The Distributed Constraint Satisfaction Problem: Formalization and Algorithms

Makoto Yokoo, Edmund H. Durfee, *Member, IEEE,*
Toru Ishida, *Member, IEEE,* and Kazuhiro Kuwabara

Abstract—In this paper, we develop a formalism called a *distributed constraint satisfaction problem* (distributed CSP) and algorithms for solving distributed CSPs. A distributed CSP is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents. Various application problems in Distributed Artificial Intelligence can be formalized as distributed CSPs. We present our newly developed technique called *asynchronous backtracking* that allows agents to act asynchronously and concurrently without any global control, while guaranteeing the completeness of the algorithm. Furthermore, we describe how the asynchronous backtracking algorithm can be modified into a more efficient algorithm called an *asynchronous weak-commitment search*, which can revise a bad decision without exhaustive search by changing the priority order of agents dynamically. The experimental results on various example problems show that the asynchronous weak-commitment search algorithm is, by far more, efficient than the asynchronous backtracking algorithm and can solve fairly large-scale problems.

Index Terms—Backtracking algorithms, constraint satisfaction problem, distributed artificial intelligence, iterative improvement algorithm, multiagent systems.

1 INTRODUCTION

DISTRIBUTED Artificial Intelligence (DAI) [1] is a subfield of artificial intelligence that is concerned with interaction, especially 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 vital area of research in artificial intelligence.

In this paper, we develop a formalism called a *distributed constraint satisfaction problem* (distributed CSP). A distributed CSP is a constraint satisfaction problem (CSP) in which variables and constraints are distributed among multiple automated agents. A CSP is a problem to find a consistent assignment of values to variables. Even though the definition of a CSP is very simple, a surprisingly wide variety of AI problems can be formalized as CSPs. Similarly, various application problems in DAI that are concerned with finding a consistent combination of agent actions can be formalized as distributed CSPs.

For example, a multiagent truth maintenance system [2] is a distributed version of a truth maintenance system [3]. In this system, there exist multiple agents, each of which has its own truth maintenance system. Each agent has uncertain data that can be IN or OUT, i.e., believed or not believed, and each shares some data with other agents. Each agent must determine the label of its data consistently, and shared data must have the same label. The multiagent truth maintenance task can be formalized as a distributed CSP, where each of uncertain data is a variable whose value can be IN or OUT.

Another example is a distributed resource allocation problem in a communication network, which is described in [4]. In this problem, each agent has its own tasks, and there are several ways (plans) to perform each task. Since resources are shared among agents, there exist constraints/contention between plans. The goal is to find the combination of plans that enables all the tasks to be executed simultaneously. This problem can be formalized as a distributed CSP by representing each task as a variable, and possible plans as variable values.

Many other application problems that are concerned with finding a consistent combination of agent actions/decisions (e.g., distributed scheduling [5] and distributed interpretation problems [6]) can be formalized as distributed CSPs. Since a variety of DAI application problems can be formalized as distributed CSPs, we can consider distributed algorithms for solving distributed CSPs as an important infrastructure in DAI.

- *M. Yokoo is with the NTT Communication Science Laboratories, 2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan. E-mail: yokoo@cslab.kecl.ntt.co.jp.*
- *E.H. Durfee is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: durfee@umich.edu.*
- *T. Ishida is with the Department of Information Science, Kyoto University, Yoshida-honmachi, Sakyo-ku, Kyoto 606-01, Japan. E-mail: ishida@kuis.kyoto-u.ac.jp.*
- *K. Kuwabara is with NTT Research and Development Headquarters, Nippon Telegraph and Telephone Corporation, 3-19-2 Nishi-shinjuku, Shinjuku-ku, Tokyo 163-19, Japan. E-mail: kuwabara@rdh.ecl.ntt.co.jp.*

Manuscript received 21 Apr. 1996; revised 10 June 1997. This paper is the extended version of the authors' previous conference papers [21], [22].
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104304.

It must be noted that although algorithms for solving distributed CSPs seem to be similar to parallel/distributed processing methods for solving CSPs [7], [8], research motivations are fundamentally different. The primary concern in parallel/distributed processing is 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. 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 not only the communication costs but also the costs of translating one's knowledge into an exchangeable format. For example, a constraint might be stored as a very complicated specialized internal function within an agent. In order to communicate the knowledge of this constraint to other agent, which might be implemented on different computer architecture, the agent must translate the knowledge into an exchangeable format, such as a table of allowed (or not allowed) combinations of variable values. These costs of centralizing all information to one agent could be prohibitively high.

Furthermore, in some application problems, gathering all information to one agent is undesirable or impossible for security/privacy reasons. In such cases, multiple agents have to solve the problem without centralizing all information.

In this paper, we develop a basic algorithm for solving distributed CSPs called *asynchronous backtracking*. In this algorithm, agents act asynchronously and concurrently based on their local knowledge without any global control, while the completeness of the algorithm is guaranteed.

Furthermore, we describe how this asynchronous backtracking algorithm can be modified into a more efficient algorithm called *asynchronous weak-commitment search*, which is inspired by the weak-commitment search algorithm for solving CSPs [9]. 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 each 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 is strongly committed to the selected value, 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 in the selection of values becomes fatal since such an exhaustive search is virtually impossible. This drawback is common to all backtracking algorithms. In the asynchronous weak-commitment search, when an agent cannot 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 selecting a value, 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 is changed.

We will show that the asynchronous weak-commitment search algorithm can solve various problems, such as the distributed 1,000-queens problem, the distributed graph-coloring problem, and the network resource allocation problem [10] that the asynchronous backtracking algorithm fails to solve within a reasonable amount of time. These results are interesting since they 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, if we assume that the priority order represents a hierarchy of agent authority, i.e., the priority order of decision making.

In the following sections, we describe the definition of a distributed CSP (Section 2). Then, we show two trivial algorithms for solving distributed CSPs (Section 3), and describe the asynchronous backtracking algorithm in detail (Section 4). We show how the asynchronous weak-commitment search algorithm is obtained by modifying the asynchronous backtracking algorithm (Section 5). Then, we present empirical results that compare the efficiency of these algorithms (Section 6).

2 Distributed Constraint Satisfaction Problem

2.1 CSP

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 that 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.

2.2 Distributed CSP

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 the 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.

