An Asynchronous Complete Method for Distributed Constraint Optimization

Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe
University of Southern California
Computer Science Department
Marina del Rey, CA 90292, USA
{modi, shen, tambe}@isi.edu

Makoto Yokoo
NTT Communication Science Labs
2-4 Hikaridai, Seika-cho
Soraku-gun, Kyoto 619-0237 Japan
{yokoo}@cslab.kecl.ntt.co.jp

ABSTRACT

We present a new polynomial-space algorithm, called *Adopt*, for distributed constraint optimization (DCOP). DCOP is able to model a large class of collaboration problems in multi-agent systems where a solution within given quality parameters must be found. Existing methods for DCOP are not able to provide theoretical guarantees on global solution quality while operating both efficiently and asynchronously. Adopt is guaranteed to find an optimal solution, or a solution within a user-specified distance from the optimal, while allowing agents to execute asynchronously and in parallel. Adopt obtains these properties via a distributed search algorithm with several novel characteristics including the ability for each agent to make local decisions based on currently available information and without necessarily having global certainty. Theoretical analysis shows that Adopt provides provable quality guarantees, while experimental results show that Adopt is significantly more efficient than synchronous methods. The speedups are shown to be partly due to the novel search strategy employed and partly due to the asynchrony of the algorithm.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent Systems*

General Terms

Algorithms

Keywords

Constraint Satisfaction/Optimization, Multi-agent Coordination

1. INTRODUCTION

A large class of multi-agent coordination and distributed resource allocation problems can be modelled via distributed constraint optimization (DCOP)[5], [2]. In DCOP, a set of collaborative agents must *optimize* over a distributed set of constraints, i.e., find solutions that meet some quality requirements. Multi-agent teamwork [10][12], distributed scheduling [5] and distributed sensor networks [8] are some examples of these types of applications. For instance, in distributed sensor networks, a set of agents must optimally allocate a set of resources (e.g., sensors) to a set of tasks (e.g., targets), where each resource is controlled by a different agent. Or, in autonomous space exploration, a constellation of satellites orbiting a planet must construct a joint plan that maximizes the amount of scientific information collected.

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 modelled as a set of constraints, and each agent knows about the constraints in which it is 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 [13] [11][7], which has relied on a satisfaction based representation. In DisCSP, 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. A DCOP method for the types of real-world applications previously mentioned must meet three key requirements. First, since the problem is 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). Second, we require a method that is able to find solutions quickly by allowing agents to operate asynchronously. A 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. Figure 1 shows groups of loosely connected agent subcommunities which could potentially execute search in parallel rather than sitting idle. Thus, in order to be efficient, the method must allow parallel execution and asynchronous communication between agents. 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 efficiently finds provably optimal solutions whenever possible and also allows 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 distributedness and asynchrony, is currently missing from the multi-agent literature. A well-known method for solving DisCSP is the Asynchronous Backtracking (ABT) algorithm of Yokoo, Durfee, Isida, and Kuwabara [13]. Simple extensions of ABT for optimization have relied on converting an optimization problem into a sequence

of satisfaction problems in order to allow the use of a DisCSP algorithm [3]. This approach has applied only to limited types of optimization problems (e.g. Hierarchical DisCSPs, Maximal DisCSPs), but has failed to apply to general DCOP problems, even rather natural ones such as minimizing the total number of constraint violations (MaxCSP). Other existing algorithms that provide quality guarantees for optimization problems such as the Synchronous Branch and Bound (SynchBB) algorithm [2] discussed later, are prohibitively slow since they require synchronous, sequential communication. Other fast, asynchronous solutions such as variants of local search [2] [14] 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. 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, the ABT algorithm concludes with certainty that the current partial solution will not lead to a global solution when a single agent locally detects an unsatisfiable constraint. While agents are able to asynchronously change variable values in ABT, that is only because of the limited representation of DisCSP, where only one constraint needs to be broken for a candidate solution to be globally inconsistent. Extensions of ABT for optimization problems [3] have continued to rely on a satisfaction-based representation and have failed to apply to general DCOP also for this reason. On the other hand, SynchBB concludes with certainty that the current partial solution will not lead to a global solution whenever the cost exceeds a synchronously computed global upper bound. This approach to backtrack search 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.

