An Asynchronous Complete Method for General Distributed Constraint Optimization

Pragnesh Jay Modi[1], Wei-Min Shen[1], Milind Tambe[1], and Makoto Yokoo[2]

[1] University of Southern California/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292, USA
{modi,shen,tambe}@isi.edu
[2] NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho
Soraku-gun, Kyoto 619-0237 Japan
yokoo@cslab.kecl.ntt.co.jp

Abstract. Distributed constraint optimization requires the optimization of a global objective function that is distributed as a set of valued constraints among a set of autonomous, communicating agents. To date, there does not exist an asynchronous, complete algorithm for general distributed constraint optimization problems. This paper presents *Adopt*, the first such algorithm that is asynchronous, operates on a general representation, uses linear space and is guaranteed to find optimal solutions. The main idea behind Adopt is a new distributed search strategy that is similar to iterative deepening search and that allows concurrent execution by a set of distributed agents. We show that Adopt outperforms Synchronous Branch&Bound, the only existing optimal algorithm for distributed constraint optimization. Furthermore, in order to isolate whether the speed-ups are due to Adopt's new search strategy or its exploitation of asynchrony, we compare to a synchronous version of iterative deepening, which simulates the centralized iterative deepening search strategy in a distributed environment. We show that the cause of speed-up is partly due to Adopt's new search strategy and partly due to its asynchrony.

1 Introduction

Whenever multiple autonomous agents must collaborate to accomplish a given task, they must reason about how their individual actions interact with other agents' actions and with the global objective. An agent's local action may positively or negatively affect the global outcome and this effect may be dependent on other agents' choice of actions. Human/agent organizations [1], self-reconfigurable robots [7] and distributed sensor networks[8] are some examples of multi-agent applications where the global effect of an agent's local action is dependent on others' choices. Furthermore, these domains are inherently distributed due to constraints such as privacy, autonomy, fault-tolerance, etc. Thus, a solution strategy that redistributes or centralizes the problem is not possible.

One effective way to model the interactions between agent activities and still maintain requirements of decentralization is through the distributed constraint reasoning paradigm. In this approach, global state is described by a set of variables and each

variable is assigned to an agent who has control of its value. Interactions between the local choices of each agent are modelled as constraints between variables belonging to different agents. Yokoo, Durfee, Ishida, Kuwabara and Hirayama have presented significant results for satisfaction based problems. The Distributed Constraint Satisfaction (DisCSP) [10][9] framework and associated algorithms allows agents to find an assignment of values to variables such that no constraint is violated, or determine that no such assignment is possible. While this is an important advance, DisCSP is not adequate to address many real-world problems. In particular, DisCSP requires that problem solutions be characterized with a binary designation of "satisfactory or unsatisfactory". However in many domains, solutions may have degrees of quality or cost. Overconstrained problems [3] are an example where no satisfactory solution may be possible and rather than simply returning failure, agents must find high quality solutions, i.e, a solution that is closest to a satisfactory solution. The Distributed Constraint Optimization Problem (DCOP) [5] [2] is a way to model problems where solutions have degrees of quality or cost. It requires a set of collaborative agents to optimize a global objective function that is distributed among them 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. In this paper, we will deal only with binary constraints. Finally, note that DCOP is different from parallel computing in the sense that the distribution of the objective function is mandated by the nature of the problem, not artificially imposed or manipulated for reasons of computational efficiency or parallel processing.

This paper presents *Adopt* (Asynchronous Distributed Optimization), the first complete asynchronous algorithm for general distributed constraint optimization problems. Our contribution is two-fold: a) a new distributed search strategy for general distributed constraint optimization problems that is based on iterative deepening search and requires linear space, b) exploiting this new search strategy to allow concurrent, asynchronous execution. The main idea in Adopt is for agents to asynchronously update/increase lower bounds on global solution quality. The idea of increasing lower bounds comes from the iterative deepening search strategy in centralized search [4]. As with other distributed constraint algorithms, the Adopt algorithm requires agents to be prioritized in a total order. Given this ordering, an agent communicates its variable value to "linked" lower priority agents (as defined in Section 3.1) and communicates a lower bound to a single higher priority agent. Since all communication is completely asynchronous, the algorithm is able to exploit concurrency whenever possible. Despite the fact that agents are asynchronously and concurrently choosing values for their variables, Adopt is able to guarantee globally optimal solution quality.

