of Theorem 1 proceeds by induction. The base case follows from the fact $LB = OPT(x_i\ , CurrentContext) = UB$ is always true at a leaf agent. The inductive hypothesis assumes that LB (UB) sent by x_i to its parent is never greater (less) than the cost of the optimal solution in the subtree rooted at x_i. The proof also relies on the fact that costs are reported to only one parent so there is no double counting of costs.

Theorem 1: $\forall x_i \in V$,

$$LB \leq OPT(x_i, CurrentContext) \leq UB$$

.

Proof: By induction on agent priorities.

Base Case I: x_i is a leaf. The equations for LB and UB simplify to $\min_{d \in D_i} \delta(d)$ since x_i has no children. Property 1 simplifies to $OPT(x_i, CurrentContext) = \min_{d \in D_i} \delta(d)$ for the same reason. So we conclude $LB = \min_{d \in D_i} \delta(d) = OPT(x_i, CurrentContext) = UB$. Done.

Base Case II: Every child of x_i is a leaf. We will show $LB \leq OPT(x_i, CurrentContext)$. The proof for $OPT(x_i, CurrentContext) \leq UB$ is analogous.

Since all children x_l are leaves, we know from Base Case I that $lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d))$. Furthermore, each child x_l sends COST messages only to x_i, so costs are not double-counted. We substitute $OPT(x_l, CurrentContext \cup (x_i, d))$ for $lb(d, x_l)$ into the definition of LB to get the following:

$$LB = \min_{d \in D_i} \delta(d) + \sum_{x_l \in Children} lb(d, x_l) \leq$$
$$\min_{d \in D_i} \delta(d) + \sum_{x_l \in Children} OPT(x_l, CurrentContext \cup (x_i, d))$$

Now we can simply substitute Property 1 into the above to get

$$LB \leq OPT(x_i, CurrentContext)$$

and we are done.

Inductive Hypothesis: $\forall d \in D_i, \forall x_l \in Children$,

$$lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d)) \leq ub(d, x_l)$$

The proof of the general case is identical to that of Base Case II, except we assume $lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d))$ from the Inductive Hypothesis, rather than from the assumption that x_l is a leaf. □

In Theorem 2, we show that Adopt will eventually terminate. Adopt's termination condition is shown in line 45 of Figure 4. A non-root agent terminates when $threshold = UB$ is true and a TERMINATE message has been received from its parent. The proof follows from the fact that once an agent's parent terminates, LB is monotonically increasing. Since LB is monotonically increasing, and Theorem 1 showed that LB has an upper bound, LB must eventually stop changing. A similar argument is true for UB. Finally, the Threshold-Invariant forces $threshold$ to stay between LB and UB until ultimately $threshold = UB$ occurs.

Theorem 2: $\forall x_i \in V$, if $CurrentContext$ is fixed, then $threshold = UB$ will eventually occur.

Proof: By induction on agent priorities.

Base Case: x_i is a leaf. $LB = UB$ is always true at x_i because it is a leaf. Every agent maintains the ThresholdInvariant $LB \leq threshold \leq UB$. So $threshold = UB$ must always be true at a leaf.

Inductive Hypothesis: If $CurrentContext$ is fixed and x_i fixes its variable value to d_i, then $\forall x_l \in Children$, $threshold = UB$ will eventually occur at x_l and it will report an upper bound ub via a COST message, where $ub = t(d_i, x_l)$.

Assume $CurrentContext$ is fixed. To apply the Inductive Hypothesis, we must show that x_i will eventually fix its variable value. To see this, note that x_i changes its variable value only when $LB(d_i)$ increases. By Theorem 1, LB is always less than the cost of the optimal solution. LB cannot increase forever and so x_i must eventually stop changing its variable value. We can now apply the Inductive Hypothesis which says that when x_i fixes its value, each child will eventually report an upper bound $ub = t(d_i, x_l)$. This means $t(d_i, x_l) = ub(d_i, x_l)$ will eventually be true at x_i. We can substitute $t(d_i, x_l)$ for $ub(d_i, x_l)$ into the definition of UB to get the following:

$$\begin{array}{rcl} UB \overset{def}{\leq} UB(d_i) & \overset{def}{=} & \delta(d_i) + \sum_{x_l \in Children} ub(d_i, x_l) \\ & = & \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l) \end{array}$$

Using the AllocationInvariant $threshold = \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$, we substitute $threshold$ into the above to get $UB \leq threshold$. The right-hand side of the ThresholdInvariant states $threshold \leq UB$. So we have both $UB \leq threshold$ and $threshold \leq UB$. So $threshold = UB$ must be true and the Theorem is proven. □

As an aside, we note that the algorithm behaves differently depending on whether x_i's $threshold$ is set below or above the cost of the optimal solution. If $threshold$ is less than the cost of the optimal solution, then when LB increases above $threshold$, x_i will raise $threshold$ until ultimately, $LB = threshold = UB$ occurs. On the other hand, if $threshold$ is greater than the cost of the optimal solution, then when UB decreases below $threshold$, x_i will lower $threshold$ so $threshold = UB$ occurs. In the second case, LB may remain less than UB at termination.

Theorem 2 is sufficient to show algorithm termination because the root has a fixed (empty) $CurrentContext$ and will therefore immediately terminate when $threshold = UB$ occurs. Before it terminates, it sends a TERMINATE message to its children informing them of its final value (line 48). It is clear to see that when a TERMINATE message is received from the parent, an agent knows that its current context will no longer change since all higher agents have already terminated.

Finally, Theorem 3 shows that the final value of $threshold$ is equal to the cost of the optimal solution.

