Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems

Makoto Yokoo
NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun,
Kyoto 619-02 Japan
e-mail: yokoo@cslab.kecl.ntt.jp

Katsutoshi Hirayama
Kobe University of Mercantile Marine
5-1-1 Fukae-minami-machi, Higashinada-ku,
Kobe 658, Japan
e-mail: hirayama@ti.kshosen.ac.jp

Abstract

This paper presents a new algorithm for solving distributed constraint satisfaction problems (distributed CSPs) called the *distributed breakout* algorithm, which is inspired by the breakout algorithm for solving centralized CSPs. In this algorithm, each agent tries to optimize its evaluation value (the number of constraint violations) by exchanging its current value and the possible amount of its improvement among neighboring agents. Instead of detecting the fact that agents as a whole are trapped in a local-minimum, each agent detects whether it is in a quasi-local-minimum, which is a weaker condition than a local-minimum, and changes the weights of constraint violations to escape from the quasi-local-minimum. Experimental evaluations show this algorithm to be much more efficient than existing algorithms for critically difficult problem instances of distributed graph-coloring problems.

Introduction

A constraint satisfaction problem (CSP) is a general framework that can formalize various problems in AI, and many theoretical and experimental studies have been performed (Mackworth 1992). In (Yokoo *et al.* 1992), a distributed constraint satisfaction problem (distributed CSP) is formalized as a CSP in which variables and constraints are distributed among multiple automated agents. Various application problems in DAI which are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems (Conry *et al.* 1991), distributed scheduling problems (Sycara *et al.* 1991), distributed interpretation tasks (Mason & Johnson 1989) and multi-agent truth maintenance tasks (Huhns & Bridgeland 1991)) can be formalized as distributed CSPs. Therefore, we can consider a distributed CSP as a general framework for DAI, and distributed algorithms for solving distributed CSPs as an important infrastructure in DAI.

It must be noted that although algorithms for solving distributed CSPs seem to be similar to parallel/distributed processing methods for solving CSPs (Collin, Dechter, & Katz 1991; Zhang & Mackworth 1991), research motivations are fundamentally different. The primary concern in parallel/distributed processing is the efficiency, and we can choose any type of parallel/distributed computer architecture for solving a given problem efficiently. In contrast, in a distributed CSP, there already exists a situation where knowledge about the problem (i.e., variables and constraints) is distributed among automated agents. Therefore, the main research issue is how to reach a solution from this given situation.

The authors have developed a series of algorithms for solving distributed CSPs, i.e., (a) a basic algorithm called *asynchronous backtracking* (Yokoo *et al.* 1992), in which agents act asynchronously and concurrently based on their local knowledge without any global control, (b) a more efficient algorithm called *asynchronous weak-commitment search* (Yokoo 1995), in which the priority order of agents is changed dynamically, and (c) a distributed iterative improvement algorithm (Hirayama & Toyoda 1995), where agents can escape from local-minima by forming coalitions among themselves.

In this paper, we develop a new distributed iterative improvement algorithm called the *distributed breakout* algorithm, which is inspired by the breakout algorithm (Morris 1993) for solving centralized CSPs. The main characteristic of this algorithm is as follows.

- Instead of detecting the fact that agents as a whole are trapped in a local-minimum, each agent detects the fact that it is in a quasi-local-minimum, which is a weaker condition than a local-minimum, and changes the weights of constraint violations.

Experimental results on particularly difficult instances of distributed graph-coloring problems show that the distributed breakout algorithm is much more efficient than existing algorithms (i.e., a fifteen-fold speed-up can be obtained).

In the remainder of this paper, we show the definition of a distributed CSP. Then, we briefly describe the breakout algorithm for solving centralized CSPs, and describe the basic ideas and details of the distributed breakout algorithm. Furthermore, we show empirical results which illustrate the efficiency of this algorithm. Finally, we compare characteristics of the distributed

breakout algorithm and existing algorithms.

Distributed Constraint Satisfaction Problem

A CSP consists of n variables $x_1, x_2, \ldots, x_n$, whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively, and a set of constraints on their values. A constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \cdots, x_{kj})$ is a predicate which is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied.