The only existing complete method for DCOP is the Synchronous Branch and Bound (SynchBB) algorithm described by Hirayama and Yokoo[2]. The search strategy in branch and bound search is to update/decrease upper bounds during search. Using synchronous computation, SynchBB simulates branch and bound search in a distributed environment. It requires agents to perform computation in a sequential manner in which only one agent executes at a time. The order of execution is determined by a priority ordering. Lemaitre and Verfaille describe another synchronous algorithm based on greedy repair search [5], but this algorithm is incomplete and requires a central agent to collect global state. One asynchronous approach to DCOP has been to use *iterative*

thresholding. Briefly, the approach relies on converting an optimization problem into a satisfaction problem by setting a threshold a priori and applying a DisCSP algorithm. If no satisfactory solution can be found, the threshold is iteratively lowered until a solution is found (Or conversely, the threshold is raised until no satisfactory solution is possible). Using this approach, Hirayama and Yokoo[2] [3] present iterative algorithms for specific classes of optimization problems, in particular, Hierarchical DisCSP [3] and Maximal DisCSP[2]. However, the iterative thresholding method cannot guarantee optimality for more general optimization problems such as those discussed in this paper, because agents cannot asynchronously determine that a global threshold is met.

The paper is structured as follows: Section 2 defines the DCOP problem. Section 3 describes the Adopt algorithm and proves that it is optimal. Section 4 presents our experimental results. The results show that Adopt significantly outperforms SynchBB on both overconstrained and weighted DCOPs and that the cause of the speedup is partly due to the novel search strategy employed and partly due to the asynchrony of the algorithm. Section 5 concludes the paper.

2 Distributed Constraint Optimization Problem

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. We will assume each agent is assigned only one variable and use the term agent/variable interchangeably[1]. The goal is to choose values for variables such that an objective function is minimized or maximized. For clarity, we will deal mainly with an objective function described as addition over costs, where cost is represented as a natural number. 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 [6].

Thus, for each pair of variables x_i, x_j, we are given a *cost function* $f_{ij} : D_i \times D_j \rightarrow N \cup \infty$. The cost functions in DCOP are the analogue of constraints from DisCSP (for convenience in this paper, we sometimes refer to cost functions as constraints). They take values of variables as input and, instead of returning "satisfied or unsatisfied", they return a valuation. We will deal only with binary constraints. Two agents x_i, x_j are *neighbors* if their cost function f_{ij} is not a constant. Figure 1.a shows an example constraint graph with four agents and associated cost function. In the example, all constraints are the same, but this is not necessary. 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 wish to find $\mathcal{A}$ $(= \mathcal{A}^*)$ such that $F(\mathcal{A})$ is minimized, where the objective function F is defined as

[1] Yokoo and Hirayama describe some methods for converting multiple variables per agent to single variable per agent for DisCSP problems[11]

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

For example, in Figure 1.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, $\mathcal{A}^* = \{(x_1, 1),(x_2, 1), (x_3, 1), (x_4, 1)\}$.

3 Asynchronous Search for DCOP

Section 3.1 defines some terms we will use to describe the Adopt algorithm. For exposition purposes, Section 3.2 will describe a "bare-bones" version of the Adopt algorithm, called Simple-Adopt, and Section 3.3 proves that Simple-Adopt is sound and complete. Section 3.4 will then describe the Adopt algorithm, which includes additional features over Simple-Adopt that contribute to efficiency rather than correctness.

3.1 Preliminaries

A set of variable/value pairs specifying a (possibly incomplete) assignment is called a *view*).

– **Definition:** A *view* is a set of variable/value pairs of the form $\{(x_i,d_i), (x_j, d_j)...\}$. A variable can appear in a view no more than once. Two views are *compatible* if they do not disagree on any variable assignment.

The cost of an assignment of value to a variable, with respect to a view, is determined by the sum of its local cost functions.

– **Definition:** The *local cost* δ incurred at x_i, wrt to a given view vw is defined as

$$\delta(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j) \text{ , } where\ x_i \leftarrow d_i, \\ x_j \leftarrow d_j\ in\ vw$$

For example, in Figure 1.a, if view $vw = \{(x_1, 0), (x_2, 0), (x_3, 0)\}$, then $\delta(x_3, vw) = 2$ because of cost of 1 between x_1 and x_3, plus cost of 1 between x_2 and x_3.

The Adopt algorithm requires variables to have a fixed total priority order. Any ordering is sufficient and lexicographic ordering is the simplest method. Figure 1.b shows an example priority ordering of the constraint graph in Figure 1.a. We will use the term *parent* to refer to an agent's immediate higher priority agent in the ordering and *child* to refer to an agent's immediate lower priority agent in the ordering. In Figure 1.b, x_1 is the parent of x_2, x_2 is the parent of x_3 and x_3 is the parent of x_4. Two agents x_i, x_j are *linked* if they are neighbors, if x_i is the parent or child of x_j, or if they are both linked to a common descendent. Note that non-neighbors may be linked. The solid arrows in Figure 1.b show links. We use *linked descendents (ancestors)* to refer to linked agents lower (higher) in priority ordering. In Figure 1.b, $x4$ is a linked descendent of $x2$. The priority ordering can be formed in a preprocessing step, or alternatively, can be discovered during algorithm execution. For simplicity of description of the algorithm, we will assume the ordering is done in a preprocessing step and every agent knows its parent, child, and linked descendents.

