5th IEEE International Conference on Tools with Artificial Intelligence '93, pp.56–63

Constraint Relaxation in Distributed Constraint Satisfaction Problems

Makoto Yokoo
NTT Communication Science Laboratories
Hikaridai, Seika-cho,
Soraku-gun, Kyoto 619-02 Japan
e-mail: yokoo@cslab.kecl.ntt.jp

Abstract

The distributed constraint satisfaction problem (DCSP) formulation has recently been identified as a general framework for formalizing various Distributed Artificial Intelligence problems. In this paper, we extend the DCSP formalization by introducing the notion of importance values of constraints. With these values, we define a solution criterion for DCSPs that are over-constrained (where no solution satisfies all constraints completely). We show that agents can find an optimal solution with this criterion by using the *asynchronous incremental relaxation algorithm*, in which the agents iteratively apply the *asynchronous backtracking algorithm* [10] to solve a DCSP, while incrementally relaxing less important constraints. In this algorithm, agents act asynchronously and concurrently, in contrast to traditional sequential backtracking techniques, while guaranteeing the completeness of the algorithm and the optimality of the optimality. Furthermore, we show that, in this algorithm, agents can avoid redundant computation and achieve a five-fold speed-up in example problems by maintaining the dependencies between constraint violations (nogoods) and constraints.

1 Introduction

Distributed Artificial Intelligence (DAI) is a subfield of AI that is concerned with how a set of artificially intelligent agents can work together to solve problems. In [10], we presented the idea of viewing DAI problems as distributed constraint satisfaction problem in order to develop a general framework for DAI. A distributed constraint satisfaction problem (DCSP) is a very general class of problems, and it is pointed out in [1], [6], [9] that distributed resource allocation problems, distributed scheduling problems, and multi-agent truth maintenance tasks, respectively, can be represented in this DCSP formalization. In [10], we developed an algorithm called *asynchronous backtracking*, which can find a solution that satisfies all constraints.

On the other hand, [3], [4], [5] have shown that when real-life problems are formalized as centralized, non-distributed CSPs, these problems are often over-constrained, where no solution satisfies all constraints completely; thus constraint relaxation is inevitable. Because many realistic problems are also inherently distributed, it is important to extend the DCSP formalization to over-constrained situations.

When finding a solution by relaxing constraints, we need some criterion to measure the goodness of solutions, each of which satisfies only a subset of constraints. In this paper, we extend the formalization of the DCSP by introducing importance values of constraints, and define a solution criterion that uses these importance values of constraints.

We have developed an algorithm called the *asynchronous incremental relaxation* algorithm. In this algorithm, agents iteratively apply the asynchronous backtracking algorithm to solve a DCSP, while incrementally relaxing less important constraints. These agents act asynchronously and concurrently, in contrast to traditional sequential backtracking techniques, while guaranteeing the completeness of the algorithm and the optimality of the solution. Furthermore, we show that the agents can avoid redundant computation and achieve almost a five-fold speed-up in example problems by maintaining the dependencies between constraint violations (nogoods) and constraints[1].

[1] These dependencies are totally different from the dependencies used in dependency-directed backtracking [2], in which dependencies between variable values and constraint violations are maintained. In our algorithm, dependencies between constraint violations (nogoods) and constraints are maintained, since constraints are changed dynamically by constraint relaxation.

In the remainder of this paper, we first introduce the formalization and algorithms for centralized CSPs, to which importance values of constraints are introduced (Section 2). Then, we show the formalization of the DCSP (Section 3), and describe the basic asynchronous incremental relaxation algorithm, its extension by introducing dependencies between constraint violations (nogoods) and constraints (Section 4), and show the experimental results (Section 5). Furthermore, we discuss the applicability of this algorithm to a more general class of solution criteria (Section 6).

2 Centralized constraint satisfaction problem

2.1 Formalization

The definition of a CSP to which importance values of constraints are introduced is as follows.