Theorem 3: $\forall x_i \in V$, x_i's final $threshold$ value is equal to $OPT(x_i, CurrentContext)$.

Base Case: x_i is the root. The root terminates when its (final) $threshold$ value is equal UB. $LB = threshold$ is always true at the root because $threshold$ is initialized to zero and is increased as LB increases. The root does not receive THRESHOLD messages so this is the only way $threshold$ changes. We conclude $LB = threshold = UB$ is true when the root terminates. This means the root's final $threshold$ value is the cost of a global optimal solution.

Inductive Hypothesis: Let x_p denote x_i's parent. x_p's final $threshold$ value is equal to $OPT(x_p, CurrentContext)$.

We proceed by contradiction. Suppose x_i's final threshold is an overestimate. By the inductive hypothesis, x_p's final threshold is not an overestimate. It follows from the AllocationInvariant that if the final threshold given to x_i (by x_p) is too high, x_p must have given some other child (a sibling of x_i), say x_j, a final threshold that is too low (See Figure 7). Let d denote x_p's current value. Since x_j's threshold is too low, it will be unable to find a solution under the given threshold and will thus increase its own threshold. It will report lb to x_p, where $lb > t(d, x_j)$. Using Adopt's invariants, we can conclude that $threshold = UB$ cannot be true at x_p, so x_p cannot have already terminated. By the ChildThresholdInvariant, x_p will increase x_j's threshold so that $lb(d, x_j) \leq t(d, x_j)$.

Eventually, $lb(d, x_j)$ will reach an upper bound and $lb(d, x_j) = t(d, x_j) = ub(d, x_j)$ will hold. This contradicts the statement that x_j's final threshold is too low. By contradiction, x_j's final threshold value cannot be too low and x_i's final threshold cannot be too high. □

The worst-case time complexity of Adopt is exponential in the number of variables n, since constraint optimization is known to be NP-hard. To determine the worst-case space complexity at each agent, note that an agent x_i needs to maintain a $CurrentContext$ which is at most size n, and an $lb(d, x_l)$ and $ub(d, x_l)$ for each domain value and child, which is at most $\mid D_i \mid \times n$. The $context(d, x_l)$ field can require n^2 space in the worst case. Thus, we can say the worst-case space complexity of Adopt is polynomial in the number of variables n. However, it can be reduced to linear at the potential cost of efficiency. Since $context(d, x_l)$ is always compatible with $CurrentContext$, $CurrentContext$ can be used in the place of each $context(d, x_l)$, thereby giving a space complexity of $\mid D_i \mid \times n$. This can be inefficient since an agent must reset all $lb(d, x_l)$ and $ub(d, x_l)$ whenever $CurrentContext$ changes, instead of only when $context(d, x_l)$ changes.

6. Bounded-Error Approximation

We consider the situation where the user provides Adopt with an error bound b, which is interpreted to mean that any solution whose cost is within b of the optimal is acceptable. For example in over-constrained graph coloring, if the optimal solution requires violating 3 constraints, $b = 5$ indicates that 8 violated constraints is an acceptable solution. Note that this measure allows a user to specify an error bound without a priori knowledge of the cost of the optimal solution. Adopt can be guaranteed to find a global solution within bound b of the optimal by allowing the root's backtrack threshold to overestimate by b. The root agent uses b to modify its ThresholdInvariant as follows:

- **ThresholdInvariant For Root (Bounded Error):** $min(LB+b, UB) = threshold$. The root agent always sets $threshold$ to b over the currently best known lower bound LB, unless the upper bound UB is known to be less than $LB + b$.

Let us revisit the example shown in figure 2. We will re-execute the algorithm, but in this case the user has given Adopt an error bound $b = 4$. Instead of initializing $threshold$ to zero, the root agent x_1 will initialize $threshold$ to b. Note that LB is zero and UB is Inf upon initialization. Thus, $min(LB + b, UB) = min(4, Inf) = 4$ and the thresholdInvariant above requires x_1 to set $threshold = 4$. In addition, the AllocationInvariant requires x_1 to set $t(0, x_2) = 4$ since the invariant requires that $threshold = 4 = \delta(0)+t(0, x_2) = 0+t(0, x_2)$ hold.

In figure 8.a, all agents again begin by concurrently choosing value 0 for their variable and sending VALUE messages to linked descendants. In addition, x_1 sends a THRESHOLD message to x_2. Upon receipt of this message, x_2 sets $threshold = 4$ (line 8).

Each agent computes LB and UB and sends a COST message to its parent (figure 8.b). This was described previously in section 4.2 and shown in figure 6.b. The execution path is the same here.

Next, x_1 receives x_2's COST message. As before, the received costs will be stored in lines 32-33 as $lb(0, x_2) = 1$ and $ub(0, x_2) = Inf$. In line 37, execution enters the **backTrack** procedure. x_1 computes $LB(1) = \delta(1)+lb(1, x_2) = 0+0 = 0$ and $LB(0) = \delta(0)+lb(0, x_2) =$

$0 + 1 = 1$. Since $LB(1) < LB(0)$, we have $LB = LB(1) = 0$. $UB(0)$ and $UB(1)$ are computed as Inf, so $UB = Inf$. Since $threshold = 4$ is not equal $UB = Inf$, the test in line 38 fails. So far, the execution is exactly as before. Now however, the test in line 40 fails because $LB(d_i) = LB(0) = 1$ is not greater than $threshold = 4$. Thus, x_1 *will not switch value* to $x_1 = 1$ and will instead keep its current value of $x_1 = 0$.