A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents. We assume that the communication between agents is done by sending messages. Each agent has some variables and tries to determine their values. The value assignment must satisfy the inter-agent constraints.

Without loss of generality, we make the following assumptions while describing our algorithm for simplicity. Relaxing these assumptions to general cases is relatively straightforward.

- Each agent has exactly one variable.
- All constraints are binary.

We can represent a distributed CSP in which all constraints are binary as a network, where variables are nodes and constraints are links between nodes. Since each agent has exactly one variable, a node also represents an agent. In the following, we use the same identifier x_i to represent an agent and its variable.

For an agent x_i, we call a set of agents, each of which is directly connected to x_i by a link, as *neighbors* of x_i. Also, a *distance* between two agents is defined as the number of links of the shortest path connecting these two agents. For example, Figure 1 shows one instance of a distributed graph-coloring problem, in which six agents exist. Each agent tries to determine its color so that neighbors do not have the same color (possible colors are white and black). The neighbors of x_1 are $\{x_2, x_6\}$, and the distance between x_1 and x_4 is 3.

Distributed Breakout Algorithm

In this section, we briefly describe the breakout algorithm for solving CSPs (Morris 1993), and describe how a similar algorithm can be implemented for distributed CSPs.

Breakout algorithm

The breakout algorithm (Morris 1993) is one kind of iterative improvement algorithms (Minton *et al.* 1992; Selman, Levesque, & Mitchell 1992). In these algorithms, a *flawed* solution containing some constraint violations is revised by local changes until all constraints are satisfied. In the breakout algorithm, a weight is defined for each pair of variable values that does not satisfy constraints (the initial weight is 1), and the summation of the weights of constraint violating pairs is used to evaluate the flawed solution. In the initial state, the summation is equal to the number of the constraint violations. In the breakout algorithm, a variable value is changed to decrease the evaluation value (i.e., the number of constraint violations). This strategy is called the *min-conflict* heuristic (Minton *et al.* 1992).

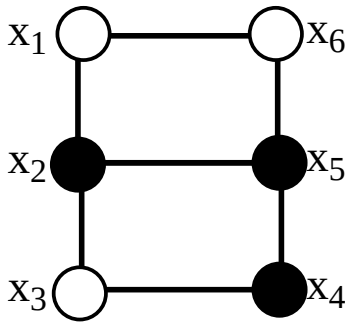

Figure 1: Example of a Constraint Network

If the evaluation value can not be decreased by changing the value of any variable, the current state is called a local-minimum. When trapped in a local-minimum, the breakout algorithm increases the weights of constraint violating pairs in the current state by 1 so that the evaluation value of the current state becomes larger than the neighboring states; thus the algorithm can escape from a local-minimum. Although the breakout algorithm is very simple, Morris shows in (Morris 1993) that it outperforms other iterative improvement algorithms (Minton *et al.* 1992; Selman, Levesque, & Mitchell 1992).

Basic Ideas

In applying the breakout algorithm to distributed CSPs, we encounter the following difficulties.

- If we allow only one agent to change its value at a time, we can not take advantage of parallelism. On the other hand, if two neighboring agents are allowed to change their values at the same time, the evaluation value may not be improved, and an oscillation (the agents continue the same actions repeatedly) may occur.
- To detect the fact that agents as a whole are trapped in a local-minimum, the agents have to globally exchange information among themselves.

In order to solve these difficulties, we introduce the following ideas.

- Neighboring agents exchange values of possible improvements, and only the agent that can maximally

improve the evaluation value is given the right to change its value. Note that if two agents are not neighbors, it is possible for them to change their values concurrently.

- Instead of detecting the fact that agents as a whole are trapped in a local-minimum, each agent detects the fact that it is in a quasi-local-minimum, which is a weaker condition than a local-minimum and can be detected via local communications.

In this algorithm, two kinds of messages (ok? and improve) are communicated among neighbors. The ok? message is used to exchange the current value assignment of the agent, and the improve message is used to communicate the possible improvement of the evaluation value, by a change in the agent's value. By exchanging the improve messages among neighbors and giving only to the agent that can maximally improve the evaluation value the right to change its value, the neighbors will not change their values concurrently, while non-neighbors can.