- There exist m variables $x_1, x_2, ..., x_m$, each of which takes its value from a finite, discrete domain $D_1, D_2, ..., D_m$, respectively.
- There exists a set of constraints P. One constraint $p_k(x_{k1}, \ldots, x_{kj})$ is a predicate defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the instantiations of the variables are consistent with each other.
- For each constraint predicate p_k, an importance value c_k is defined (c_k is a positive real value). This value represents the importance of the constraint, i.e., the greater this value is, the more important the constraint is.
- The goal is to find a solution that satisfies as many important constraints as possible. We define that a solution S is preferred to solution S', iff S satisfies all constraints whose importance values are greater than some positive real value c, while S' does not satisfy at least one constraint whose importance value is greater than c. The goal is to find a solution S, where no other solution is preferred to S.

Let's define an evaluation function F for solutions, where $F(S)$ returns 0 if S satisfies all constraints (recall that importance values are positive); otherwise, $F(S)$ returns the maximum of the importance values of constraints not satisfied by S. We can represent the preference relation by using this evaluation function, i.e., a solution S is preferred to solution S' iff $F(S)$ is less than $F(S')$. Therefore, the goal can be stated as finding a solution S that minimizes $F(S)$.

When we are formalizing real-life problems as CSPs, it is natural to assume that all constraints are not equally important, and that the designer of a problem can give a criterion for relaxing constraints. In this formalization, we assume that such a criterion can be represented as importance values of constraints[2]. A similar criterion called *locally-predicate-better* is used in [4] for dealing with constraints and preferences in the graphical display of a physical simulation.

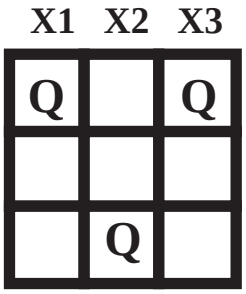

Figure 1: Three-queens Problem

Of course, there can be alternative ways for defining a solution criterion, such as maximizing the summation of importance values of satisfied constraints. However, there usually exists a trade-off between the strictness of the solution criterion definition and the required computational efforts to find a solution. We expect that this simple criterion can give intuitively natural solutions at a reasonable computational cost. We discuss the generalization of this solution criterion in Section 6.

We show an example problem in Figure 1. This problem is to place three queens in a 3×3 chess board so that these queens do not threaten each other. This problem can be formalized as a CSP, where there exist variables x_1, x_2, x_3, and the domain $\{1, 2, 3\}$. It is obvious that this problem is over-constrained. We define the importance values of constraints between two queens as being inverse to the square distance between them.

- There exists a constraint $p_{ij,k}$ between variables x_i and x_j ($k \in \{1, 2, 4, 8\}$), whose importance value is $1/k$. $p_{ij,k}$ is true iff $((x_i \neq x_j) \wedge (|x_i - x_j| \neq |i - j|)) \vee (x_i - x_j)^2 + (i - j)^2 > k$.

The placement of queens in Figure 1 satisfies all constraints whose importance values are greater than 1/4. Since no solution satisfies all constraints whose importance values are greater than or equal to 1/4, this placement is an optimal solution.

[2] The importance value represents the subjective preference of the problem designer, and is not related to whether the constraint is difficult to achieve or not.

By this formalization, we can deal with the situation where the domain of a variable is very large, and all possible values are not enumerated beforehand. For example, if the domain of a variable x_i is all prime numbers smaller than 10,000, it would be costly to enumerate the entire domain of this variable. Let's assume that we put additional constraints on the values of this variable, i.e., $p_c(x_i) \equiv x_i \leq c$, where $c \in \{1000, 2000, 3000, \ldots, 10000\}$ and the importance value of p_c is c. These additional constraints represent our preferences for smaller prime numbers, which can be relaxed in the order of importance values if needed. Therefore, we can systematically generate as many values of x_i (from smaller values to greater values) as necessary. This example shows that, even if the original problem is not over-constrained, we can still sometimes solve the problem more efficiently by adding additional constraints that can be relaxed if needed.

2.2 Algorithms

We can use general state-space search techniques [8] (A*, Depth-first Branch & Bound, etc.) to find a solution that minimizes the value of the evaluation function F. In [5], an algorithm based on Depth-first Branch & Bound which optimizes a given solution criterion is presented.

3 Formalization of distributed CSP

3.1 Communication model

We assume the following communication model.

