Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS-2000), pp.169–177 (2000)

The Effect of Nogood Learning in Distributed Constraint Satisfaction*

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

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

Abstract

We present resolvent-based learning as a new nogood learning method for a distributed constraint satisfaction algorithm. This method is based on a look-back technique in constraint satisfaction algorithms and can efficiently make effective nogoods. We combine the method with the asynchronous weak-commitment search algorithm (AWC) and evaluate the performance of the resultant algorithm on distributed 3-coloring problems and distributed 3SAT problems. As a result, we found that the resolvent-based learning works well compared to previous learning methods for distributed constraint satisfaction algorithms. We also found that the AWC with the resolvent-based learning is able to find a solution with fewer cycles than the distributed breakout algorithm, which was known to be the most efficient algorithm (in terms of cycles) for solving distributed constraint satisfaction problems.

1. Introduction

A *distributed constraint satisfaction problem* [23, 24] is a constraint satisfaction problem (CSP) where variables and constraints are distributed among multiple agents. Even though the definition of a CSP is very simple, a surprisingly wide variety of problems in artificial intelligence can be formalized as CSPs. Similarly, various application problems in Multiagent Systems (MAS) that are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems [7], distributed scheduling problems [20], distributed interpretation tasks [16], and multi-agent truth maintenance tasks [14]) can be formalized as distributed CSPs. Therefore, we have considered an efficient distributed algorithm for solving a distributed CSP as an important infrastructure in MAS.

The author has proposed the *asynchronous weak-commitment search algorithm* (AWC) [22, 24] for solving a distributed CSP. All agents in this algorithm communicate their tentative values to their variables, and concurrently and asynchronously change the values to find consistent values. In the AWC, when an agent receives the latest information from another agent, it updates an *agent_view*, a list of 3-tuples: (*agent's id*, *variable's id*, *variable's value*), and tries to find a consistent value for its variable under the current agent_view. However, an agent meets a *deadend* under a certain agent_view. Namely, an agent fails to find a consistent value for the variable because the agent_view prohibits all possible values.

In the AWC, an agent makes a *nogood* at a deadend and sends it to relevant agents. A nogood is a subset of its agent_view under which no variable value is consistent. We can look at such a nogood as a new constraint that was not explicitly stated in the beginning. Thus, we refer to making (and recording) nogoods as *nogood learning* or just *learning*. When recording all discovered nogoods, the AWC is guaranteed to be complete, i.e., it finds a solution if one exists and insolubleness if none exists [22, 24].

In previous studies, some researchers presented the following learning methods for distributed constraint satisfaction algorithms.

- In the *asynchronous backtracking algorithm* (ABT) [23, 24], which is an ancestor of the AWC, an agent uses an agent_view itself as a nogood. The cost of this

*©2000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

method is virtually zero because it does not search in the subset space of the agent_view. However, the obtained nogood is not so effective.

- Consequently, the AWC presented in [22, 24] provides a selective or no learning strategy where an agent makes a limited number of nogoods.[1] Unfortunately, such a learning strategy makes the AWC incomplete.

- Mammen and Lesser use a method where an agent identifies a *minimum conflict set* in an agent_view and uses it as a nogood [15]. A minimum conflict set is the smallest subset of an agent_view that causes a deadend. This nogood can be the most effective one because it may prune a large portion of the search space. However, the cost of identifying such a set is usually very high.

Despite all these learning methods for distributed constraint satisfaction algorithms, none of the researchers have focused on their effects and fully investigated this issue. In this paper, we first provide *resolvent-based learning* as a new nogood learning method for a distributed constraint satisfaction algorithm, which is based on a look-back technique in the CSP literature [5, 10, 11, 19]. Then, we evaluate its performance through experiments on distributed 3-coloring problems and distributed 3SAT problems.

This paper is organized as follows. We first present the background of this work, which includes the definition of a distributed CSP and the outline of the AWC (Section 2). Next, we introduce the resolvent-based learning (Section 3), and then experimentally evaluate the AWC combined with the resolvent-based learning (Section 4). Finally, we conclude the work and discuss future directions (Section 5).

2. Background

2.1. Distributed Constraint Satisfaction Problem

A CSP consists of a set of *variables* and a set of *nogoods* (*constraints*). A variable has a finite and discrete *domain*, that is, a set of possible *values* for the variable. A nogood is a set of values for some variables stating that the set of values is prohibited for the variables. A solution to a CSP is a set of values for all variables violating no nogood. The goal of a CSP is to find a solution.

