Autonomous Agents and Multi-Agent Systems, Vol.3, No.2, pp.198–212, 2000

Algorithms for Distributed Constraint Satisfaction: A Review

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

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

Abstract. When multiple agents are in a shared environment, there usually exist constraints among the possible actions of these agents. A distributed constraint satisfaction problem (distributed CSP) is a problem to find a consistent combination of actions that satisfies these inter-agent constraints. Various application problems in multi-agent systems can be formalized as distributed CSPs. This paper gives an overview of the existing research on distributed CSPs. First, we briefly describe the problem formalization and algorithms of normal, centralized CSPs. Then, we show the problem formalization and several MAS application problems of distributed CSPs. Furthermore, we describe a series of algorithms for solving distributed CSPs, i.e., the asynchronous backtracking, the asynchronous weak-commitment search, the distributed breakout, and distributed consistency algorithms. Finally, we show two extensions of the basic problem formalization of distributed CSPs, i.e., handling multiple local variables, and dealing with over-constrained problems.

Keywords: Constraint Satisfaction, Search, distributed AI

1. Introduction

A Constraint satisfaction problem (CSP) is a problem to find a consistent assignment of values to variables. A typical example of a CSP is a puzzle called n-queens. The objective is to place n chess queens on a board with $n \times n$ squares so that these queens do not threaten each other (Figure 1). A problem of this kind is called a constraint satisfaction problem since the objective is to find a configuration that satisfies the given conditions (constraints). Even though the definition of a CSP is very simple, a surprisingly wide variety of AI problems can be formalized as CSPs. Therefore, the research on CSP has a long and distinguished history in AI [15].

A distributed CSP is a CSP in which variables and constraints are distributed among multiple automated agents. Various application problems in Multi-agent Systems (MAS) that are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation problems [4], distributed scheduling problems [21],

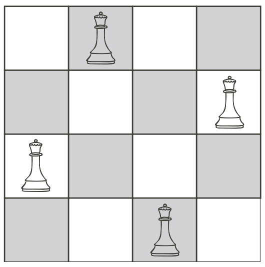

Figure 1. Example of a Constraint Satisfaction Problem (4-queens)

distributed interpretation tasks [16], and multi-agent truth maintenance tasks [13]) can be formalized as distributed CSPs. Therefore, we can consider distributed algorithms for solving distributed CSPs as an important infrastructure in MAS.

This paper gives an overview of the existing research on distributed CSPs. First, we show the problem definition of a normal, centralized CSP and algorithms for solving CSPs (Section 2). Next, we show the problem definition of distributed CSPs (Section 3). Then, we describe several MAS application problems that can be formalized as distributed CSPs (Section 4). Next, we show a series of algorithms for solving distributed CSPs (Section 5). Then, we show the way for extending the basic problem formalization of distributed CSPs, i.e., handling multiple local variables, and dealing with over-constrained problems (Section 6). Finally, we compare the efficiency of these algorithms (Section 7).

2. Constraint Satisfaction Problem

2.1. Problem Definition

Formally, a CSP consists of n variables $x_1, x_2, \ldots, x_n$, whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively, and a set of constraints on their values. In general, 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. Since constraint satisfaction is NP-complete in general, a trial-and-error exploration of alternatives is inevitable.

Note that there is no restriction about the form of the predicate. It can be a mathematical or logical formula, or any arbitrary relation defined by a tuple of variable values. In particular, we sometimes use a prohibited combination of variable values for representing a constraint. This type of constraint is called a *nogood.*

For example, in the 4-queens problem, it is obvious that only one queen can be placed in each row (or each column). Therefore, we can formalize this problem as a CSP, in which there are four variables x_1, x_2, x_3, and x_4, each of which corresponds to the position of a queen in each row. The domain of a variable is $\{1, 2, 3, 4\}$. A solution is a combination of the values of these variables. The constraints that the queens do not threaten each other can be represented as predicates, e.g., a constraint between x_i and x_j can be represented as $x_i \neq x_j \wedge \mid i - j \mid \neq \mid x_i - x_j \mid$.

2.2. Algorithms for Solving CSPs

Algorithms for solving CSPs can be divided into two groups, i.e., search algorithms and consistency algorithms. The search algorithms for solving CSPs can be further divided into two groups, i.e., backtracking algorithms and iterative improvement algorithms.

2.2.1. *Backtracking*

A backtracking algorithm is a basic, systematic search algorithm for solving CSPs. In this algorithm, a value assignment to a subset of variables that satisfies all of the constraints within the subset is constructed. This value assignment is called a partial solution. A partial solution is expanded by adding new variables one by one, until it becomes a complete solution. When for one variable, no value satisfies all of the constraints with the partial solution, the value of the most recently added variable to the partial solution is changed. This operation is called backtracking. Although backtracking is a simple depth-first tree search algorithm, many issues must be considered to improve efficiency. For example, the order of selecting variables and values greatly affects the efficiency of the algorithm. Various heuristics have been developed during the long history of CSP studies.

A value-ordering heuristic called min-conflict heuristic [17] is a very successful one among these heuristics. In the min-conflict backtracking, each variable has a tentative initial value. The tentative initial value is revised when the variable is added to the partial solution. Since it is a backtracking algorithm, the revised value must satisfy all of the constraints with the variables in the partial solution. If there exist multiple values that satisfy all constraints with the partial solution, we choose the one that satisfies as many constraints with tentative initial values as possible.

2.2.2. *Iterative Improvement*

In iterative improvement algorithms, as in the min-conflict backtracking, all variables have tentative initial values. However, no consistent partial solution is constructed. A *flawed* solution that contains all variables is revised by using hill-climbing search. The min-conflict heuristic can be used for guiding the search process, i.e., a variable value is changed so that the number of constraint violations is minimized.

Since these algorithms are hill-climbing search algorithms, occasionally they will be trapped in *local-minima*. Local-minima are states that violate some constraints, but the number of constraint violations cannot be decreased by changing any single variable value. Various methods have been proposed for escaping from local-minima. For example, in the breakout algorithm [18], a weight is defined for each constraint (the initial weight is 1). The summation of the weights of violated constraints is used as an evaluation value. When trapped in a local-minimum, the breakout algorithm increases the weights of violated constraints in the current state by 1 so that the evaluation value of the current state becomes larger than those of the neighboring states.