Next, x_2 receives the COST messages sent from x_3 and x_4. The received costs will be stored in lines 32-33 as $lb(0, x_3) = 2$, $ub(0, x_3) = 2$, $lb(0, x_4) = 1$, and $ub(0, x_4) = 1$. In line 37, execution enters the **backTrack** procedure. x_2 computes $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 1 + 2 + 1 = 4$ and $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 2 + 0 + 0 = 2$. Thus, $LB = LB(1) = 2$. Similarly, x_2 computes $UB(0) = \delta(0)+ub(0, x_3)+ub(0, x_4) = 1+2+1 = 4$ and $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 2 + Inf + Inf = Inf$. Thus, $UB = UB(0) = 4$. Since $threshold = UB = 4$, the test in line 38 succeeds. However, x_2 will not switch value since its current value is the one that minimizes $UB(d)$. Note that the equivalent test in line 45 succeeds, but the test in line 46 fails since x_2 has not yet received a TERMINATE message from x_1. So, x_2 does not terminate. Instead, execution proceeds to line 52 where a COST message is sent to x_1. This is depicted in figure 8.c.

Next, x_1 receives x_2's COST message. The received costs will be stored as $lb(0, x_2) = 2$ and $ub(0, x_2) = 4$. x_1 now computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 + 0$ and $LB(0) = \delta(0)+lb(0, x_2) = 0+2 = 2$. Similarly, x_1 computes $UB(1) = \delta(1)+ub(1, x_2) = 0+Inf = Inf$ and $UB(0) = \delta(0) + ub(0, x_2) = 0 + 4 = 4$. Thus, $UB = UB(0) = 4$. So, now we have the test $threshold == UB$ in line 45 evaluate to true, since $threshold = UB = 4$. Since x_1 is the root, the test in line 47 succeeds and x_1 will terminate with value $x_1 = 0$. It will send a TERMINATE message to x_2 and the other agents will terminate in a similar manner.

In this way, we see the agents have ultimately settled on a configuration with all values equal to 0, with a total cost of 4. Since the optimal solution has cost 0, the obtained solution is indeed within the given error bound of $b = 4$. The solution was found faster because less of the solution space was explored. In particular, note that x_1 never had to explore solutions with $x_1 = 1$.

Theorems 1 and 2 still hold with the bounded-error modification to the ThresholdInvariant. Also, agents still terminate when *threshold* value is equal UB. The root's final *threshold* value is the cost of a global solution within the given error bound. Using this error bound, Adopt is able to find a solution faster than if searching for the optimal solution, thereby providing a method to trade-off computation time for solution quality. This trade-off is principled because a theoretical quality guarantee on the obtained solution is still available.

7. Evaluation

As in previous experimental set-ups[11], we experiment on distributed graph coloring with 3 colors. One node is assigned to one agent who is responsible for choosing its color. Cost of solution is measured by the total number of violated constraints. We will experiment with graphs of varying *link density* – a graph with link density d has dn links, where n is the number of nodes in the graph. For statistical significance, each datapoint representing number of cycles is the average over 25 random problem instances. The randomly generated instances were not explicitly made to be over-constrained, but note that link density 3 is

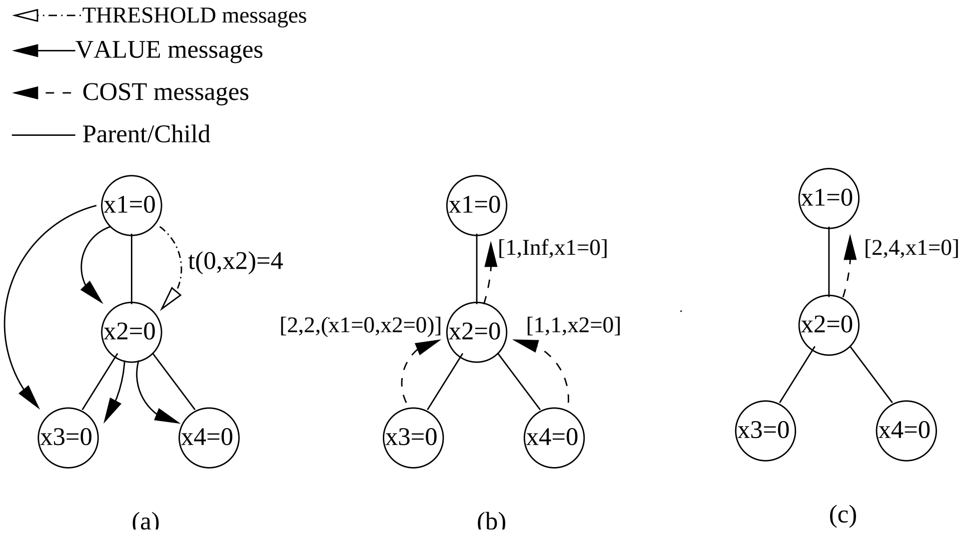

Figure 8: Example Adopt execution for the DCOP shown in figure 2, with error bound $b = 4$.

beyond phase transition, so randomly generated graphs with this link density are almost always over-constrained. The tree-structured DFS priority ordering for Adopt was formed in a preprocessing step. To compare Adopt's performance with algorithms that require a chain (linear) priority ordering, a depth-first traversal of Adopt's DFS tree was used.

As in [11], we measure "time to solution" in terms of synchronous cycles. One *cycle* is defined as all agents receiving all incoming messages and sending all outgoing messages simultaneously. The synchronous cycle metric allows repeatable experiments because it is not sensitive to differing computation speeds at different agents or fluctuations in message delivery time. Indeed, these factors are often unpredictable and we would like to control for them when performing systematic experiments. However, we note that a weakness of this metric is that it does not take into account the time required for local processing.