A distributed CSP is a CSP where variables and nogoods are distributed among multiple agents. The problem consists of:

- a set of *agents*, $1, 2, \ldots, l$
- a set of CSPs, $P_1, P_2, \ldots, P_l$, such that P_i belongs to an agent i.

We usually assume that a P_i includes all nogoods that are relevant to variables in P_i and such nogoods include *inter-agent nogoods*, which are defined over variables both in agent i and in some other agents. A solution to a distributed CSP is a set of solutions to all agents' CSPs. The goal of a distributed CSP is also to find a solution.

It is important that we do not confuse a distributed CSP with a method for solving a CSP in a distributed/parallel manner. If we want to solve a CSP in a distributed/parallel manner, we can choose any distribution of problems. On the other hand, since a distributed CSP is a problem for handling a MAS application problem, where multiple agents exist and have requirements for solving their local problems, the distribution of local problems is given in advance.

2.2. Asynchronous Weak-commitment Search Algorithm

To solve distributed CSPs, we could consider a centralized algorithm, where agents run some leader election distributed algorithm to elect one leader; agents send their local CSPs to the leader, and the leader finally solves the gathered CSPs using some constraint satisfaction algorithm while other agents are idle. If we were only interested in efficiency and not in other aspects, such a centralized algorithm might do well because it can make better use of the global knowledge of the entire problem. However, considering other aspects like privacy or security for example, we believe such an algorithm is not suitable for MAS application problems. Therefore, we have developed a series of distributed algorithms [13, 21, 22, 23, 24, 25, 26], where agents' knowledge of the entire problem stays limited throughout the execution of the algorithms.

Among these algorithms, the AWC is basically designed for a distributed CSP where an agent has a CSP with one variable. In the AWC, a *priority* is defined for each variable. An agent starts the algorithm by selecting some initial value to its variable and sending the variable's value and the variable's priority (initialized as zero) to relevant agents with *ok?* messages.

When receiving an *ok?* message, an agent i with a variable x_i updates its agent_view and tests whether some nogood is violated. The agent only performs this test for a nogood whose priority is higher than x_i's priority (we call such a nogood a *higher nogood*). The priority of a nogood is defined as the lowest priority among variables except x_i in the nogood. For example, suppose an agent 5 has a variable x_5 and a nogood: ((1, x_1, red)(2, x_2, green)(5, x_5, yellow)). Also suppose that the priority for x_1, x_2 and x_5 are 2, 1 and

[1] In the ABT, the only way for breaking a deadend is to make and send a nogood. On the other hand, an agent can break a deadend in the AWC by making and sending a nogood as well as by raising the priority of the deadend variable. Thus, the algorithm never gets stuck at a deadend even if an agent does not make a nogood.

0, respectively. In this case, the agent 5 has to test the nogood because the priority of the nogood (1) is higher than that of x_5 (0). All ties in priorities are broken due to the alphabetical order of variables' ids.

According to the test results, an agent i does the following.

- When no higher nogood is violated, an agent does nothing.
- When some higher nogoods are violated and the violation can be repaired by changing x_i's value, an agent changes the value and sends *ok?* messages. If there are multiple candidates for a new value, the agent selects the value causing the minimum violation on *lower nogoods*. The lower nogood is a nogood whose priority is lower than x_i's priority.
- When some higher nogoods are violated and the violation cannot be repaired, an agent makes a new nogood out of its agent_view and sends it with a *nogood* message to every agent that has the variable in the nogood. Then, the agent raises the priority of x_i, changes x_i's value to the one causing the minimum violation on all its nogoods, and sends *ok?* messages. If the new nogood is the same as the previously generated nogood, the agent does nothing. This step is required to ensure the completeness of the algorithm [22, 24].

When receiving a *nogood* message, an agent appends the nogood to its nogood set and performs the nogood violation test. If the new nogood includes an unknown variable, the agent has to request the corresponding agent to send its value.

3. Nogood Learning

Constraint satisfaction algorithms can be enhanced by *look-back* techniques, which exploit information about search that has already been done [2, 3, 5, 8, 9, 10, 11, 19]. This paper provides a new learning method for a distributed constraint satisfaction algorithm that is based on a look-back technique in the CSP literature.

3.1. Resolvent-based Learning

We use a similar method to that presented in [5, 10, 11, 19]. This method can be summarized as follows: for each possible value for a deadend variable, select one nogood that prohibits the value, then make a new nogood out of the aggregation of these selected nogoods. The nogood made in this way is virtually equivalent to a *resolvent* in the propositional logic, so we refer to this method as *resolvent-based learning*.