In iterative improvement algorithms, a mistake can be revised without conducting an exhaustive search, that is, the same variable can be revised again and again. Therefore, these algorithms can be efficient, but their completeness cannot be guaranteed.

There exist several hybrid-type algorithms of backtracking and iterative improvement. For example, the weak-commitment search algorithm [24] is based on the min-conflict backtracking. However, in this algorithm, when for one variable no value satisfies all of the constraints with the partial solution, instead of changing one variable value, the whole partial solution is abandoned. The search process is restarted using the current value assignments as new tentative initial values. This algorithm is similar to iterative improvement type algorithms since the new tentative initial values are usually better than the initial values in the previous iteration.

2.2.3. *Consistency Algorithms*

Consistency algorithms [15] are preprocessing algorithms that reduce futile backtracking. Consistency algorithms can be classified by the notion of k-consistency [8]. A CSP is k-consistent iff given any instantiation of any $k-1$ variables satisfying all the constraints among those variables, it is possible to find an instantiation of any kth variable such that the k values satisfy all the constraints among them. If there are n variables in a CSP and the CSP is k-consistent for all $k \leq n$, then a solution can be obtained immediately without any backtracking. However, achieving such a high degree of consistency requires too many computational costs, so we must find an appropriate combination of consistency algorithms and backtracking so that the total search costs are minimized.

For further readings on CSPs, Tsang's textbook [22] on constraint satisfaction covers topics from basic concepts to recent research results. There are several concise overviews of constraint satisfaction problems, such as [6, 15].

3. Problem Definition of Distributed CSP

A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents. Finding a value assignment to variables that satisfies inter-agent constraints can be viewed as achieving coherence or consistency among agents. Achieving coherence or consistency is one of the main research topics in MAS. As described in Section 4, various application problems in MAS can be formalized as distributed CSPs, by extracting the essential part of the problems. Once we formalize our problem as a distributed CSP, we don't have to develop the algorithms for solving it from scratch, since various algorithms for solving distributed CSPs have been developed.

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.

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 l knows a constraint predicate p_k is represented as $\mathit{known}(p_k, l)$.

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 $x_j = d_j$.

It must be noted that although algorithms for solving distributed CSPs seem similar to parallel/distributed processing methods for solving CSPs [3, 30], the research motivations are fundamentally different[1]. 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. For example, when each agent is designed/owned by a different person/organization, there already exist multiple agents, each of which has different and partial knowledge about the global problem. 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 a single agent, this agent can solve the problem alone by 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. 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.

4. Application Problems of Distributed CSPs

Various application problems in MAS can be formalized as distributed CSPs. For example, a multi-agent truth maintenance system [13] is a distributed version of a truth maintenance system [7]. In this system, there exist multiple agents, each of which has its own truth maintenance system (Figure 2). 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 multi-agent truth maintenance task can be formalized as a distributed CSP, where a piece of uncertain data is a variable whose value can be IN or OUT, and dependencies are constraints. For example, in Figure 2, we can represent the dependencies as nogoods, such as {(fly, IN), (has-wing, OUT)}, {(mammal, IN), (bird, IN)}, {(bird, IN),(penguin, OUT),(fly, OUT)}, etc.

[1] Of course, even the research motivations are different, the same algorithm might be useful for both. However, as far as the authors' know, existing parallel/distributed processing methods for solving CSPs are not suitable for distributed CSPs, since they usually require some global knowledge/control among agents.

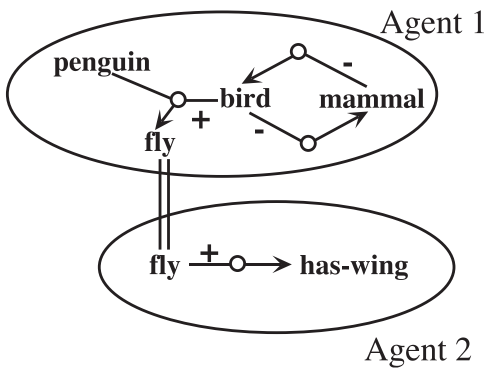

Figure 2. Multi-agent Truth Maintenance System

Another class of problems that can be formalized as a distributed CSP is resource allocation problems. If the problem is allocating tasks or resources to agents and there exist inter-agent constraints, such a problem can be formalized as a distributed CSP by viewing each task or resource as a variable and the possible assignments as values. For example, the multi-stage negotiation protocol [4] deals with the case in which tasks are not independent and there are several ways to perform a task (plans). The goal of the multi-stage negotiation is to find the combination of plans which enables all tasks can be executed simultaneously.

We show the example problem of a communication network used in the multi-stage negotiation protocol in Figure 3. This communication network consists of multiple communication sites (e.g., A-1, B-2), and communication links (e.g., L-5, L-11). These sites are geographically divided into several regions (e.g., A, B), and each region is controlled by different agents. These agents try to establish communication channels according to connection requests (goals) under the capacity constraints of communication links. Each agent recognizes a part of a global plan for establishing a channel called a plan fragment. For example, let us assume one goal is connecting A-1 and D-1. Agent A recognizes two plan fragments 1A and 2A, where 1A uses L-1 and L-2, and 2A uses L-1 and L-12. Such a problem can be easily formalized as distributed CSPs, namely, each agent has a variable that represents each goal, and possible values of the variable are plan fragments.

Also, in [11], a multi-agent model for resource allocation problems was developed. In this formalization, there are task agents and resource agents, and these agents cooperatively allocate shared resources with a limited capacity. Such problems also can be formalized as distributed CSPs. However, in this case, a task agent that has a variable/task does

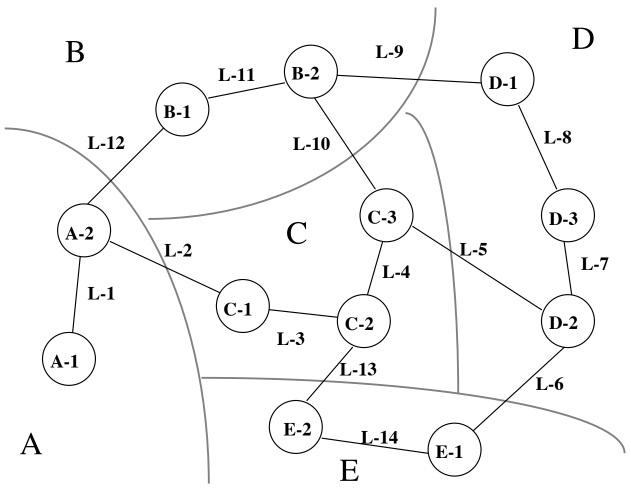

Figure 3. Distributed Resource Allocation Problem

not have knowledge about the constraints associated with its variable. Therefore, a task agent must communicate with constraint agents to find out whether the related constraints are satisfied or not.

