International Conference on Principles and Practice of Constraint Programming
1995, pp.88–102

Asynchronous Weak-commitment Search 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

Abstract. A distributed constraint satisfaction problem (Distributed CSP) is a CSP in which variables and constraints are distributed among multiple automated agents, and various application problems in Distributed Artificial Intelligence can be formalized as Distributed CSPs. We develop a new algorithm for solving Distributed CSPs called *asynchronous weak-commitment search*, which is inspired by the weak-commitment search algorithm for solving CSPs. This algorithm can revise a bad decision without an exhaustive search by changing the priority order of agents dynamically. Furthermore, agents can act asynchronously and concurrently based on their local knowledge without any global control, while guaranteeing the completeness of the algorithm.
The experimental results on various example problems show that this algorithm is by far more efficient than the asynchronous backtracking algorithm for solving Distributed CSPs, in which the priority order is static. The priority order represents a hierarchy of agent authority, i.e., the priority of decision making. Therefore, these results imply that a flexible agent organization, in which the hierarchical order is changed dynamically, actually performs better than an organization in which the hierarchical order is static and rigid.

1 Introduction

Distributed Artificial Intelligence (DAI) is a subfield of AI that is concerned with the interaction, especially the coordination among artificial automated agents. Since distributed computing environments are spreading very rapidly due to the advances in hardware and networking technologies, there are pressing needs for DAI techniques, thus DAI is becoming a very vital area in AI.

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. It is well known that surprisingly a wide variety of AI problems can be formalized as CSPs. Similarly, various application problems in DAI which are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems [3], distributed scheduling problems [9], and multi-agent truth maintenance tasks [5]) 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 [2, 14], 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 the 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. If all knowledge about the problem can be gathered into one agent, this agent can solve the problem alone using normal centralized constraint satisfaction algorithms. However, collecting all information about a problem requires certain communication costs, which could be prohibitively high. Furthermore, in some application problems, gathering all information to one agent is not desirable or impossible for security/privacy reasons. In such cases, multiple agents have to solve the problem without centralizing all information. The author has developed a basic algorithm for solving Distributed CSPs called *asynchronous backtracking* [13]. In this algorithm, agents act asynchronously and concurrently based on their local knowledge without any global control.

In this paper, we develop a new algorithm called *asynchronous weak-commitment search*, which is inspired by the weak-commitment search algorithm for solving CSPs [12]. The main characteristic of this algorithm is as follows.

- Agents can revise a bad decision without an exhaustive search by changing the priority order of agents dynamically.

In the asynchronous backtracking algorithm, the priority order of agents is determined, and an agent tries to find a value satisfying the constraints with the variables of higher priority agents. When an agent sets a variable value, the agent commits to the selected value strongly, i.e., the selected value will not be changed unless an exhaustive search is performed by lower priority agents. Therefore, in large-scale problems, a single mistake of value selection becomes fatal since doing such an exhaustive search is virtually impossible. This drawback is common to all backtracking algorithms. On the other hand, in the asynchronous weak-commitment search, when an agent can not find a value consistent with the higher priority agents, the priority order is changed so that the agent has the highest priority. As a result, when an agent makes a mistake in value selection, the priority of another agent becomes higher; thus the agent that made the mistake will not commit to the bad decision, and the selected value will be changed.

We will show that the asynchronous weak-commitment search algorithm can solve problems such as the distributed 1000-queens problem, the distributed graph-coloring problem, and the network resource allocation problem [8] that the asynchronous backtracking algorithm fails to solve within a reasonable amount of time. We can assume that the priority order represents a hierarchy of agent authority, i.e., the priority order of decision making. Therefore, these results imply that a flexible agent organization, in which the hierarchical order is changed dynamically, actually performs better than an organization in which the hierarchical order is static and rigid.

In the following, we briefly describe the definition of a Distributed CSP and the asynchronous backtracking algorithm (Section 2). Then, we show the basic ideas and details of the asynchronous weak-commitment search algorithm (Section 3), and empirical results which show the efficiency of the algorithm (Section 4). Finally, we examine the complexity of the algorithm (Section 5).