- Communication between agents is done by sending messages. An agent can send messages to other agents iff the agent knows the addresses (identifiers) of the other agents .
- The delay in delivering a message is finite, but random. For the transmission between any pair of agents, messages are received in the order in which they were sent.

3.2 Distributed CSP

A distributed CSP is a CSP whose variables and constraints are distributed among multiple agents.

- There exist n agents $1, 2, \ldots, n$.
- Each agent has several variables.
- Agent i knows all constraint predicates relevant to its variables (constraint predicates which take i's variables as arguments). We represent this set of constraints as P_i. Agent i also knows the addresses of other agents having relevant variables (variables included in the arguments of P_i).

In a DCSP, we define the importance values of constraints and the evaluation function in the same manner as in a CSP. The goal of the agents is to find a solution S that minimizes the evaluation value $F(S)$. In the example problem of Figure 1, if we assume that there exist three agents, and each agent tries to place its queen in one column, this problem can be formalized as a DCSP.

4 Asynchronous incremental relaxation algorithm

4.1 Basic asynchronous incremental relaxation algorithm

The asynchronous backtracking algorithm can find a solution that satisfies all constraints if there exists one; if not, this algorithm discovers this fact and terminates. Therefore, agents can obtain the optimal solution by iteratively applying the asynchronous backtracking algorithm.

- Agents establish a *threshold value* (its initial value is some appropriate lower-bound or 0), and try to find a solution that satisfies all constraints whose importance values are greater than the *threshold* by using the asynchronous backtracking algorithm. If no solution satisfies all of these active constraints, the agents revise the threshold to the minimum of the importance values of these constraints [3], then try to find a solution that satisfies all constraints whose importance values are greater than the new *threshold*, and so on.

An alternative algorithm can be the Depth-first Branch & Bound search algorithm using asynchronous backtracking, in which constraints are tightened incrementally. Such an algorithm, however, becomes inappropriate when additional constraints are introduced to variable domains. Since these additional constraints are relaxed at the initial stage of the Depth-first Branch & Bound search, there is a chance that agents may enumerate many variable values that cannot be a part of an optimal solution. In contrast, the asynchronous incremental relaxation algorithm relaxes constraints only in inevitable situations. Therefore, agents never enumerate unnecessary values.

In the following, we describe an extension of this basic asynchronous incremental relaxation algorithm, in which agents avoid redundant computation by maintaining dependencies between constraint violations (nogoods) and constraints.

[3] We assume that agents can agree on this minimum value via communication.

4.2 Asynchronous incremental relaxation algorithm with nogood dependency

4.2.1 Nogood dependency

In asynchronous backtracking, agents communicate information about constraint violations (nogoods) with each other. A nogood is a set of variable values that causes some constraint violation. For example, in Figure 1, if $x_1 = 1$ and $x_2 = 3$, there exists no consistent value for x_3 with these values. Therefore, $\{(x_1,1),(x_2,3)\}$ is characterized as a nogood. A set of variable values that is a superset of a nogood can not be a final solution. If an empty set is found to be a nogood, no combination of variable values can be a final solution and the problem is over-constrained.

We extend the notion of nogoods so that a nogood N_k is coupled with its importance value c_k. The importance value of a nogood represents the dependencies between this nogood and constraints that contribute to this nogood, i.e., when all constraints whose importance values are equal to c_k are relaxed, then this nogood becomes obsolete.

The importance value of a nogood is defined as follows.

- The importance value of a nogood is the minimum of the importance values of constraints that contribute to this nogood.