Time-tabling tasks are another class of application problems that can be formalized as distributed CSPs. For example, the nurse time-tabling task described in [20] involves assigning nurses to shifts in each department of a hospital. Although the time-table of each department is basically independent, there exist inter-agent constraints involving the transportation of nurses. In this work, a real-life problem with 10 departments, 20 nurses in each department, and 100 weekly assignments was solved using distributed CSP techniques.

Many other application problems that are concerned with finding a consistent combination of agent actions/decisions (e.g., distributed scheduling [14, 21] and distributed interpretation problems [16]) can be formalized as distributed CSPs.

5. Algorithms for Solving Distributed CSPs

In this section, we make the following assumptions for simplicity in describing the algorithms.

1. Each agent has exactly one variable.

2. All constraints are binary.

3. Each agent knows all constraint predicates relevant to its variable.

Relaxing assumptions 2 and 3 to general cases is relatively straightforward. We show how to deal with the case where an agent has multiple local variables in Section 6.1. 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. For an agent x_i, we call a set of agents, each of which is directly connected to x_i by a link, as *neighbors* of x_i.

We show the classification of the search algorithms described in this paper (Table I). These algorithms are classified by the basic algorithm that is based on (backtracking, iterative improvement, and hybrid), and the type of problems it mainly deals with (distributed CSPs with a single local variable, multiple local variables, and distributed partial CSPs). These algorithms will be described in the following two sections.

Table I. Algorithms for Solving Distributed CSPs

	single	multiple	partial
backtracking	asynchronous BT (Section 5.1)		asynchronous IR (Section 6.2.3)
Iterative Improvement	distributed breakout (Section 5.3)		iterative DB (Section 6.2.2)
hybrid	asynchronous WS (Section 5.2)	agent-ordering AWS variable-ordering AWS (Section 6.1)	

5.1. Asynchronous Backtracking

5.1.1. *Algorithm*

The asynchronous backtracking algorithm [26, 27] is a distributed, asynchronous version of a backtracking algorithm. The main message types communicated among agents are *ok?* messages to communicate the current value, and *nogood* messages to communicate a new constraint. The procedures executed at agent x_i after receiving an *ok?* message and a *nogood* message are described in Figure 4 (i) and Figure 4 (ii), respectively.

In the asynchronous backtracking algorithm, the priority order of variables/agents is determined, and each agent communicates its tentative value assignment to neighboring agents via *ok?* messages. Each agent maintains the current value assignment of other agents from its viewpoint called *agent_view*. The priority order is determined by the alphabetical order of the variable identifiers, i.e., preceding variables in alphabetical order have higher priority. An agent changes its assignment if its current value assignment is not consistent with the assignments of higher priority agents. If there exists no value that is consistent with the higher priority agents, the agent generates a new constraint (called a *nogood*), and communicates the nogood to a higher priority agent (Figure 4 (iii)); thus the higher priority agent changes its value.

A nogood is a subset of an *agent_view*, where the agent is not able to find any consistent value with the subset. Ideally, the nogood generated in (Figure 4 (iii)) should be *minimal*, i.e., no subset of them should be a nogood. However, since finding minimal nogoods requires certain computation costs, an agent can make do with non-minimal nogoods. In the simplest case, it can use the whole *agent_view* as a nogood.

It must be noted that since each agent acts asynchronously and concurrently and agents communicate by sending messages, the *agent_view* may contain obsolete information. Even if x_i's *agent_view* says that x_j's current assignment is 1, x_j may already have changed its value. Therefore, if x_i does not have a consistent value with the higher priority agents according to its *agent_view*, we cannot use a simple control method such as x_i orders a higher priority agent to change its value, since the *agent_view* may be obsolete. Therefore, each agent needs to generate and communicate a new nogood, and the receiver of the new nogood must check whether the nogood is actually violated based on its own *agent_view*.

The completeness of the algorithm (i.e., always finds a solution if one exists, and terminates if no solution exists) is guaranteed [27]. The outline of the proof is as follows. First, we can show that agent x_1, which has the highest priority, never falls into an infinite processing loop. Then, assuming that agents x_1 to x_{k-1} ($k > 2$) are in a stable state, we can show that agent x_k never falls into an infinite processing loop. Therefore, we can prove that the agents never fall into an infinite processing loop by using mathematical induction.

5.1.2. *Example*

We show an example of an algorithm execution in Figure 5. In Figure 5 (a), after 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*, a new nogood $\{(x_1, 1), (x_2, 2)\}$ is generated. x_3 chooses the lowest priority agent in the nogood, i.e., x_2, and sends a *nogood* message. After receiving this *nogood* message, x_2 records it. This nogood, $\{(x_1, 1), (x_2, 2)\}$, contains agent x_1, which is not a neighbor of x_2. Therefore, a new link must be added between x_1 and x_2. x_2 requests x_1 to send x_1's value to x_2, adds $(x_1, 1)$ to

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

when received (nogood, $x_j$, nogood) do — (ii)
  record nogood as a new constraint;
  when nogood contains an agent $x_k$ that is not its neighbor
    do request $x_k$ to add $x_i$ as a neighbor,
      and add $x_k$ to its neighbors; end do;
  old_value ← current_value; check_agent_view;
  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;
    else select $d \in D_i$ where agent_view and $d$ are consistent;
      current_value ← $d$;
      send (ok?, ($x_i$, $d$)) to neighbors; end if; end do;