To solve this challenging problem, we propose a new distributed constraint optimization algorithm, called *Adopt* (Asynchronous Distributed Optimization). Adopt, to the best of our knowledge, is the first algorithm for distributed constraint optimization that can find either an 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 get 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 partial solutions to be abandoned before suboptimality is proved. This increases asynchrony because an agent does not need global information to make its local decisions. 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. 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.

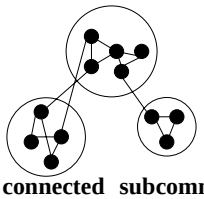

Figure 1: Loosely connected subcommunities of problem solvers

Adopt's ability to provide quality guarantees 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-quality/computation-time tradeoffs in the real world. Approaches that use incomplete search to find solutions quickly have 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 [8]. We also present empirical results demonstrating an important feature of the algorithm, namely, the ability to perform bounded-error approximation.

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 is to choose values for variables such that a given objective function is minimized or maximized. The objective function is described as an aggregation over a set of cost functions, or valued constraints. For clarity, we will deal mainly with addition as an aggregation operator. However, the techniques described in this paper can be applied to any associative, commutative, monotonic aggregation operator defined over a totally ordered set of valuations, with minimum and maximum element. This class of optimization functions is described formally by Schiex, Fargier and Verfaillie as Valued CSPs [9].

The cost functions in DCOP are the analogue of constraints from DisCSP (for convenience, we refer to cost functions as constraints). They take values of variables as input and, instead of returning "satisfied or unsatisfied", they return a valuation. Thus, for each pair of variables x_i, x_j, we may be given a *cost function* $f_{ij} : D_i \times D_j \rightarrow N \cup \infty$. Figure 2.a shows an example constraint graph with four agents. In the example, all constraints are identical only for simplicity. 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 because a constraint exists between them, but x_1 and x_4 are not neighbors because they have no constraint. The objective is to find an assignment $\mathcal{A}^*$ of values to variables such that the total cost, denoted F, is minimized and every variable has a value. Stated formally, we

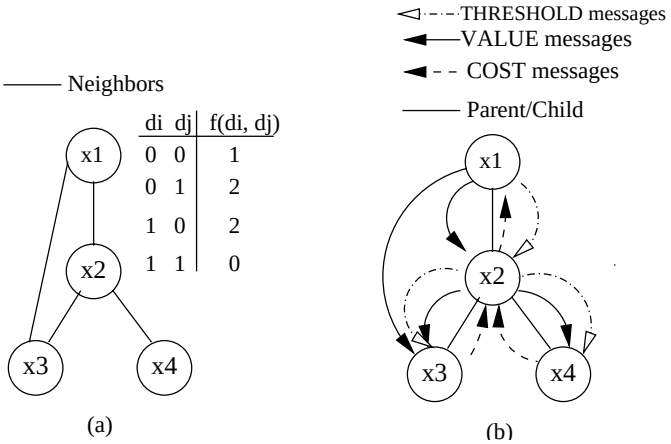

Figure 2: (a) Constraint graph. (b) Communication graph.

wish to find $\mathcal{A}$ ($= \mathcal{A}^*$) such that $F(\mathcal{A})$ is minimized, where the objective function F is defined as

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

For example, 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 are interested in managing interdependencies between different agents' choices. Therefore, we will assume each agent is assigned a single variable and use the terms "agent" and "variable" interchangeably. Since agents sometimes have complex local problems, this is an assumption to be relaxed in future work. Yokoo et al. [13] describe some methods for dealing with multiple variables per agent in DisCSP and such methods may also apply to DCOP. We will also assume that constraints are binary. Note that generalization to n-ary constraints has been achieved in the DisCSP case without significant revisions to algorithms that were originally developed for binary constraints. 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.

3. ASYNCHRONOUS SEARCH FOR DCOP

3.1 Basic Ideas

3.1.1 Opportunistic search

Adopt brings forth three new ideas not seen in previous distributed constraint reasoning algorithms. First, Adopt performs 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 [2]) that require agents to change value only when cost exceeds a global upper bound (which proves that the current solution must be suboptimal). Lower bounds are more suitable for asynchronous search because 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. Because this search strategy allows agents to abandon partial solutions before they have proved the solution is definitely suboptimal, they may be forced to reexplore previously considered solutions. The next idea in Adopt addresses this issue.

