Distributed Constraint Satisfaction Algorithm for Complex Local Problems

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

Katsutoshi Hirayama
Kobe University of Mercantile Marine
5-1-1 Fukae-minami-machi, Higashinada-ku, Kobe 658-0022, Japan
hirayama@ti.kshosen.ac.jp

Abstract

A distributed constraint satisfaction problem can formalize various application problems in MAS, and several algorithms for solving this problem have been developed. One limitation of these algorithms is that they assume each agent has only one local variable. Although simple modifications enable these algorithms to handle multiple local variables, obtained algorithms are neither efficient nor scalable to larger problems.

We develop a new algorithm that can handle multiple local variables efficiently, which is based on the asynchronous weak-commitment search algorithm. In this algorithm, a bad local solution can be modified without forcing other agents to exhaustively search local problems. Also, the number of interactions among agents can be decreased since agents communicate only when they find local solutions that satisfy all of the local constraints. Experimental evaluations show that this algorithm is far more efficient than an algorithm that uses the prioritization among agents.

1 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 [8]. In [13], 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 Multiagent Systems (MAS) that are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems [5], distributed scheduling problems [11], distributed interpretation tasks [9], and multi-agent truth maintenance tasks [7]) can be formalized as distributed CSPs. Therefore, we can consider distributed algorithms for solving distributed CSPs as an important infrastructure in MAS.

It must be noted that although algorithms for solving distributed CSPs seem to be similar to parallel/distributed processing methods for solving CSPs [4, 15], 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* [13], 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* [12], in which the priority order of agents is changed dynamically, and (c) distributed iterative improvement algorithms [6, 14].

One limitation of these algorithms is that they assume each agent has only one local variable. This assumption cannot be satisfied when the local problem of each agent becomes large and complex. Although these algorithms can be applied to the situation where one agent has multiple local variables by the following methods, both methods are neither efficient nor scalable to large problems.

Method 1: each agent finds all solutions of its local problem first.

By finding all solutions, the given problem can be reformalized as a distributed CSP, in which each agent has one local variable, whose domain is a set of obtained local solutions. Then, agents can apply algorithms for the case of a single local variable. The drawback of this method is that when a local problem becomes large and complex, finding all the solutions of a local problem becomes virtually impossible.

Method 2: an agent creates multiple virtual agents, each of which corresponds to one local variable, and simulates the activities of these virtual agents.
For example, if agent k has two local variables x_i, x_j, we assume that there exist two virtual agents, each of which corresponds to either x_i or x_j. Then, agent k simulates the concurrent activities of these two virtual agents. In this case, each agent does not have to predetermine all the local solutions. However, since communicating with other agents is usually more expensive than performing local computations, it is wasteful to simulate the activities of multiple virtual agents and not to distinguish the communications between virtual agents (that are within a single real agent) and the communications between real agents.

In [1], the prioritization among agents is introduced for handling multiple local variables. In this algorithm, each agent tries to find a local solution that is consistent with the local solutions of higher priority agents. If there exists no such local solution, backtracking or modification of the prioritization occurs. Various heuristics for determining good ordering among agents are examined [1].

One limitation of this approach is that, if a higher priority agent selects a bad local solution (i.e., a local solution that cannot be a part of a global solution), a lower priority agent must exhaustively search its local problem in order to change the bad decision made by the higher priority agent. When a local problem becomes large and complex, conducting such an exhaustive search becomes impossible. This approach is similar to that in method 1 described above, except that each agent searches for its local solutions only as required, instead of finding all solutions in advance. However, if the local solution selected by a higher priority agent is bad, a lower priority agent is forced to exhaustively search its local problem after all.

In this paper, we develop a new algorithm that is similar to that in method 2, but in this algorithm, an agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution that satisfies all local constraints. Experimental evaluations using example problems show that this algorithm is far more efficient than an algorithm that employs the prioritization among agents, or a simple extension of the asynchronous weak-commitment search algorithm for the case of a single local variable.

In the following, we briefly describe the definition of a distributed CSP and the asynchronous weak-commitment search algorithm for the case of a single local variable (Section 2). Then, we present the basic ideas and details of the asynchronous weak-commitment search algorithm for the case of multiple local variables (Section 3). Finally, we show empirical results that illustrate the efficiency of our newly developed algorithm (Section 4).