procedure backtrack
  generate a nogood $V$ — (iii)
  when $V$ is an empty nogood do
    broadcast to other agents that there is no solution,
        terminate this algorithm; end do;
  select $(x_j, d_j)$ where $x_j$ has the lowest priority in a nogood;
  send (nogood, $x_i$, $V$) to $x_j$;
  remove $(x_j, d_j)$ from agent_view;
  check_agent_view;
```

Figure 4. Procedures for Receiving Messages (Asynchronous Backtracking)

its *agent_view* (Figure 5 (b)), and checks whether its value is consistent with the *agent_view*. The *agent_view* $\{(x_1, 1)\}$ and the assignment $(x_2, 2)$ violate the received nogood $\{(x_1, 1), (x_2, 2)\}$. However, there is no other possible value for x_2. Therefore, x_2 generates a new nogood $\{(x_1, 1)\}$, and sends a *nogood* message to x_1 (Figure 5 (c)).

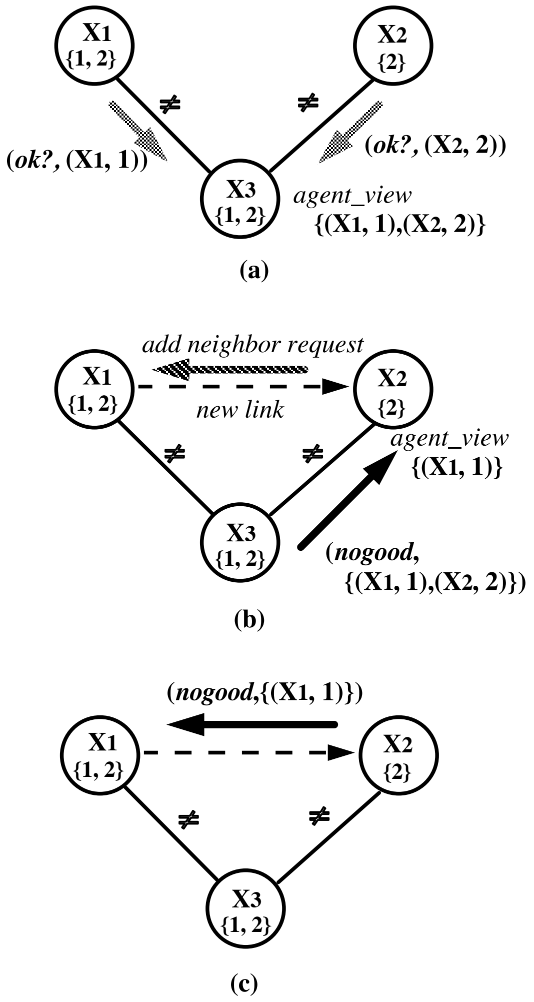

Figure 5. Example of an Algorithm Execution (Asynchronous Backtracking)

5.2. Asynchronous Weak-commitment Search

5.2.1. *Algorithm*

One limitation of the asynchronous backtracking algorithm is that the agent/variable ordering is statically determined. If the value selection of a higher priority agent is bad, the lower priority agents need to perform an exhaustive search to revise the bad decision. The asynchronous weak-commitment search algorithm [25, 27] introduces the min-conflict heuristic to reduce the risk of making bad decisions. Furthermore, the agent ordering is dynamically changed so that a bad decision can be revised without performing an exhaustive search.

In Figure 6, the procedure executed at agent x_i for checking *agent_view* is described (other procedures are basically identical to those for the asynchronous backtracking algorithm). The differences between this procedure and that for the asynchronous backtracking algorithm are as follows.

- A *priority value* is determined for each variable, and the priority value is communicated through the *ok?* message. The priority order is determined by the communicated priority values, i.e., the variable/agent with a larger priority value has higher priority (ties are broken using the alphabetical order).
- 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 using the min-conflict heuristic (Figure 6 (i)).
- When x_i cannot find a consistent value with its *agent_view*, x_i sends *nogood* messages to other agents, and increases its priority value (Figure 6 (ii)). If x_i cannot generate a new nogood, x_i will not change its priority value but will wait for the next message. This procedure is needed to guarantee the completeness of the algorithm.

The completeness of the algorithm is guaranteed. An overview of the proof is as follows. The priority values are changed if and only if a new nogood is found. Since the number of possible nogoods is finite, the priority values cannot be changed infinitely. Therefore, after a certain time point, the priority values will be stable. 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 to be complete, the asynchronous weak-commitment search algorithm is also complete.

Note that the completeness of the algorithm is guaranteed by the fact that the agents record all nogoods found so far. In practice, we can restrict the number of recorded nogoods, i.e., each agent records only a fixed number of the most recently found nogoods.

5.2.2. *Example*

An execution of the algorithm is illustrated using the distributed 4-queens problem. The initial values are shown in Figure 7 (a). 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

```
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; — (i)
        *current_value* $\leftarrow d$;
        send (**ok?**, ($x_i$, $d$, *current_priority*)) to *neighbors*;
      **end if**; **end do**;