For example, the nogood $\{(x_1,1),(x_2,3)\}$ described above is a constraint violation if all constraints must be satisfied, but is no longer a constraint violation if some constraints are relaxed. A set of variable values $\{(x_1,1),(x_2,3),(x_3,1)\}$ satisfies all constraints whose importance values are greater than 1/4. Therefore, this set of variable values is not a constraint violation if all constraints whose importance values are smaller than or equal to 1/4 are relaxed. Similarly, $\{(x_1,1),(x_2,3),(x_3,2)\}$ and $\{(x_1,1),(x_2,3),(x_3,3)\}$ are not constraint violations if constraints whose importance values are less than or equal to 1/2 and 1 are relaxed respectively. Therefore, the importance value of a nogood $\{(x_1,1),(x_2,3)\}$ is 1/4. By using the importance value of nogoods, agents can avoid redundant computation as follows:

avoiding search for wasteful threshold:
When an empty nogood is found and the importance value of this nogood is c_k, agents can tell that the new *threshold* should be c_k. Since the importance value of a nogood is the minimum of the importance values of constraints that contribute to this nogood, if agents set the new *threshold* to a value less than c_k (i.e., relaxing constraints whose importance values are less than c_k), then this nogood is still effective and no solution can be obtained. Thus the search for this new *threshold* is wasteful.

In the example of Figure 1, when the *threshold* is set to 0 (all constraints are considered), the problem is over-constrained and an empty nogood is obtained; the importance value of this nogood is 1/4. Therefore, although there exists a constraint whose importance value is 1/8, this constraint does not contribute to this nogood, and agents can tell that the new *threshold* should be 1/4.

avoiding redundant re-computation:
If agents do not maintain the importance values of discovered nogoods, then when the *threshold* is revised, they have to throw away all the nogoods obtained in previous backtracking search. They thus start the search process from scratch. On the other hand, if the agents do maintain the importance values of nogoods, when the *threshold* is revised, those nogoods whose importance values are greater than the new *threshold* are still effective and do not need to be thrown away. Agents can avoid redundant re-computation by utilizing these nogoods.

In the example of Figure 1, the importance value of nogood $\{(x_1,1)\}$ is 1/4 and the importance value of nogood $\{(x_1,2)\}$ is 1/2. When the *threshold* is increased to 1/4, the former is obsolete while the latter is still effective and the agents avoid setting the value of x_1 to 2.

The communication/computation overhead of maintaining this dependency is very small. When communicating a nogood, only one integer value is added to each message. When selecting a consistent value, by evaluating constraints in decreasing order of their importance values, an agent can tell the importance value of a newly discovered nogood, i.e., the value is equal to the minimum of the importance values of evaluated constraints.

It must be mentioned that these dependencies are totally deferent from the dependencies used in dependency-directed backtracking [2]. In dependency-directed backtracking, the dependencies between variable values and constraint violations are maintained in order to reduce unnecessary backtracking. In our algorithm, the dependencies between constraint violations (nogoods) and constraint predicates are maintained, since constraints are changed dynamically by constraint relaxation.

4.2.2 Details of the algorithm

In the following description of the algorithm, for simplicity, we assume that one agent has exactly one variable. This assumption, however, can be relaxed straightforwardly to a general case where one agent has multiple variables. We use the same identifier x_i for representing agent i and its variable x_i.

In this algorithm, agents act concurrently and asynchronously, rather than in a predefined sequential order. We show the initialization procedure for agent x_i in Figure 2 (i), and show the procedures that are invoked at agent x_i by the reception of three kinds of messages (*ok?, nogood, revise_threshold*) in Figure 2(ii), (iii), (iv), respectively.

A summary of these procedures is as follows:

- When initialized, each agent concurrently and asynchronously selects its value and sends the value to relevant agents (Figure 2(i)). After that, the agent waits for and responds to messages.
- Agent x_i divides relevant agents into two groups; one contains agents having smaller identifiers (*sender_list*), and the other contains agents having larger identifiers (*receiver_list*). Agent x_i sends its value assignment (*current_value*) to agents in the *receiver_list* (Figure 2(v-b,i-b)).
- On the other hand, agent x_i receives *ok?* messages from agents in the *sender_list*. Agent x_i puts these values in the *agent_view* (Figure 2(ii)). It then tries to find a value consistent with its *agent_view*. Directing *ok?* messages by agent identifiers is necessary to avoid infinite processing loops [10].
- If agent x_i cannot find a value consistent with its *agent_view*, it sends a nogood message to one of the agents in the *agent_view*. (Figure 2 (vi-c)), and asks the agent to change its value.