2 Distributed Constraint Satisfaction Problem

2.1 Formalization

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 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 the following communication model.

- Agents communicate by sending messages. An agent can send messages to other agents iff the agent knows the addresses of those agents.
- The delay in delivering a message is finite, though random. For the transmission between any pair of agents, messages are received in the order in which they were sent.

Each agent has some variables and tries to determine their values. However, there exist inter-agent constraints, and the value assignment must satisfy these inter-agent constraints. Formally, there exist m agents $1, 2, \ldots, m$. Each variable x_j belongs to one agent i (this relation is represented as *belongs*(x_j, i)). If x_j belongs to agent i, we call x_j a *local variable* of i. Constraints are also distributed among agents. The fact that an agent k knows a constraint predicate p_l is represented as *known*(p_l, k). We call a constraint defined only on local variables of one agent a *local constraint*.

We say that a distributed CSP is solved iff the following conditions are satisfied.

- $\forall$ i, $\forall x_j$ where belongs(x_j, i), the value of x_j is assigned to d_j,
  and $\forall$ k, $\forall p_l$ where known(p_l, k), p_l is true under the assignment $x_j = d_j$.

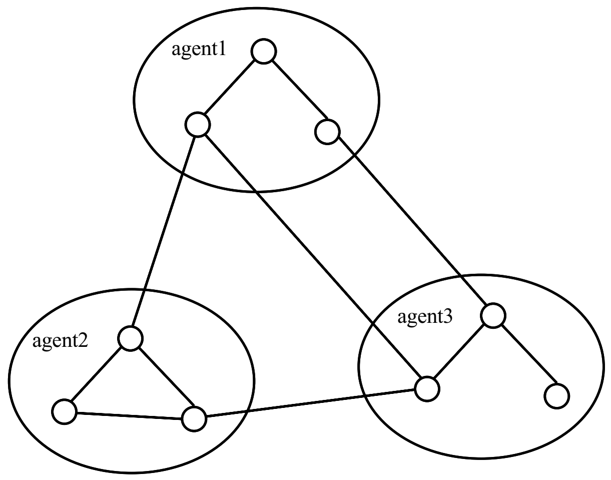

Figure 1. Example of a constraint network

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 knows all constraint predicates relevant to its variables.
- 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 (Figure 1). An agent can be represented as a set of variables, which is shown as a large circle in the figure.

2.2 Asynchronous Weak-Commitment Search Algorithm (single local variable)

In the asynchronous weak-commitment search algorithm (where each agent has exactly one local variable), each agent concurrently assigns a value to its variable, and sends the value to other agents. After that, agents wait for and respond to incoming messages. There are two kinds of messages: *ok?* messages to communicate the current value, and *nogood* messages to communicate information about constraint violations. The overview of the algorithm is given as follows.

- For each variable/agent, a non-negative integer value representing the priority order of the variable/agent is defined. We call this value the *priority value*.
- The order is defined such that any variable/agent with a larger priority value has higher priority.
- If the priority values of multiple agents are the same, the order is determined by the alphabetical order of the identifiers.
- For each variable/agent, the initial priority value is 0.
- After receiving an *ok?* message, an agent records the values of other agents in its *agent_view*. The *agent_view* represents the state of the world recognized by this agent.
- If the current value satisfies the constraints with higher priority agents in the *agent_view*, we say that the current value is consistent with the *agent_view*[1]. If the current value is not consistent with the *agent_view*, the agent selects a new value which is consistent with the *agent_view*.
- If there exists no consistent value for x_i, the priority value of x_i is changed to *max*+1, where *max* is the largest priority value of related agents. The agent sends *nogood* messages to relevant agents[2]. A *nogood* message contains a set of variable values that cannot be a part of any final solution.

By using this algorithm, if a solution exists, agents will reach a stable state where all constraints are satisfied. If there exists no solution, an empty nogood will be found and the algorithm will terminate[3].

3 Asynchronous Weak-Commitment Search Algorithm (multiple local variables)

3.1 Basic Ideas

We are going to modify the asynchronous weak-commitment search algorithm for the case of a single local variable by the following ways.