7.1 Efficiency

We present the empirical results from experiments using three different algorithms for DCOP – Synchronous Branch and Bound (SynchBB), Synchronous Iterative Deepening (SynchID) and Adopt. We illustrate that Adopt outperforms SynchBB[10], a distributed version of branch and bound search and the only known algorithm for DCOP that provides optimality guarantees. In addition, by comparing with SynchID we show that the speed-up comes from two sources: a) Adopt's novel search strategy, which uses lower bounds instead of upper bounds to do backtracking, and b) the asynchrony of the algorithm, which enables concurrency.

SynchID is an algorithm we have constructed in order to isolate the causes of speed-ups obtained by Adopt. SynchID simulates iterative deepening search[13] in a distributed environment by requiring agents to execute sequentially and synchronously. SynchID's

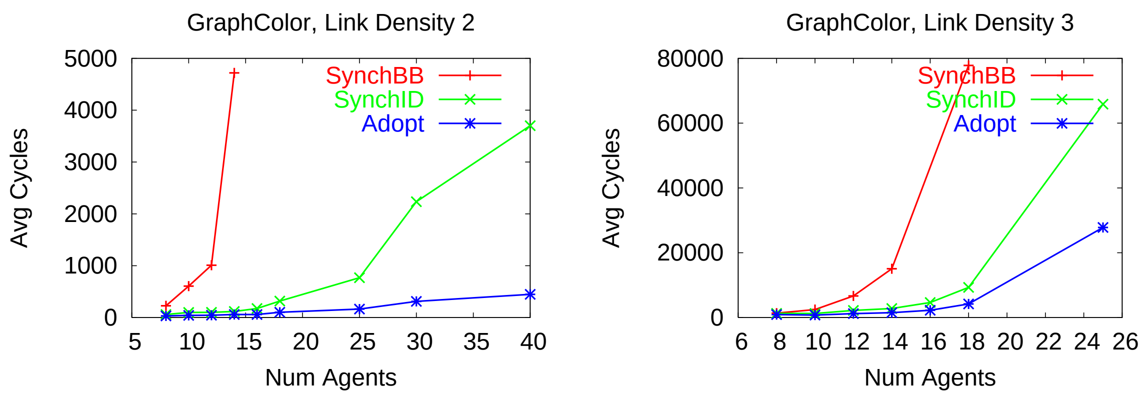

Figure 9: **Average number of cycles required to find the optimal solution (MaxCSP)**

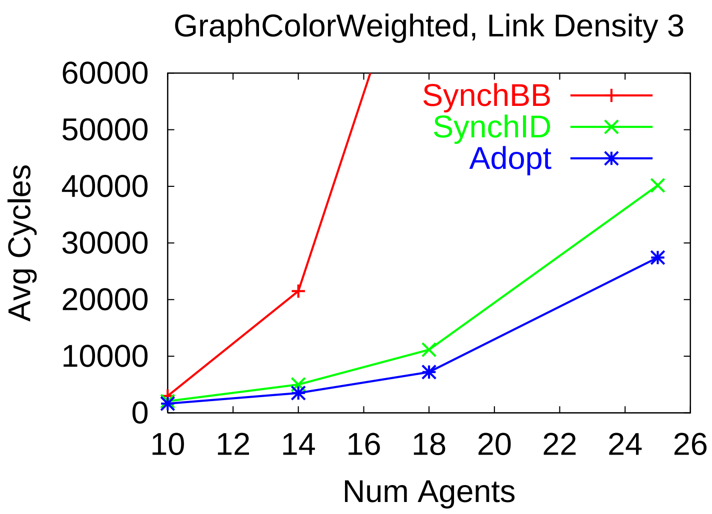

Figure 10: **Average number of cycles required to find the optimal solution (Weighted CSP)**

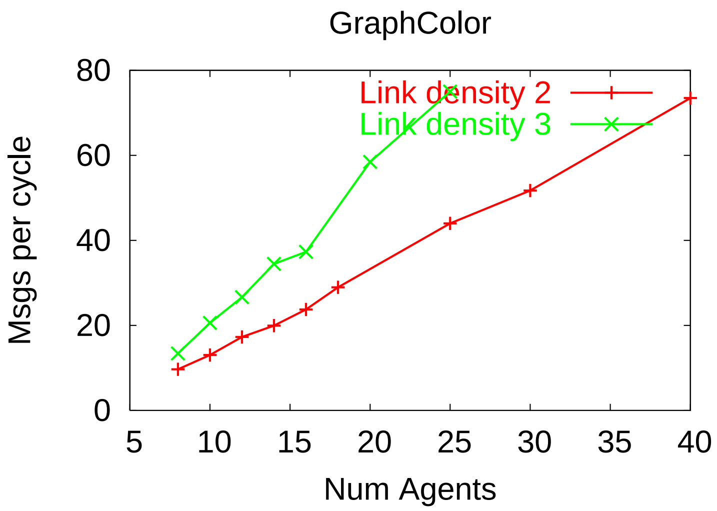

Figure 11: **Average number of messages per cycle required to find the optimal solution.**

search strategy is similar to Adopt in that both algorithms use the lower bounds to do backtracking. The central difference is that SynchID is sequential while Adopt is concurrent. In SynchID, the agents are ordered into a linear chain. (A depth-first traversal of Adopt's DFS tree was used in our experiments.) Briefly, SynchID operates as follows: the highest priority agent chooses a value for its variable first and initializes a global lower bound to zero. The next agent in the chain attempts to extend this solution so that the cost remains under the lower bound. If an agent finds that it cannot extend the solution so that the cost is less than the lower bound, a backtrack message is sent back up to the parent. Once the highest priority agent receives a backtrack message, it increases the global lower bound and the process repeats.