It must be noted that 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 the agents and do not care about the implementation of the physical communication network. This is because our primary concern is cooperation among intelligent agents, rather than solving CSPs by certain multiprocessor architectures.

Each agent has some variables and tries to determine their values. However, there exist interagent constraints, and the value assignment must satisfy these interagent constraints. Formally, there exist m agents 1, 2, …, m. Each variable x_j belongs to one agent i (this relation is represented as *belongs*(x_j, i)).[1] Constraints are also distributed among agents. The fact that an agent l knows a constraint predicate p_k is represented as *known*(p_k, l).

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 $\forall l$, $\forall p_k$ where known(p_k, l), p_k is true under the assignment $x_j = d_j$.

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.
- 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.

3 Trivial Algorithms for Solving Distributed CSP

The methods for solving CSPs can be divided into two groups, namely search algorithms (e.g., backtracking algorithms), and consistency algorithms [12]. Consistency algorithms are preprocessing procedures that are invoked before search. Consistency algorithms in the Assumption-based Truth Maintenance System framework [13] are essentially monotonic and can be applied straightforwardly to distributed CSPs. Namely, if each agent has its own Assumption-based Truth Maintenance System, these agents can execute the consistency algorithm by exchanging their possible values, generating new constraints (nogoods) using hyper-resolution rules, and further exchanging obtained nogoods [14]. Therefore, in this paper hereafter, we focus on search algorithms for distributed CSPs. We can consider two trivial algorithms for solving distributed CSPs.

3.1 Centralized Method

The most trivial algorithm for solving a distributed CSP is to select a leader agent among all agents, and gather all information about the variables, their domains, and their constraints, into the leader agent. The leader then solves the CSP alone using normal centralized constraint satisfaction algorithms.

However, as discussed in Section 1, the cost of collecting all information about a problem can be prohibitively high. Furthermore, in some application problems, such as software agents in which each agent acts as a secretary of an individual, gathering all information to one agent is undesirable or impossible for security / privacy reasons.

3.2 Synchronous Backtracking

The standard backtracking algorithm for CSP can be modified to yield the synchronous backtracking algorithm for distributed CSPs. Assume the agents agree on an instantiation order for their variables (such as agent x_1 goes first, then agent x_2, and so on). Each agent, receiving a partial solution (the instantiations of the preceding variables) from the previous agent, instantiates its variable based on the constraints that it knows about. If it finds such a value, it appends this to the partial solution and passes it on to the next agent. If no instantiation of its variable can satisfy the constraints, then it sends a *backtracking* message to the previous agent.

While this algorithm does not suffer from the same communication overhead as the centralized method, determining the instantiation order still requires certain communication costs. Furthermore, this algorithm cannot take advantage of parallelism.[3] Because, at any given time, only one agent is receiving the partial solution and acting on it, the problem is solved sequentially.

4 Asynchronous Backtracking

4.1 Overview

Our asynchronous backtracking algorithm removes the drawbacks of synchronous backtracking by allowing agents to run concurrently and asynchronously. Each agent instantiates its variable and communicates the variable value to the relevant agents.

We represent a distributed CSP in which all constraints are binary as a network, where variables are nodes and constraints are links between nodes.[4] Since each agent has exactly one variable, a node also represents an agent. We use the same identifier for representing an agent and its variable. We also assume that every link (constraint) is *directed*. In other words, one of the two agents involved in a constraint is assigned that constraint, and receives the other agent's value. A link is directed from the value-sending agent to the constraint-evaluating agent. For example, in Fig. 1 there are three agents, x_1, x_2, x_3, with variable domains {1, 2}, {2}, {1, 2}, respectively, and constraints $x_1 \neq x_3$ and $x_2 \neq x_3$.

1. We can consider the case that several agents share a variable. However, such a case can be formalized as these agents have different variables, and there exist constraints that these variables must have the same value.

2. In [11], an algorithm in which each agent has multiple variables is described.

3. In [7], a variation of the synchronous backtracking algorithm called the *Network Consistency Protocol* is presented. In this algorithm, agents construct a depth-first search tree. Agents act synchronously by passing *privilege*, but the agents that have the same parent in the search tree can act concurrently.

4. It must be emphasized that this constraint network has nothing to do with the physical communication network. The link in the constraint network is not a physical communication link, but a logical relation between agents.

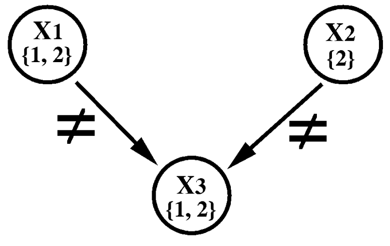


Fig. 1. Example of a constraint network.

Each agent instantiates its variable concurrently and sends the value to the agents which are connected by outgoing links. After that, the agents wait for and respond to messages. Fig. 2 describes procedures executed by agent x_i for receiving two kinds of messages.[5] One kind is an *ok?* message, that a constraint-evaluating agent receives from a value-sending agent asking whether the value chosen is acceptable (Fig. 2i). The second kind is a *nogood* message that a value-sending agent receives, indicating that the constraint-evaluating agent has found a constraint violation (Fig. 2ii).

Each agent has a set of values from the agents that are connected by incoming links. These values constitute the agent's *agent_view*. The fact that x_1's value is 1 is represented by a pair of the agent identifier and the value, (x_1, 1). Therefore, an *agent_view* is a set of these pairs, e.g., {(x_1, 1), (x_2, 2)}. If an *ok?* message is sent on an incoming link, the evaluating agent adds the pair to its *agent_view* and checks whether its own value assignment (represented as (x_i, *current_value*)) is *consistent* with its *agent_view*. Its own assignment is consistent with the *agent_view* if all constraints the agent evaluates are true under the value assignments described in the *agent_view* and (x_i, *current_value*), and if all communicated *nogoods* are not *compatible*[6] with the *agent_view* and (x_i, *current_value*). If its own assignment is not consistent with the *agent_view*, agent x_i tries to change the *current_value* so that it will be consistent with the *agent_view*.

A subset of an *agent_view* is called a *nogood* if the agent is not able to find any consistent value with the subset. For example, in Fig. 3a, if agents x_1 and x_2 instantiate their variables to 1 and 2, the *agent_view* of x_3 will be {(x_1, 1), (x_2, 2)}. Since there is no possible value for x_3 which is consistent with this agent_view, this *agent_view* is a nogood. If an agent finds a subset of its *agent_view* is a nogood,[7] the assignments of other agents must be changed. Therefore, the agent causes a *backtrack* (Fig. 2iii)) and sends a *nogood* message to one of the other agents.