3.1.2 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 is that if a parent agent knows from previous search experience that LB is a lower bound on the cost for its (possibly multiple) children subcommunities, it should inform them not to bother searching for a solution whose cost is less than LB. In this way, a parent agent calculates backtrack threshold using LB and sends the threshold to its children. The backtrack threshold is used as an allowance on solution cost – a child agent will not change its variable value so long as its lower bound is less than the backtrack threshold. The backtrack threshold is calculated using a previously known lower bound and is ensured to be less than or equal to the cost of the optimal solution – so we know the optimal solution will not be missed.

To make the backtrack threshold approach work when multiple subcommunities search in parallel, a parent agent must distribute its backtrack threshold, denoted $threshold$, correctly to its multiple children. This is a challenging task because the parent does not remember how $threshold$ was accumulated from its children in the past. In Adopt, if a parent agent chooses variable value d which has a local cost of $\delta(d)$, it subdivides the remaining cost, $threshold - \delta(d)$, arbitrarily among its children. After some search, a child agent x_i may discover that its portion, $t(d, x_i)$, is too low because the lower bound on the cost in its subcommunity, $lb(d, x_i)$, exceeds $t(d, x_i)$. When this happens, x_i unilaterally raises its own backtrack threshold and reports to its parent. The parent agent then redistributes $threshold$ among its children by increasing $t(d, x_i)$ and decreasing the portions given to the other children. Informally, the parent maintains an **AllocationInvariant** (described later) which states that its local cost plus the sum of $t(d, x_i)$ over all children x_i must equal its backtrack threshold $threshold$ and a **ChildThresholdInvariant**, which states that no child should be given $t(d, x_i)$ less than its lower bound $lb(d, x_i)$. Using these invariants (and feedback from its children), the parent continually re-balances the subdivision of backtrack threshold among its children until the correct threshold is given to each child.

3.1.3 Built-in Termination Detection

Finally, the third key idea is the use of bound intervals for tracking the progress towards the optimal solution and, 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. While previous distributed search algorithms have required a separate termination detection algorithm, bound intervals in Adopt provide a natural termination detection criterion integrated within the algorithm. 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. Bound intervals can also 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. Agents can find an approximate solution faster than the optimal one but still provide a theoretical guarantee on solution quality.

3.2 Details of Algorithm

Agents are prioritized in a Depth-First Search (DFS) tree in which each agent has a single *parent* and multiple *children*. Thus, unlike previous algorithms such as SynchBB, Adopt does *not* require a linear priority ordering on all the agents. 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. Constraints are allowed between an agent and any of its ancestors or descendents (there is a constraint between x_1 and x_3), but there can be no constraints between nodes in different subtrees of the DFS tree. We assume parent and children are neighbors. The requirement of a DFS ordering places no restrictions on the constraint network itself – every connected constraint network can be ordered into some DFS tree[6]. Distributed algorithms for forming DFS trees are also presented in[1] [6]. We will assume the DFS ordering is done in a preprocessing step so every agent knows its parent and children.

The communication in Adopt is shown in Figure 2.b. The algorithm begins by all agents choosing their variable values concurrently. Variable values are sent down constraint edges via VALUE messages – 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 second type of message, a THRESHOLD message, is sent only from parent to child. A THRESHOLD message contains a single number representing a backtrack threshold, initially zero. Upon receipt of any type of message, an agent i) calculates cost and possibly changes variable value and/or modifies its backtrack threshold, ii) sends VALUE messages to its lower neighbors and THRESHOLD messages to its children and iii) sends a third type of message, a COST message, to its parent. 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. (x_i is informed of costs at agents in the subtree rooted at x_i by COST messages it receives from its own children.) To summarize the communication, variable value assignments (VALUE messages) are sent down the DFS tree while cost feedback (COST messages) for the higher agents' value choices 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 sent down the tree to reduce redundant search.

Procedures from Adopt are shown in Figure 3 and 4. We first focus on Figure 3. x_i represents the agent's local variable and d_i represents its current value. $CurrentContext$ is a *context* which holds x_i's view of the assignments of higher neighbors:

- **Definition:** A *context* is a partial solution of the form {(x_i,d_i), (x_j, d_j)...}. A variable can appear in a context no more than once. Two contexts are *compatible* if they do not disagree on any variable assignment.

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 [13]. 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 iii). Similarly, the ub field is stored as $ub(d, x_l)$ and the $context$ field is stored as $context(d, x_l)$. 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 ii-a, ii-b, ii-c).

x_i calculates cost as local cost plus any cost feedback received from its children. Procedures for calculation of cost are not shown in Figure 3 but are implicitly given by procedure calls, such as **LB** and **UB**, defined next. The *local cost* at x_i, 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, p$\delta(0) = 1 + 1 = 2$.