Figure 9 shows how SynchBB, SynchID and Adopt scale up with increasing number of agents on graph coloring problems. The results in Figure 9 (left) show that Adopt significantly outperforms both SynchBB and SynchID on graph coloring problems of link density 2. The speed-up of Adopt over SynchBB is 100-fold at 14 agents. The speed-up of Adopt over SynchID is 7-fold at 25 agents and 8-fold at 40 agents. The speedups due to search strategy are significant for this problem class, as exhibited by the difference in scale-up between SynchBB and SynchID. In addition, the figure also show the speedup due exclusively to the asynchrony of the Adopt algorithm. This is exhibited by the difference between SynchID and Adopt, which employ a similar search strategy, but differ in amount of asynchrony. In SynchID, only one agent executes at a time so it has no asynchrony, whereas Adopt exploits asynchrony when possible by allowing agents to choose variable values in parallel. In summary, we conclude that Adopt is significantly more effective than SynchBB on sparse constraint graphs and the speed-up is due to both its search strategy and its exploitation of asynchronous processing. Adopt is able to find optimal solutions very efficiently for large problems of 40 agents.

Figure 9 (right) shows the same experiment as above, but for denser graphs, with link density 3. We see that Adopt still outperforms SynchBB – around 10-fold at 14 agents and at least 18-fold at 18 agents (experiments were terminated after 100000 cycles). The speed-up between Adopt and SynchID, i.e, the speed-up due to concurrency, is 2.06 at 16 agents, 2.22 at 18 agents and 2.37 at 25 agents. Finally, Figure 10 shows results from a weighted version of graph coloring where each constraint is randomly assigned a weight between 1 and 10. Cost of solution is measured as the sum of the weights of the violated constraints. We see similar results on the more general problem with weighted constraints.

Figure 11 shows the average total number of messages sent by all the agents per cycle of execution. As the number of agents is increased, the number of messages sent per cycle increases only linearly. This is because, in Adopt, agent communicates with only neighboring agents and not with all other agents. This is in contrast to a broadcast mechanism where we would expect an exponential increase in the number of messages.

7.2 Approximating Solutions

We evaluate the effect on time to solution (as measured by cycles) and the total number of messages exchanged, as a function of error bound b in Figure 12. Error bound $b = 0$ indicates a search for the optimal solution. Figure 12 (left) shows that increasing the error bound significantly decreases the number of cycles to solution. At 18 agents, Adopt finds a solution that is guaranteed to be within a distance of 5 from the optimal in under 200 cycles, a 30-fold decrease from the number of cycles required to find the optimal solution. Similarly, figure 12 (right) shows that the total number of messages exchanged per agent decreases significantly as b is increased.

We evaluate the effect on cost of obtained solution as a function of error bound b. Figure 13 shows the cost of the obtained solution for the same problems in Figure 12. (Data for problems instances of 18 agents is shown, but the results for the other problem instances are similar.) The x-axis shows the "distance from optimal" (cost of obtained solution minus cost of optimal solution for a particular problem instance) and the y-axis shows the percentage of 25 random problem instances where the cost of the obtained solution was at the given distance from optimal. For example, the two bars labeled "$b = 3$" show that when b is set to 3, Adopt finds the optimal solution for 90 percent of the examples and a solution whose cost is at a distance of 1 from the optimal for the remaining 10 percent of the examples. The graph shows that in no cases is the cost of the obtained solution beyond the allowed bound, validating our theoretical results. The graph also shows that the cost of the obtained solutions are often much better than the given bound, in some cases even optimal.

The above results support our claim that varying b is an effective method for doing principled trade-offs between time-to-solution and quality of obtained solution. These results are significant because, in contrast to incomplete search methods, Adopt provides the ability to find solutions faster when time is limited but without giving up theoretical guarantees on solution quality.

8. Related Work

This section discusses related work in distributed constraint reasoning for multiagent domains. Section 8.1 provides an discussion of work on distributed constraint satisfaction

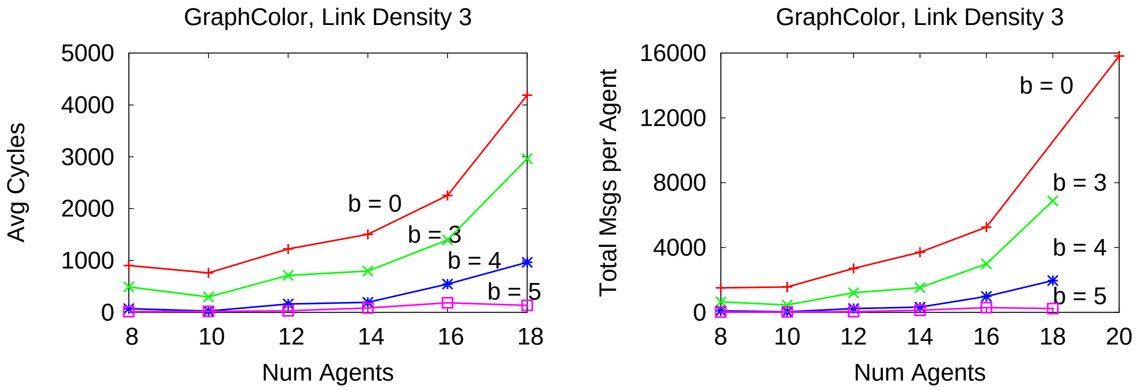

Figure 12: **Average number of cycles required to find a solution (left) and the average number of messages exchanged per agent (right) for given error bound b.**