- An agent sequentially changes the values of its local variables. More specifically, it selects a variable x_k that has the highest priority among variables that are violating constraints with higher priority variables, and modifies x_k's value so that constraints with higher priority variables are satisfied.

[1] More precisely, the agent must satisfy not only initially given constraint predicates, but also the new constraints communicated by *nogood* messages.

[2] This procedure is necessary to guarantee the completeness of the algorithm. We can omit this procedure if the algorithm completeness is not required. Actually, when solving large-scale problems, the algorithm completeness has only theoretical importance.

[3] A set of variable values that is a *superset* of a nogood cannot be a final solution. If an empty set becomes a nogood, it means that there is no solution, since any set is a superset of an empty set.

- If there exists no value that satisfies all constraints with higher priority variables, the agent increases x_k's priority value.

- By iterating the above procedures, when all local variables satisfy constraints with higher priority variables, the agent sends changes to related agents.

Each variable must satisfy constraints with higher priority variables. Therefore, changing the value of a lower priority variable before the value of a higher priority variable is fixed is usually wasteful. Therefore, an agent changes the value of the highest priority variable first. Also, by sending messages to other agents only when an agent finds a consistent local solution, agents can reduce the number of interactions among agents.

By using this algorithm, if the local solution selected by a higher priority agent is bad, a lower priority agent does not have to exhaustively search its local problem. It simply increases the priority values of certain variables that violate constraints with the bad local solution.

3.2 Details of Algorithm

In the asynchronous weak-commitment search algorithm for the case of multiple local variables, each agent assigns values to its variables, and sends the values and the priority values to related agents. After that, agents wait for and respond to incoming messages[4]. In Figure 2, the procedures executed by agent i in receiving an *ok?* message are described[5].

In order to guarantee the completeness of the algorithm, the agent needs to record and communicate nogoods. Agents try to avoid situations previously found to be nogoods. However, due to the delay of messages, an *agent_view* of an agent can occasionally be a superset of a previously found nogood. In order to avoid reacting to unstable situations, and performing unnecessary changes to priority values, if an agent identifies an identical nogood it has already sent, the agent will not change the priority value but wait for the next message. By these procedures, the completeness of the algorithm is guaranteed, since the priority value of a variable is changed only when a new nogood is created.

[4]Although the following algorithm is described in a way that an agent reacts to messages sequentially, an agent can handle multiple messages concurrently, i.e., the agent first revises *agent_view* according to the messages, and performs **check_agent_view** only once.

[5]It must be mentioned that the way to determine that agents as a whole have reached a stable state is not contained in this algorithm. To detect a stable state, agents must use distributed termination detection algorithms such as [2].

```
when received (ok?, (sender_id, variable_id,
        variable_value, priority)) do
  add (sender_id, variable_id, variable_value, priority)
        to agent_view;
  when agent_view and current_assignments
        are not consistent do
    check_agent_view; end do;

procedure check_agent_view
  if agent_view and current_assignments
        are consistent then
    communicate changes to related agents;
  else select $x_k$, which has the highest priority and
      violating some constraint with
      higher priority variables;
    if no value in $D_k$ is consistent with
        agent_view and current_assignments then
      record and communicate a nogood, i.e., the subset
          of agent_view and current_assignments,
        where $x_k$ has no consistent value;
      when the obtained nogood is new do
        set $x_k$'s priority value to the highest priority
            value of related variables + 1;
        select $d \in D_k$ where $d$ minimizes the number of
          constraint violations with
              lower priority variables;
        set the value of $x_k$ to $d$;
        check_agent_view; end do;
    else select $d \in D_k$ where $d$ is consistent
        with agent_view and current_assignments,
        and minimizes the number of constraint
        violations with lower priority variables;
      set the value of $x_k$ to $d$;
      check_agent_view; end if; end if;
```

Figure 2. Procedure for handling *ok?* messages (asynchronous weak-commitment search for the case of multiple local variables)

3.3 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 goal is to assign a color to each node so that the nodes connected by a link have different colors. The possible colors for each node are black, white, or gray. There are two agents, i.e., agent1 and agent2, each of which has three variables.