The differences between this algorithm and the basic asynchronous backtracking algorithm are as follows:

- An agent does not try to satisfy all constraints, but it tries to satisfy constraints whose importance values are greater than the current *threshold* (Figure 2(i-a, v-a)). In this algorithm, agent x_i uses an evaluation function F_i, to which the checks with nogoods are introduced. $F_i(S)$ returns 0 if S satisfies all constraints in P_i, and S is not a superset of any nogoods in the *nogood_list*. If not, $F_i(S)$ returns the maximum of the importance values of constraints and the nogoods, which are not satisfied by S or any subsets of S.

when initialized do — (i)
  select $d \in D_i$ where $F_i(\{(x_i, d)\}) \leq$ *threshold*; — (i - a)
  current_value$\leftarrow d$;
  send (**ok?**, x_i, d) to *receiver_list*; **end do**; — (i - b)

when received (**ok?**, x_j, d_j) **do** — (ii)
  add (x_j, d_j) to *agent_view*;
  check_agent_view; **end do**;

when received (**nogood**,
  x_k,*nogood, importance_value*) **do** — (iii)
  add (*nogood, importance_value*) to *nogood_list*;
  when (x_j,d_j) where $x_j \notin$ *sender_list* is in *nogood* **do**
    request x_j to add x_i to its *receiver_list*;
    add x_j to *sender_list*; add (x_j,d_j) to *agent_view*;
  end do;
  old_value $\leftarrow$ *current_value*;
  check_agent_view;
  when *old_value* = *current_value* **do**
    send (**ok?**, x_i, *current_value*) to x_k; **end do**; **end do**;

when received (**revise_threshold**,
  x_j,*new_threshold*) **do** — (iv)
  when *new_threshold* > *threshold* **do**
    threshold $\leftarrow$ *new_threshold*;
    send (**revise_threshold**, x_i,*new_threshold*)
      to other agents except x_j; **end do**; **end do**;

procedure **check_agent_view** — (v)
  when F_i(*agent_view* $\cup$ {(x_i, *current_value*)})
    > *threshold* **do**
    if there exists $d \in D_i$ where
      F_i(*agent_view* $\cup$ { (x_i, d)}) $\leq$ *threshold* **then**
      current_value $\leftarrow d$; — (v - a)
      send (**ok?**, x_i, d) to *receiver_list*;— (v - b)
    else backtrack; **check_agent_view**; **end if**; **end do**;

procedure **backtrack** — (vi)
  $V \leftarrow$ *agent_view*;
  new_importance_value
    $\leftarrow \min_{d \in D_i} F_i(V \cup \{(x_i, d)\})$ — (vi - a)
  if V={} **then** *threshold* $\leftarrow$ *new_importance_value*;
    send (**revise_threshold**,
      x_i, *threshold*) to other agents; — (vi - b)
    else select (x_j, v_j) from V where j is the largest;
    send (**nogood**,
      x_i, V, *new_importance_value*) to x_j;— (vi - c)
    remove (x_j, v_j) from *agent_view*; **end if**;

Figure 2: Procedures for receiving messages

- A *nogood* message is coupled with its importance value (Figure 2(iii,vi-c)). The importance value of a new nogood is obtained by the minimal value of F_i among possible values (Figure 2(vi-a))
- When an empty nogood is found, agents exchange *revise_threshold* messages and revise the *threshold* (Figure 2(iv,vi-b)).

In this algorithm, agents act asynchronously and concurrently, and eventually reach a stable state where all variable values satisfy all constraints whose importance values are greater than the current *threshold*, and all agents are waiting for messages [4].

4.2.3 Example of algorithm execution

We show an algorithm execution example of the distributed three-queens problem. In Figure 3, an agent is represented by a circle and a constraint between agents is represented as a link between circles. For simplicity, we restrict the domain of x_1 to {1,2} (Since the problem is symmetrical, we do not lose the optimal solution by this restriction).

Step1: Each agent initially sets its value like x_1=1, x_2=3, x_3=2. Agent x_1 sends *ok?* messages to x_2 and x_3, and agent x_2 sends an *ok?* message to x_3. By receiving the *ok?* message from x_1, agent x_2 does not change its value since it is consistent with $\{(x_1, 1)\}$. On the other hand, agent x_3 cannot find a consistent value with the *agent_view* {(x_1,1), (x_2,3)}. Therefore, agent x_3 sends a *nogood* message (nogood, {(x_1,1), (x_2,3)}, 1/4) to x_2 (Figure 3 (a)).