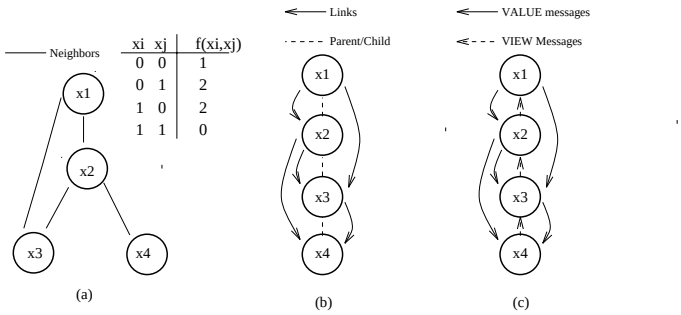

Fig. 1. (a) Example constraint graph. (b) An example ordering formed from the constraint graph in (a). (c) Flow of VALUE and VIEW messages between agents.

3.2 Simple Adopt

Procedures from the Simple-Adopt algorithm are shown in Figure 2. x_i represents the agent's local variable and d_i represents its current value. The algorithm begins by each agent choosing a value for its variable concurrently and sending this value to all its linked descendents via a VALUE message. After this, agents asynchronously wait for and respond to incoming messages. Upon receiving a VALUE message, an agent stores the current value of the linked ancestors in its $Currentvw$ variable, which represents x_i's *current context*. It then reports to its parent the current lower bound for its current context. This information is sent via a VIEW message. Figure 1.c shows the flow of VALUE and VIEW messages between agents as the algorithm executes asynchronously. To be more concrete,

- **Initialize:** For each value d, set $c(d)$, *the current lower bound* for value d, to zero. Go to hill_climb.
- **hill_climb:** For each value d, compute *estimate of lower bound*, denoted $e(d)$ (Figure 2, Line (iii)). If x_i is a leaf agent, $e(d)$ is just the local cost, $\delta(x_i, Currentvw \cup (x_i, d))$; If x_i is not a leaf agent, $e(d)$ will also include the current lower bound reported to x_i from its child. Choose d such that $e(d)$ is minimized (Figure 2, Line (iv)) and make it the current value. Send VALUE message to all linked descendents. Send the lower bound to parent via a VIEW message, but since global context could change before parent receives the VIEW message, attach the current context under which the cost was computed.
- **when received VALUE:** Update current context. If there is a context change, delete all stored lower bounds. Go to hill_climb.
- **when received VIEW:** Compare the received context against own current context (Figure 2,Line (i)) to see if they are compatible. If not compatible, throw message away; If compatible, i.e., the reported lower bound is still valid, only store new lower bound if it has increased from previously reported lower bound. If lower bound has increased, go to hill_climb.

In the **when received VIEW** procedure, it is correct to throw away VIEW messages when there is an incompatibility between $Currentvw$ and the context attached

```
# Currentvw: Current view of linked ancestor's variable values
# x_i: Local agent/variable
# d_i: x_i's current variable value
# c(d): Current lower bound on cost for subtree rooted at child, given x_i chooses value d
Initialize: Currentvw ← {};d_i ← null;
  ∀d ∈ D_i :
    c(d) ← 0
  hill_climb;
when received (VALUE, (x_j, d_j))
    add (x_j, d_j) to Currentvw;
    # context change
    if Currentvw changed then
        ∀d ∈ D_i :
          c(d) ← 0
    end if;
    hill_climb;
when received (VIEW, vw, cost)
    d ← value of x_i in vw
    if vw is compatible with                          — (i)
    Currentvw ∪ {(x_i, d)} then
        c(d) ← max(c(d), cost);                       — (ii)
        if c(d) changed then
          hill_climb;
        end if;
    end if;
procedure hill_climb
    ∀d ∈ D_i:
        # e(d) is x_i's estimate of cost if it chooses d
        e(d) ← δ(x_i, Currentvw ∪ {(x_i, d)}) + c(d);— (iii)
    choose d that minimizes e(d)                      — (iv)
        prefer current value d_i for tie;
    d_i ← d;
    SEND (VALUE, (x_i, d_i)) to all linked descendents;
    SEND (VIEW, Currentvw, e(d_i)) to
        parent;
```

Fig. 2. Procedure for asynchronous search (Simple-Adopt)

to the VIEW message. To see this, realize there can be two cases in which the contexts are incompatible; either x_i has a more up-to-date current view than its child x_l or x_l has a more current up-to-date view than its parent x_i. Either case may occur when x_i and x_l have a linked ancestor in common. In case 1, x_l must eventually receive a VALUE message, triggering a new VIEW message to be sent to x_i with the up-to-date context. In case 2, x_i must eventually receive a VALUE message causing it to update its $Currentvw$ and triggering it to send x_l a VALUE message. This in turn will cause x_l to resend a VIEW message to x_i. When x_i receives this message, the contexts will match. So in either case, it is safe for x_i to throw away VIEW messages when there is a context mismatch since the information will eventually be resent.