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

6. A nogood is compatible with the *agent_view* and (x_i, *current_value*) if all variables in the nogood have the same values in the *agent_view* and (x_i, *current_value*).

7. Ideally, the nogoods detected in Fig. 2iii-a should be *minimal*, i.e., no subset of them should be a nogood. However, since finding all minimal nogoods requires certain computation costs, an agent can make do with nonminimal nogoods. In the simplest case, it can use the whole *agent_view* as a nogood.

4.2 Avoiding Infinite Processing Loops

If agents change their values again and again and never reach a stable state, they are in an infinite processing loop. An infinite processing loop can occur if there exists a value changing loop of agents, such as if a change in x_1 causes x_2 to change, then this change in x_2 causes x_3 to change, which then causes x_1 to change again, and so on. In the network representation, such a loop is represented by a cycle of directed links.

One way to avoid cycles in a network is to use a total order relationship among nodes. If each node has an unique identifier, we can define a priority order among agents by using the alphabetical order of these identifiers (the preceding agent in the alphabetical order has higher priority). If a link is directed by using this priority order (from the higher priority agent to the lower priority agent), there will be no cycle in the network. This means that for each constraint, the lower priority agent will be an evaluator, and the higher priority agent will send an *ok?* message to the evaluator. Furthermore, if a nogood is found, a *nogood* message is sent to the lowest priority agent in the nogood (Fig. 2iii-b). Similar techniques to this unique identifier method are used for avoiding deadlock in distributed database systems [15].

The knowledge each agent requires for this unique identifier method is much more local than that needed for synchronous backtracking. In synchronous backtracking, agents must act in a predefined sequential order. Such a sequential order cannot be obtained easily just by giving an unique identifier to each agent. Each agent must know the previous and next agent, which means polling all of the other agents to find the closest identifiers above and below it. On the other hand, in the unique identifier method for asynchronous backtracking, each agent has to know only the identifiers of an agent with which it must establish a constraint in order to direct the constraint.

4.3 Handling Asynchronous Changes

Because agents change their instantiations asynchronously, an *agent_view* is subject to incessant changes. This can lead to potential inconsistencies, because a constraint-evaluating agent might send a *nogood* message to an agent that has already changed the value of an offending variable as a result of other constraints. In essence, the *nogood* message may be based on obsolete information, and the value-sending agent should not necessarily change its value again.

We introduce the use of *context attachment* to deal with these potential inconsistencies. In context attachment, an agent couples its message with the nogood that triggered it. This nogood is the context of backtracking. After receiving this message, the recipient only changes its value if the nogood is *compatible* with its current *agent_view* and its own assignment (Fig. 2ii-a). Since the nogood attached to a *nogood* message indicates the cause of the failure, asynchronous backtracking includes the function of dependency-directed backtracking in CSPs [12].

A nogood can be viewed as a new constraint derived from the original constraints. By incorporating such a new constraint, agents can avoid repeating the same mistake. For example, in Fig. 3b, the nogood {(x_1, 1), (x_2, 2)} represents

```
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 — (ii)
  add nogood to nogood_list;
  when ($x_k$,$d_k$) where $x_k$ is not connected is contained in nogood do
    request $x_k$ to add a link from $x_k$ to $x_i$;
    add ($x_k$, $d_k$) to agent_view; end do;
  old_value ← current_value; check_agent_view; — (ii-a)
  when old_value = current_value do
      send (ok?, ($x_j$, current_value)) to $x_j$; end do; 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; — (iii)
    else select $d \in D_i$ where agent_view and $d$ are consistent;
      current_value ← $d$;
      send (ok?, ($x_i$, $d$)) to outgoing links; end if; end do;

procedure backtrack
  nogoods ← {$V$ | $V$= inconsistent subset of agent_view}; — (iii-a)
  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 $V$; — (iii-b)
      send (nogood, $x_i$, $V$) to $x_j$;
    remove $(x_j, d_j)$ from agent_view; end do;
  check_agent_view;
  end do;
```

Fig. 2. Procedures for receiving messages (asynchronous backtracking).

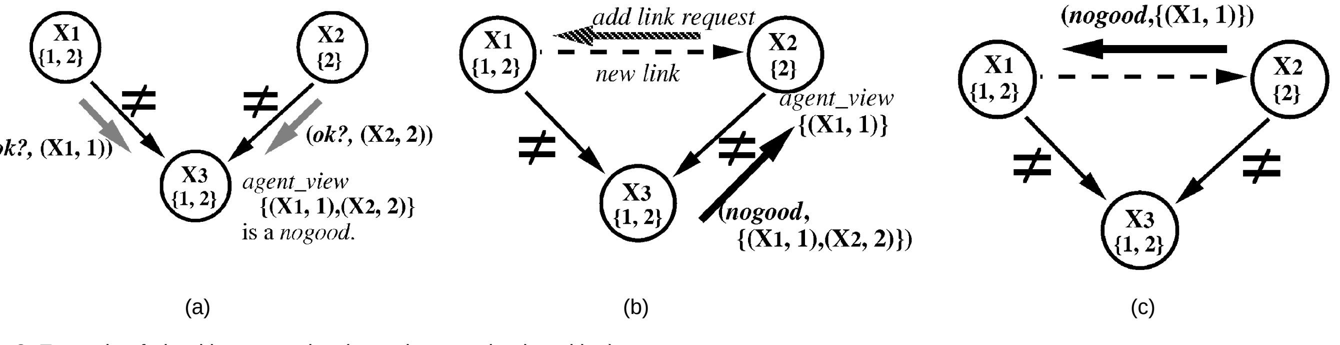


Fig. 3. Example of algorithm execution (asynchronous backtracking).

a constraint between x_1 and x_2. Since there is no link between x_1 and x_2 originally, a new link must be added between them.[8] Therefore, after receiving the nogood message, agent x_2 asks x_1 to add a link between them. In general, even if all the original constraints are binary, newly derived constraints can be among more than two variables. In such a case, one of the agents, which has the lowest priority in the constraint, will be an evaluator and the links will be added between each of the nonevaluator agents and the evaluator.