We assume that the initial values are chosen as in Figure 3 (a). Each agent communicates these initial values via *ok?* messages. In the initial state, priority values of all variables are 0. Each agent checks whether the current value assignments are consistent with higher priority variables. Since the priority values are all equal, the priority order is determined by the alphabetical order of variable identifiers. Therefore, all variables of agent1 are ranked higher than those of agent2, so agent1 does not need to change the values of its variables.

On the other hand, for agent2, while x_4, which has the highest priority within agent2, satisfies all constraints with higher priority variables, x_5 does not satisfy the constraint between x_2. Therefore, agent2 changes x_5's value to gray, which satisfies the constraints between x_2 and x_4. By this change, the constraint between x_5 and x_6 is violated. Agent2 tries to change x_6's value, but there exists no value that satisfies all constraints since all colors are taken by higher priority variables (x_3 is black, x_4 is white, and x_5 is gray). Therefore, agent2 increases x_6's priority value to 1.

It changes x_6's value so that it satisfies as many constraints between lower priority variables as possible. In this case, each color violates one constraint, so agent2 randomly selects x_6's color (black is selected in this case). Also, agent2 records and communicates a nogood $\{(x_3,$ black), (x_4, white), (x_5, gray)$\}$, if the completeness of the algorithm is required. As a result, all variables of agent2 satisfy all constraints with higher priority variables, so it communicates the changes to agent1 (Figure 3 (b)).

Then, for agent1, while x_1 and x_2 satisfy constraints with higher priority variables, x_3 violates a constraint with x_6, which a priority value of 1. Therefore, agent1 changes x_3's value to gray, then a globally consistent solution is obtained (Figure 3 (c)).

Actually, there exists no local solution for agent2 that is consistent with agent1's initial local solution. Therefore, if we use the prioritization among agents, agent2 needs to exhaustively search its local problem. Conversely, in this algorithm, since a priority value is associated to each variable, and it is changed dynamically, a bad local solution can be modified without exhaustively searching a local problem.

4 Evaluations

In this section, we evaluate the efficiency of distributed constraint satisfaction algorithms using 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 a 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 order to compare the efficiency of our proposed algorithm (multiple local variables asynchronous weak-commitment search, multi-AWC), we use an algorithm that employs prioritization among agents, and the priorities are changed when one agent finds that there exists no consistent local solution with higher priority agents (asynchronous weak-commitment search with agent priority, AWC+AP), and an algorithm in which each agent simulates the activities of multiple virtual agents (single variable asynchronous weak-commitment search, single-AWC). AWC+AP is basically identical to the algorithm using the *decaying nogoods* heuristic described in [1]. However, in [1], agents are assumed to act in a sequential order. To make the comparison fair, we let agents act concurrently in AWC+AP. Also, each agent performs *min-conflict backtracking* [10] in AWC+AP.

We use a 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) cannot use the same channels to avoid interference.

A graph-coloring problem can be characterized by three parameters, i.e., the number of nodes/variables n, the number of possible colors for each nodes k, and the number of links between nodes l. A parameter called link density (l/n) affects the difficulty of a problem instance, and when $k = 3$, the setting $l/n = 2.7$ has been identified as a critical setting that produces particularly difficult problem instances [3].

In Table 1, we show the results where the number of agents m is 10, the number of possible colors k is 3, and the number of links l is set to $n \times 2.7$, varying the number of variables for each agent n/m. Each data point is the average of the trials for 100 randomly generated problem instances. Also, in Table 2, we show the results obtained by varying the number of agents m, while setting the number of variables for each agent to 10.