Next, note that in the **hill_climb** procedure, an agent always reports to its parent the cost of its variable value that minimizes its estimate of lower bound $e(d)$. In this way, the lower bounds are always conservative. This is necessary in order to guarantee completeness. If an agent does not do this, the cost of a solution may be overestimated and a good solution may be erroneously discarded by the parent.

Finally, in the **when received VALUE** procedure, an agent updates its current context by updating the current value of its linked ancestor stored in the $Currentvw$ variable. It must also delete its stored cost $c(d)$, since it may now be invalid. By storing only one current view and storing only the costs that are relevant to this current view, Simple-Adopt has space requirements (at each agent) linear in the number of variables.

Example In the example from Figure 3.a, assume that the all agents concurrently choose value 0 for their variable. Each agent sends its value to all linked descendents. Since the algorithm is asynchronous, there are many possible execution paths from here – we describe one possible execution path. x_2 will have $Currentvw = \{(x_1, 0)\}$ and will choose a value that minimizes its cost. Its best choice is $x_2 = 0$. x_2 sends a VIEW message with cost of 1 to x_1. It is safe for x_2 to report this cost because, given x_1's choice, 1 is a lower bound on *global* solution cost. This is true because 0 is a lower bound on cost below x_2 and 1 is a lower bound on local cost at x_2. Since $1 + 0 = 1$, the agents below x_1 can do no better than 1. Now concurrently with x_2's execution, x_3 will evaluate its constraints with higher agents and realize that it is incurring a cost of 2 due its interaction with x_1 and x_2. We have $\delta(x_3, \{(x_1, 0), (x2, 0), (x_3, 0)\}) = 2$. A change of value to $x_3 = 1$ would incur a cost of 4, so instead, x_3 will stick with $x_3 = 0$. x_3 will send a VIEW message of cost 2, with associated context $\{(x_1, 0), (x_2, 0)\}$, to its parent x_2. x_2 will store this cost as $c(0) = 2$.

When x_1 receives x_2's VIEW message, it will change its value to one with lower cost, namely $x_1 = 1$. It will again send VALUE messages to its linked descendents (Figure 3.b). When x_2 receives x_1's new value, it will choose the best value it can. Note that x_2 will not use the previous cost report of 2 from x_3 to choose its best value since its current context $\{(x_1, 1)\}$ is no longer compatible with the cost report context from x_3, which was $\{(x_1, 0), (x_2, 0)\}$. Figure 3.c shows the change in both x_2 and x_3 values after receiving x_1's VALUE message. In this way, the agents will ultimately settle on the optimal configuration with all values equal to 1 (total cost = 0).

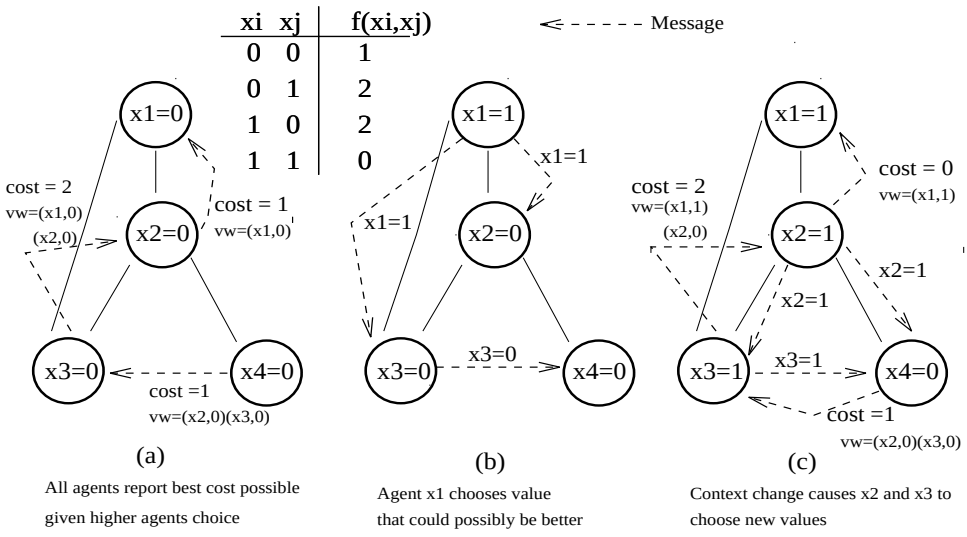

Fig. 3. Example of algorithm execution

3.3 Algorithm Correctness

We show that if Simple-Adopt reaches a stable state (all agents are waiting for incoming messages), then the complete assignment chosen by the agents is equal to the optimal assignment, and we also show that Simple-Adopt eventually reaches a stable state. Lemma 1 states that an agent's estimate of lower bound is never an overestimate, and Lemma 2 says that in a stable state, an agent's estimate of lower bound for its final choice is equal to the minimum cost possible for that choice. Let $C(x_i, vw)$ denote the local cost at x_i plus the cost of the optimal assignment to descendents of x_i, given that x_i and its ancestors have values fixed to those given in vw.