2 Distributed Constraint Satisfaction Problem and Asynchronous Backtracking

2.1 Formalization

A CSP consists of n variables $x_1, x_2, ..., x_n$, whose values are taken from finite, discrete domains $D_1, D_2, ..., D_n$ respectively, and a set of constraints on their values. A constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \ldots, x_{kj})$ 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 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 the agents[1].
- 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 $\mathit{belongs}(x_j, i)$). Constraints are also distributed among agents. The fact that an agent k knows a constraint predicate p_l is represented as $\mathit{known}(p_l, k)$.

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

[1] This model does not necessarily mean that the physical communication network must be fully connected (i.e., a complete graph). Unlike most parallel/distributed algorithm studies, in which the topology of the physical communication network plays an important role, we assume the existence of a reliable underlying communication structure among agents and do not care about the implementation of the physical communication network. This is because our primary concern is the cooperation of intelligent agents, rather than solving CSPs by certain multi-processor architectures.

– $\forall$ 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 $x_j = d_j$.

2.2 Asynchronous Backtracking

A basic algorithm for solving Distributed CSPs called *asynchronous backtracking* is developed in [13]. In this algorithm, agents act asynchronously and concurrently, in contrast to traditional sequential backtracking techniques, while guaranteeing the completeness of the algorithm.

Without loss of generality, we make the following assumptions while describing our algorithms for simplicity. Relaxing these assumptions to general cases is relatively straightforward[2].

– Each agent has exactly one variable.
– All constraints are binary.
– There exists a constraint between any pair of agents.
– Each agent knows all constraint predicates relevant to its variable.

In the following, we use the same identifier x_i to represent an agent and its variable. We assume that each agent (and its variable) has a unique identifier.

In the asynchronous backtracking algorithm, 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 procedures executed at agent x_i by receiving an *ok?* message and a *nogood* message are described in Fig. 1. An overview of these procedures is given as follows.

– 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 (Fig. 1 (i)).
– The priority order of variables/agents is determined by the alphabetical order of the identifiers, i.e., preceding variables/agents in the alphabetical order have higher priority. 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*[3]. If the current value is not consistent with the *agent_view*, the agent selects a new value which is consistent with the *agent_view* (Fig. 1 (ii)).

[2] In [11], an algorithm in which each agent has multiple variables is described. If there exists no explicit constraint between two agents, there is a chance that an implicit constraint exists between them. In such a case, a new relation between these agents must be generated dynamically. The procedure for adding a new relation is described in [13]. The idea of making originally implicit constraints explicit can be found in the consistency algorithms for CSPs. For example, the adaptive consistency procedure [4] adds links to a constraint network while transforming the constraint network to a backtrack-free constraint network.

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

– If the agent can not find a value consistent with the *agent_view*, the agent sends *nogood* messages to higher priority agents (Fig. 1 (iii)). A *nogood* message contains a set of variable values that can not 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[4].

```
when received (ok?, ($x_j$, $d_j$)) do — (i)
  add ($x_j$, $d_j$) to agent_view;
  check_agent_view;
end do;

when received (nogood, $x_j$, nogood) do
  add nogood to nogood_list;
  check_agent_view;
end do;

procedure check_agent_view
  when agent_view and current_value are not consistent do — (ii)
    if no value in $D_i$ is consistent with agent_view then backtrack; — (iii)
    else select $d \in D_i$ where agent_view and $d$ are consistent;
      current_value $\leftarrow d$;
      send (ok?, ($x_i$, $d$)) to other agents; end if; end do;

procedure backtrack
  nogoods $\leftarrow \{V \mid V=$ inconsistent subset of agent_view$\}$;
  when an empty set is an element of nogoods do
      broadcast to other agents that there is no solution,
          terminate this algorithm; end do;
  for each $V \in$ nogoods do;
      select $(x_j, d_j)$ where $x_j$ has the lowest priority in $V$;
      send (nogood, $x_i$, $V$) to $x_j$;
  end do;
```

Fig. 1. Procedures for receiving messages (asynchronous backtracking)

[4] A set of variable values that is a *superset* of a nogood can not 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.

