Springer Series on Agent Technology

Series Editors: T. Ishida N. Jennings K. Sycara

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Makoto Yokoo

Distributed Constraint Satisfaction

Foundations of Cooperation in Multi-agent Systems

With 82 Figures and 22 Tables

Springer

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

Library of Congress Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Yokoo, Makoto:
Distributed constraint satisfaction: foundations of cooperation in
multi-agent systems; with tables/Makoto Yakoo. - Berlin;
Heidelberg; New York; Barcelona; Hong Kong; London; Milan; Paris;
Singapore; Tokyo: Springer, 2001
(Springer series on agent technology)
ISBN-13: 978-3-642-64020-9 e-ISBN-13: 978-3-642-59546-2
DOI: 10.1007/978-3-642-59546-2

ACM Subject Classification (1998): I.2.11, C.2.4, F.2

ISBN 978-3-642-64020-9 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

Softcover reprint of the hardcover 1st edition 2001

Typesetting: Camera-ready by the author
Cover Design: design + production, Heidelberg
SPIN 10722913 - 06/3142SR - 5 4 3 2 1 0

Preface

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 in which the goal is to find a consistent combination of actions that satisfies these inter-agent constraints. More specifically, a distributed CSP is a constraint satisfaction problem (CSP) in which multiple agents are involved.

A constraint satisfaction problem is a problem in which the goal is to find a consistent assignment of values to variables. Even though the definition of a CSP is very simple, a surprisingly wide variety of artificial intelligence (AI) problems can be formalized as CSPs. Therefore, the research on CSPs has a long and distinguished history in AI (Mackworth 1992; Dechter 1992; Tsang 1993; Kumar 1992). A distributed CSP is a CSP in which variables and constraints are distributed among multiple autonomous agents. Various application problems in Multi-agent Systems (MAS) that are concerned with finding a consistent combination of agent actions can be formalized as distributed CSPs. Therefore, we can consider distributed CSPs as a general framework for MAS, and algorithms for solving distributed CSPs as important infrastructures for cooperation in MAS.

This book gives an overview of the research on distributed CSPs, as well as introductory material on CSPs. In Chapter 1, we show the problem definition of normal, centralized CSPs and describe 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., systematic depth-first tree search algorithms called *backtracking* and hill-climbing algorithms called *iterative improvement*. In this chapter, we also describe a hybrid-type algorithm of backtracking and iterative improvement called the *weak-commitment search* algorithm. Furthermore, we examine which kinds of problem instances are most difficult for complete search algorithms and hill-climbing algorithms. Also, we describe an extension of the basic CSP formalization called *partial CSPs*, which can handle over-constrained CSPs.

Next, in Chapter 2, we show the problem definition of distributed CSPs and describe several MAS application problems that can be formalized as distributed CSPs. These problems include recognition problems, resource allo-

cation problems, multi-agent truth maintenance tasks, and scheduling/timetabling tasks.

From Chapter 3 to Chapter 5 we describe the search algorithms for solving distributed CSPs. In Chapter 3, we describe the basic backtracking algorithm for solving distributed CSPs called the *asynchronous backtracking* algorithm. In this algorithm, agents act concurrently and asynchronously without any global control, while the completeness of the algorithm is guaranteed.

In Chapter 4, we describe how the asynchronous backtracking algorithm can be modified to a more efficient algorithm called the *asynchronous weak-commitment search* algorithm, which is inspired by the weak-commitment search algorithm for solving CSPs. In this algorithm, when an agent cannot find a value consistent with higher priority agents, the priority order is changed so that the agent has the highest priority. As a result, when an agent makes a mistake in selecting a value, the priority of another agent becomes higher; accordingly the agent that made the mistake will not commit to the bad decision, and the selected value is changed. Experimental evaluations show that this algorithm can solve large-scale problems that cannot be solved by the asynchronous backtracking algorithm within a reasonable amount of time.

In Chapter 5, we describe an iterative improvement algorithm called the *distributed breakout* algorithm. This algorithm is based on the breakout algorithm for solving centralized CSPs. In this algorithm, neighboring agents exchange values that provide possible improvements so that only an agent that can maximally improve the evaluation value can change its variable value, and agents detect quasi-local-minima instead of real local-minima. Experimental evaluations show that the distributed breakout algorithm is more efficient than the asynchronous weak-commitment search algorithm when problem instances are particularly difficult.

In Chapter 6, we describe distributed consistency algorithms. We first describe a distributed problem-solving model called a *distributed ATMS*, in which each agent has its own ATMS, and these agents communicate hypothetical inference results and nogoods among themselves. Then, we show how distributed consistency algorithms are implemented using a distributed ATMS.

In Chapter 7, we consider the case where each agent has multiple local variables. Although the algorithms described in Chapters 3 to 5 can be applied to such a situation after slight modification, more sophisticated methods are needed to handle large and complex local problems. In this chapter, we explain how we can modify the asynchronous weak-commitment search algorithm so that each agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution that satisfies all local constraints. Experimental evaluations using example problems show that this algorithm is far more efficient than an

algorithm that employs the prioritization among agents, or than a simple extension of the asynchronous weak-commitment search algorithm.