If we simply generate links at random, the number of links within an agent becomes very small. For example, if there exist 10 agents, each of which has 10 variables, al-

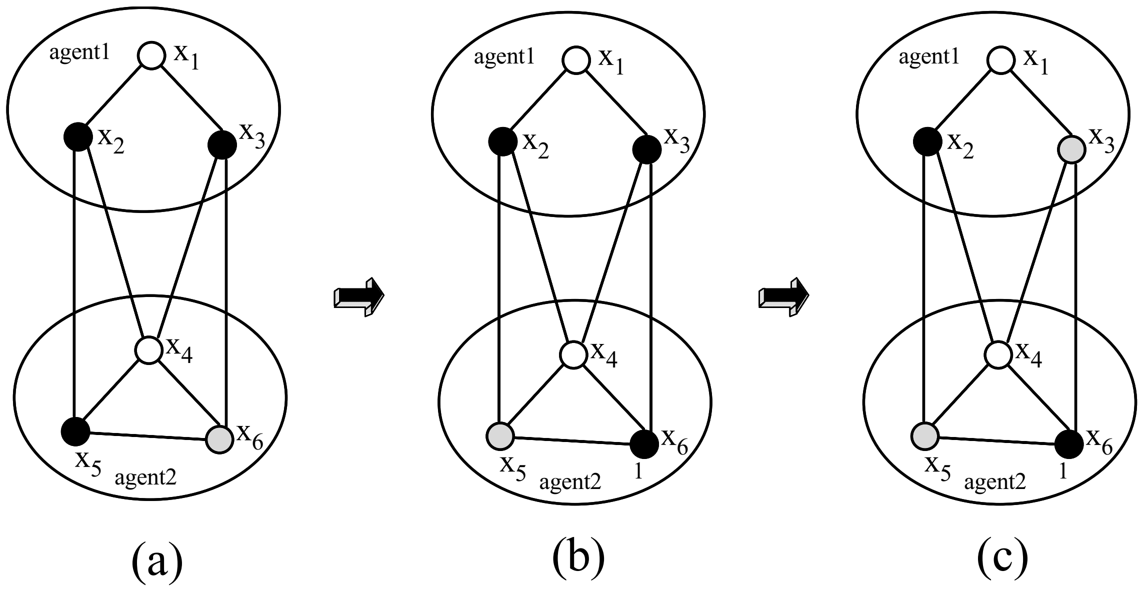

Figure 3. Example of algorithm execution

though there are 270 links in all, only less than 10% are local constrains. For each agent, there exist only two or three local constraints. Since a local problem of each agent should be a meaningful cluster of variables, it is natural to assume that local constraints should be at least as tight as inter-agent constraints. Therefore, we are going to assign half of the links to local constraints, and another half to inter-agent constraints. We randomly generate a problem with these parameter settings using the method described in [10], so that the graph is connected and the problem has a solution. The initial value of each variable is determined randomly.

To conduct the experiments within a reasonable amount of time, we limited the number of cycles to 10000 for each trial, and terminated the algorithm if this limit was exceeded; we counted the result as 10000. We show the ratio of trials successfully completed to the total number of trials in the table. Furthermore, to obtain an idea of how much local computation is performed, we measure the number of consistency checks. Namely, for each cycle, we select an agent that performs the most consistency checks (a bottleneck agent for each cycle), and show the summation of consistency checks for these bottleneck agents.

The following facts can be derived from these results.

- For all parameter settings in our experiments, multi-AWC outperforms single-AWC and AWC+AP (both in the number of required cycles and the number of consistency checks), and this becomes greater as the size of local problems or the number of agents increases.

- The number of required cycles for multi-AWC is smaller than for single-AWC, since in multi-AWC, agents communicate only when they find consistent local solutions. On the other hand, in single-AWC, each agent simply simulates the activities of multiple virtual agents, and agents communicate even if the values of the virtual agents within a single real agent are not consistent with the others. Although the number of consistency checks for each cycle in single-AWC is smaller than that of multiple-AWC, it never compensates for the increase in the number of required cycles.

- The number of required cycles for multi-AWC is smaller than for AWC+AP. At first glance, this result seems somewhat surprising, since in AWC+AP, each agent is so diligent that it exhaustively searches for its local problem to find a local solution consistent with higher priority agents, while in multi-AWC, each agent is rather lazy and tries to increase the priority values of its variables instead of trying to satisfy constraints with higher priority variables. However, in reality, diligently trying to find a consistent local solution with higher priority agents should not necessarily be good for agents as a whole. While the consistent local solution satisfies all constraints with higher priority agents, it may violate many constraints with lower priority agents. Therefore, the convergence to a global solution can be slower than with multi-AWC, where each agent simply increases the priority values