We define the fact that agent x_i is in a quasi-local-minimum as follows.

- x_i is violating some constraint, and the possible improvement of x_i and all of x_i's neighbors is 0.

It is obvious that if the current situation is a (real) local-minimum, each of the constraint violating agents is in a quasi-local-minimum, but not vice versa. For example, in Figure 1, although x_1 is in a quasi-local-minimum, this situation is not a real local-minimum since x_5 can improve the evaluation value.

Increasing the weights in quasi-local-minima that are not real local-minima may adversely affect the performance if the evaluation function is modified too much. We will evaluate the effect of increasing the weights in quasi-local-minima rather than real local-minima.

These ideas are basically equivalent to those used in the authors' previous work on the distributed iterative improvement algorithm presented in (Hirayama & Toyoda 1995). However, the algorithm is much simpler than the one presented in (Hirayama & Toyoda 1995), since agents independently increase the weights of constraint violations in a quasi-local-minimum, rather than negotiate to form coalitions.

Details of Algorithm

In this algorithm, each agent randomly determines its initial value, then sends ok? messages to its neighbors. After receiving ok? messages from all of its neighbors, it calculates its current evaluation value (the summation of the weights of constraint violating pairs related to its variable) and the possible improvement of the evaluation value, and sends improve messages to its neighbors.

The procedures executed at agent x_i when receiving ok? and improve messages are shown in Figure 2. The agent alternates in the wait_ok? mode (Figure 2 (i)) and the wait_improve mode (Figure 2 (ii)).

In the wait_ok? mode, x_i records the value assignment of a neighbor in its *agent_view*. After receiving ok? messages from all neighbors, it calculates its current evaluation value and a possible improvement, sends improve messages to its neighbors, and enters the wait_improve mode.

The meanings of the state variables used in this algorithm are as follows:

can_move: represents whether x_i has the right to change its value. If the possible improvement of a neighbor x_j is larger than the improvement of x_i, or the possible improvements are equal and x_j precedes x_i in alphabetical order, *can_move* is changed to *false*.

quasi_local_minimum: represents whether x_i is in a quasi-local-minimum. If the possible improvement of a neighbor x_j is positive, *quasi_local_minimum* is changed to *false*.

my_termination_counter: is used for the termination detection of the algorithm. If the value of *my_termination_counter* is d, every agent whose distance from x_i is within d satisfies all of its constraints. We assume that x_i knows the maximal distance to other agents or an appropriate upper-bound *max_distance*. If the value of *my_termination_counter* becomes equal to *max_distance*, x_i can confirm that all agents satisfy their constraints.

The correctness of this termination detection procedure can be inductively proven using the fact that the *my_termination_counter* of x_i is increased from d to $d+1$ iff each neighbor of x_i satisfies all of its constraints and the *my_termination_counter* of each neighbor is equal to or larger than d.

After receiving improve messages from all neighbors, x_i changes the weights of constraint violations if the state variable *quasi_local_minimum* is *true*. Since each agent independently records the weights, neighboring agents do not need to negotiate about increasing the weights. If the state variable *can_move* is *true*, x_i's value is changed; otherwise, the value remains the same. The agent sends ok? messages and enters the wait_ok? mode.