8. Since a link in the constraint network represents a logical relation between agents, adding a link does not mean adding a new physical communication path between agents.

4.4 Example

In Fig. 3a, by receiving *ok?* messages from x_1 and x_2, the *agent_view* of x_3 will be {(x_1, 1), (x_2, 2)}. Since there is no possible value for x_3 consistent with this *agent_view*, this *agent_view* is a nogood. Agent x_3 chooses the lowest priority agent in the *agent_view*, i.e., agent x_2, and sends a *nogood* message with the nogood. By receiving this *nogood* message, agent x_2 records this nogood. This nogood, {(x_1, 1), (x_2, 2)} contains agent x_1, which is not connected with x_2 by a link. Therefore, a new link must be added between x_1 and x_2. Agent x_2 requests x_1 to send x_1's value to x_2, and adds (x_1, 1) to its *agent_view* (Fig. 3b). Agent x_2 checks whether its value is consistent with the agent_view. Since the nogood

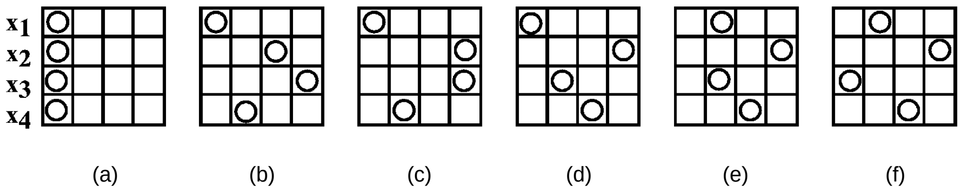


Fig. 4. Example of algorithm execution (asynchronous backtracking).

received from agent x_3 is compatible with its assignment (x_2, 2) and its *agent_view* {(x_1, 1)}, the assignment (x_2, 2) is inconsistent with the *agent_view*. The *agent_view* {(x_1, 1)} is a nogood because x_2 has no other possible values. There is only one agent in this nogood, i.e., agent x_1, so agent x_2 sends a *nogood* message to agent x_1 (Fig. 3c).

Furthermore, 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 in each row. The goal of the agents is to find positions on a 4×4 chess board so that the queens do not threaten each other. It must be noted that the trace of the algorithm execution can vary significantly according to the timing/delay of the messages, and this example shows one possible trace of execution.

The initial values are shown in Fig. 4a. Agents communicate these values with each other. The priority order is determined by the alphabetical order of identifiers. The agents except x_1 change their value, so that the new value is consistent with its *agent_view* (Fig. 4b), i.e., agent x_2 changes its value to 3, which is consistent with x_1's value. Agent x_3 changes its value to 4, which is consistent with x_1's and x_2's value (since x_2 changes its value, x_3's value is no longer consistent with the new value.). Since there is no consistent value for agent x_4, it sends a *nogood* message to x_3, and changes its value so that the value is consistent with its *agent_view*, except the value of x_3. Note that x_3 will ignore this nogood message since it has changed its value before it receives this message. The agents send *ok?* messages to other agents. Then, x_3 does not satisfy constraints with x_1 and x_2, and there is no consistent value, while other agents' values are consistent with their *agent_view*. Therefore, x_3 sends a nogood message to x_2. After receiving this nogood message, x_2 changes its value to 4 (Fig. 4c). Then, x_3 changes its value to 2. There is no consistent value for agent x_4, it sends a *nogood* message to x_3, and changes its value so that the value is consistent with its *agent_view*, except the value of x_3 (Fig. 4d). Again, this nogood message is ignored. There is no consistent value for agent x_4, it sends a *nogood* message to x_3. After receiving this message, x_3 has no other consistent value, so x_3 sends a nogood message to x_2. After receiving this message, x_2 also has no other consistent value, so x_2 sends a nogood message to x_1. Then, x_1 changes its value to 2 (Fig. 4e). Then, x_3 changes its value to 1. There is no consistent value for agent x_4, it sends a *nogood* message to x_3, and changes its value so that the value is consistent with its *agent_view*, except the value of x_3. Again, this nogood message is ignored, and a solution is found (Fig. 4f).

4.5 Algorithm Soundness and Completeness

We will show that if there exists a solution, this algorithm reaches a stable state where all the variable values satisfy all the constraints, and all agents are waiting for an incoming message,[9] and if no solution exists, this algorithm discovers this fact and terminates. For the agents to reach a stable state, all their variable values must perforce satisfy all the constraints. Thus, the soundness of the algorithm is clear. Furthermore, the algorithm is complete, in that it finds a solution if one exists and terminates with failure when there is no solution.

A solution does not exist when the problem is overconstrained. In an overconstrained situation, our algorithm eventually generates a nogood corresponding to the empty set. Because a nogood logically represents a set of assignments that leads to a contradiction, an empty nogood means that any set of assignments leads to a contradiction. Thus, no solution is possible. Our algorithm thus terminates with failure if and only if an empty nogood is formed.

So far, we have shown that when the algorithm leads to a stable state, the problem is solved, and when it generates an empty nogood, the algorithm terminates with failure. What remains is to show that the algorithm reaches one of these conclusions in finite time. The only way that our algorithm might not reach a conclusion is when at least one agent is cycling among its possible values in an infinite processing loop. Given our algorithm, we can prove by induction that this cannot happen as follows.

In the base case, assume that the agent with the highest priority, x_1, is in an infinite loop. Because it has the highest priority, x_1 only receives *nogood* messages. When it proposes a possible value, x_1 either receives a *nogood* message back, or else gets no message back. If it receives *nogood* messages for all possible values of its variable, then it will generate an empty nogood (any choice leads to a constraint violation) and the algorithm will terminate. If it does not receive a nogood message for a proposed value, then it will not change that value. Either way, it cannot be in an infinite loop.

Now, assume that agents x_1 to x_{k-1} ($k > 2$) are in a stable state, and agent x_k is in an infinite processing loop. In this case, the only messages agent x_k receives are *nogood* messages from agents whose priorities are lower than k, and these *nogood* messages contain only the agents x_1 to x_k. Since agents x_1 to x_{k-1} are in a stable state, the *nogoods* agent x_k receives must be compatible with its *agent_view*, and so x_k

9. We should mention 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, distributed termination detection algorithms such as [16] are needed.

will change instantiation of its variable with a different value. Because its variable's domain is finite, x_k will either eventually generate a value that does not cause it to receive a nogood (which contradicts the assumption that x_k is in an infinite loop), or else it exhausts the possible values and sends a nogood to one of $x_1 \ldots x_{k-1}$. However, this nogood would cause an agent, which we assumed as being in a stable state, to not be in a stable state. Thus, by contradiction, x_k cannot be in an infinite processing loop.