Lemma 1: An agent's estimate of the cost of a solution is never greater than the actual cost. $\forall x_i \in V, \forall d \in D_i$,

$$e(d) \leq C(x_i, Currentvw \cup \{(x_i, d)\})$$

proof: See appendix.

Lemma 2: If Simple-Adopt is in a stable state and d_i is x_i's final value, then,

$$e(d_i) = C(x_i, Currentvw \cup \{(x_i, d_i)\})$$

proof: See appendix.

Theorem 1: Assume Simple-Adopt is in a stable state. Then, there exists $\mathcal{A}^*$ such that $\forall x_i \in V$, if d_i is the value of x_i, then $(x_i, d_i) \in \mathcal{A}^*$.

proof: Assume x_i is the root agent. For the current assignment d_i, $e(d_i) = C(x_i, \{(x_i, d_i)\})$ by Lemma 2. x_i chooses d_i so that $e(d)$ is minimized. Using Lemma 1, $e(d_i) =$

$C(x_i, \{(x_i, d_i)\}) \leq e(d) \leq C(x_i, \{(x_i, d)\})$ for all $d \in D_i$. Thus, $e(d_i) = \min_d C(x_i, \{(x_i, d)\})$. By definition, the cost of $\mathcal{A}^*$ is equal to $\min_{d \in D_i} C(x_i, \{(x_i, d)\})$, which means the total cost is minimized when $x_i = d_i$. □

Finally, we have left to show that the algorithm does indeed reach a stable state in which all agents are waiting for incoming messages. We first prove Lemma 3 which states that an agent's estimate of cost never decreases.

Lemma 3: If x_i's Currentvw does not change, then for all $d \in D_i$, the value of $c(d)$ is non-decreasing over time.

proof: Figure 2, Line (ii) ensures that $c(d)$ never decreases. □.

Theorem 2: Simple-Adopt will reach a stable state.

proof: We show by induction on priorities that each agent stops sending messages. **Base case:** Assume x_i is the highest priority agent. x_i only sends VALUE messages and only in reponse to the receipt of a VIEW message that changes cost $c(d)$. The highest priority agent has an empty non-changing $Currentvw$. By Lemma 3, $c(d)$ is non-decreasing and by Lemma 1, has an upper bound. So, eventually $c(d)$ must stop changing and x_i will stop sending messages. **Inductive Step:** Assume all agents of higher priority than x_i have stopped sending messages. x_i will never receive another VALUE message and so x_i's $Currentvw$ has stopped changing. As in the base case, we can apply Lemma 3 and combine with Lemma 1 to see that x_i will eventually stop sending messages. By induction, the system reaches a stable state. □.

In summary, we have shown that Simple-Adopt is indeed guaranteed to find the optimal solution while performing distributed asynchronous search.

3.4 Algorithm Improvements

We make two modifications to the Simple-Adopt algorithm to improve efficiency. The first modification allows an agent to store more context information in order to avoid deleting its stored lower bounds unnecessarily when a context switch occurs. The second modification requires a parent to send a lower bound down to its child, thereby allowing the child to find an optimal solution faster.

One source of inefficiency in the Simple-Adopt algorithm is that all stored lower bounds must be deleted whenever context changes occur (a linked ancestor changes variable value). This is wasteful since some of the stored bounds may not be affected by the context change and thus may still be valid. For example, let x_k be the parent of x_i, x_i be the parent of x_l and let x_i be the *only* linked descendent of x_k. Suppose x_i has stored a lower bound $c(d) = a$ that was reported from child x_l. Then, suppose x_i receives a VALUE message from its parent x_k which changes its current view. Simple-Adopt will require x_i to set $c(d) = 0$, deleting the stored cost of a. But the lower bound $c(d) = a$ is still valid since there is no interaction between x_k and x_l, except through x_i. Thus, x_i is unnecessarily forced to rediscover the lower bound of a. We can avoid this inefficiency by requiring agents to store context associated with each reported cost. For example, upon receiving message (VIEW, vw, a) from x_l, x_i will store not only $c(d) = a$, but also $context(d) = vw$. In the above example, vw will not contain x_k since x_k and x_l are not linked. Then, when x_k changes its variable value, x_i can compare $Currentvw$ with $context(d)$, realize that they are still compatible, and

therefore not delete $c(d)$. This modification preserves completeness because agents still delete lower bounds when context changes occur but only when absolutely necessary.