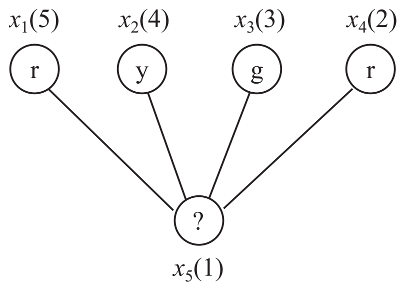

Figure 1. (A part of) Distributed 3-coloring problem

Suppose an agent i has a variable x_i with a domain D_i, and every possible value in D_i violates some higher nogoods under the current agent_view. An agent i first selects one nogood for each value $d \in D_i$ as follows.

1. An agent identifies higher nogoods that are violated under the current agent_view and $x_i = d$.
2. Next, it selects the smallest nogood among these nogoods. Ties are broken by selecting the one with the highest priority.

Then, the agent makes a new nogood by removing all of the elements including x_i from the union of these selected nogoods.

An agent selects the smallest nogood because we want to make the resultant nogood as small as possible. Furthermore, an agent selects the highest nogood when there are ties for the smallest nogood because such the highest nogood includes variables with high priorities. A highly-prioritized variable generally makes a strong commitment to the current value, so we should notify the agent with such a variable as early as possible if such a value is wrong.

3.2. Example

We illustrate the resolvent-based learning using Figure 1. Figure 1 shows a part of a distributed 3-coloring problem. In this problem, an agent i is responsible for one node x_i and the agent tries to paint the node red, yellow, or green, so that each adjacent pair of nodes has a different color. We also show a priority for a node (variable) in parentheses.

In this figure, agent 5 is going to select a color for node x_5. Suppose other variables, which are not shown in this figure, have lower priorities than x_5. Also suppose that agent 5 has the following nogoods that come from the arcs in the figure: $\{((x_1, \text{r})(x_5, \text{r})), ((x_1, \text{y})(x_5, \text{y})), ((x_1, \text{g})(x_5, \text{g})), \ldots,$ $((x_4, \text{g})(x_5, \text{g}))\}$, and the nogood received from some agent: $((x_3, \mathrm{g})(x_4, \mathrm{r})(x_5, \mathrm{y}))$ (an agent's id is omitted in a nogood). We can see that there is no consistent value for x_5 in this situation. In the resolvent-based learning, agent 5 makes a new nogood in the following way.

The value 'r' to x_5 will violate $((x_1, \mathrm{r})(x_5, \mathrm{r}))$ and $((x_4, \mathrm{r})(x_5, \mathrm{r}))$. Both have the same size, but their priorities are 5 and 2, respectively. Thus, the former nogood is selected for 'r'. Next, the value 'y' will violate $((x_2, \mathrm{y})(x_5, \mathrm{y}))$ and $((x_3, \mathrm{g})(x_4, \mathrm{r})(x_5, \mathrm{y}))$. The former nogood is selected for 'y' because it is smaller than the latter. Finally, the value 'g' will violate $((x_3, \mathrm{g})(x_5, \mathrm{g}))$ alone, so it is selected for 'g'. Agent 5 makes $((x_1, \mathrm{r})(x_2, \mathrm{y})(x_3, \mathrm{g}))$ as a new nogood from these selected nogoods.

4. Evaluation

We combine the resolvent-based learning with the AWC and evaluate the performance of the resultant algorithm through experiments on distributed 3-coloring problems and distributed 3SAT problems.

A distributed 3-coloring problem is a 3-coloring problem where n nodes (variables) and m arcs (constraints) are distributed among multiple agents. We generate a solvable problem instance with $m = 2.7n$ using the method in [17], and distribute one variable and its relevant nogoods to one agent. This setting is known to be hard in 3-coloring problems [6].[2] We generate 10 instances with this method for each $n \in \{60, 90, 120, 150\}$. For each instance, we randomly generate 10 sets of initial values for the variables. Thus, we make 100 trials for each n.

A distributed 3SAT is a 3SAT where n Boolean variables and m clauses are distributed among multiple agents. We use solvable 3SAT instances of the AIM problems [4], which are generated by 3SAT-GEN and 3ONESAT-GEN, and distribute one Boolean variable and its relevant clauses to one agent.

The 3SAT-GEN generates satisfiable 3SAT instances with a specified clause/variable ratio [4]. With this generator, we generate 25 instances with $m = 4.3n$ for each $n \in \{50, 100, 150\}$. These setting is hard enough according to experimental results in [4]. For each instance, we randomly generate 4 sets of initial values for variables. Thus, 100 trials are made for each n.