Since constraint satisfaction is NP-complete in general, the worst-case time complexity of the asynchronous backtracking algorithm becomes exponential in the number of variables n. The worst-case space complexity of the algorithm is determined by the number of recorded nogoods. In the asynchronous backtracking algorithm, an agent can forget old nogoods after it creates a new nogood from them. Also, an agent does not need to keep the nogoods that are not compatible with the *agent_view*. Therefore, each agent x_i needs to record at most $|D_i|$ nogoods, where $|D_i|$ is the number of possible values of x_i.

5 Asynchronous Weak-Commitment Search

In this section, we briefly describe the weak-commitment search algorithm for solving CSPs [9] and describe how the asynchronous weak-commitment search algorithm is constructed by modifying the asynchronous backtracking algorithm.

5.1 Weak-Commitment Search Algorithm

In the weak-commitment search algorithm, all the variables have tentative initial values. 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 the constraints between the variables included in the partial solution, and satisfies as many constraints as possible between variables that are not included in the partial solution. This value ordering heuristic is called the *min-conflict* heuristic [17]. When there exists no value for one variable that satisfies all the constraints between the variables included in 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. Experimental results on various example problems in [9] illustrate that this algorithm is three to 10 times more efficient than the min-conflict backtracking [17] or the breakout algorithm [18].

5.2 Basic Ideas

The main characteristics of the weak-commitment search algorithm described in the previous subsection 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 the 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, 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 a distributed constraint satisfaction algorithm that commits to the partial solution weakly can be constructed 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 nonnegative 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_i, the priority value of x_i is changed to $k + 1$, where k is the largest priority value of related agents.

It must be noted that the asynchronous weak-commitment search algorithm is fundamentally different from backtracking algorithms with dynamic variable ordering (e.g., dynamic backtracking [19] and dependency-directed backtracking [12]). In backtracking algorithms, a partial solution is never modified unless it is sure that the partial solution cannot be a part of any complete solution (dynamic backtracking or dependency-backtracking is a way for finding out the true cause of the failure/backtracking). In the asynchronous weak-commitment search algorithm, a partial solution is not modified but completely abandoned after one failure/backtracking.

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 identical to a previously found nogood. In order to avoid reacting to such 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 identical to a nogood that it has already sent, the agent will not change the priority value and waits for the next message.

```
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;
  when ($x_k$,$d_k$,priority) where $x_k$ is not in neighbors
      is contained in nogood do
    add $x_k$ to neighbors, add ($x_k$,$d_k$,priority) to agent_view; end do;
  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
          with lower priority agents; — (ii)
      current_value ← $d$;
      send (ok?, ($x_i$, $d$, current_priority)) to neighbors; end if; end do;

procedure backtrack — (iii)
  nogoods ← $\{V \mid V =$ 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 $(x_j, d_j, p_j)$ 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 ← $1 + p_{max}$;
      select $d \in D_i$ where $d$ minimizes the number of constraint violations
        with lower priority agents;
      current_value ← $d$;
      send (ok?, ($x_i$, $d$, current_priority)) to neighbors; end do;
```

Fig. 5. Procedures for receiving messages (asynchronous weak-commitment search).

5.3 Details of Algorithm

In the asynchronous weak-commitment search, each agent concurrently assigns a value to its variable, and sends the variable value to other agents. After that, agents wait for and respond to incoming messages.[10] In Fig. 5, the procedures executed at agent x_i by receiving an *ok?* message and a *nogood* message are described.[11] The differences between these procedures and the procedures for the asynchronous backtracking algorithm are as follows:

- In the asynchronous backtracking algorithm, each agent sends its variable value only to related lower priority agents, while in the asynchronous weak-commitment search algorithm, each agent sends its variable value to both lower and higher priority agents connected by constraints. We call these related agents *neighbors*.
- The priority value, as well as the current value assignment, is communicated through the *ok?* message (Fig. 5i).
- The priority order is determined using 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. 5ii).
- When x_i cannot 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 its priority value but will wait for the next message (Fig. 5iii).

10. As in the asynchronous backtracking, 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 the *agent_view* and *nogood_list* according to the messages, and performs **check_agent_view** only once.

11. As in the asynchronous backtracking, the way to determine that agents as a whole have reached a stable state is not contained in this algorithm.

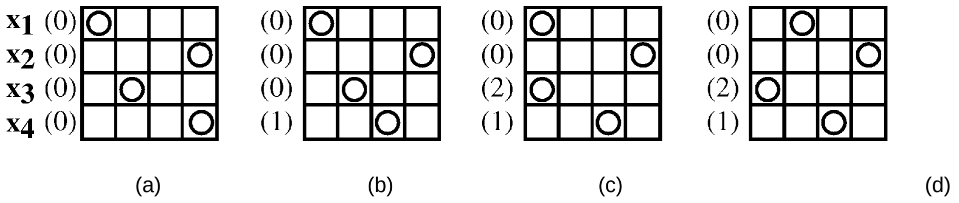


Fig. 6. Example of algorithm execution (asynchronous weak-commitment search).

5.4 Example

We illustrate the execution of the algorithm using the distributed 4-queens problem described in Section 4.4.

The initial values are shown in Fig. 6a. Agents communicate these values with each other. 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*. 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 with x_3 only. Therefore, x_4 selects 3 and sends *ok?* messages to the other agents (Fig. 6b). 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 the number of constraint violations is 1 or 2. In this example, x_3 selects 1 and sends *ok?* messages to the other agents (Fig. 6c). After that, x_1 changes its value to 2, and a solution is obtained (Fig. 6d).

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

5.5 Algorithm Completeness

The priority values are changed if and only if a new nogood is found. Since the number of possible nogoods is finite,[12] the priority values cannot 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:

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

If situation 1) 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 that all the 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 1) 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, situation 2) will not occur.

From the fact that neither situation 1) nor 2) will occur, we can guarantee that the asynchronous weak-commitment search algorithm will always find a solution, or find the fact that no solution exists.