Step2: Agent x_2 tries to change its value after receiving this *nogood* message. However, there is no other value consistent with $\{(x_1, 1)\}$. Therefore, x_2 sends a *nogood* message (nogood, {(x_1,1)}, 1/4) to x_1 (Figure 3 (b)).

Step3: After receiving this *nogood* message, agent x_1 changes its value to 2, and sends *ok?* messages to the other agents. However, there is no consistent value with {(x_1,2)} for x_2. Therefore, x_2 sends a *nogood* message (nogood, {(x_1,2)}, 1/2) to x_1 (Figure 3 (c)).

Step4: Having received this *nogood* message, x_1 finds that there is no possible value (since we restrict the domain of x_1 to {1, 2}). Therefore, it generates an empty *nogood*; the importance value of this *nogood* is 1/4 (the minimum of 1/4 and 1/2). Agent x_1 changes the *threshold* to 1/4 from 0, and sends *revise_threshold* messages to x_2 and x_3. It also changes its value to 1 (since x_1=2 is still a constraint violation by (nogood, {(x_1,2)}, 1/2)), and sends *ok?* messages (Figure 3 (d)). Then, the agents reach a stable state x_1=1, x_2=3, and x_3=1, which is the optimal solution (since the *threshold* is 1/4, $x_3 = 1$ satisfies all constraints whose importance values are larger than this *threshold*.).

4.2.4 Completeness and Complexity of the algorithm

By the completeness of the asynchronous backtracking algorithm [10], the completeness of this algorithm is also guaranteed. Since the number of constraints is finite, the number of possible importance values is also finite. In the asynchronous incremental relaxation algorithm, the asynchronous backtracking algorithm is applied iteratively, and after one iteration, the *threshold* is always increased to one of the possible importance values of constraints. Therefore, after a finite number of iterations, the agents find a solution, or the *threshold* reaches the greatest importance value of constraints. In the latter case, all constraints are relaxed in the next iteration and agents eventually find a solution.

In order to measure the time complexity of this algorithm, we use the following computation model. We assume that, in one stage, an agent reads all incoming messages, does internal computation, and sends messages to other agents. Messages issued at one stage are available to other agents in the next stage. Unfortunately, the number of required stages even for one iteration is exponential in the number of variables m in the worst case. The upper bound of the number of iterations is given by the number of constraints. The fact that the number of required stages becomes exponential in the worst case is somewhat inevitable since a CSP is an NP-complete problem.

5 Experimental results

We experimentally examine the effect of introducing importance values of nogoods. We use the computation model described in Section 4.2.4 to measure the efficiency of the algorithms.

For the evaluation, we employ an over-constrained distributed n-queens problem, in which the domains of variables are restricted to $\{1, \ldots, n/2\}$ for vari-

[4] It must be mentioned that the way to determine that agents as a whole have reached a stable state is not contained in this algorithm. To detect the stable state, agents must use distributed termination detection algorithms such as [7].