3 Asynchronous Weak-commitment Search

In this section, we briefly describe the weak-commitment search algorithm for solving CSPs [12], and describe how the asynchronous backtracking algorithm can be modified into the asynchronous weak-commitment search algorithm.

3.1 Weak-commitment Search Algorithm

In the weak-commitment search algorithm (Fig. 2), all variables have tentative initial values. We can execute this algorithm by calling **weak-commitment** $(\{(x_1, d_1), (x_2, d_2), \ldots, (x_n, d_n)\}, \{\})$, where d_i is the tentative initial value of x_i. In this algorithm, a consistent partial solution is constructed for a subset of variables, and this partial solution is extended by adding variables one by one until a complete solution is found. When a variable is added to the partial solution, its tentative initial value is revised so that the new value satisfies all constraints between the partial solution, and satisfies as many constraints between variables that are not included in the partial solution as possible. This value ordering heuristic is called the *min-conflict* heuristic [6]. The essential difference between this algorithm and the min-conflict backtracking [6] is the underlined part in Fig. 2. When there exists no value for one variable that satisfies all constraints between the partial solution, this algorithm abandons the whole partial solution, and starts constructing a new partial solution from scratch, using the current value assignment as new tentative initial values.

This algorithm records the abandoned partial solutions as new constraints, and avoids creating the same partial solution that has been created and abandoned before. Therefore, the completeness of the algorithm (always finds a solution if one exists, and terminates if no solution exists) is guaranteed. The experimental results on various example problems in [12] show that this algorithm is 3 to 10 times more efficient than the min-conflict backtracking [6] or the breakout algorithm [7].

3.2 Basic Ideas

The main characteristics of the weak-commitment search algorithm are as follows.

1. The algorithm uses the min-conflict heuristic as a value ordering heuristic.
2. It abandons the partial solution and restarts the search process if there exists no consistent value with the partial solution.

Introducing the first characteristic into the asynchronous backtracking algorithm is relatively straightforward. When selecting a variable value, if there exist multiple values consistent with the *agent_view* (those that satisfy all constraints with variables of higher priority agents), the agent prefers the value that minimizes the number of constraint violations with variables of lower priority agents.

In contrast, introducing the second characteristic into the asynchronous backtracking is not straightforward, since agents act concurrently and asynchronously,

```
procedure weak-commitment(left, partial-solution)
  when all variables in left satisfy all constraints do
      terminate the algorithm, current value assignment is a solution; end do
  $(x_i, d) \leftarrow$ a variable and value pair in left that does not satisfy some constraint;
  values $\leftarrow$ the list of $x_i$'s values that are consistent with partial-solution;
  if values is an empty list;
    if partial-solution is an empty list
      then terminate the algorithm since there exists no solution;
      else record partial-solution as a new constraint (nogood);
        remove each element of partial-solution and add to left;
        call weak-commitment(left, partial-solution); end if;
    else value $\leftarrow$ the value within values that minimizes
        the number of constraint violations with left;
      remove $(x_i, d)$ from left;
      add $(x_i, value)$ to partial-solution;
      call weak-commitment(left, partial-solution); end if;
```

Fig. 2. Weak-commitment search algorithm

and no agent has exact information about the partial solution. Furthermore, multiple agents may try to restart the search process simultaneously.

In the following, we show that the agents can commit to their decisions weakly by changing the priority order dynamically. We define the way of establishing the priority order by introducing *priority values*, and change the priority values by the following rules.

- 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.
- If there exists no consistent value for x_l, the priority value of x_l is changed to $k + 1$, where k is the largest priority value of other agents.

Furthermore, in the asynchronous backtracking algorithm, 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 of priority values, each agent performs the following procedure.

- Each agent records the nogoods that it has sent. When the *agent_view* is a superset of a nogood that it has already sent, the agent will not change the priority value and waits for the next message.

3.3 Details of Algorithm

In the asynchronous weak-commitment search, 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[5]. In Fig. 3, the procedures executed at agent x_i by receiving an *ok?* message and a *nogood* message are described[6]. The differences between these procedures and the procedures for the asynchronous backtracking algorithm are as follows.