Due to the delay of messages or differences in the processing speeds of agents, x_i may receive an improve message even though it is in the wait_ok? mode, or vice versa. In such a case, x_i postpones the processing of the message, and waits for a next message. The postponed message is processed after x_i changes its mode. On the other hand, since an agent can not send ok? messages unless it receives improve messages from all neighbors, x_i in the wait_ok? mode will never receive an ok? message that should be processed in the next or previous cycle. The same is true for improve messages.

```
wait_ok? mode — (i)
when received (ok?, x_j, d_j) do
  counter ← counter + 1;
  add (x_j, d_j) to agent_view;
  when counter = number_of_neighbors do
      send_improve; counter ← 0;
      goto wait_improve mode; end do;
  goto wait_ok mode; end do;

procedure send_improve
  current_eval ← evaluation value of current_value;
  my_improve ← possible maximal improvement;
  new_value ←
        the value which gives the maximal improvement;
  if current_eval = 0 then consistent ← true;
    else consistent ← false;
      my_termination_counter ← 0; end if;
  if my_improve > 0
    then can_move ← true; quasi_local_minimum ← false;
    else can_move ← false;
      quasi_local_minimum ← true; end if;
  send (improve, x_i, my_improve, current_eval,
          my_termination_counter) to neighbors;

wait_improve? mode — (ii)
when received (improve, x_j, improve,
                      eval, termination_counter) do
  counter ← counter + 1;
  my_termination_counter ←
      min(termination_counter, my_termination_counter)
  when improve > my_improve do
    can_move ← false;
    quasi_local_minimum ← false; end do;
  when improve = my_improve and x_j precedes x_i do
    can_move ← false; end do;
  when eval > 0 do
    consistent ← false; end do;
  when counter = number_of_neighbors do
    send_ok; counter ← 0; clear agent_view;
    goto wait_ok mode; end do;
  goto wait_improve mode; end do;

procedure send_ok
  when consistent = true do
    my_termination_counter ← my_termination_counter +1;
      when my_termination_counter = max_distance do
        notify neighbors that a solution has been found;
            terminate the algorithm;
      end do; end do;
  when quasi_local_minimum = true do
    increase the weights of constraint violations; end do;
  when can_move = true do
    current_value ← new_value; end do;
  send (ok?, x_i, current_value) to neighbors;
```

Figure 2: Procedures for receiving messages

Example of Algorithm Execution

We show an example of algorithm execution in Figure 3. This problem is an instance of a distributed graph-coloring problem, where the possible colors of agents are black and white.

We assume that initial values are chosen as in Figure 3(a). Each agent communicates this initial value via ok? messages. After receiving ok? messages from all of its neighbors, each agent calculates *current_evaluation* and *my_improvement*, and exchanges improve messages. Initially, all weights are equal to 1. In the initial state, the improvements of all agents are equal to 0. Therefore, the weights of constraint violating pairs (x_1=white and x_6=white, x_2=black and x_5=black, and x_3=white and x_4=white) are increased by 1 (Figure 3(b)). Then, the improvements of x_1, x_3, x_4, and x_6 are 1, and the improvements of x_2 and x_5 are 0. The agents that have the right to change their values are x_1 and x_3 (each of which precedes in the alphabetical order within its own neighborhood). They each change their value from white to black (Figure 3(c)). Then, the improvement of x_2 is 4, while the improvements of the other agents are 0. Therefore, x_2 changes its value to white, and all constraints are satisfied (Figure 3(d)).

Evaluation

In this section, we evaluate the efficiency of distributed constraint satisfaction algorithms by a discrete event simulation, where each agent maintains its own simulated clock. An agent's time is incremented by one simulated time unit whenever it performs one cycle of computation. One cycle consists of reading all incoming messages, performing local computation, and then sending messages. We assume that a message issued at time t is available to the recipient at time $t+1$. We analyze the performance in terms of the number of cycles required to solve the problem. One cycle corresponds to a series of agent actions, in which an agent recognizes the state of the world, then decides its response to that state, and communicates its decisions. In the distributed breakout algorithm, each mode (wait_ok? or wait_improve) requires one cycle. Therefore, each agent can change its value at most once in two cycles.

We are going to compare the distributed breakout algorithm (DB), the asynchronous weak-commitment search algorithm (AWC) (Yokoo 1995), and the iterative improvement algorithm presented in (Hirayama & Toyoda 1995). We call this algorithm *hill-climbing+coalition* (HCC). In HCC, we use the altruistic strategy (Hirayama & Toyoda 1995) in coalitions, and make all coalitions broken up to restart with new initial values if one of coalitions grows up to the state where it includes over 5 agents. Furthermore, to examine the effect of changing weights in quasi-local-minima, rather than real local-minima, we show the result of an algorithm in which each agent broadcasts messages not only to its neighbors but to all