Another source of inefficiency in the Simple-Adopt algorithm is that frequent context changes may cause the same parts of the search space to be revisited over and over again. To see this, suppose for a moment that all of x_i's linked ancestors have already chosen optimal values and x_i must now decide what to choose. Suppose x_i has two possible values, d_1 or d_2, and $c(d_1) = 0$ and $c(d_2) = 0$. x_i chooses d_1 as its variable value and sends a VALUE message to all its linked descendents, including its child x_l. x_l subsequently sends x_i a VIEW message with some non-zero cost a. After receiving this cost report, x_i will immediately switch to d_2, since $c(d_2) = 0 < c(d_1) = a$. x_i will send VALUE messages and descendents will then begin exploring their own values given this choice. If the lower bound for d_2 ever exceeds the lower bound of a for d_1, x_i will switch back to d_1, triggering context changes in descendents who are forced to explore a previously explored solution from scratch. This results in wasteful search. We can mitigate this effect in the following way. In the above scenario, when x_i switches its value from d_2 back to d_1, it can send $c(d_1) = a$ to x_l as a cost *threshold*. Since x_l has previously reported a as a lower bound on the costs for the subtree rooted at x_l, x_l and its descendents can use this threshold to find *any* solution at this cost and be guaranteed it is the best one. In particular, x_l only need switch values when $e(d_l)$ (x_l's estimate of cost for its current value) exceeds the cost of this threshold. This is similar to cost-depth limits in iterative deepening search. Note that this modification does not affect the lower bound reporting from x_l to x_i – x_l will still report the same lower bound as before – but it only prevents unnecessary context switches by x_l. This modification has space requirements (at each agent) linear in $n \times \mid D_i \mid$. Generalizing this technique to the case where an agent has more than one child, i.e., the priority ordering is a tree, while still maintaining linear space requirements seems to be a difficult task. In summary, Simple-Adopt can be modified to take advantage of cost thresholds to reduce repeated search.

Figure 4 shows the Adopt algorithm, a modification of Simple-Adopt which implements the above features.

4 Experimental Evaluation

In this section, we present the empirical results from experiments using three different distributed optimization algorithms for the distributed constraint optimization – Synchronous Branch and Bound (SynchBB), Synchronous Iterative Deepening (SynchID) and Adopt. We illustrate that Adopt outperforms SynchBB, the only known complete algorithm for distributed constraint optimization. Furthermore, we would like to know whether Adopt's speed-ups are due to its new iterative deepening search strategy or its asynchrony (which enables concurrency), or both. To answer this question, we have constructed a synchronous version of iterative deepening search, called SynchID, which performs distributed iterative deepening search in a synchronous manner.

In the previous SynchBB algorithm[2], each agent is assigned a priority and the highest priority agent chooses a value for its variable first. It informs the second priority agent of its choice, who then chooses a value. The second agent then sends the first

 # threshold: Lower bound on solution cost
 # $context(d)$: Context associated with $c(d)$
Initialize: threshold ← 0;
  $\forall d \in D_i$:
    $context(d) \leftarrow \{\}$
when received (**VALUE**, (x_j, d_j), $limit$)
  add (x_j, d_j) to $Currentvw$;
  $\forall d \in D_i$
    # change from Simple-Adopt – check for context incompatiblities
    if $context(d)$ incompatible with $Currentvw$ then
      $c(d) \leftarrow 0$
      $context(d) \leftarrow \{\}$
    end if;
  if x_j is parent
    threshold ← $limit$
  end if;
  hill_climb;
when received (**VIEW**, vw, $cost$)
  $d \leftarrow$ value of x_i in vw
  #child is my neighbor
  if vw contains (x_i, d) then
    remove (x_i, d) from vw;
    if vw compatible with $Currentvw$ and $cost > c(d)$ then
      $c(d) \leftarrow cost$;
      $context(d) \leftarrow vw$;
  #child is not my neighbor
  else $\forall d' \in D_i$:
    if vw compatible with $Currentvw$ and $cost > c(d')$ then
      $c(d') \leftarrow cost$;
      $context(d') \leftarrow vw$;
  end if;
  if $c(d_i)$ changed then
    hill_climb;
procedure hill_climb
  $\forall d \in D_i$:
    $e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$;
  choose d that minimizes $e(d)$
    prefer current value d_i for tie;
  # change from Simple-Adopt – switch value only if cost exceeds threshold
  if $e(d_i) >$ threshold
    $d_i \leftarrow d$;
  childLimit ← max($c(d_i)$, threshold - $\delta(x_i, Currentvw \cup \{(x_i, d_i)\})$)
  SEND (**VALUE**, (x_i, d_i), childLimit) to all linked descendents
  # only choose variables relevant to local cost
  $Neighborvw$ = $\{(x_j, d_j) \in Currentvw \mid x_j$ is neighbor of $x_i\}$
  $viewContext \leftarrow Neighborvw \cup \{\bigcup_{d \in D_i} context(d)\}$
  # to preserve completeness – VIEW is for best value d, not current value d_i
  SEND (**VIEW**, $viewContext$, $e(d)$) to
    parent;