- The priority value, as well as the current value, is communicated through the *ok?* message (Fig. 3 (i)).
- The priority order is determined by the communicated priority values. If the current value is not consistent with the *agent_view*, i.e., some constraint with variables of higher priority agents is not satisfied, the agent changes its value so that the value is consistent with the *agent_view*, and also the value minimizes the number of constraint violations with variables of lower priority agents (Fig. 3 (ii)).
- When x_i can not find a consistent value with its *agent_view*, x_i sends *nogood* messages to other agents, and increments its priority value. If x_i has already sent an identical nogood, x_i will not change the priority value and will wait for the next message (Fig. 3 (iii)).

3.4 Example of Algorithm Execution

We illustrate the execution of the algorithm using a distributed version of the well-known n-queens problem (where n=4). There exist four agents, each of which corresponds to a queen of each row. The goal of the agents is to find positions on a 4×4 chess board so that they do not threaten one another.

The initial values are shown in Fig. 4 (a). Agents communicate these values to one another. The values within parentheses represent the priority values. The initial priority values are 0. Since the priority values are equal, the priority order is determined by the alphabetical order of identifiers. Therefore, only the value of x_4 is not consistent with its *agent_view*, i.e., only x_4 is violating constraints with higher priority agents. Since there is no consistent value, agent x_4 sends *nogood* messages and increments its priority value. In this case, the value minimizing the number of constraint violations is 3, since it conflicts only with x_3. Therefore, x_4 selects 3 and sends *ok?* messages to other agents (Fig. 4 (b)). Then, x_3 tries to change its value. Since there is no consistent value, agent x_3 sends *nogood* messages, and increments its priority value. In this case, the value that minimizes

[5] 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* and *nogood_list* according to the messages, and performs **check_agent_view** only once.

[6] 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 the stable state, agents must use distributed termination detection algorithms such as [1].

```
**when received** (**ok?**, ($x_j$, $d_j$, *priority*)) **do** — (i)
  add ($x_j$, $d_j$, *priority*) to *agent_view*;
  **check_agent_view**;
**end do**;

**when received** (**nogood**, $x_j$, *nogood*) **do**
  add *nogood* to *nogood_list*;
  **check_agent_view**;
**end do**;

procedure **check_agent_view**
  **when** *agent_view* and *current_value* are not consistent **do**
    **if** no value in $D_i$ is consistent with *agent_view* **then backtrack**;
    **else** select $d \in D_i$ where *agent_view* and $d$ are consistent
        and $d$ minimizes the number of constraint violations; — (ii)
      *current_value* $\leftarrow d$;
      send (**ok?**, ($x_i$, $d$, *current_priority*)) to other agents; **end if**; **end do**;

procedure **backtrack** — (iii)
  *nogoods* $\leftarrow \{V \mid V =$ inconsistent subset of *agent_view*$\}$;
  **when** an empty set is an element of *nogoods* **do**
      broadcast to other agents that there is no solution,
          terminate this algorithm; **end do**;
  **when** no element of *nogoods* is included in *nogood_sent* **do**
      **for each** $V \in$ *nogoods* **do**;
          add $V$ to *nogood_sent*
          **for each** $(x_j, d_j)$ in $V$ **do**;
              send (**nogood**, $x_i$, $V$) to $x_j$; **end do**; **end do**;
      $p_{max} \leftarrow max_{(x_j,d_j,p_j) \in agent\_view}(p_j)$;
      *current_priority* $\leftarrow 1 + p_{max}$;
      select $d \in D_i$ where $d$ minimizes the number of constraint violations;
      *current_value* $\leftarrow d$;
      send (**ok?**, ($x_i$, $d$, *current_priority*)) to other agents; **end do**;
```

Fig. 3. Procedures for receiving messages (asynchronous weak-commitment search)

the number of constraint violations is 1 or 2. In this example, x_3 selects 1 and sends *ok?* messages to other agents (Fig. 4 (c)). After that, x_1 changes its value to 2, and a solution is obtained (Fig. 4 (d)).

In the distributed 4-queens problem, there exists no solution when x_1's value is 1. We can see that a bad decision can be revised without an exhaustive search in the asynchronous weak-commitment search.