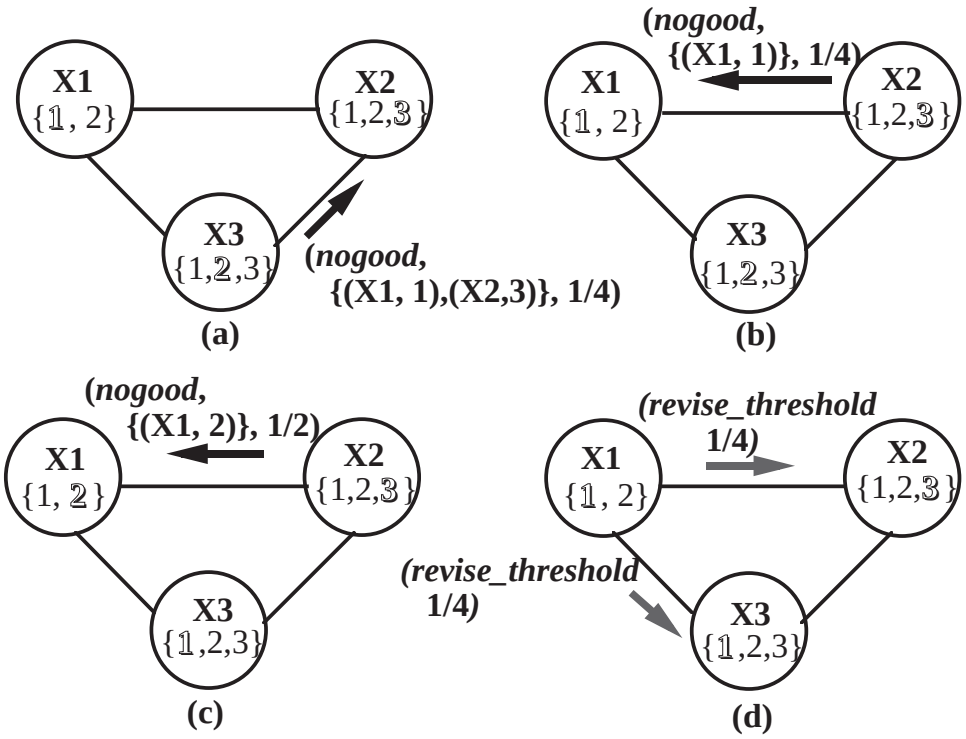

Figure 3: Example of algorithm execution

ables $i = 1, \ldots, n/2$, and $\{n/2+1, \ldots, n\}$ for variables $i = n/2+1, \ldots, n$ (we assume that n is an even number.). We define the importance values of constraints in the same manner as for the 3-queens problem in Figure 1. Since this problem is over-constrained, some constraints must be relaxed.

Figure 4 shows the required stages for the basic asynchronous incremental relaxation algorithm (basic AIR) and for the asynchronous incremental relaxation algorithm that introduces nogood dependency (AIR with nogood dependency) while varying the number of queens. In the basic AIR, since importance values of nogoods are not introduced, the agents perform search at wasteful *threshold* values, and the agents are required to throw away all nogoods after each iteration. This figure shows that the required stages for the AIR with nogood dependency is about 1/5 of the stages required for the basic AIR. As described in Section 4.2.1, the overhead of maintaining nogood dependency is very small. Therefore, we can expect that the reduction of computation time will be proportional to this reduction of required stages.

Figure 5 shows the history of *threshold* changes in solving the over-constrained distributed 12-queens problem. The initial value of the *threshold* is 0, and it is gradually increased; the optimal solution is obtained when the *threshold* becomes $1/72 \approx 0.014$. This figure shows that the AIR with nogood dependency avoids wasteful *threshold* values, compared with the basic AIR. Furthermore, the required number of stages for the same *threshold* is reduced by utilizing the nogoods obtained in the previous computation. For example, the AIR with nogood dependency requires only 23 stages for the search at *threshold*=$1/162 \approx 0.0062$, while the basic AIR requires 249 stages.

6 Discussion

In this paper, we defined a solution criterion by using the importance values of constraints. However, the asynchronous incremental relaxation algorithm can be applied to a more general class of solution criteria. In the asynchronous incremental relaxation algorithm, agents try to minimize the maximal value (the worst value) of local evaluation functions F_i, and the definition of F_i is immaterial to the essential part of this algorithm. By changing the definition of F_i, this algorithm can be applied to other solution criteria. For example, if F_i is defined as the number of constraint violations related to agent i, the asynchronous incremental relaxation algorithm finds a solution which minimizes the maximal number of constraint violations