On the other hand, the 3ONESAT-GEN generates satisfiable 3SAT instances that have exactly one solution with a specified clause/variable ratio [4]. We use four instances with $m = 3.4n$ for each $n \in \{50, 100, 200\}$, which were got from the DIMACS benchmark site.[3] These instances are shown to be very hard for non-systematic search in [19]. Then we randomly generate 25 sets of initial values for variables in each instance. Thus, we also make 100 trials for each n.

Since the distributed algorithms used in these experiments are designed for a *fully asynchronous distributed system*, we can implement the algorithms on any distributed systems including a fully asynchronous distributed system itself. However, in our experiments, we assume a *synchronous distributed system* for simplicity and implement the algorithms on a simulator of such a distributed system. A synchronous distributed system is one of possible distributed systems, where all processes (agents) do their *cycles* synchronously. One cycle consists of activities so that all agents read incoming messages, do their local computation, and send messages to relevant agents.[4]

For each trial, we measure *cycle* (cycles consumed until a solution is found) and *maxcck* (sum of the maximal number of nogood checks performed by agents at each cycle) on the simulator. Broadly speaking, the former represents the communication cost of an algorithm and the latter represents the computational cost of an algorithm. We evaluate the performance of an algorithm with the averages of these measures over 100 trials for each n. We set the upper bound of cycles to 10000 and cut off a trial when it goes beyond this limit. If this happens, we use the data at that time.

4.1. Comparison with Other Learning Methods

We compare the resolvent-based learning with the following learning methods.

Mcs-based learning This method uses a minimum conflict set (mcs) as a new nogood like the method in [15]. Such a set is searched in this way: make a nogood with the resolvent-based learning and test whether a subset of the nogood is a conflict set or not from larger subsets to smaller subsets.

No learning In this method, an agent doesn't make a nogood when meeting deadends. This is also used in the AWC in [22, 24].

We combined the methods with the AWC and conducted experiments. The results are shown in Table 1-3. Note that '%' in the tables indicates the percentage of trials finished within the upper bound.

We first compare the resolvent-based learning (Rslv) and the mcs-based learning (Mcs). For distributed 3-coloring

[2] It is not clear for now where the hard instances lie for distributed 3-coloring problems (and also for distributed 3SAT problems). However, we can expect that an instance generated in this way, i.e., making a hard (centralized) instance and then distributing variables and nogoods, will be hard enough when every agent has one variable.

[3] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

[4] A computational load of one cycle varies according to algorithms. In addition, it varies during one run of some algorithms.

n	$learn$	$cycle$	$maxcck$	%
60	Rslv	83.2	58084.4	100
	Mcs	88.8	119019.2	100
	No	458.2	52601.6	100
90	Rslv	125.4	135569.8	100
	Mcs	133.2	275099.1	100
	No	2923.9	358486.1	91
120	Rslv	178.5	263115.1	100
	Mcs	172.3	494266.7	100
	No	6121.9	793280.3	60
150	Rslv	173.9	273823.3	100
	Mcs	177.1	512657.0	100
	No	8800.5	1188345.1	21

Table 1. Comparison with other learning methods on distributed 3-coloring problems

n	$learn$	$cycle$	$maxcck$	%
50	Rslv	125.0	76256.2	100
	Mcs	120.7	180122.0	100
	No	360.0	15959.3	100
100	Rslv	215.3	233003.8	100
	Mcs	238.9	830660.5	100
	No	3949.8	188182.3	80
150	Rslv	275.3	399146.6	100
	Mcs	286.0	1146204.1	100
	No	7793.8	382634.7	41

Table 2. Comparison with other learning methods on distributed 3SAT problems by 3SAT-GEN

n	$learn$	$cycle$	$maxcck$	%
50	Rslv	140.4	64011.0	100
	Mcs	120.3	90813.5	100
	No	1378.1	47784.3	62
100	Rslv	155.4	81086.1	100
	Mcs	138.2	132518.7	100
	No	9179.5	340172.3	14
200	Rslv	263.8	294334.5	100
	Mcs	237.4	544732.6	100
	No	-	-	0

Table 3. Comparison with other learning methods on distributed 3SAT problems by 3ONESAT-GEN

problems and distributed 3SAT problems by 3SAT-GEN, both methods are competitive for cycle, but the resolvent-based learning does very well for maxcck in all cases. As a result, the resolvent-based learning can make a good-quality nogood at a reduced computational cost for these problems. On the other hand, for distributed 3SAT problems by 3ONESAT-GEN, while the resolvent-based learning is always better for maxcck, it is slightly worse for cycle (10-15% larger). An instance of the problem has a relatively small number of clauses ($m = 3.4n$), but the clauses are selected to have only one solution. We expect that such an instance implicitly has many small-sized nogoods because all but one complete sets of values to variables are rejected by the small number of explicit clauses. With the mcs-based learning, small-sized nogoods are usually found at an early cycle and hence the communication cost is reduced. However, finding such nogoods by the mcs-based learning is computationally expensive.