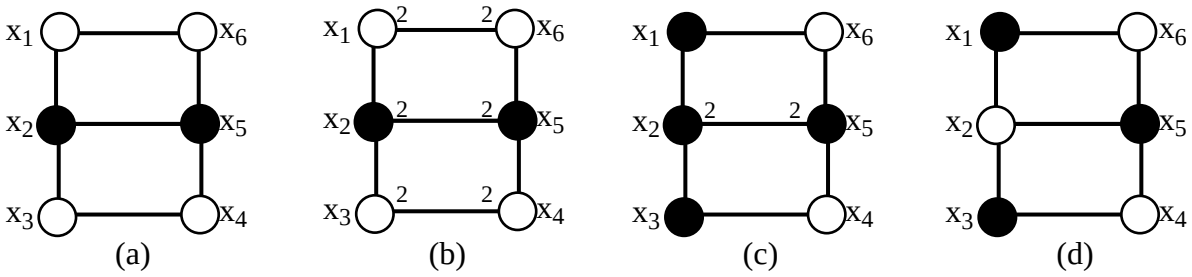

Figure 3: Example of algorithm execution

agents, and increases the weights only in a real local-minimum. We call this algorithm *distributed breakout with broadcasting* (DB+BC). In this algorithm, messages from non-neighbors are used only for detecting local-minima, and other procedures are equivalent to the original distributed breakout algorithm.

We use the distributed graph-coloring problem for our evaluations. This problem can represent various application problems such as channel allocation problems in mobile communication systems, in which adjoining cells (regions) can not use the same channels to avoid interference. A distributed graph-coloring problem can be characterized by three parameters, i.e., the number of agents/variables n, the number of possible colors of each agent k, and the number of links between agents m. We randomly generated a problem with n agents/variables and m arcs by the method described in (Minton *et al.* 1992), so that the graph is connected and the problem has a solution.

First, we evaluated the problem for $n = 90$, 120, and 150, where $m = n \times 2$, k=3. This parameter setting corresponds to the "sparse" problems for which Minton *et al.* reported poor performance of the min-conflict heuristic in (Minton *et al.* 1992). We generated 10 different problems, and for each problem, 10 trials with different initial values were performed (100 trials in all). The initial values were set randomly. The results are summarized in Table 1. For each trial, in order to conduct the experiments within a reasonable amount of time, we set the limit for the number of cycles at 10000, and terminated the algorithm if this limit was exceeded; we counted the result as 10000. We show the ratio of trials completed successfully to the total number of trials in the table.

Furthermore, we show results where the number of links $m = n \times 2.7$ and $m = n \times (n-1)/4$ in Table 2 and Table 3, respectively. The setting where $k = 3$ and $m = n \times 2.7$ has been identified as a critical setting which produces particularly difficult problems in (Cheeseman, Kanefsky, & Taylor 1991). The setting where $m = n \times (n-1)/4$ represents the situation in which the constraints among agents are dense.

Finally, we evaluated the problem for $n = 60$, 90, and 120, where $m = n \times 4.7$ and k=4. This parameter setting has also been identified as critical setting which produces particularly difficult problems in (Cheeseman, Kanefsky, & Taylor 1991). The results are summarized in Figure 4. The limit of the cycles was set to 40000 in this setting.

We can see the following facts from these results.

For "sparse" and "dense" problems, the asynchronous weak-commitment search algorithm is most efficient. Since these problems are relatively easy, the overhead for controlling concurrent actions among neighbors does not pay. Note that in the distributed breakout algorithm, an agent can change its value at most once in two cycles, and in the hill-climbing algorithm, an agent can change its value once in three cycles (i.e., sending negotiate, sending reply, and sending state), while in the asynchronous backtracking algorithm, an agent can change its value in every cycle.

For critically difficult problem instances, the distributed breakout algorithm is most efficient. On the other hand, in centralized CSPs, it has been pointed out that even for critically difficult problems, the weak-commitment search algorithm (the centralized version of the asynchronous weak-commitment search algorithm) is much more efficient than the breakout algorithm (Yokoo 1994). What causes this difference?

First, in the (centralized) weak-commitment search algorithm, various variable ordering heuristics such as forward-checking and the first-fail principle (Haralick & Elliot 1980) are introduced, and by performing a look-ahead search before determining a variable value, the algorithm can avoid choosing a value that leads to an immediate failure. Although it is possible to introduce variable/agent ordering heuristics in the asynchronous weak-commitment search algorithm, it is difficult to perform a look-ahead search in distributed CSPs since the knowledge about the problem is distributed among multiple agents.