Fig. 4. Efficiency Modifications to Simple-Adopt (Adopt algorithm)

agent's choice plus its choice, to the third agent, etc. Also sent is cost of the current partial solution. When a solution is complete, its cost is stored as an upper bound and backtracking occurs. In this way, agents synchronously and systematically search for the optimal solution, backtracking whenever current partial solution cost exceeds the cost of the best solution found so far. In SynchID, each agent is assigned a priority and the highest priority agent chooses a value for its variable first. It informs the second priority agent of its choice along with a "cost-depth limit" [4], which is initially zero. The second agent attempts to find a value for its variable such that the cost is less than the search limit and sends a message to the third agent, etc. If an agent finds that it has no choice of variable value such that the total cost is less than the current search limit, a backtrack message is sent back up the chain. In this way, agents synchronously search for a global solution under a given search limit. Once the highest priority agent receives a backtrack message, it increases the search limit and the process repeats.

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. Agents must find a coloring that minimizes the total number of constraint violations. As in previous expermintal set-ups, time to solution is measured in terms of synchronous cycles. One *cycle* is defined as each agent receiving all its incoming messages and sending out all its outgoing messages.

Table 1 and 2 show how SynchBB, SynchID and Adopt scale up with increasing number of variables on graph coloring problems. Table 1 shows the results for sparse graphs, with link density 2. A graph with *link density* d has dn links, where n is the number of nodes in the graph. The median number of cycles over 25 random problem instances for each datapoint is reported. The problems were not explicitly made to be overconstrained but subsequent inspection showed that more than half of the randomly generated problems were in fact overconstrained in which case the solution with a minimum number of violations must be found – an optimization problem. The agents use a random priority ordering and arbitrary (but deterministic) value ordering. The results in Table 1 show that Adopt significantly outperforms both SynchBB and SynchID. The speed-up of Adopt over SynchBB is 20-fold at 14 variables and at least 25-fold at 16 variables. The speed-up of Adopt over SynchID is 3-fold at 16 variables and 4-fold at 18 variables. 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 table also show the speedup due exclusively to the asynchrony of the Adopt algorithm. The speedup due to asynchrony is exhibited by the difference between SynchID and Adopt, which employ the same search strategy, but differ in amount of concurrency. In SynchID, only one agent executes at a time so it has no concurrency, whereas Adopt exploits concurrency 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 concurrent processing. Table 2 shows the same experiment as above, but for denser graphs, with link density 3. We see that Adopt still outperforms SynchBB – around 3-fold at 14 variables and at least 5-fold at 16 variables. The speed-up between Adopt and SynchID, i.e, the speed-up due to concurrency, is around 2-fold for both 16 variables and 18 variables. This shows that in denser graphs, a larger portion of the

speed-up of Adopt over SynchBB is due to asynchrony rather than search strategy, as compared to our results from sparser graphs.

Finally, Table 3 shows a different experimental setup in which the three algorithms are compared on random weighted CSPs. In a weighted CSP, each constraint is assigned a random weight between 1 and 10. If the constraint is broken, the agents pay a cost equal to its weight and the goal is to minimize the total cost of a solution. Problem instances are described by four parameters: $\langle n, m, p_1, p_2 \rangle$. n is the number of variables, m is the number of values for each variable, p_1 is the proportion of variable pairs that are constrained and p_2 is the proportion of value pairs that are nogoods. For a random problem in a given problem class, we randomly select $n(n-1)p_1/2$ variable pairs to be constrained and $m^2 p_2$ value pairs to be nogoods. For 12 variables, Adopt obtains a 10-fold speed-up over SynchBB and a 6-fold speed-up over SynchID. For 15 variables, Adopt obtains at least a 12-fold speed-up over SynchBB and a 4-fold speed-up over SynchID. In summary, we can see that Adopt continues to obtain significant speed-ups even for the more general constraint optimization problem where constraints can have weights.