Table 1. Evaluation by varying the number of variables per agent n/m $(k = 3, l = 2.7 \times n, m = 10)$

	multi-AWC			AWC+AP			single-AWC		
n/m	ratio	cycles	checks	ratio	cycles	checks	ratio	cycles	checks
5	100%	26.9	2989.6	100%	35.9	3617.6	100%	323.0	13630.7
10	100%	89.5	22481.2	100%	577.7	155026.1	79%	4713.0	369195.0
15	100%	189.5	87688.8	79%	3951.8	1978801.9	14%	9083.4	1031475.1
20	100%	488.1	320312.6	37%	7529.6	6691615.5	0%	—	—

Table 2. Evaluation by varying the number of agents m $(k = 3, l = 2.7 \times n$**,** $n/m = 10)$

	multi-AWC			AWC+AP			single-AWC		
m	ratio	cycles	checks	ratio	cycles	checks	ratio	cycles	checks
10	100%	89.5	22481.2	100%	577.7	155026.1	79%	4713.0	369195.0
15	100%	214.7	62049.3	90%	3039.2	888422.6	10%	9573.6	779022.9
20	100%	615.6	190718.2	54%	6568.1	2083577.1	0%	—	—

of their variables, then tries to minimize the number of constraint violations as a whole.

Figure 4 shows the trace of the number of constraint violations when solving one problem instance with 10 agents, 10 variables. We can see that reducing the total number of constraint violations becomes rather difficult if agents devote too much energy to satisfying constraints with higher priority agents.

If each agent tries to find a consistent local solution that not only satisfies all constraints with higher priority agents, but also minimizes the number of constraint violations with lower priority agents, the convergence to a global solution can be hastened. However, it requires too many computations for an agent. Note that although each agent uses the min-conflict backtracking in AWC+AP, there is no guarantee that the obtained local solution minimizes the number of constraint violations with lower priority agents.

- In [1], more sophisticated heuristics for prioritization among agents are presented. However, the evaluation results in [1] show that the speedup obtained by employing these heuristics are not very drastic (at most two-fold) compared with the simple *decaying nogoods* heuristic used in AWC+AP. Therefore, we cannot assume that AWC+AP will outperform multi-AWC by employing more sophisticated prioritization heuristics when local problems are large.

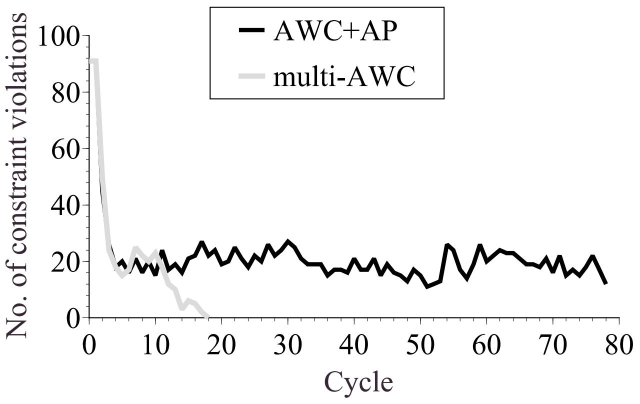

Figure 4. Traces of No. of constraint violations

5 Conclusions

In this paper, we developed a new algorithm that can efficiently solve a distributed CSP, in which each agent has multiple local variables. This algorithm is based on the asynchronous weak-commitment search algorithm for the case of a single local variable, but an agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution (which satisfies all local constraints). By using this algorithm, a bad local solution can be modified without forcing other agents to exhaustively search their local solutions, and the number of interactions among agents can be decreased. Experimental evaluations showed that this algorithm is far more efficient than the algorithm that employs the prioritization among agents, or a simple extension of the asynchronous weak-commitment search algorithm for the case of a single local variable.

Acknowledgments

The authors wish to thank Koichi Matsuda and Fumio Hattori for their supports in this work.

References

[1] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 620–625, 1997.

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

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

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

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

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

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

[8] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley-Interscience Publication, New York, 1992.

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

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

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

[12] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science 976)*, pages 88–102. Springer-Verlag, 1995.

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

[14] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 401–408. MIT Press, 1996.

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