Furthermore, the breakout algorithm requires more computations in each cycle (i.e., changing one variable value or changing weights) than the weak-commitment

Table 1: Evaluation with "sparse" problems ($k = 3, m = n \times 2$)

	DB		DB+BC		AWC		HCC	
n	ratio	cycles	ratio	cycles	ratio	cycles	ratio	cycles
90	100%	150.8	100%	230.4	100%	70.1	98%	533.5
120	100%	210.1	100%	253.4	100%	106.4	100%	538.4
150	100%	278.8	100%	344.5	100%	159.2	99%	1074.2

Table 2: Evaluation with "critical" problems ($k = 3, m = n \times 2.7$)

	DB		DB+BC		AWC		HCC	
n	ratio	cycles	ratio	cycles	ratio	cycles	ratio	cycles
90	100%	517.1	100%	397.1	97%	1869.6	66%	5305.7
120	100%	866.4	100%	693.0	65%	6428.4	37%	7788.2
150	100%	1175.5	100%	687.7	29%	8249.5	19%	8874.0

Table 3: Evaluation with "dense" problems ($k = 3, m = n \times (n-1)/4$)

	DB		DB+BC		AWC		HCC	
n	ratio	cycles	ratio	cycles	ratio	cycles	ratio	cycles
90	100%	31.2	100%	31.2	100%	9.9	100%	65.6
120	100%	34.6	100%	34.4	100%	9.3	100%	70.2
150	100%	34.9	100%	34.9	100%	9.6	100%	70.7

Table 4: Evaluation with "critical" problems ($k = 4, m = n \times 4.7$)

	DB		DB+BC		AWC		HCC	
n	ratio	cycles	ratio	cycles	ratio	cycles	ratio	cycles
60	100%	591.3	100%	497.1	100%	1733.6	61%	19953.8
90	100%	1175.8	100%	691.8	83%	14897.3	26%	32923.9
120	100%	2218.1	100%	1616.7	25%	34771.6	9%	38028.1

search algorithm. More specifically, when selecting a variable to modify its value, the weak-commitment search algorithm can choose any of constraint violating variables, while the breakout algorithm must choose a variable so that the number of constraint violations can be reduced. Therefore, in the worst case (when the current state is a local-minimum), the breakout algorithm has to check all the values for all constraint violating variables.

On the other hand, in the distributed breakout algorithm, the computations for each variable are performed concurrently by multiple agents. Therefore, the required computations of an agent for each cycle in the distributed breakout algorithm are basically equivalent to those in the asynchronous weak-commitment search algorithm. In other words, the potential parallelism in the breakout algorithm is greater than that in the weak-commitment search algorithm.

Increasing the weights in quasi-local-minima that are not real local-minima does not adversely affect the performance in "sparse" and "dense" problems. The performance is even increased in "sparse" problems. This result can be explained as follows. Assume x_i is in a quasi-local-minimum, while the current state is not a real local-minimum since x_j (which is a non-neighbor of x_i) can improve the evaluation value. If the constraints among agents are sparse, the effect of changing x_j's value to x_i would be relatively small. Therefore, the chance that x_i will be not in a quasi-local-minimum after x_j changes its value is very slim, i.e., eventually the state would be a real local-minimum. When the constraints are dense, the chance of trapped in a quasi or real local-minimum is very small.

In critically difficult problems, increasing the weights in quasi-local-minima that are not real local-minima does adversely affect the performance. For example, in the case where $n = 120, k = 4$, and $m = n \times 4.7$, the distributed breakout algorithm requires 40% more steps than the distributed breakout algorithm with broadcasting. However, the broadcasting algorithm requires a lot more (more than 10 times as many) messages than the original distributed breakout algorithm. Considering the cost of sending/receiving these additional messages, the performance degradation by increasing weights in quasi-local-minima seems to be acceptable.

The hill-climbing+coalition is not very efficient for all problems, especially for "critical" problems. One reason is that the current bound of the coalition size (i.e., 5) is too small for difficult problems, in which coalitions tend to be very large. However, a larger coalition consumes a great amount of constraint checks and degrades the overall performance.