Next we compare the resolvent-based learning (Rslv) with the no learning (No). The resolvent-based learning (and the mcs-based learning) overwhelmingly outperform the no learning for cycle. This tells us that the nogood learning has a great impact on the AWC's communication cost. To understand the reason for this, we measured the total number of redundant nogoods generated with the AWC using the following two methods.

Rslv/rec An agent makes a nogood with the resolvent-based learning, and other agents record the nogood.

Rslv/norec An agent makes a nogood with the resolvent-based learning, but no other agent records it.

The Rslv/rec is equivalent to the previously described resolvent-based learning. In the AWC with this method, the redundant generation of nogoods can still occur due to the concurrent activities of agents.

The results are shown in Table 4. We can see that an agent repeatedly makes the same nogoods if the previously generated nogoods are not recorded by other agents. However, such redundant generation dramatically declines when nogoods are recorded. Thus, we conjecture that learning helps agents adequately decide values to variables and thus reduces the communication cost.

For maxcck, on the other hand, we can see that the no learning is sometimes better than the resolvent-based learning. Since the no learning doesn't learn nogoods, its computational cost at each cycle is relatively low. However, if the size of problems increases, the no learning spends many cycles and consequently the maxcck of the no learning becomes larger than that of the resolvent-based learning.

problem	n	Rslv/rec	Rslv/norec
d3c	60	69.1	1612.3
	90	208.1	24399.3
	120	432.5	69784.6
	150	565.3	135502.5
d3s	50	195.3	1105.3
	100	908.0	42998.7
	150	1947.2	133162.6
d3s1	50	276.6	5523.3
	100	651.9	86595.8
	200	2683.4	190501.8

Table 4. Total number of redundant nogood generation (averaged over 100 trials) for distributed 3-coloring problems (d3c), distributed 3SAT problems by 3SAT-GEN (d3s) and distributed 3SAT problems by 3ONESAT-GEN (d3s1).

4.2. Size-bounded Learning

There is one drawback to the look-back techniques for constraint satisfaction algorithms. For a certain problem instance, a great number of nogoods can be produced, so the computational cost of checking nogoods may increase. We call such a problem *nogood-explosion*.

An agent in the AWC, on the other hand, only handles nogoods that are relevant to its variable, so the nogood-explosion for each agent is not so serious.[5] However, we cannot say that the AWC with nogood learning is completely free of the nogood-explosion.

In the CSP literature, two approaches, *size-bounded learning* [9, 10] and *relevance-bounded learning* [2] have been proposed to handle the nogood-explosion. The size-bounded learning is a simple strategy that bounds the size of the recorded nogood. The relevance-bounded learning, on the other hand, records nogoods of arbitrary size, but only maintains i-relevant nogoods, i.e., nogoods that differ from the working assignment in at most i variable-value pairs.

The relevance-bounded type of strategy cannot be applied to our learning method because the AWC is basically a non-systematic algorithm. Thus we consider that the size-bounded type of strategy is promising for our learning method. We test the AWC combined with *size-bounded resolvent-based learning*. KthRslv refers to the resolvent-based learning where agents only record the nogoods of size k or less.[6] Table 5-7 show the results for distributed 3-coloring problems and distributed 3SAT problems.

[5]The total number of nogoods over agents can be large because an agent sends a generated nogood to multiple agents.