Since constraint satisfaction is NP-complete in general, the worst-case time complexity of the asynchronous weak-commitment search algorithm becomes exponential in the number of variables *n*. Furthermore, the worst-case space complexity is 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 in the asynchronous weak-commitment search algorithm, i.e., each agent records only a fixed number of the most recently found nogoods. In this case, however, the theoretical completeness cannot 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 can still find solutions for all example problems when the number of recorded nogoods is restricted to 10.

5.6 Security/Privacy of Agents

One reason for solving a distributed CSP in a distributed fashion is that agents might not want to communicate all the information to the centralized leader agent. Then, how much information do agents reveal using the asynchronous backtracking/weak-commitment search algorithm?

In both algorithms, agents communicate current value assignments and nogoods. By observing the value assignments of agent x_i, other agents can gradually accumulate the information about the domain of x_i. However, other agents cannot tell whether the obtained information of x_i's domain is complete or not. There might be other values of x_i, which are not selected because they violate some constraints with higher priority agents.

Furthermore, agent x_i never reveals the information about its constraints directly. A nogood message sent from x_i is a highly summarized information about its constraints and nogoods sent from other agents.

12. The number of possible nogoods is exponential in the number of variables *n*.

Therefore, we can see that the amount of information revealed by these algorithms are much smaller than the centralized methods, in which agents must declare precise information about their variable domains and constraints.

6 Evaluations

In this section, we evaluate the efficiency of algorithms by a discrete event simulation, where each agent maintains its own simulated clock. An agent's time is incremented by one simulated time unit whenever it performs one cycle of computation. One cycle consists of reading all incoming messages, performing local computation, and then sending messages. We assume that a message issued at time t is available to the recipient at time $t + 1$. We analyze the performance in terms of the number of cycles required to solve the problem.[13]

One cycle corresponds to a series of agent actions, in which an agent recognizes the state of the world (the value assignments of other agents), then decides its response to that state (its own value assignment), and communicates its decisions.

6.1 Comparison Between Synchronous and Asynchronous Backtracking

First, we are going to compare the synchronous backtracking algorithm and the asynchronous backtracking algorithm. Since agents can act concurrently in the asynchronous backtracking algorithm, we can expect that the asynchronous backtracking algorithm will be more efficient than the synchronous backtracking algorithm. However, the degree of speed-up is affected by the strength of the constraints among agents. If the constraints among agents are weak, we can expect that the agents can easily reach a solution, even if they concurrently set their values. On the other hand, if the constraints among agents are strong, we can assume that until higher priority agents set their values properly, the lower priority agents cannot choose consistent values; thus the overall performance of the asynchronous backtracking algorithm becomes close to the one for synchronous backtracking.

To verify these expectations, we performed experimental evaluations on the distributed n-queens problem explained in the previous section. Each agent corresponds to each queen in a row. Therefore, the distributed n-queens problem is solved by n agents. In the distributed n-queens problem, constraints among agents become weak as n increases. Our results are summarized in the graph shown in Fig. 7. To make the comparisons fair, we included dependency-directed backtracking in the synchronous backtracking. Each agent randomly selects a value among the consistent values with higher priority agents. The graph shows the average of 100 trials.[14] In the distributed n-queens problem, there exist constraints between any pair of agents. Therefore, the synchronous backtracking algorithm is basically equivalent to the Network Consistency Protocol described in [7]. As we expected, the obtained parallelism of the asynchronous backtracking becomes larger as n increases. When $n > 18$, the asynchronous backtracking is approximately two times as fast as the synchronous backtracking.[15]

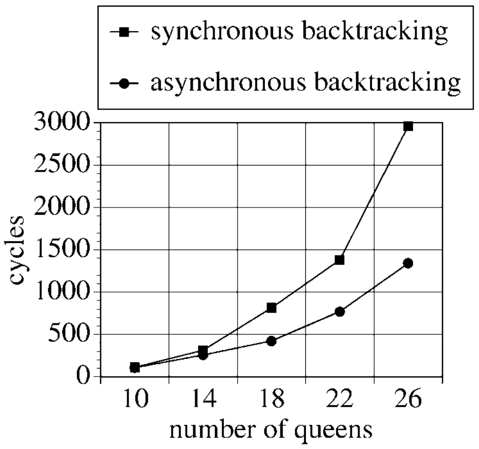


Fig. 7. Comparison between synchronous and asynchronous backtracking (distributed n-queens).

Traditionally, distributed artificial intelligence applications involve having agents work on nearly independent, loosely coupled subproblems [1]. These results confirm that, if the local subproblems are loosely coupled, solving the problem asynchronously by multiple agents is worthwhile.

6.2 Comparison Between Asynchronous Backtracking and Asynchronous Weak-Commitment Search

We are going to compare the following three kinds of algorithms:

1) asynchronous backtracking, in which a variable value is selected randomly from consistent values, and the priority order is determined by alphabetical order,
2) asynchronous backtracking with *min-conflict* heuristic, in which the *min-conflict* heuristic is introduced, but the priority order is statically determined by alphabetical order, and
3) asynchronous weak-commitment search.[16]

We first applied these three algorithms to the distributed n-queens problem described in the previous section, varying n from 10 to 1,000. The results are summarized in Table 1.[17] For each n, we generated 100 problems, each of

13. One drawback of this model is that it does not take into account the costs of communication. However, introducing the communication costs into the model is difficult since we don't have any standard way for comparing communication costs and computation costs.

14. In this evaluation, we did not include the cost of determining the sequential order in the synchronous backtracking, nor the cost of the termination detection in the asynchronous backtracking.

15. Since the asynchronous backtracking algorithm requires more messages than the synchronous backtracking for each cycle, the synchronous backtracking might be as efficient as the asynchronous backtracking due to the communication overhead, even though it requires more cycles.

16. The amount of communication overhead of these algorithms are almost equivalent. The amounts of local computation performed in each cycle for 2) and 3) are equivalent. The amount of local computation for 1) 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.

17. Since the the min-conflict heuristic is very effective when n is very large [9], [17], we did not include the results for $n > 1{,}000$.