Discussions

One drawback of the distributed breakout algorithm is that the completeness of the algorithm can not be guaranteed since it may fall into an infinite processing loop. We say that an algorithm is complete if the algorithm is guaranteed to find one solution eventually when solutions exist; and when there exists no solution, the algorithm is guaranteed to find out the fact that there exists no solution and terminate. The distributed breakout algorithm may fall into an infinite processing loop, so it can not guarantee that it finds a solution even if a solution does exist. On the other hand, the asynchronous weak-commitment search algorithm and the hill-climbing+coalition algorithm are guaranteed to be complete.

One advantage of the distributed breakout algorithm is that the termination detection procedure is embedded in the algorithm, while the other two algorithms must run separate procedures such as (Chandy & Lamport 1985) for termination detection.

In (Davenport *et al.* 1994), an algorithm similar to the breakout algorithm is implemented by connectionist architecture. In this algorithm, concurrent changes of neighboring clusters (which correspond to agents in distributed CSPs) are not prohibited. Therefore, there is a chance that the network of clusters will oscillate[1]. Furthermore, a local-minimum is detected by the fact the network as a whole is not changed in a certain time period. Such a global control is difficult to introduce into the distributed CSPs.

Conclusions

In this paper, we developed a new algorithm for solving distributed CSPs called the distributed breakout algorithm, which is inspired by the breakout algorithm for solving centralized CSPs. In this algorithm, each agent tries to minimize the number of constraint violations by exchanging the current value assignment and the amount of its possible improvement among neighboring agents. Instead of detecting a local-minimum, each agent detects a quasi-local-minimum, and changes the weights of constraint violations to escape from the quasi-local-minimum. Experiment evaluations showed that this algorithm is much more efficient than existing algorithms for particularly difficult problem instances.

Our future work includes showing the effectiveness of the distributed breakout algorithm in other example problems and practical applications such as channel allocation problems in mobile communication systems.

Acknowledgments

The initial idea of this research emerged during the discussions at a workshop of Multiagent Research community in Kansai (MARK). The authors wish to thank

[1] If the communications among clusters is fast and each cluster runs asynchronously, such an oscillation would not be very frequent.

members of MARK for their discussions, and Keihanna Interaction Plaza Inc. for supporting MARK. We also thank Koichi Matsuda, Nobuyasu Osato, Seiji Yamada, and Jun'ichi Toyoda for their support in this work.

References

Chandy, K., and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems* 3(1):63–75.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. 1991. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 331–337.

Collin, Z.; Dechter, R.; and Katz, S. 1991. On the feasibility of distributed constraint satisfaction. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 318–324.

Conry, S. E.; Kuwabara, K.; Lesser, V. R.; and Meyer, R. A. 1991. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man and Cybernetics* 21(6):1462–1477.

Davenport, A.; Tsang, E.; Wang, C. J.; and Zhu, K. 1994. Genet: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 325–330.

Haralick, R., and Elliot, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14:263–313.

Hirayama, K., and Toyoda, J. 1995. Forming coalitions for breaking deadlocks. In *Proceedings of the First international Conference on Multiagent Systems*, 155–162.

Huhns, M. N., and Bridgeland, D. M. 1991. Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics* 21(6):1437–1445.

Mackworth, A. K. 1992. Constraint satisfaction. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*. New York: Wiley-Interscience Publication. 285–293.

Mason, C., and Johnson, R. 1989. DATMS: A framework for distributed assumption based reasoning. In Gasser, L., and Huhns, M., eds., *Distributed Artificial Intelligence vol.II*. Morgan Kaufmann. 293–318.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. 1992. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58(1–3):161–205.

Morris, P. 1993. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 40–45.

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 440–446.

Sycara, K. P.; Roth, S.; Sadeh, N.; and Fox, M. 1991. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man and Cybernetics* 21(6):1446–1461.

Yokoo, M.; Durfee, E. H.; Ishida, T.; and Kuwabara, K. 1992. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the Twelfth IEEE International Conference on Distributed Computing Systems*, 614–621.

Yokoo, M. 1994. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 313–318.

Yokoo, M. 1995. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, 88–102. Springer-Verlag.

Zhang, Y., and Mackworth, A. 1991. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, 394–397.