procedure **backtrack**
    generate a nogood $V$;
    **when** $V$ is an empty nogood **do**
      broadcast to other agents that there is no solution,
      terminate this algorithm; **end do**;
    **when** $V$ is a new nogood **do** — (ii)
      send $V$ to the agents in the nogood;
      *current_priority* $\leftarrow 1 + p_{max}$,
          where $p_{max}$ is the maximal priority value of neighbors;
      select $d \in D_i$ where *agent_view* and $d$ are consistent,
        and $d$ minimizes the number of constraint violations
        with lower priority agents;
      *current_value* $\leftarrow d$;
      send (**ok?**, ($x_i$, $d$, *current_priority*)) to neighbors; **end do**;
```

Figure 6. Procedure for Checking *agent_view* (Asynchronous Weak-commitment Search)

the alphabetical order of the identifiers. Therefore, only the value of x_4 is not consistent with its *agent_view*.

Since there is no consistent value, x_4 sends *nogood* messages and increases 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 (Figure 7 (b)). Then, x_3 tries to change its value. Since there is no consistent value, x_3 sends *nogood* messages, and increases 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 (Figure 7 (c)). After that, x_1 changes its value to 2, and a solution is obtained (Figure 7 (d)).

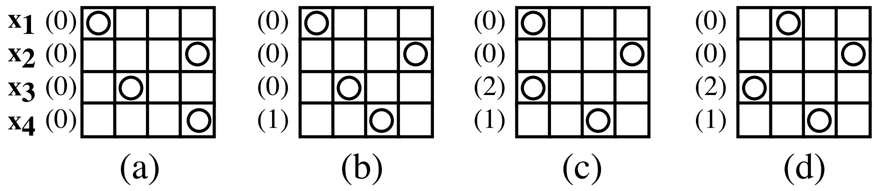

Figure 7. Example of an Algorithm Execution (Asynchronous Weak-commitment Search)

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 (assigning its value to 1) can be revised without an exhaustive search.

5.3. Distributed Breakout Algorithm

5.3.1. *Algorithm*

As described in Section 2.2.2, in the breakout algorithm, a weight is defined for each constraint, and the summation of the weights of constraint violating pairs is used as an evaluation value. The basic ideas for implementing the distributed version of the breakout algorithm are as follows.

- To guarantee that the evaluation value is improved, neighboring agents exchange values of possible improvements, and only the agent that can maximally improve the evaluation value is given the right to change its value. Note that if two agents are not neighbors, it is possible for them to change their values concurrently.

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

In this algorithm, two kinds of messages (*ok?* and *improve*) are communicated among neighbors. The procedures executed at agent x_i when receiving *ok?* and *improve* messages are shown in Figure 9. The agent alternates between the wait_ok? mode (Figure 9 (i)) and the wait_improve mode (Figure 9 (ii)). The *improve* message is used to communicate the possible improvement of the evaluation value.

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

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

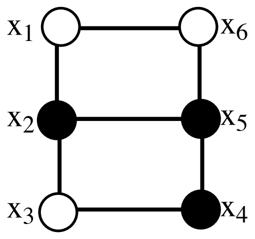

Figure 8. Example of a Distributed Graph-coloring Problem

It is obvious that if the current situation is a real local-minimum, each of the constraint-violating agents is in a quasi-local-minimum, but not vice versa. For example, Figure 8 shows one instance of a distributed graph-coloring problem, in which six agents exist. Each agent tries to determine its color so that neighbors do not have the same color (possible colors are white and black). Although x_1 is in a quasi-local-minimum, this situation is not a real local-minimum since x_5 can improve the evaluation value.

5.3.2. *Example*

We show an example of the algorithm execution in Figure 10. We assume that initial values are chosen as in Figure 10 (a). Each agent communicates this initial value via ok? messages. After receiving ok? messages from all of its neighbors, each agent calculates *current_eval* and *my_improve*, and exchanges *improve* messages. Initially, all weights are equal to 1. In the initial state, the improvements of all agents are equal to 0. Therefore, the weights of constraints (nogoods), $\{(x_1,$white$),$ $(x_6,$white$)\}$, $\{(x_2,$ black$),$ $(x_5,$ black$)\}$, and $\{(x_3,$ white$),$ $(x_4,$ white$)\}$, are increased by 1 (Figure 10 (b)).

Then, the improvements of x_1, x_3, x_4, and x_6 are 1, and the improvements of x_2 and x_5 are 0. The agents that have the right to change their values are x_1 and x_3 (each of which precedes in alphabetical order within its own neighborhood). These agents change their value from white to black (Figure 10 (c)). Then, the improvement of x_2 is 4, while the improvements of the other agents are 0. Therefore, x_2 changes its value to white, and all constraints are satisfied (Figure 10 (d)).

5.4. Distributed Consistency Algorithm

Achieving 2-consistency by multiple agents is relatively straightforward, since the algorithm can be achieved by the iteration of local processes. In [19], a distributed system that achieves arc-consistency for resource allocation tasks was developed. This system also main-

```
wait_ok? mode — (i)
when received (ok?, $x_j$, $d_j$) do
  add ($x_j$, $d_j$) to agent_view;
  when received ok? messages from all neighbors do
      send_improve;
      goto wait_improve mode; end do;
  goto wait_ok mode; end do;

procedure send_improve
  current_eval ← evaluation value of current_value;
  my_improve ← possible maximal improvement;
  new_value ← the value which gives the maximal improvement;
  send (improve, $x_i$, my_improve, current_eval) to neighbors;

wait_improve? mode — (ii)
when received (improve, $x_j$, improve, eval) do
  record this message;
  when received improve? messages from all neighbors do
    send_ok; clear agent_view;
    goto wait_ok mode; end do;
  goto wait_improve mode; end do;