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

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

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

- **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 vi). (Realize that LB need not correspond to x_i's current value, i.e., LB need not equal $LB(d_i)$). 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 if x_i is a leaf agent, it does not receive COST messages so $\delta(d) = LB(d) = UB(d)$ for all value choices $d \in D_i$ and LB is always equal to UB in every COST message. 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 just the minimum local cost over all value choices.

x_i's backtrack threshold is stored in the $threshold$ variable, initialized to zero (line i). Whenever $LB(d_i)$ exceeds $threshold$, x_i changes its variable value to one with smaller lower bound if possible (line iv). Since $threshold$ may be less than the cost of the optimal solution, x_i cannot prove that its current value is definitely suboptimal, but it changes value to one with smaller cost anyway –

thereby realizing the opportunistic search strategy described in Section 3.1. If no value can be chosen so that lower bound is less than $threshold$, i.e., $LB > threshold$, x_i will unilaterally increase the threshold to LB (see **maintainThresholdInvariant**, Figure 4). Similarly, if x_i determines definitively that its $threshold$ is too high, i.e., $threshold > UB$, it decreases $threshold$ to UB. This 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.

The agent consumes some of the threshold through its local cost for its choice of variable. It then allocates the remaining threshold among its children and sends them THRESHOLD messages (see **maintainAllocationInvariant**, Figure 4). 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. 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.

3.3 Example

Figure 5 shows a portion of a DFS tree. The constraints are not shown. x_p has parent x_q and two children x_i and x_j. For simplicity, assume x_p has only one value in its domain ($D_p = \{d\}$). x_p received $threshold = 11$ from its parent x_q via a THRESHOLD message. x_p must choose value d for its variable and suppose this value has a local cost of 1 ($\delta(d) = 1$). In Figure 5.a, x_p subdivides the remaining threshold of 10 among its two children so that $t(d, x_i) = 5$ and $t(d, x_j) = 5$. This is an arbitrary subdivision that satisfies the AllocationInvariant – there are many other values of $t(d, x_i)$ and $t(d, x_j)$ that could be used. Subsequently, in Figure 5.b, x_p receives a COST message from its right child x_j indicating that the lower bound in that subtree is 6. x_p will set $lb(d, x_j) = 6$ (line iii) and will detect that the ChildThresholdInvariant is violated since $lb(d, x_j) = 6 \not\leq t(d, x_j) = 5$. In order to correct this, x_p increases $t(d, x_j)$ to 6. This causes the AllocationInvariant to be violated since $threshold = 11 \neq \delta(d) + t(d, x_i) + t(d, x_j) = 1 + 5 + 6 = 12$. So x_p must lower $t(d, x_i)$ to 4 to satisfy the invariant. In Figure 5.c, x_p sends the new threshold values to its children.

3.4 Algorithm Correctness and Complexity

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 $context$. For example, if x_i is a leaf, then $OPT(x_i, context) = \min_{d \in D_i} \delta(d)$, i.e., the optimal solution for a leaf is the variable value that minimizes its local cost.

Theorem 1 proves that given a fixed $CurrentContext$, the cost of the optimal solution within the subtree rooted at x_i is always guaranteed to be within the bound interval $[LB, UB]$.

Theorem 1: $\forall x_i \in V$, $LB \leq OPT(x_i, CurrentContext) \leq UB$.

The proof proceeds by induction. The base case follows from the fact $LB = OPT(x_i, CurrentContext) = UB$ is always true at

initialize
(i) $threshold \leftarrow 0$; $CurrentContext \leftarrow \{\}$
forall $d \in D_i, x_l \in Children$ **do**
(ii-a) $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
$ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$; **enddo**
$d_i \leftarrow d$ that minimizes **LB(***d***)**
backTrack

when received (**THRESHOLD**, t, $context$)
if $context$ compatible with $CurrentContext$:
$threshold \leftarrow t$
maintainThresholdInvariant
backTrack; **endif**

when received (**TERMINATE**, $context$)
record TERMINATE received from parent
$CurrentContext \leftarrow context$
backTrack

when received (**VALUE**, (x_j,d_j))
if TERMINATE not received from parent:
add (x_j,d_j) to $CurrentContext$
forall $d \in D_i, x_l \in Children$ **do**
if $context(d, x_l)$ incompatible with $CurrentContext$:
(ii-b) $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
$ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$; **endif; enddo**
maintainThresholdInvariant
backTrack; endif