TABLE 1
COMPARISON BETWEEN ASYNCHRONOUS BACKTRACKING AND ASYNCHRONOUS WEAK-COMMITMENT SEARCH (DISTRIBUTED N-QUEENS)

	asynchronous backtracking		asynchronous backtracking with min-conflict heuristic		asynchronous weak-commitment	
n	ratio	cycles	ratio	cycles	ratio	cycles
10	100%	105.4	100%	102.6	100%	41.5
50	50%	325.4	56%	326.8	100%	59.1
100	14%	510.0	30%	504.3	100%	50.8
1000	0%	—	16%	323.8	100%	29.6

TABLE 2
COMPARISON BETWEEN ASYNCHRONOUS BACKTRACKING AND ASYNCHRONOUS WEAK-COMMITMENT SEARCH (DISTRIBUTED GRAPH-COLORING PROBLEM)

	asynchronous backtracking		asynchronous backtracking with min-conflict heuristic		asynchronous weak-commitment	
n	ratio	cycles	ratio	cycles	ratio	cycles
60	13%	364.6	12%	481.7	100%	59.4
90	0%	—	2%	725.0	100%	70.1
120	0%	—	0%	—	100%	106.4

which had different randomly generated initial values, and averaged the results for these problems. For each problem, in order to conduct the experiments within a reasonable amount of time, we set the limit for the number of cycles at 1,000, and terminated the algorithm if this limit was exceeded. We show the average of the successful trials, and the ratio of problems completed successfully to the total number of problems in Table 1.

The second example problem is the distributed graph-coloring problem. This 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 generated a problem with n nodes/agents and m arcs by the method described in [17], so that the graph is connected and the problem has a solution. We evaluated the problem for $n = 60$, 90, and 120, where $m = n \times 2$ and $k = 3$. This parameter setting corresponds to the "sparse" problems for which [17] reported poor performance of the min-conflict heuristic. We generated 10 different problems, and for each problem, 10 trials with different initial values were performed (100 trials in all). As in the distributed n-queens problem, the initial values were set randomly. The results are summarized in Table 2.

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 [10]. In this problem, there exist requests for allocating circuits between switching nodes of NTT's communication network in Japan (Fig. 8). For each request, there exists an agent assigned to handle it, and the candidates for the circuits are given. The goal is to find a set of circuits that satisfies the resource constraints. This problem can be formalized 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 a 400 Mbps backbone network extracted from the network configuration management database developed in NTT Optical Network Systems Laboratories [20]. In each problem, there exist 10 randomly generated circuit allocation requests, and for each request, 50 candidates are given. These candidates represent reasonably short circuits for satisfying the request. We assume that these candidates are calculated beforehand. The constraints between requests are that they do not assign the same circuits. 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

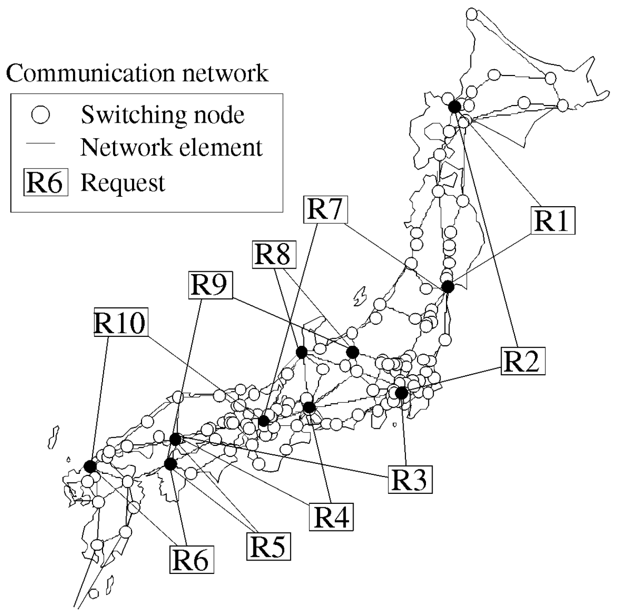


Fig. 8. Example of the network resource allocation problem.

problems, the limit for the required number of cycles was set at 1,000. 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 cannot be solved within a reasonable amount of computation time by asynchronous backtracking algorithms. By using only the *min-conflict* heuristic, although a certain amount of speed-up can be obtained, the algorithm fails to solve many problem instances.
- 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 1,000. 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, by changing the priority order dynamically and selecting values cooperatively, the misjudgments of specific agents do not have fatal effects, since bad decisions are weeded out, and only good decisions survive. These results are intuitively natural, since they imply that a flexible agent organization performs better than a static and rigid organization.

7 Conclusions

In this paper, we develop the formalism for distributed constraint satisfaction problems, which can represent various application problems in Distributed Artificial Intelligence. We developed a basic algorithm for solving distributed CSPs, called *asynchronous backtracking*, in which agents act asynchronously and concurrently without any global control. Furthermore, we developed a more efficient algorithm called *asynchronous weak-commitment search*, which can revise a bad decision without exhaustive search, just as the *weak-commitment search* algorithm does for CSPs. We presented a series of experimental results to compare the efficiency of these algorithms. These results show that the asynchronous weak-commitment search algorithm can solve fairly large-scale problems such as the distributed 1,000-queens problem, the distributed graph-coloring problem, and the network resource allocation problem, within a reasonable amount of time.

Our future work includes examining the effectiveness of the asynchronous weak-commitment search algorithm in more practical applications, introducing other heuristics (e.g., forward-checking) into the asynchronous weak-commitment search algorithm, and clarifying the appropriate agent/variable ordering heuristics when each agent has multiple variables.

TABLE 3
Comparison Between Asynchronous Backtracking and Asynchronous Weak-Commitment Search (Network Resource Allocation Problem)

asynchronous backtracking		asynchronous backtracking with min-conflict heuristic		asynchronous weak-commitment	
ratio	cycles	ratio	cycles	ratio	cycles
32%	952.5	63%	92.8	100%	17.3

ACKNOWLEDGMENTS

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

REFERENCES

[1] *Readings in Distributed Artificial Intelligence*, A.H. Bond and L. Gasser, eds. Morgan Kaufmann, 1988.

[2] M.N. Huhns and D.M. Bridgeland, "Multiagent Truth Maintenance," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 21, no. 6, pp. 1,437–1,445, 1991.

[3] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, pp. 231–272, 1979.

[4] S.E. Conry, K. Kuwabara, V.R. Lesser, and R.A. Meyer, "Multistage Negotiation for Distributed Constraint Satisfaction," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 21, no. 6, pp. 1,462–1,477, 1991.