[6]The size-bounded learning makes the AWC incomplete because it does not record all nogoods.

n	$learn$	$cycle$	$maxcck$	%
60	Rslv	83.2	58084.4	100
	3rdRslv	85.6	40594.2	100
	4thRslv	90.6	66622.4	100
90	Rslv	125.4	135569.8	100
	3rdRslv	126.4	76923.5	100
	4thRslv	136.0	151973.7	100
120	Rslv	178.5	263115.1	100
	3rdRslv	171.8	124226.1	100
	4thRslv	167.3	217033.4	100
150	Rslv	173.9	273823.3	100
	3rdRslv	186.1	153139.2	100
	4thRslv	180.4	249459.3	100

Table 5. AWC with size-bounded resolvent-based learning on distributed 3-coloring problems

n	$learn$	$cycle$	$maxcck$	%
50	Rslv	125.0	76256.2	100
	4thRslv	124.7	37717.9	100
	5thRslv	113.0	49770.3	100
100	Rslv	215.3	233003.8	100
	4thRslv	387.9	311048.8	100
	5thRslv	216.0	171115.7	100
150	Rslv	275.3	399146.6	100
	4thRslv	595.7	522191.2	100
	5thRslv	255.5	246534.5	100

Table 6. AWC with size-bounded resolvent-based learning on distributed 3SAT problems by 3SAT-GEN

For distributed 3-coloring problems, the 3rdRslv is competitive with the Rslv (the unrestricted resolvent-based learning) for cycle, but it performs better than the Rslv for maxcck. For distributed 3SAT problems by 3SAT-GEN, the 4thRslv is worse than the Rslv for both cycle and maxcck in instances with a large n. We conjecture that the instances of distributed 3SAT problems with a large n are so hard that we need to record larger nogoods. We should notice that the 5thRslv works well in such hard instances. For distributed 3SAT problems by 3ONESAT-GEN, the 4thRslv performs better for maxcck. Since the problem implicitly has many small nogoods, a large nogood is likely to become redundant after a smaller nogood is discovered. We can say that such redundant nogoods increase maxcck in the Rslv and the 5thRslv.

n	$learn$	$cycle$	$maxcck$	%
50	Rslv	140.4	64011.0	100
	4thRslv	130.8	38892.5	100
	5thRslv	128.9	46611.6	100
100	Rslv	155.4	81086.1	100
	4thRslv	167.8	68777.9	100
	5thRslv	162.8	84404.4	100
200	Rslv	263.8	294334.5	100
	4thRslv	265.7	181491.7	100
	5thRslv	272.6	290999.9	100

Table 7. AWC with size-bounded resolvent-based learning on distributed 3SAT problems by 3ONESAT-GEN

From these results, we can say that the optimal setting for k depends on problems. Since we do not have a way to determine it optimally for now, it should be set empirically. Generally speaking, making k smaller leads to lightening a computational load on agents in a cycle, but that may require a lot of cycles especially in hard problem instances. On the other hand, making k larger may burden agents with a relatively heavy computational load in a cycle, but that enables agents to solve hard problem instances with fewer cycles.

4.3. Comparison with Distributed Breakout Algorithm

The authors have presented the *distributed breakout algorithm* (DB) for solving distributed CSPs and experimental results that show the DB is very efficient (in terms of cycles) especially for difficult problem instances with solutions [25].

This algorithm is characterized by concurrent hill-climbing while excluding neighbors' simultaneous value changes [12] and the *breakout* strategy [18] as a method for escaping from *quasi-local-minima* (a weak notion of a real local-minimum). In the DB, each agent first initializes its variable value arbitrarily, sends its value to neighbors with $ok?$ messages, and then repeats the following:

- when receiving $ok?$ messages from all neighbors, an agent measures the cost of the current variable value as a weighted sum of violated constraints and its possible maximal improvement (called $improve$). Note that a weight, a positive integer, is defined for each constraint. After this calculation, an agent sends these results to all neighbors with $improve$ messages.

- when receiving $improve$ messages from all neighbors, an agent compares each of them with its own $improve$, and transfers the right to change its variable value by skipping its next change if the neighbor's $improve$ is greater than its own $improve$ or does not transfer this right if it's smaller. Ties are broken by comparing agent identifiers. Only the winners for the right to change actually change their variable values, and then all agents send the current variable values to neighbors with $ok?$ messages.

This repeated process sometimes leads to a solution to a distributed CSP. However, in many cases some agent falls into a quasi-local-minimum, where it has at least one constraint violation and has no way to improve the cost. In that case, an agent escapes from a quasi-local-minimum by the breakout strategy, i.e., increasing weights of violated constraints at the quasi-local-minimum.

We compare the DB and the AWC combined with the most effective resolvent-based learning (3rdRslv for distributed 3-coloring problems, 5thRslv for distributed 3SAT problems by 3SAT-GEN, and 4thRslv for distributed 3SAT problems by 3ONESAT-GEN). Table 8-10 show the results.[7]

The AWC combined with the size-bounded resolvent-based learning is worse for maxcck in all cases. Since the DB does not learn nogoods, the number of nogoods in an agent never increases. This means that the load of local computation for the DB is very small compared to the AWC combined with the size-bounded resolvent-based learning. On the other hand, the AWC combined with the size-bounded resolvent-based learning is better for cycle in all cases. This is because agents in the DB use special messages ($improve$ messages) to mutually exclude their value changes, and thus extra cycles are spent.