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

procedure backTrack
if $threshold ==$ **UB**:
$d_i \leftarrow d$ that minimizes **UB(***d***)**
(iv) else if LB(d_i**)** $> threshold$:
$d_i \leftarrow d$ that minimizes **LB(***d***);endif**
SEND (**VALUE**, (x_i, d_i))
to each lower priority neighbor
maintainAllocationInvariant
(v-a) if $threshold ==$ **UB**:
if TERMINATE received from parent or x_i is root:
(v-b) SEND (**TERMINATE**, $CurrentContext \cup \{(x_i, d_i)\}$)
to each child
Terminate execution; **endif;endif**
(vi) SEND (**COST**, x_i, $CurrentContext$, **LB**, **UB**) to parent

Figure 3: Procedures for receiving messages (Adopt algorithm)

procedure maintainThresholdInvariant
  if $threshold <$ **LB**
    $threshold =$ **LB**; **endif**
  if $threshold >$ **UB**
    $threshold =$ **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 4: Procedures for updating backtrack thresholds

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 2 shows that Adopt will eventually terminate because its termination condition, $threshold = UB$ (line v-a), will eventually occur.

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

The proof follows from the fact that agents continually receive cost reports LB and UB from their children and pass costs up to their parent. Theorem 1 showed that LB has an upper bound and UB has a lower bound, so LB must eventually stop increasing and UB must eventually stop decreasing. The ThresholdInvariant forces $threshold$ to stay between LB and UB until ultimately $threshold = UB$ occurs. 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 this case, LB may remain less than UB at termination since some variable values may not be re-explored.

From Theorem 1, if the condition $threshold = UB$ occurs at x_i, then there exists at least one solution within the subtree rooted at x_i whose cost is less than or equal $threshold$. From Theorem 2, the condition $threshold = UB$ necessarily occurs. Next, Theorem 3 shows that the solution is optimal.

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

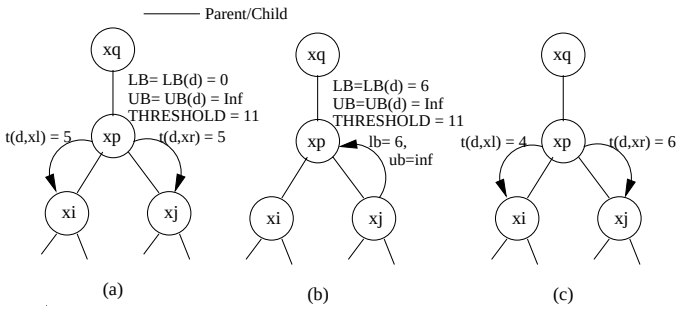

Figure 5: Example of algorithm execution

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 5). 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. □

Adopt is guaranteed to terminate because the root has an fixed (empty) $CurrentContext$, so the root can safely terminate when $threshold = UB$. Before the root terminates, its sends a TERMINATE message to its children informing them of its final value (line v-b). Upon receipt of this message, the root's children know that their $CurrentContext$ is fixed and they can safely terminate when $threshold = UB$ – and so on down the DFS tree.

The worst-case time complexity of Adopt is exponential in the number of variables, n, since constraint optimization is known to be NP-complete. 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.

3.5 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 has 3 violated

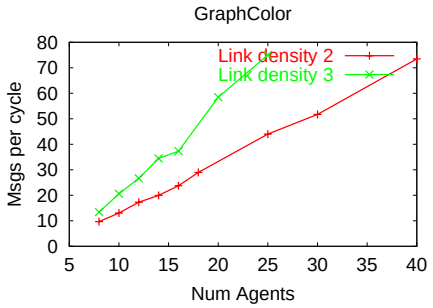

Figure 7: Average number of messages per cycle required to find the optimal solution.

constraints, $b = 5$ indicates that 8 violated constraints is an acceptable solution. This 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 $LB + b$.

Theorems 1 and 2 still hold with this modification. Also, agents still terminate when $threshold$ value is equal UB. This means 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.

4. EVALUATION

As in previous experimental set-ups[3], 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 signficance, each datapoint representing number of cycles is the average over 25 random problem instances. The randomly generated instances were not explicitly made to be overconstrained, but note that link density 3 is beyond phase transition, so randomly generated graphs with this link density are almost always overconstrained. Also as in [3], time to solution is measured in terms of synchronous cycles. One *cycle* is defined as all agents receiving all their incoming messages and sending out all their outgoing messages. 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.