[5] K.P. Sycara, S. Roth, N. Sadeh, and M. Fox, "Distributed Constrained Heuristic Search," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 21, no. 6, pp. 1,446–1,461, 1991.

[6] C. Mason and R. Johnson, "DATMS: A Framework for Distributed Assumption Based Reasoning," L. Gasser and M. Huhns, eds., *Distributed Artificial Intelligence*, vol. II, pp. 293–318. Morgan Kaufmann, 1989.

[7] Z. Collin, R. Dechter, and S. Katz, "On the Feasibility of Distributed Constraint Satisfaction," *Proc. 12th Int'l Joint Conf. Artificial Intelligence*, pp. 318–324, 1991.

[8] Y. Zhang and A. Mackworth, "Parallel and Distributed Algorithms for Finite Constraint Satisfaction Problems," *Proc. Third IEEE Symp. Parallel and Distributed Processing*, pp. 394–397, 1991.

[9] M. Yokoo, "Weak-Commitment Search for Solving Constraint Satisfaction Problems," *Proc. 12th Nat'l Conf. Artificial Intelligence*, pp. 313–318, 1994.

[10] Y. Nishibe, K. Kuwabara, T. Ishida, and M. Yokoo, "Speed-Up of Distributed Constraint Satisfaction and Its Application to Communication Network Path Assignments," *Systems and Computers in Japan*, vol. 25, no. 12, pp. 54–67, 1994.

[11] M. Yokoo, "Dynamic Variable/Value Ordering Heuristics for Solving Large-Scale Distributed Constraint Satisfaction Problems," *Proc. 12th Int'l Workshop Distributed Artificial Intelligence*, pp. 407–422, 1993.

[12] A.K. Mackworth, "Constraint Satisfaction," S.C. Shapiro, ed., *Encyclopedia of Artificial Intelligence*, second ed., pp. 285–293. New York: Wiley-Interscience, 1992.

[13] J. de Kleer, "A Comparison of ATMS and CSP Techniques," *Proc. 11th Int'l Joint Conf. Artificial Intelligence*, pp. 290–296, 1989.

[14] M. Yokoo, T. Ishida, and K. Kuwabara, "Distributed Constraint Satisfaction for DAI Problems," *Proc. 10th Int'l Workshop Distributed Artificial Intelligence*, 1990.

[15] D. Rosenkrantz, R. Stearns, and P. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Systems*, vol. 3, no. 2, pp. 178–198, 1978.

[16] K. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[17] 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*, vol. 58, nos. 1–3, pp. 161–205, 1992.

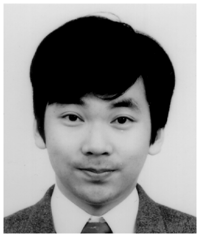
[18] P. Morris, "The Breakout Method for Escaping from Local Minima," *Proc. 11th Nat'l Conf. Artificial Intelligence*, pp. 40–45, 1993.

[19] M. Ginsberg, "Dynamic Backtracking," *J. Artificial Intelligence Research*, vol. 1, pp. 25–46, 1993.

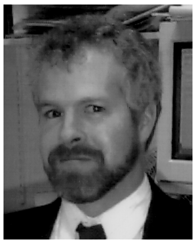
[20] H. Yamaguchi, H. Fujii, Y. Yamanaka, and I. Yoda, "Network Configuration Management Database," *NTT R&D*, vol. 38, no. 12, pp. 1,509–1,518, 1989.

[21] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving," *Proc. 12th IEEE Int'l Conf. Distributed Computing Systems*, pp. 614–621, 1992.

[22] M. Yokoo, "Asynchronous Weak-Commitment Search for Solving Distributed Constraint Satisfaction Problems," *Proc. First Int'l Conf. Principles and Practice of Constraint Programming*, pp. 88–102, Springer-Verlag, 1995.



Makoto Yokoo received the BE and ME degrees in electrical engineering in 1984 and 1986, respectively; and the PhD degree in information and communication engineering in 1995, all from the University of Tokyo, Japan. He is currently a senior research scientist at NTT Communication Science Laboratories, Kyoto, Japan. He was a visiting research scientist in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, from 1990 to 1991. His current research interests include distributed artificial intelligence, multiagent systems, constraint satisfaction, and search.



Edmund H. Durfee received his AB degree in chemistry and physics from Harvard University in 1980; and his MS and PhD degrees from the University of Massachusetts in computer science and engineering in 1984 and 1987, respectively. His PhD research developed an approach for planning coordinated actions and interactions in a network of distributed AI problem-solving systems, and is described in his book *Coordination of Distributed Problem Solvers* (Kluwer Academic Press). He is now an associate professor of electrical engineering and computer science and holds a joint appointment at the School of Information at the University of Michigan, Ann Arbor. His research interests are in distributed artificial intelligence, planning, cooperative robotics, and real-time problem solving. He received a Presidential Young Investigator award from the National Science Foundation in 1991 to support this work, and was an invited speaker at the National Conference on Artificial Intelligence in 1992. He is an associate editor of *IEEE Transactions on Systems, Man, and Cybernetics* and program cochair of the Third International Conference on MultiAgent Systems. He is a member of the IEEE.

Toru Ishida received the BE, MEng, and DEng degrees from Kyoto University, Kyoto, Japan, in 1976, 1978 and 1989, respectively. He is currently a professor in the Department of Information Science at Kyoto University, Kyoto. From 1978 to 1993, he was a research scientist for NTT Laboratories. He was a visiting research scientist in the Department of Computer Science at Columbia University from 1983 to 1984. He authored *Parallel, Distributed, and Multiagent Production Systems* (Springer, 1994) and *Real-time Search for Learning Autonomous Agents* (Kluwer Academic, 1997). He first proposed parallel rule firing and extended it to distributed rule firing. Organizational self-design was then introduced into distributed production systems for increasing adaptiveness. With colleagues in 1994, he initiated work on *Communityware: Towards Global Collaboration* (John Wiley and Sons, 1998). He is a member of the IEEE.

Kazuhiro Kuwabara received the BE and ME degrees in electrical engineering from the University of Tokyo, Japan, in 1982 and 1984, respectively. He joined Nippon Telegraph and Telephone Corporation (NTT) in 1984 and has been engaged in research and development on knowledge-based systems and multiagent systems. He was a visiting research scientist at the University of Massachusetts at Amherst in 1988. He is currently doing research at NTT Research and Development headquarters.