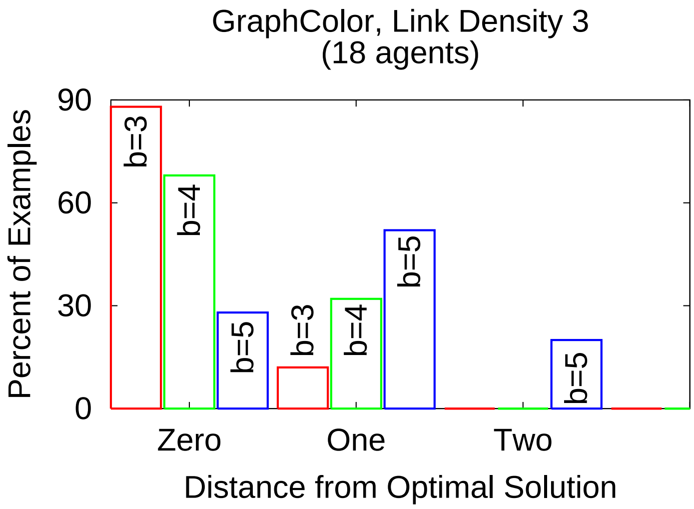

Figure 13: **For each error bound b, the percentage of problem instances where the obtained cost was at the given distance from optimal.**

Table 1: Characteristics of Distributed Constraint Optimization Methods

Method	Asynch?	Optimal?	Dist?
Satisfaction-Based Search[16][11]	N	N	Y
Local Search [10][6]	Y	N	Y
Synchronous Search [10]	N	Y	Y
Greedy Repair [14]	N	N	N
Asynchronous Best-First Search (Adopt)	Y	Y	Y

relevant to DCOP, while section 8.2 provides an overview of various existing approaches to DCOP.

8.1 Distributed Constraint Satisfaction

Yokoo, Hirayama and others have studied the DisCSP problem in depth and a family of sound and complete algorithms for solving these types of problems in a decentralized manner exist [29]. This has been an important advance and provides key insights that influence the work presented here. However, existing distributed search methods for DisCSP do not generalize easily to DCOP.

Armstrong and Durfee [1] investigate the effect of agent priority orderings on efficiency in DisCSP. They show that variable ordering heuristics from CSP can be reused as priority orderings in DisCSP and that dynamic reordering is also a useful technique. These results could potentially be generalized and applied to DCOP. Silaghi, Sam-Haroud and Faltings [26] present an alternative representation of DisCSP in which constraints are assigned to agents while variables are shared between agents. This approach allows the distributed constraint paradigm to be applied in distributed domains where constraints cannot be shared, perhaps for privacy reasons, but variables may be assigned to multiple agents. Representing DCOP in this manner is an interesting direction of future work.

8.2 Distributed Constraint Optimization

Table 1 outlines the state of the art in existing approaches to DCOP. Methods are parameterized by communication model (asynchronous or synchronous), completeness (guaranteed optimal solutions for DCOP), and "distributedness". We assume that a method is not distributed if all agents are required to communicate directly with a single agent irrespective of the underlying constraint network. The individual approaches are discussed further below.

Satisfaction-based methods. This method leverages existing DisCSP search algorithms to solve special classes of DCOP, e.g. over-constrained DisCSP. In over-constrained DisCSP, the goal is to optimize a global objective function by relaxing constraints since no completely satisfactory solution may be possible. The approach typically relies on converting the DCOP into a sequence of satisfaction problems in order to allow the use of a DisCSP algorithm. This can be done by iteratively removing constraints from the problem until a satisfactory solution is found. However, a drawback of this approach is that agents need to repeatedly synchronize to remove constraints (although the satisfaction-based search component may be asynchronous). Hirayama and Yokoo [11] show that this approach can find optimal solutions for a limited subclass of optimization problems, namely over-constrained DisCSP in which solutions can be structured into hierarchical classes. Liu and Sycara [16] present another similar iterative relaxation method, Anchor&Ascend, for heuristic search in a job-shop scheduling problem. As discussed in Section 1, these satisfaction-based methods fail to generalize to DCOP defined in this paper since agents are not able to asynchronously determine which constraints should be relaxed to obtain the optimal solution.

Local Search Methods. In this approach, agents are oblivious to non-local costs and simply attempt to minimize costs with respect to neighboring agents. Methods such as random value change or dynamic priority ordering may be used for escaping local minima. In this method, no guarantees on solution quality are available even if given unlimited execution time. Furthermore, agents cannot know the quality of the solution they have obtained. Examples of this approach include the Iterative Distributed Breakout (IDB) algorithm[10]. This algorithm utilizes the Satisfaction-Based approach described above, and so is limited in the type of DCOP it can address. In particular, IDB is applicable to a particular class of DCOP in which agents wish to minimize the maximum cost incurred at any agent. This type of criterion function has the special property that some agent can always locally determine the global cost of the current solution without knowledge of the cost incurred at other agents. For this class of DCOP, IDB is empirically shown to find good solutions quickly but cannot guarantee optimality.

Fitzpatrick and Meertens [6] present a simple distributed stochastic algorithm for minimizing the number of conflicts in an over-constrained graph coloring problem. Agents change variable value with some fixed probability in order to avoid concurrent moves. No method for escaping local minimum is used. The algorithm is shown empirically to quickly reduce the number of conflicts in large sparse graphs, even in the face of noisy/lossy communication. It is unknown how this approach would work in general since no quality guarantees are available.

Synchronous Search. This approach can be characterized as simulating a centralized search method in a distributed environment by imposing synchronous, sequential execution on the agents. It is seemingly straightforward to simulate centralized search algorithms in this manner. Examples include SynchBB (Synchronous Branch and Bound) [10] and the SynchID (Synchronous Iterative Deepening) algorithm described in Section 7 of this paper. While this approach yields an optimal distributed algorithm, the imposition of synchronous, sequential execution can be a significant drawback.