An answer to the question, which algorithm is efficient?, depends on a characteristic of the distributed system, the ratio of the communication delay to the computational time-unit. Figure 2 shows the rough estimation of the efficiency of both algorithms for $n = 50$ of distributed 3SAT problems by 3ONESAT-GEN. We assume that one nogood check amounts to one computational time-unit and a communication delay between cycles amounts to the designated number of time-unit. The figure illustrates total number of time-unit vs. communication delay when each algorithm consumes cycle and maxcck shown in Table 10. For these problems, when a communication delay is more than around 50 time-unit (i.e., 50 nogood checks), the AWC+4thRslv seems to become efficient. However, the point at which the AWC+$k$thRslv becomes efficient varies according to the problems. For example, this point is around

[7]The DB in this work is slightly different from [25]. The DB requires a *weight* of a constraint, which reflects a cost violating the constraint. When solving a distributed graph-coloring problem, the DB in [25] assigns a weight to a pair of variables. The DB in this work assigns it to a nogood. Our experiments showed that the latter is better.

n	alg	$cycle$	$maxcck$	$\%$
60	AWC+3rdRslv	85.6	40594.2	100
	DB	164.9	7730.0	100
90	AWC+3rdRslv	126.4	76923.5	100
	DB	282.1	14228.5	100
120	AWC+3rdRslv	171.8	124226.1	100
	DB	522.4	26931.5	100
150	AWC+3rdRslv	186.1	153139.2	100
	DB	523.7	29207.0	100

Table 8. Comparison with distributed breakout algorithm on distributed 3-coloring problems

n	alg	$cycle$	$maxcck$	$\%$
50	AWC+5thRslv	113.0	49770.3	100
	DB	322.6	6461.3	100
100	AWC+5thRslv	216.0	171115.7	100
	DB	847.2	19870.8	100
150	AWC+5thRslv	255.5	246534.5	100
	DB	1257.2	31717.2	100

Table 9. Comparison with distributed breakout algorithm on distributed 3SAT problems by 3SAT-GEN

n	alg	$cycle$	$maxcck$	$\%$
50	AWC+4thRslv	130.8	38892.5	100
	DB	690.1	11691.1	100
100	AWC+4thRslv	167.8	68777.9	100
	DB	1917.4	38210.5	97
200	AWC+4thRslv	265.7	181491.7	100
	DB	5246.5	117277.4	69

Table 10. Comparison with distributed breakout algorithm on distributed 3SAT problems by 3ONESAT-GEN

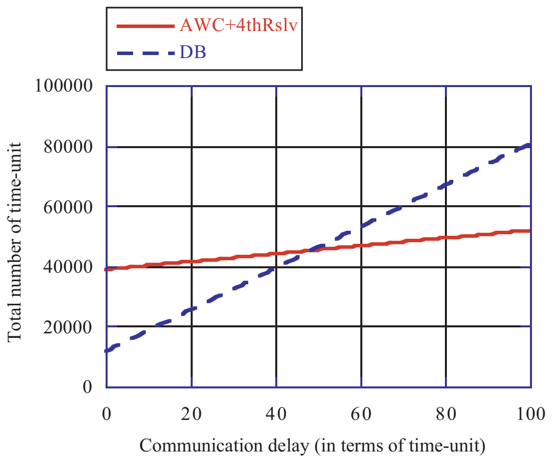

Figure 2. Estimated efficiency on $n = 50$ of distributed 3SAT problems by 3ONESAT-GEN (Note: one nogood check for one time-unit)

210 time-unit for distributed 3SAT problems by 3SAT-GEN with $n = 150$ and around 370 time-unit for distributed 3-coloring problems with $n = 150$.

5. Conclusions and Future Work

We have presented the resolvent-based learning, which is based on a look-back technique in the CSP literature, and combined it with the asynchronous weak-commitment search algorithm. Such a combination is promising because our experimental results show that:

- The learning methods (both the mcs- and resolvent-based learning) dramatically reduce cycles consumed to find a solution to a distributed CSP.
- The resolvent-based learning can produce an effective nogood with fewer nogood checks.
- By introducing the size-bounded strategy, the number of nogood checks can be reduced without having a bad effect on cycle.
- The AWC with the resolvent-based learning can be more efficient than the DB when the communication delay is large.