procedure send_ok
  when its improvement is largest among neighbors do
    current_value ← new_value; end do;
  when it is in a quasi-local-minimum do
    increase the weights of constraint violations; end do;
  send (ok?, $x_i$, current_value) to neighbors;
```

Figure 9. Procedures for Receiving Messages (Distributed Breakout)

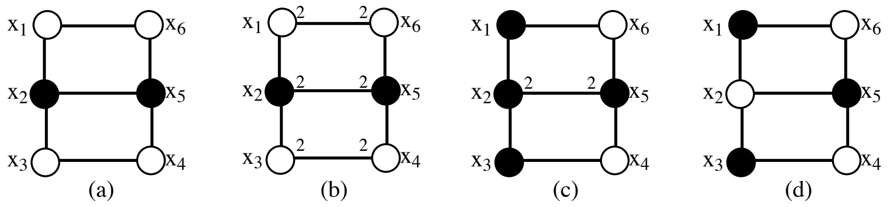

Figure 10. Example of Algorithm Execution (Distributed Breakout)

tains arc-consistency, i.e., it can re-achieve arc-consistency after dynamic changes in variables/values/constraints with a small amount of computational effort by utilizing dependencies.

Also, a higher degree of consistency can be achieved using the hyper-resolution-based consistency algorithm [5]. In [29], a distributed consistency algorithm that achieves k-consistency is described. In this algorithm, agents communicates nogoods among themselves, and generate new nogoods whose length are less than k using the hyper-resolution rule.

6. Extensions of Problem Formalization

6.1. Handling Multiple Local Variables

So far, we assume that each agent has only one local variable. Although the developed algorithms can be applied to the situation where one agent has multiple local variables by the following methods, both methods are neither efficient nor scalable to large problems.

Method 1: each agent finds all solutions to its local problem first.
By finding all solutions, the given problem can be re-formalized as a distributed CSP, in which each agent has one local variable whose domain is a set of obtained local solutions. Then, agents can apply algorithms for the case of a single local variable. The drawback of this method is that when a local problem becomes large and complex, finding all the solutions of a local problem becomes virtually impossible.

Method 2: an agent creates multiple virtual agents, each of which corresponds to one local variable, and simulates the activities of these virtual agents.
For example, if agent k has two local variables x_i, x_j, we assume that there exist two virtual agents, each of which corresponds to either x_i or x_j. Then, agent k simulates the concurrent activities of these two virtual agents. In this case, each agent does not have to predetermine all the local solutions. However, since communicating with other agents is usually more expensive than performing local computations, it is wasteful to simulate the activities of multiple virtual agents without distinguishing the communications between virtual agents within a single real agent, and the communications between real agents.

In [1], prioritization among agents was introduced to handle multiple local variables (we call this algorithm *agent-ordering AWS*). In this algorithm, each agent tries to find a local solution that is consistent with the local solutions of higher priority agents. If there exists no such local solution, backtracking or modification of the prioritization occurs. Various heuristics for determining good ordering among agents were examined in [1]. This approach is similar to method 1 described above, except that each agent searches for its local solutions only as required, instead of finding all solutions in advance.

In [28], an extension of the asynchronous weak-commitment search algorithm that can handle multiple local variables (*variable-ordering AWS*) was developed. Although this algorithm is similar to method 2, it has the following characteristics.

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

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

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

Each variable must satisfy constraints with higher priority variables. Therefore, changing the value of a lower priority variable before the value of a higher priority variable is fixed is usually wasteful. Accordingly, an agent changes the value of the highest priority variable first. Also, by sending messages to other agents only when an agent finds a consistent local solution, agents can reduce the number of interactions among themselves. By using this algorithm, if the local solution selected by a higher priority agent is bad, a lower priority agent does not have to exhaustively search its local problem. It simply increases the priority values of certain variables that violate constraints with the bad local solution.

6.2. Distributed Partial Constraint Satisfaction

Many application problems in MAS that can be formalized as distributed CSPs might be over-constrained, i.e., a problem instance has too many constraints and there exists no solution that satisfies all constraints completely. In that case, the distributed constraint satisfaction algorithms described in Section 5 fail to provide any useful information, i.e., if the algorithm is complete, it simply reports that the problem has no solution, and if the algorithm is incomplete, it never terminates. However, in many application problems, we would rather have an incomplete solution that satisfies as many constraints as possible.

For example, in a distributed resource allocation problem [4], agents try to find the combination of plans that enables all agents' tasks to be executed under resource constraints. If agents need to perform their tasks with scarce resources, the problem instance can be mapped into an over-constrained distributed CSP. For such a problem instance, we want to know how the instance should be modified in order to make it solvable.

Furthermore, in a distributed interpretation problem [16], each agent is assigned a task to interpret a part of sensor data. The goal of agents is to find a globally consistent interpretation by communicating their local interpretations. If some agents make incorrect interpretations due to noisy sensors, the problem instance can be over-constrained. For such a problem instance, we would like to get an approximate solution.

In this section, we first show a general framework for dealing with over-constrained distributed CSPs called *distributed partial CSPs*. Then, we describe two subclasses of distributed partial CSPs, i.e., *distributed maximal CSPs*, where each agent tries to find the variable values that minimize the maximal number of violated constraints over agents, and *distributed hierarchical CSPs*, where each agent tries to find the variable values that minimize the maximal degree of importance of violated constraints over agents.

6.2.1. *Problem Formalization*

Intuitively, in a distributed partial CSP, agents try to find a solvable distributed CSP and its solution by relaxing an original over-constrained distributed CSP. How much the original problem is relaxed is measured by a globally defined function (global distance function). Agents prefer the problem closer to the original problem, and in some case they want to make the relaxation minimal.

A distributed partial CSP can be formalized using terms in a *partial CSP*, which has been presented in [9] for dealing with over-constrained centralized CSPs. It can be defined using the following components:

- a set of agents, $A = \{1, 2, \ldots, m\}$,
- $\langle (P_i, U_i), (PS_i, \leq), M_i \rangle$ for each agent i,
- $(G, (N, S))$.

For each agent i, P_i is an original CSP (a part of an original distributed CSP), and U_i is a set of universes[2], i.e., a set of potential values for each variable in P_i. Furthermore, PS_i is a set of (relaxed) CSPs including P_i, and $\leq$ is a partial order over PS_i. Also, M_i is a locally-defined distance function over the problem space. G is a global distance function over distributed problem spaces, and (N, S) are necessary and sufficient bounds on the global distance between the original distributed CSP (a set of P_is of all agents) and some solvable distributed CSP (a set of solvable CSPs of all agents, each of which comes from PS_i).

A *solution* to a distributed partial CSP is a combination of a solvable distributed CSP and its solution, where the global distance between an original distributed CSP and the solvable distributed CSP is less than N. Any solution to a distributed partial CSP will suffice if the global distance between an original distributed CSP and the solvable distributed CSP is not more than S. An *optimal solution* to a distributed partial CSP is a solution in which the global distance between an original distributed CSP and the solvable distributed CSP is minimal.