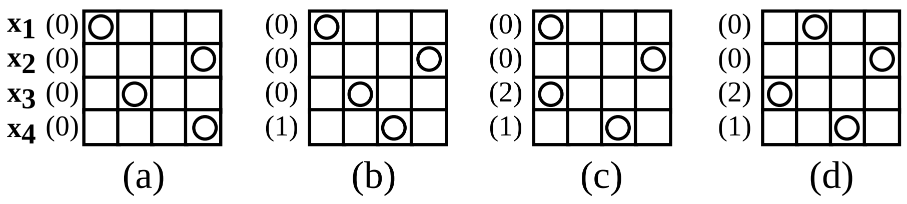

Fig. 4. Example of algorithm execution

3.5 Algorithm Completeness

The priority values are changed if and only if a new nogood is found[7]. Since the number of possible nogoods is finite, the priority values can not be changed infinitely. Therefore, after a certain time point, the priority values will be stable. Then, we show that the situations described below will not occur when the priority values are stable.

(i) There exist agents that do not satisfy some constraints, and all agents are waiting for incoming messages.
(ii) Messages are repeatedly sent/received, and the algorithm will not reach a stable state (infinite processing loop).

If situation (i) occurs, there exist at least two agents that do not satisfy the constraint between them. Let us assume that the agent ranking k-th in the priority order does not satisfy the constraint between the agent ranking j-th (where $j < k$), and all agents ranking higher than k-th satisfy all constraints within them. The only case that the k-th agent waits for incoming messages even though the agent does not satisfy the constraint between the j-th agent is that the k-th agent has sent nogood messages to higher priority agents. This fact contradicts the assumption that higher priority agents satisfy constraints within them. Therefore, situation (i) will not occur.

Also, if the priority values are stable, the asynchronous weak-commitment search algorithm is basically identical to the asynchronous backtracking algorithm. Since the asynchronous backtracking is guaranteed not to fall into an infinite processing loop [13], situation (ii) will not occur.

From the fact that situation (i) or (ii) will not occur, we can guarantee that the asynchronous weak-commitment search algorithm will always find a solution, or find the fact that there exists no solution.

4 Evaluations

In this section, we evaluate the efficiency of algorithms by discrete event simulation, where each agent maintains its own simulated clock. An agent's time

[7] To be exact, different agents may find an identical nogood simultaneously.

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 sending messages. We assume that a message issued at time t is available to the recipient at time $t + 1$. We analyze performance in terms of the amount of cycles required to solve the problem. Given this model, we compare the following three kinds of algorithms: (a) asynchronous backtracking, in which a variable value is selected randomly from consistent values, and the priority order is determined by alphabetical order, (b) *min-conflict* only, in which the min-conflict heuristic is introduced into the asynchronous backtracking, but the priority order is statically determined by alphabetical order, and (c) asynchronous weak-commitment search[8].

We first applied these three algorithms to the distributed n-queens problem described in the previous section, varying n from 10 to 1000. The results are summarized in Table 1. For each n, we generated 100 problems, each of which had different randomly generated initial values, and averaged the results for these problems. For each problem, in order to conduct the experiments in a reasonable amount of time, we set the bound for the number of cycles to 1000, and terminated the algorithm if this limit was exceeded; we counted the result as 1000. The ratio of problems completed successfully to the total number of problems is also described in Table 1.

Table 1. Required cycles for distributed n-queens problem

	asynchronous backtracking		min-conflict only		asynchronous weak-commitment	
n	ratio	cycles	ratio	cycles	ratio	cycles
10	100%	105.4	100%	102.6	100%	41.5
50	50%	662.7	56%	623.0	100%	59.1
100	14%	931.4	30%	851.3	100%	50.8
1000	0%	—	16%	891.8	100%	29.6

The second example problem is the distributed graph-coloring problem. The distributed graph-coloring problem is a graph-coloring problem, in which each node corresponds to an agent. The graph-coloring problem involves painting nodes in a graph by k different colors so that any two nodes connected by an arc do not have the same color. We randomly generate a problem with n nodes/agents and m arcs by the method described in [6], so that the graph is connected and the problem has a solution. We evaluate the problem $n = 60, 90,$

[8] The amounts of local computation performed in each cycle for (b) and (c) are equivalent. The amounts of local computation for (a) can be smaller since it does not use the min-conflict heuristic, but for the lowest priority agent, the amounts of local computation of these algorithms are equivalent.

and 120, where $m = n \times 2$ and k=3. This parameter setting corresponds to the “sparse” problems for which poor performance of the min-conflict heuristic is reported in [6]. We generate 10 different problems, and for each problem, 10 trials with different initial values are performed (100 trials in all). As in the distributed n-queens problem, the initial values are set randomly. The results are summarized in Table 2.