Table 1. GraphColor (Link density=2)

	Median Cycles		
n	SynchBB	SynchID	Adopt
8	767	212	125
10	2239	390	255
12	7401	544	345
14	20899	1062	423
16	>50000	5880	1851
18	–	14604	3304

Table 2. GraphColor (Link density=3)

	Median Cycles		
n	SynchBB	SynchID	Adopt
8	1955	2220	1717
10	5251	3046	2413
12	14713	7468	5589
14	61847	28610	17425
16	>100000	46348	21714
18	–	127476	58846

Table 3. Weighted CSP (Non-Uniform Weights)

Class	Median Cycles		
⟨ n,m,p1,p2⟩	SynchBB	SynchID	Adopt
⟨ 8, 3, 0.4, 0.4 ⟩	81	20	13
⟨ 10, 3, 0.4, 0.4 ⟩	844	80	32
⟨ 12, 3, 0.4, 0.4 ⟩	10180	640	136
⟨ 15, 3, 0.4, 0.4 ⟩	>50000	11964	3087

5 Conclusion

Distributed constraint optimization is an important problem in many real multi-agent domains where problem solutions are characterized by degrees of quality or cost and a set of agents must find optimal solutions in a distributed decentralized manner. We have presented the first general-purpose asynchronous algorithm for distributed constraint optimization that is guaranteed to converge to the optimal solution. The main idea behind the presented algorithm is a new distributed search strategy that allows agents to asynchronously increase lower bounds on solution quality and enables agents to operate asynchronously, thereby exploiting parallelism when possible. We have shown significant orders of magnitude speedups over the best previous algorithm SynchBB. We also showed that the speed-ups obtained by the algorithm can be attributed to both its novel search strategy and its ability to exploit parallelism. In future work, we will obtain further speed-ups by investigating memory/time tradeoffs and more efficient variable orderings.

References

1. 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.
2. K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In G. Smolka, editor, *Principles and Practice of Constraint Programming – CP97*, pages 222–236. 1997.
3. K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proc. of the 4th Intl. Conf. on Multi-Agent Systems(ICMAS)*, July 2000.
4. Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
5. 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.
6. T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *International Joint Conference on Artificial Intelligence*, 1995.
7. WM Shen and P. Will. Docking in self-reconfigurable robots. In *Proceedings of IROS*, 2001.
8. BAE Systems. Ecm challenge problem, http://www.sanders.com/ants/ecm.htm. 2001.

9. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
10. M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of International Conference on Multi-Agent Systems*, 1996.
11. M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *ICMAS*, July 1998.

Appendix

Lemma 1: An agents estimate of the cost of a solution is never greater than the actual cost. $\forall x_i \in V, \forall d \in D_i$,

$$e(d) \leq C(x_i, Currentvw \cup \{(x_i, d)\})$$

proof: By induction on agent priorities. In the base case, assume x_i is the lowest priority. Then $e(d) = \delta(x_i, Currentvw \cup \{(x_i, d)\}) = C(x_i, Currentvw \cup \{(x_i, d)\})$ and we are done. So the base case is proved. By the inductive assumption, for all $d_l \in D_l$,

$$e_l(d_l) \leq C(x_l, Currentvw \cup \{(x_i, d)\} \cup \{(x_l, d_l)\}) \quad (1)$$

By definition,

$$\begin{aligned} &C(x_i, Currentvw \cup \{(x_i, d)\}) = \\ &\quad \delta(x_i, Currentvw \cup \{(x_i, d)\}) \ + \\ &\quad \min_{d_l} C(x_l, Currentvw \cup \{(x_i, d)\} \cup \{(x_l, d_l)\} \end{aligned}$$

From 1 we have,

$$\min_{d_l} e_l(d_l) \leq \min_{d_l} C(x_l, Currentvw \cup \{(x_i, d)\} \cup \{(x_l, d_l)\})$$

Line (iii) of Figure 2 gives

$$e(d) = \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$$

x_l reports $\min_{d_l} e(d_l)$ to only x_i. If x_i has received a message (VIEW, $Currentvw \cup \{(x_i, d)\}$, $e_l(d_l)$), then $c(d) = e_l(d_l)$. Otherwise, $c(d) = 0$. In either case,

$$\begin{aligned} e(d) &= \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d) \\ &\leq \delta(x_i, Currentvw \cup \{(x_i, d)\}) + \min_{d_l} e(d_l) \\ &\leq \delta(x_i, Currentvw \cup \{(x_i, d)\}) + \\ &\quad \min_{d_l} C(x_l, Currentvw \cup \{(x_i, d)\} \cup \{(x_l, d_l)\}) \\ &= C(x_i, Currentvw \cup \{(x_i, d)\}) \end{aligned}$$

which is to be shown. By induction, the Lemma is proved.□

Lemma 2: If Simple-Adopt is in a stable state and d_i is x_i's final value, then,

$$e(d_i) = C(x_i, Currentvw \cup \{(x_i, d_i)\})$$

proof sketch: The structure of the proof is identical to Lemma 1, except in the last step we know all messages have been received, so $c(d) = e_l(d_l)$.