Greedy Repair. Lemaitre and Verfaille [14] describe an incomplete method for solving general constraint optimization problems. They address the problem of distributed variables by requiring a leader agent to collect global cost information. Agents then perform a greedy repair search where only one agent is allowed to change variable value at a time. Since all agents must communicate with a single leader agent, the approach may not apply in situations where agents may only communicate with neighboring agents.

8.3 Other Work in DCOP

R. Dechter, A. Dechter, and Pearl [5] present a theoretical analysis of the constraint optimization problem establishing complexity results in terms of the structure of the constraint graph and global optimization function. In addition, they outline an approach for distributed search for the optimal solution based on dynamic programming, but no algorithm or implementation is presented. While their approach has certain similarities to the methods presented here, they do not deal with asynchronous changes to global state or timeliness of solution.

Parunak *et al* [23] describe the application of distributed constraint optimization to the design of systems that require interdependent sub-components to be assembled in a manufacturing domain. The domain illustrates the unique difficulties of interdependencies between sub-problems in distributed problem solving and illustrates the applicability of the distributed constraint representation. Frei and Faltings [7] focus on modelling bandwidth resource allocation as a CSP. Although they do not deal with distributed systems, they show how the use of abstraction techniques in the constraint modelling of real problems results in tractable formulations.

9. Conclusion

Distributed constraint optimization is an important problem in domains where problem solutions are characterized by degrees of quality or cost and agents must find optimal solutions in a distributed manner. We have presented the Adopt algorithm that is guaranteed to converge to the optimal solution while using only localized, asynchronous communication and only polynomial space at each agent. The three key ideas in Adopt are a) to perform distributed backtrack search using a novel search strategy where agents are able to locally explore partial solutions asynchronously, b) backtrack thresholds for more efficient search and c) built-in termination detection. These three ideas in Adopt naturally lead to a bounded-error approximation technique for performing trade-offs between solution quality and time-to-solution. We showed that a certain class of optimization problems can be solved efficiently and optimally by Adopt and that it obtains significant orders of magnitude speedups over distributed branch and bound search.

10. Algorithmic Variations for Future Work

Adopt is one example within a space of algorithms that may be designed that exploits our key idea of using lower bounds to perform distributed optimization. In this section, we discuss some possible algorithmic modifications to Adopt. Algorithm modifications for unreliable communication are discussed in [21]. In addition, we are aware that the ordering of variables has a dramatic effect on the efficiency of the DCOP algorithm. In future work, we will develop distributed methods for discovering efficient DFS variable orderings.

Memory Usage. We consider how Adopt can be modified to obtain efficiency gains at the expense of the polynomial-space bound at each agent. In Adopt, each agent maintains a single $CurrentContext$ as a partial solution and all stored costs are conditioned on the variable values specified in that context. When context changes occur, agents delete all stored costs. This is necessary to maintain the polynomial-space bound. However, in some cases worst-case exponential-space requirements are tolerable either because sufficient memory is available or the worst-case is sufficiently unlikely to occur. In such cases, we may allow agents to store more than one partial solution at a time. Agents should not delete all stored costs when context changes and instead agents should maintain multiple contexts and their associated costs. In this way, if a previously explored context should become current again due to variable value changes at higher agents, then the stored costs will be readily available instead of having to be recomputed. Preliminary experiments (not reported here) have shown this technique can dramatically decrease solution time.

Reducing Communication. We consider how Adopt can be modified to reduce the number of messages communicated. In Adopt, an agent always sends VALUE and COST messages every time it receives a message from another agent, regardless of whether its variable value or costs have changed. As a consequence, an agent often sends a message that is identical to a message that it sent immediately prior. Although this is seemingly wasteful, it is a sufficient mechanism to ensure liveness. However, if other mechanisms are employed to ensure liveness, then it may be possible to reduce the number of messages dramatically. An alternative mechanism for ensuring liveness is through the use of timeouts, as discussed in [21].

Sending COST messages to non-parent ancestors. We consider how Adopt can be modified to allow COST messages to be sent to multiple ancestors instead of only to the parent. To see how such reporting may decrease solution time, consider the following scenario. Suppose x_r is the root agent and it has a constraint with neighbor x_i who is very low in the tree, i.e., the length of p is large, where p is the path from x_r to x_i obtained by traversing only parent-child edges in the tree-ordering. If x_r initially chooses a bad variable value that causes a large cost on the constraint shared with x_i, we would like x_r to be informed of this cost as soon as possible so that it may explore other value choices. In Adopt, x_i will send a COST message only to its immediate parent and not to x_r. The parent will then pass the cost up to its parent and so on up the tree. This method of passing costs up the tree is sufficient to ensure completeness, however, the drawback in this case is that since the length of p is large, it will take a long time for the information to reach x_r. Thus, it may take a long time before x_r will abandon its bad choice.

To resolve this problem, we may allow an agent to report cost directly to all its neighbors higher in the tree. The key difficulty with this is that double-counting of costs may occur. Such double-counting will violate our completeness guarantee. However, we can resolve this difficulty by attaching a list of agent names to every COST message (in addition to the information already in the COST messages). This list of names corresponds to those agents whose local costs were used to compute the cost information in the COST message. A receiving agent can use this list to determine when two COST messages contain overlapping costs.