Table 2. Required cycles for distributed graph-coloring problem

	asynchronous backtracking		min-conflict only		asynchronous weak-commitment	
n	ratio	cycles	ratio	cycles	ratio	cycles
60	13%	917.4	12%	937.8	100%	59.4
90	0%	—	2%	994.5	100%	70.1
120	0%	—	0%	—	100%	106.4

Then, in order to examine the applicability of the asynchronous weak-commitment search to real-life problems rather than artificial random problems, we applied these algorithms to the distributed resource allocation problem in a communication network described in [8]. In this problem, there exist requests for allocating circuits between switching nodes of NTT’s communication network in Japan (Fig. 5). For each request, there exists an agent assigned to handle it, and candidates for circuits are given. The goal is to find a set of circuits that satisfies the resource constraints. We can formalize this problem as a Distributed CSP by representing each request as a variable and each candidate as a possible value for the variable. We generated problems based on data from the 400 Mbps backbone network extracted from the network configuration management database developed in NTT Optical Network Systems Laboratories [10]. In each problem, there exist 10 circuit allocation requests, and for each request, 50 candidates are given. These candidates represent reasonably short circuits for satisfying the request. The goal is to find the consistent combination of the candidates, i.e., different candidates do not require the same resources.

We generated 10 different sets of randomly generated initial values for 10 different problems (100 trials in all), and averaged the results. As in the previous problems, the limit for the required number of cycles was set to 1000. The results are summarized in Table 3.

We can see the following facts from these results.

- The asynchronous weak-commitment search algorithm can solve problems that can not be solved within a reasonable amount of computation time by other algorithms. By using only the min-conflict heuristic, although the algorithm can obtain a certain amount of speed-up, it fails to solve many problem instances.

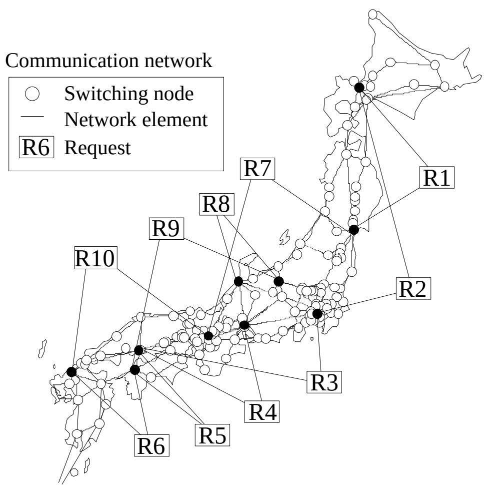

Fig. 5. Example of network resource allocation problem

- When the priority order is static, the efficiency of the algorithm is highly dependent on the selection of initial values, and the distribution of required cycles is quite large. For example, in the network resource allocation problem, when only the min-conflict heuristic is used, the average number of required cycles for 63 successfully completed trials is only 92.8. However, the number of required cycles for 37 failed trials is more than 1000. When the initial values of higher priority agents are good, the solution can easily be found. If some of these values are bad, however, an exhaustive search is required to revise these values; this tends to make the number of required cycles exceed the limit. On the other hand, in the asynchronous weak-commitment search, the initial values are less critical, and a solution can be found even if the initial values are far from the final solution, since the variable values gradually come close to the final solution.
- We can assume that the priority order represents a hierarchy of agent authority, i.e., the priority order of decision making. If this hierarchy is static, the misjudgments (bad value selections) of agents with higher priority are fatal to all agents. On the other hand, when the priority order is changed dynamically and variable values are selected cooperatively, the misjudgments of specific agents do not have fatal effects, since bad decisions will be weeded out, and only good decisions can survive. These results are intuitively natural since they imply that a flexible agent organization performs better than a static and rigid organization.