This general model can be specialized in various ways. In this paper, we are going to show two subclasses of problems: *distributed maximal CSPs* and *distributed hierarchical CSPs.*

6.2.2. *Distributed Maximal CSP*

A distributed maximal CSP is a problem where each agent tries to find the variable values that minimize the maximal number of violated constraints over agents. This problem corresponds to finding an optimal solution for the following distributed partial CSP.

- For each agent i, PS_i is made up of all possible CSPs that can be obtained by removing constraints from P_i.
- For each agent i, a distance d_i between P_i and a CSP in PS_i is measured as the number of constraints removed.
- A global distance is measured as $\max_{i \in A} d_i$.

When each agent tries to satisfy as many of its *own* constraints as possible, this solution criterion can be considered a reasonable compromise among agents, since the number of the constraint violations in the worst agent is minimized.

Two algorithms for solving distributed maximal CSPs were presented in [12]. One is the *synchronous branch and bound algorithm*,

[2] A universe is used for relaxing a problem by enlarging a variable domain. As noted in [9], all kinds of relaxation of a CSP can be expressed in terms of relaxing a constraint (enlarging permitted values for variables) by introducing a universe.

and the other is the *iterative distributed breakout algorithm*. The synchronous branch and bound algorithm is a very simple algorithm that simulates branch and bound operations in a distributed environment. Since it systematically searches in a distributed problem space in a sequential manner, it is guaranteed to be complete. On the other hand, the obvious drawbacks of this algorithm are that agents must act in a predefined sequential order, and global knowledge is required to determine such a sequential order.

The iterative distributed breakout algorithm repeatedly applies the distributed breakout algorithm to distributed CSPs. At the first stage of the iteration, an agent sets a predetermined uniform value, *ub*, to its target distance and starts the distributed breakout algorithm. This distributed breakout algorithm is modified so that an agent can detect the fact that all agents reach the state where the number of violated constraints is less than the current target distance, i.e., *ub*. By detecting this fact, an agent decreases the current target distance, propagates the new target distance to all agents using *improve* messages, and moves to the next stage. At the next stage, where all agents know the new target distance, the distributed breakout algorithm is started again for the new target distance. The agents continue this iteration until the value of the target distance becomes zero.

In the iterative distributed breakout algorithm, agents perform constraint checking and value changing in parallel. Thus, this algorithm can be more efficient than the synchronous branch and bound algorithm. However, this algorithm is not complete, i.e., it may fail to find an optimal solution because the distributed breakout algorithm at a certain stage may fail to find a solution.

In [10], a multi-agent model for solving maximal CSPs was developed. This model is an extension of the multi-agent model for the resource allocation problem described in Section 4. In this model, each variable performs its own simulated annealing process combined with the min-conflict heuristic.

6.2.3. *Distributed Hierarchical CSP*

A distributed hierarchical CSP is a problem where each agent tries to find the variable values that minimize the maximal importance level of violated constraints over agents. In this problem, each constraint is labeled a positive integer called *importance value*, which represents the importance of the constraint. A constraint with a larger importance value is considered more important.

A distributed hierarchical CSP corresponds to finding an optimal solution to the following distributed partial CSP.

- For each agent i, PS_i is made up of $\{P_i^0, P_i^1, P_i^2, \ldots\}$, where P_i^α is a CSP that is obtained from P_i by removing every constraint with an importance value of α or less.
- For each agent i, a distance d_i between P_i and P_i^α is defined as α.
- A global distance is measured as $\max_{i \in A} d_i$.

This solution criterion can be considered another reasonable compromise among agents.

A simple algorithm for solving distributed hierarchical CSPs, called the *asynchronous incremental relaxation algorithm*, was presented in [23]. This algorithm repeatedly applies the asynchronous backtracking algorithm to distributed hierarchical CSPs in the following way.

In the first stage, agents try to solve an original distributed CSP, in which all agents have all constraints, by using the asynchronous backtracking algorithm. If the problem is solved, it is not an over-constrained situation and a distributed hierarchical CSP is optimally solved. If the problem is found to be over-constrained (this fact can be identified by the agent that produces an empty nogood), agents need to give up constraints that are less important than a certain threshold. This threshold is efficiently calculated using the importance values of constraints that cause the empty nogood. At the next stage, agents again apply the asynchronous backtracking algorithm to the relaxed distributed CSP. If it is solved, the optimal solution of the distributed hierarchical CSP is found; otherwise, further relaxation occurs in the same way. Agents continue this process until a solution is found for some relaxed distributed CSP.

If the number of possible importance values is finite, this algorithm is guaranteed to be complete because the asynchronous backtracking algorithm at each stage is complete.

7. Comparison of Distributed CSP Algorithms

In this section, we compare the search algorithms for solving distributed CSPs with a single local variable, i.e., the asynchronous backtracking, the asynchronous weak-commitment search, and distributed breakout algorithm. 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. One cycle corresponds to a series of agent actions, in which an agent recognizes the state of the world, then decides its response to that state, and communicates its decisions.

We first show evaluation results using distributed graph-coloring problems, where the number of variables/agents $n = 60, 90$, and 120, and the number of constraints $m = n \times 2$, and the number of possible colors is 3. We generated 10 different problems, and for each problem, 10 trials with different initial values were performed (100 trials in all). We set the limit for the number of cycles at 1,000, and terminated the algorithm if this limit was exceeded; we counted the result as 1,000. We show the average of required cycles, and the ratio of problems completed successfully to the total number of problems in Table II.

Clearly, the asynchronous weak-commitment search outperforms the asynchronous backtracking, since in the asynchronous weak-commitment search, a mistake can be revised without conducting an exhaustive search.

Table II. Comparison between Asynchronous Backtracking and Asynchronous Weak-commitment Search ("sparse" Problems)

Algorithm		n		
		60	90	120
asynchronous backtracking	ratio	13%	0%	0%
	cycles	917.4	—	—
asynchronous weak-commitment search	ratio	100%	100%	100%
	cycles	59.4	70.1	106.4

Then, we compare the asynchronous weak-commitment search and the distributed breakout. Table III shows the results where the number of variables/agents $n = 90, 120$, and 150, and the number of constraints $m = n \times 2$, and the number of possible colors is 3. Table IV shows the results where the number of constraints $m = n \times 2.7$. When $m = n \times 2$, we can assume that constraints among agents are rather sparse. The setting where $m = n \times 2.7$ has been identified as a critical setting which produces particularly difficult, phase-transition problems in [2].