Extension to n-ary constraints. Adopt can be easily extended to operate on DCOP where constraints are defined over more than two variables. Suppose we are given a DCOP that contains a ternary constraint $f_{ijk} : D_i \times D_j \times D_k \to N$ defined over 3 variables x_i, x_j, x_k, as shown in Figure 14. The tree ordering procedure must ensure that x_i, x_j and x_k lie on a single path from root to leaf (they may not be in different subtrees since all three are considered neighbors). Suppose x_i and x_j are ancestors of x_k. With binary constraints, the ancestor would send a VALUE message to the descendant. With our ternary constraint, both x_i and x_j will send VALUE messages to x_k. x_k then evaluates the ternary constraint and sends COST messages back up the tree as normal. The way in which the COST message is received and processed by an ancestor is unchanged. Thus, we deal with an n-ary constraint by assigning responsibility for its evaluation to the lowest agent involved in the constraint. The only difference between evaluation of an n-ary constraint and a binary one is

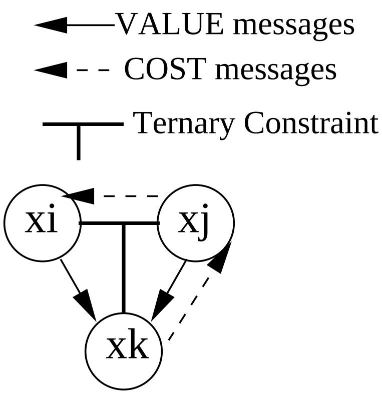

Figure 14: A ternary constraint

that the lowest agent must wait to receive all ancestors' VALUE messages before evaluating the constraint. For this reason operating on problems with n-ary constraints may decrease concurrency and efficiency of the Adopt algorithm. However this seems unavoidable due to the inherent complexity of n-ary constraints.

Acknowledgments

This research is sponsored by DARPA/ITO under contract number F30602-99-2-0507. We thank Paul Scerri for his contributions to the application of Adopt in the distributed sensor network domain. We also thank Yan Jin, Victor Lesser, Paul Rosenbloom, for their helpful comments as members of the first author's PhD thesis committee. Finally, we are grateful to the reviewers of this article for their kind suggestions.

References

[1] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proc of International Joint Conference on Artificial Intelligence*, 1997.

[2] A. Barrett. Autonomy architectures for a constellation of spacecraft. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)*, 1999.

[3] R. Caulder, J.E. Smith, A.J. Courtemanche, M.F. Mar, and A.Z. Ceranowicz. Modsaf behavior simulation and control. In *Proceedings of the Conference on Computer Generated Forces and Behavioral Representation*, 1993.

[4] H. Chalupsky, Y. Gil, C.A. Knoblock, K. Lerman, J. Oh, D.V. Pynadath, T.A. Russ, and M. Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of Innovative Applications of Artificial Intelligence Conference*, 2001.

[5] R. Dechter, A. Dechter, and J. Pearl. Optimization in constraint networks. In *Influence Diagrams, Belief Nets, and Decision Analysis*. 1990.

[6] S. Fitzpatrick and L. Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *Stochastic Algorithms: Foundations and Applications, Proceedings SAGA*, 2001.

[7] C. Frei and B. Faltings. Resource allocation in networks using abstraction and constraint satisfaction techniques. In *Proc of Constraint Programming*, 1999.

[8] E.C. Freuder and M.J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the International Joint Conference of AI*, 1985.

[9] Y. Hamadi, C. Bessiere, and J. Quinqueton. Backtracking in distributed constraint networks. In *European Conference on Artificial Intelligence*, 1998.

[10] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 222–236. 1997.

[11] K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of International Conference on Multiagent Systems*, 2000.

[12] H. Kitano, S. Todokoro, I. Noda, H. Matsubara, and T Takahashi. Robocup rescue: Search and rescue in large-scale disaster as a domain for autonomous agents research. In *Proceedings of the IEEE International Conference on System, Man, and Cybernetics*, 1999.

[13] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[14] M. Lemaitre and G. Verfaillie. An incomplete method for solving distributed valued constraint satisfaction problems. In *Proceedings of the AAAI Workshop on Constraints and Agents*, 1997.

[15] V. Lesser, C. Ortiz, and M Tambe, editors. *Distributed sensor nets: A multiagent perspective.* Kluwer academic publishers, 2003.

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

[17] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[18] P. Mesequer and M. A. Jiménez. Distributed forward checking. In *Proceedings of CP-00 Workshop on Distributed Constraint Satisfaction*, 2000.

[19] P. J. Modi, H. Jung, W. Shen, M. Tambe, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. In *Principles and Practice of Constraint Programming*, 2001.

[20] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proc of Autonomous Agents and Multi-Agent Systems*, 2003.

[21] P.J. Modi, S. M. Ali, W. Shen, and M. Tambe. Distributed constraint reasoning under unreliable communication. In *Proceedings of Distributed Constraint Reasoning Workshop at Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, 2003.

[22] Pragnesh Jay Modi. *Distributed Constraint Optimization for Multiagent Systems.* PhD thesis, University of Southern California, 2003.

[23] V. Parunak, A. Ward, M. Fleischer, J. Sauter, and T. Chang. Distributed component-centered design as agent-based distributed constraint optimization. In *Proc. of the AAAI Workshop on Constraints and Agents*, 1997.

[24] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *International Joint Conference on Artificial Intelligence*, 1995.

[25] W.-M. Shen and Mark Yim. Self-reconfigurable robots. *IEEE Transactions on Mechatronics*, 7(4), 2002.

[26] M.C. Silaghi, D. Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proceedings of National Conference on Artificial Intelligence*, 2000.

[27] BAE Systems. Ecm challenge problem, http://www.sanders.com/ants/ecm.htm. 2001.

[28] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research (JAIR)*, 7:83–124, 1997.

[29] M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.

[30] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of International Conference on Multiagent Systems*, 1998.

[31] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.