Finally, we wish to point out some matters for further investigation. Our discussion was made on one specific class of distributed CSPs, where each agent has one variable. Although all distributed CSPs can be converted into this class in principle, such conversion is sometimes unreasonable in real-life problems [1, 26]. The authors have proposed a few extended versions of the AWC to handle a problem with multi-variables per agent [26]. Perhaps, it is easy to introduce our learning method into these algorithms as well. We may be able to develop the algorithms and do further analyses.

Experimental analyses in this work are done on a synchronous distributed system for simplicity. As mentioned before, our distributed constraint satisfaction algorithms are designed for a fully asynchronous distributed system, and thereby can work on any type of distributed systems. We should analyze the performance of our algorithm on other types of distributed systems.

References

[1] Armstrong, A. and Durfee, E.: Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (1997) 620–625

[2] Bayardo, R. J. and Miranker, D. P.: A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (1996) 298–304

[3] Bayardo, R. J. and Schrag, R. C.: Using CSP Look-Back Techniques to Solve Real-World SAT Instances. *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (1997) 203–208

[4] Cha, B. and Iwama, K.: Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (1995) 304–310

[5] Cha, B. and Iwama, K.: Adding New Clauses for Faster Local Search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (1996) 332–337

[6] Cheeseman, P., Kanefsky, B., and Taylor, W.: Where the Really Hard Problems Are. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (1991) 331–337

[7] Conry, S. E., Kuwabara, K., Lesser, V. R., and Meyer, R. A.: Multistage Negotiation for Distributed Constraint Satisfaction. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6) (1991) 1462–1477

[8] de Kleer, J.: A Comparison of ATMS and CSP techniques. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989) 290–296

[9] Dechter, R.: Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41 (1990) 273–312

[10] Frost, D and Dechter, R.: Dead-end driven Learning. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994) 294–300

[11] Ginsberg, M. L.: Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1 (1993) 25–46

[12] Hirayama, K. and Toyoda, J.: Forming Coalitions for Breaking Deadlocks. *Proceedings of the First International Conference on Multi-Agent Systems* (1995) 155–162

[13] Hirayama, K. and Yokoo, M.: Distributed Partial Constraint Satisfaction Problem. In: Smolka, G. (ed.): *Principles and Practice of Constraint Programming –CP97*. Lecture Notes in Computer Science, Vol.1330. Springer-Verlag (1997) 222–236

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

[15] Mammen, D. L. and Lesser, V. R.: Problem Structure and Subproblem Sharing in Multi-Agent Systems. *Proceedings of the Third International Conference on Multi-Agent Systems* (1998) 174–181

[16] Mason, C. and Johnson, R.: DATMS: A Framework for Distributed Assumption based Reasoning. In: Gasser, L. and Huhns, M. (eds.): *Distributed Artificial Intelligence*, Vol.2. Morgan Kaufmann (1989) 293–318

[17] Minton, S., Johnston, M. D., Philips, A. B., and Laird, P.: Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1–3) (1992) 161–205

[18] Morris, P.: The Breakout Method for Escaping From Local Minima. *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 40–45

[19] Richards, E. T. and Richards, B.: Non-systematic Search and Learning: An empirical study. In: Maher, M. and Puget, J.-F. (eds.) *Principles and Practice of Constraint Programming –CP98*, Lecture Notes in Computer Science, Vol.1520. Springer-Verlag (1998) 370–384

[20] Sycara, K. P., Roth, S., Sadeh, N., and Fox, M.: Distributed Constrained Heuristic Search. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6) (1991) 1446–1461

[21] Yokoo, M.: Constraint Relaxation in Distributed Constraint Satisfaction Problem. *Proceedings of the Fifth International Conference on Tools with Artificial Intelligence* (1993) 56–63

[22] Yokoo, M.: Asynchronous Weak-Commitment Search for Solving Distributed Constraint Satisfaction Problems. *Principles and Practice of Constraint Programming –CP95*, Lecture Notes in Computer Science, Vol.976. Springer-Verlag (1995) 88–102

[23] Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K.: Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *Proceedings of the Twelfth IEEE International Conference on Distributed Computing Systems* (1992) 614–621

[24] Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K.: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5) (1998) 673–685

[25] Yokoo, M. and Hirayama, K.: Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. *Proceedings of the Second International Conference on Multi-Agent Systems* (1996) 401–408

[26] Yokoo, M and Hirayama, K.: Distributed Constraint Satisfaction Algorithm for Complex Local Problems. *Proceedings of the Third International Conference on Multi-Agent Systems* (1998) 372–379