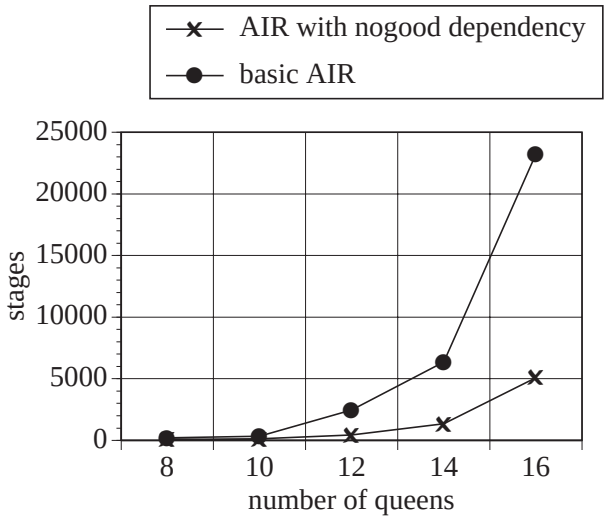

Figure 4: Required stages for over-constrained distributed n-queens problems

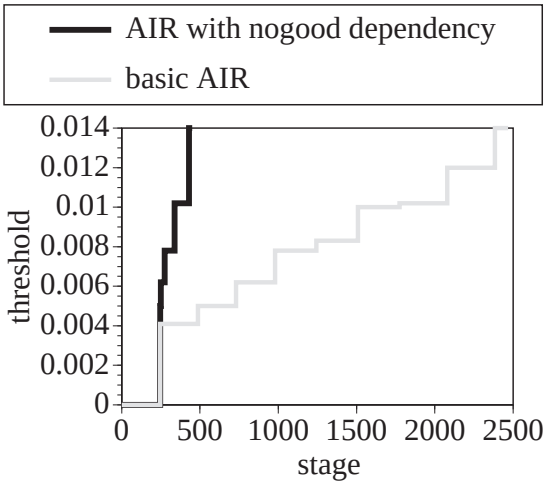

Figure 5: Threshold changes in over-constrained distributed 12-queens problem

related to each agent.

7 Conclusions

In this paper, we extended the formalization of the DCSP by introducing the notion of the importance values of constraints, and defined a solution criterion for over-constrained DCSPs. We developed an asynchronous incremental relaxation algorithm, with which agents incrementally relax less important constraints and find the solution that satisfies important constraints as much as possible. By introducing dependencies between nogoods and constraints, the agents can avoid redundant computation and achieve a five-fold speed-up in example problems.

Our ongoing research efforts involve applying this algorithm to large-scale, real-life problems, such as distributed resource allocation problems in communication networks.

Acknowledgments

The author would like to thank Prof. Edmund H. Durfee of the University of Michigan, whose discussion clarified the ideas presented in this paper. The author also would like to thank Prof. Toru Ishida of Kyoto University for his helpful comments on the earlier draft of this paper.

References

[1] Conry, S. E., Kuwabara, K., Lesser, V.R., and Meyer, R.A., Multistage Negotiation for Distributed Satisfaction, *IEEE Trans. on SMC*, Vol.21, No.6, pp. 1462–1477, 1991.

[2] de Kleer, J. : Dependency-directed Backtracking, In S.C.Shapiro editor, *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, pp. 47–48, 1987.

[3] Descotte,Y. and Latombe, J.C., Making Compromises among Antagonistic Constraints in Planner, *Artificial Intelligence*, Vol.27, No.1, pp. 183–217, 1985.

[4] Freeman-Benson, B.N., Maloney, J. and Borning, A., An Incremental Constraint Solver, Communications of the ACM, Vol.33, No.1, pp. 54–62, 1990.

[5] Fruder, E.C., Partial Constraint Satisfaction, *Proc. of IJCAI-89* , pp. 278–283, 1989.

[6] Huhns, M.N. and Bridgeland, D.M., Multiagent Truth Maintenance, *IEEE Trans. on SMC*, Vol.21, No.6, pp. 1437–1445, 1991.

[7] Chandy, K.M. and Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, Vol.3, No.1, pp. 63–75, 1985.

[8] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[9] Sycara, K.P., Roth, S., Sadeh, N., and Fox, M., Distributed Constrained Heuristic Search, *IEEE Trans. on SMC*, Vol.21, No.6, pp. 1446–1461, 1991.

[10] Yokoo, M., Durfee, E., Ishida, T. and Kuwabara, K., Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving, *Proc. of 12th IEEE International Conference on Distributed Computing Systems*, pp. 614–621, 1992.