We can see that the distributed breakout outperforms the asynchronous weak-commitment search when problem instances are critically difficult. In the distributed breakout, each mode (wait_ok? or wait_improve) requires one cycle. Therefore, each agent can change its value at most once in two cycles, while in the asynchronous weak-commitment search algorithm, each agent can change its value at every cycle. When a problem is critically difficult, it is worthwhile to introduce more control among agents, and change their values more cautiously.

Table III. Comparison between Asynchronous Weak-commitment Search and Distributed Breakout ("sparse" Problems)

Algorithm		n		
		90	120	150
distributed	ratio	100%	100%	100%
breakout	cycles	150.8	210.1	278.8
asynchronous	ratio	100%	100%	100%
weak-commitment search	cycles	70.1	106.4	159.2

Table IV. Comparison between Asynchronous Weak-commitment Search and Distributed Breakout ("critical" Problems)

Algorithm		n		
		90	120	150
distributed	ratio	100%	100%	100%
breakout	cycles	517.1	866.4	1175.5
asynchronous	ratio	97%	65%	29%
weak-commitment search	cycles	1869.6	6428.4	8249.5

8. Conclusions and Future Issues

This paper provided an overview of the existing research on distributed CSPs. We described the problem definition of distributed CSP, and showed several MAS application problems that can be formalized as distributed CSPs. Furthermore, we described a series of algorithms for solving distributed CSPs.

There are many remaining research issues concerning distributed CSPs, including the following items.

– Much more work is needed to develop better algorithms, especially for multiple local variables and distributed partial constraint satisfaction problems.

- For centralized CSPs, it is well known that the most difficult problem instances are in the phase-transition region [2], where the provability that a problem instance has a solution is about 0.5. On the other hand, we don't have a clear idea about what kinds of problem instances of distributed CSPs would be most difficult when a problem has multiple local variables. More theoretical/experimental work is needed to identify how the ratio of inter/intra constraints would affect the problem difficulty.

- We cannot expect that a single algorithm can efficiently solve all types of problems. More theoretical/experimental evaluations are needed to clarify the characteristics of algorithms.

- In MAS application problems, it is common that the problem setting (environment) changes dynamically, and agents must make their decisions under real-time constraints. Dynamic/real-time aspects should be incorporated into the distributed CSP formalization.

References

1. Armstrong, A. and E. Durfee: 1997, 'Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems'. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence.* pp. 620–625.
2. Cheeseman, P., B. Kanefsky, and W. Taylor: 1991, 'Where the really hard problems are'. In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.* pp. 331–337.
3. Collin, Z., R. Dechter, and S. Katz: 1991, 'On the Feasibility of Distributed Constraint Satisfaction'. In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.* pp. 318–324.
4. Conry, S. E., K. Kuwabara, V. R. Lesser, and R. A. Meyer: 1991, 'Multistage Negotiation for Distributed Constraint Satisfaction'. *IEEE Transactions on Systems, Man and Cybernetics* **21**(6), 1462–1477.
5. de Kleer, J.: 1989, 'A Comparison of ATMS and CSP Techniques'. In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* pp. 290–296.
6. Dechter, R.: 1992, 'Constraint Networks'. In: S. C. Shapiro (ed.): *Encyclopedia of Artificial Intelligence.* New York: Wiley-Interscience Publication, pp. 276–285.
7. Doyle, J.: 1979, 'A Truth Maintenance System'. *Artificial Intelligence* **12**, 231–272.
8. Freuder, E. C.: 1978, 'Synthesizing Constraint Expressions'. *Communications ACM* **21**(11), 958–966.
9. Freuder, E. C. and R. J. Wallace: 1992, 'Partial Constraint Satisfaction'. *Artificial Intelligence* **58**(1–3), 21–70.

10. Ghedira, K.: 1994, 'A Distributed Approach to Partial Constraint Satisfaction Problems'. In: J. W. Perram and J.-P. Müller (eds.): *Distributed Software Agents and Applications*. Springer-Verlag, pp. 106–122. Lecture Notes in Computer Science 1069.
11. Ghedira, K. and G. Verfaillie: 1992, 'A Multi-Agent Model for the Resource Allocation Problem: A Reactive Approach'. In: *Proceedings of the Tenth European Conference on Artificial Intelligence*. pp. 252–254.
12. Hirayama, K. and M. Yokoo: 1997, 'Distributed partial constraint satisfaction problem'. In: *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP-97)*. pp. 222–236. Lecture Notes in Computer Science 1330.
13. Huhns, M. N. and D. M. Bridgeland: 1991, 'Multiagent Truth Maintenance'. *IEEE Transactions on Systems, Man and Cybernetics* **21**(6), 1437–1445.
14. Liu, J.-S. and K. P. Sycara: 1996, 'Multiagent Coordination in Tightly Coupled Task Scheduling'. In: *Proceedings of the Second International Conference on Multi-Agent Systems*. pp. 181–188.
15. Mackworth, A. K.: 1992, 'Constraint Satisfaction'. In: S. C. Shapiro (ed.): *Encyclopedia of Artificial Intelligence*. New York: Wiley-Interscience Publication, pp. 285–293.
16. Mason, C. and R. Johnson: 1989, 'DATMS: A Framework for Distributed Assumption Based Reasoning'. In: L. Gasser and M. Huhns (eds.): *Distributed Artificial Intelligence vol. II*. Morgan Kaufmann, pp. 293–318.
17. Minton, S., M. D. Johnston, A. B. Philips, and P. Laird: 1992, 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems'. *Artificial Intelligence* **58**(1–3), 161–205.
18. Morris, P.: 1993, 'The Breakout Method for Escaping From Local Minima'. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*. pp. 40–45.
19. Prosser, P., C. Conway, and C. Muller: 1992, 'A Constraint Maintenance System for the Distributed Allocation Problem'. *Intelligent Systems Engineering* **1**, 76–83.
20. Solotorevsky, G. and E. Gudes: 1996, 'Solving a Real-life Time Tabling and Transportation Problem Using Distributed CSP Techniques'. In: *Proceedings of CP '96 Workshop on Constraint Programming Applications*. pp. 123–131.
21. Sycara, K. P., S. Roth, N. Sadeh, and M. S. Fox: 1991, 'Distributed Constrained Heuristic Search'. *IEEE Transactions on Systems, Man and Cybernetics* **21**(6), 1446–1461.
22. Tsang, E.: 1993, *Foundations of Constraint Satisfaction*. Academic Press.
23. Yokoo, M.: 1993, 'Constraint Relaxation in Distributed Constraint Satisfaction Problem'. In: *5th International Conference on Tools with Artificial Intelligence*. pp. 56–63.
24. Yokoo, M.: 1994, 'Weak-commitment Search for Solving Constraint Satisfaction Problems'. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence*. pp. 313–318.
25. Yokoo, M.: 1995, 'Asynchronous Weak-commitment Search for Solving Distributed Constraint Satisfaction Problems'. In: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP-95)*. pp. 88–102. Lecture Notes in Computer Science 976.
26. Yokoo, M., E. H. Durfee, T. Ishida, and K. Kuwabara: 1992, 'Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving'. In:

Proceedings of the Twelfth IEEE International Conference on Distributed Computing Systems. pp. 614–621.

27. Yokoo, M., E. H. Durfee, T. Ishida, and K. Kuwabara: 1998, 'The Distributed constraint satisfaction problem: formalization and algorithms'. *IEEE Transactions on Knowledge and Data Engineering* **10**(5), 673–685.
28. Yokoo, M. and K. Hirayama: 1998, 'Distributed constraint satisfaction algorithm for complex local problems'. In: *Proceedings of the Third International Conference on Multi-Agent Systems*.
29. Yokoo, M., T. Ishida, and K. Kuwabara: 1990, 'Distributed Constraint Satisfaction for DAI Problems'. In: *10th International Workshop on Distributed Artificial Intelligence*.
30. Zhang, Y. and A. Mackworth: 1991, 'Parallel and distributed algorithms for finite constraint satisfaction problems'. In: *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. pp. 394–397.

Address for Offprints: Makoto Yokoo
NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237 Japan
e-mail: yokoo@cslab.kecl.ntt.co.jp
phone: +81-774-93-5231
fax : +81-774-93-5285