Table 3. Required cycles for network resource allocation problem

asynchronous backtracking		min-conflict only		asynchronous weak-commitment	
ratio	cycles	ratio	cycles	ratio	cycles
32%	984.8	63%	428.4	100%	17.3

5 Discussions

Since constraint satisfaction is NP-complete in general, the worst-case time complexity of the asynchronous weak-commitment search becomes exponential in the number of variables n. The worst-case space complexity for each agent is determined by the number of recorded nogoods, which is also exponential in n. This result seems inevitable since this algorithm changes the search order flexibly while guaranteeing its completeness.

We can restrict the number of recorded nogoods, i.e., each agent records only a fixed number of the most recently found nogoods. In this case, however, the theoretical completeness can not be guaranteed (the algorithm may fall into an infinite processing loop in which agents repeatedly find identical nogoods). Yet, when the number of recorded nogoods is reasonably large, such an infinite processing loop rarely occurs. Actually, the asynchronous weak-commitment search algorithm can still find solutions for all example problems when the number of recorded nogoods is restricted to 10.

6 Conclusions

In this paper, the asynchronous weak-commitment search algorithm for solving Distributed CSPs is developed. In this algorithm, agents act asynchronously and concurrently based on their local knowledge without any global control, while guaranteeing the completeness of the algorithm. This algorithm can revise a bad decision without an exhaustive search by changing the priority order of agents dynamically. The experimental results indicate that this algorithm can solve problems such as the distributed 1000-queens problem, the distributed graph-coloring problem, and the network resource allocation problem, which can not be solved by asynchronous backtracking algorithm within a reasonable amount of time. These results imply that a flexible agent organization performs better than a static and rigid organization.

Our future work includes showing the effectiveness of the asynchronous weak-commitment search algorithm in more practical application problems, and examining ways of introducing other heuristics (e.g., forward-checking) into this algorithm, and developing iterative improvement algorithms such as the breakout algorithm [7] for solving Distributed CSPs.

Acknowledgments

The author wish to thank N. Fujii and I. Yoda for providing the network configuration management database, and Y. Nishibe for providing the example problems.

References

1. Chandy, K. and Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, Vol. 3, No. 1, (1985) 63–75
2. Collin, Z., Dechter, R., and Katz, S.: On the Feasibility of Distributed Constraint Satisfaction, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (1991) 318–324
3. 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*, Vol. 21, No. 6, (1991) 1462–1477
4. Dechter, R. and Pearl, J.: Network-based Heuristics for Constraint Satisfaction Problems, *Artificial Intelligence*, Vol. 34, No. 1, (1988) 1–38
5. Huhns, M. N. and Bridgeland, D. M.: Multiagent Truth Maintenance, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, (1991) 1437–1445
6. 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*, Vol. 58, No. 1–3, (1992) 161–205
7. Morris, P.: The Breakout Method for Escaping From Local Minima, *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 40–45
8. Nishibe, Y., Kuwabara, K., Ishida, T., and Yokoo, M.: Speed-Up of Distributed Constraint Satisfaction and Its Application to Communication Network Path Assignments, *Systems and Computers in Japan*, Vol. 25, No. 12, (1994) 54 – 67
9. Sycara, K. P., Roth, S., Sadeh, N., and Fox, M.: Distributed Constrained Heuristic Search, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, (1991) 1446–1461
10. Yamaguchi, H., Fujii, H., Yamanaka, Y., and Yoda, I.: Network Configuration Management Database, *NTT R & D*, Vol. 38, No. 12, (1989) 1509–1518
11. Yokoo, M.: Dynamic Variable/Value Ordering Heuristics for Solving Large-Scale Distributed Constraint Satisfaction Problems, *12th International Workshop on Distributed Artificial Intelligence* (1993) 407–422
12. Yokoo, M.: Weak-commitment Search for Solving Constraint Satisfaction Problems, *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994) 313–318
13. 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
14. Zhang, Y. and Mackworth, A.: Parallel and distributed algorithms for finite constraint satisfaction problems, *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing* (1991) 394–397