4.1 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[2], 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[4] in a distributed environment. SynchID's search strategy is similar to Adopt since both algorithms iteratively increase lower bounds and use the lower bounds to do backtracking. However, the difference is that SynchID maintains a single global lower bound and agents are required to execute sequentially and synchronously while in Adopt, each agent maintains its own lower bound and agents are able to execute concurrently and asynchronously. In SynchID, the agents are ordered into a linear chain. (A depth-first traversal of Adopt's DFS tree was used in our experiments.) 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 the chain. Once the highest priority agent receives a backtrack message, it increases the global lower bound and the process repeats. In this way, agents synchronously search for the optimal solution by backtracking whenever the cost exceeds a global lower bound.

Figure 6 shows how SynchBB, SynchID and Adopt scale up with increasing number of agents on graph coloring problems. The results in Figure 6 (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 6 (middle) 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 6 (right) 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 similiar results on the more general problem with weighted constraints.

Figure 7 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 in contrast to a broadcast mechanism where we would expect an exponential increase. In Adopt, an agent communicates with only neighboring agents and not with all other agents.

4.2 Approximating Solutions

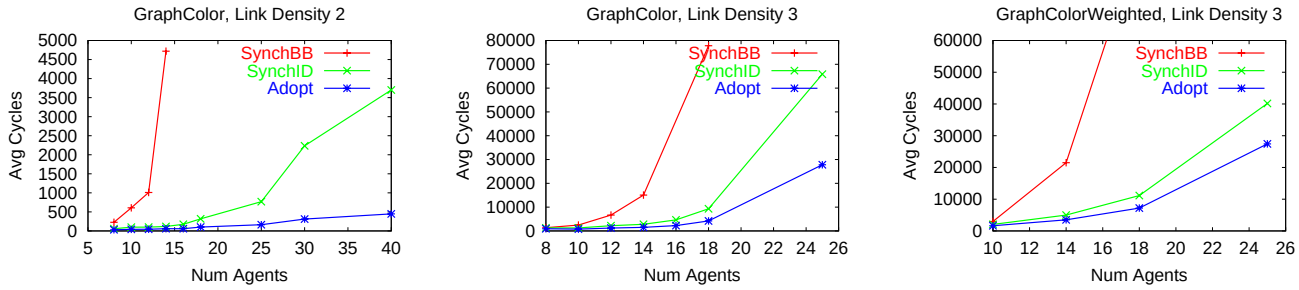

Figure 6: Average number of cycles required to find the optimal solution

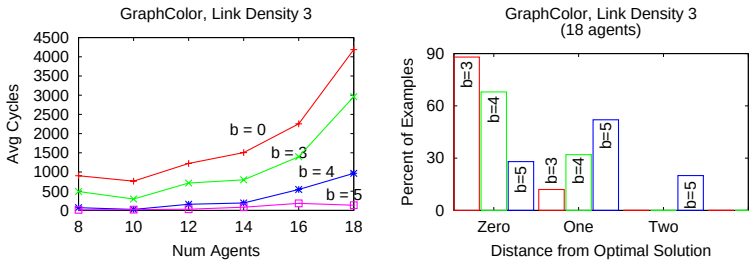

Figure 8: Average number of cycles required to find a solution within bounded error b (left) and the cost of obtained solution for 18 agents (right)

We evaluate the effect on time to solution (as measured by cycles) as a function of error bound b in Figure 8 (left). Error bound $b = 0$ indicates a search for the optimal solution. Increasing the error bound is shown to significantly decrease 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 35-fold decrease from the number of cycles required to find the optimal solution.

Figure 8 (right) shows the cost of the obtained solution for 18 agents for the same problems in Figure 8 (left). 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 bars labeled "$b = 3$" shows that Adopt finds the optimal solution for 90 percent of the examples when b is set to 3 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 tradeoffs 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.

5. 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. In future work, we will generalize Adopt to non-binary constraints and multiple variables per agent, and we will develop distributed methods for discovering efficient DFS variable orderings.

6. REFERENCES

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

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

[3] 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.

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

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

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

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

[8] 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.

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

[10] W.-M. Shen, B. Salemi, and P. Will. Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots. *IEEE Transactions on Robotics and Automation*, 2002.

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

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

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

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