When real-life problems are formalized as distributed CSPs, these problems are often over-constrained, where no solution satisfies all constraints completely; thus constraint relaxation is inevitable. In Chapter 8, we describe the formalization of *distributed partial CSPs*, where agents settle for a solution to a relaxed problem of an over-constrained distributed CSP. Also, we provide two classes of problems in this model, *distributed maximal CSPs* and *distributed hierarchical CSPs*, and describe algorithms for solving them.

This book is intended to benefit computer scientists and engineers who are interested in multi-agent systems and constraint satisfaction. A basic acquaintance with fundamental concepts in search would be helpful, but not necessary, since sufficient introductory material on CSPs is included in Chapter 1.

This book would not have been possible without the contribution of many people. The author wishes to thank Katsutoshi Hirayama, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara for their collaborative work. A large part of this book was written based on the author's conference and journal papers. Parts of Chapter 1 appeared in (Yokoo 1994; Yokoo 1997). Portions of Chapters 2, 3, and 4 appeared in (Yokoo, Durfee, Ishida, and Kuwabara 1992; Yokoo 1995; Yokoo, Durfee, Ishida, and Kuwabara 1998). Parts of Chapter 5 appeared in (Yokoo and Hirayama 1996), and an early version of Chapter 6 appeared in (Yokoo, Ishida, and Kuwabara 1990). Parts of Chapter 7 appeared in (Yokoo and Hirayama 1998), and Parts of Chapter 8 appeared in (Yokoo 1993) and (Hirayama and Yokoo 1997). The author wishes to thank the referees of these journals/conferences for their helpful remarks on these papers. Most importantly, I would like to thank my wife Yuko and my daughter Maho; without their support and encouragement, this book could never have been completed.

Makoto Yokoo

March 30, 2000
Kyoto, Japan

Table of Contents

List of Figures

List of Tables

1. Constraint Satisfaction Problem

1.1 Introduction

A constraint satisfaction problem (CSP) is a general framework that can formalize various application problems in artificial intelligence (AI). 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 (Fig. 1.1). A problem of this kind is called a constraint satisfaction problem because the objective is to find a configuration that satisfies the given conditions (constraints).

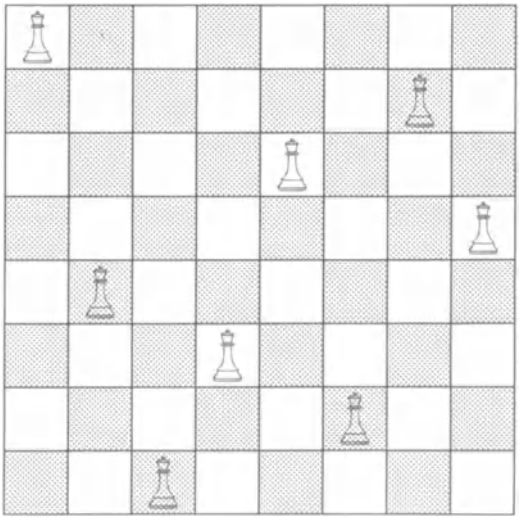

Fig. 1.1. Example of Constraint Satisfaction Problem (8-Queens)

In this chapter, we first give a formal definition of CSPs (Section 1.2), and then describe algorithms for solving CSPs (Section 1.3). These algorithms can be divided into two groups, i.e., search algorithms and consistency algorithms. Consistency algorithms are performed before applying search algorithms to

M. Yokoo, *Distributed Constraint Satisfaction*

reduce futile search efforts. Furthermore, search algorithms can be divided into two groups, i.e., systematic depth-first tree search algorithms called *backtracking* and hill-climbing algorithms called *iterative improvement*. Moreover, we show a hybrid-type algorithm called the *weak-commitment search*, which overcomes the drawbacks of backtracking algorithms and iterative improvement algorithms (Section 1.4). Next, we describe which kinds of problem instances of CSPs are most difficult for systematic search algorithms and hill-climbing algorithms (Section 1.5). Finally, we describe a *partial CSP*, which is an extension of a basic CSP formalization, that can handle the situation when constraints are too tight and there exists no complete solution for a CSP (Section 1.6).

1.2 Problem Definition

Formally, a CSP consists of the following two components.

- 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
- a set of constraints on their values

A constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \ldots, x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied. Constraint satisfaction is NP-complete in general. Actually, a satisfiability problem for propositional formulas in conjunctive normal form (SAT), which is one instance of a CSP, was the first computational task shown to be NP-complete (Cook 1971). Therefore, a trial-and-error exploration of alternatives is inevitable for solving CSPs.

This chapter presents several examples of CSPs. In the 8-queens problem, it is obvious that only one queen can be placed in each row. Therefore, we can formalize this problem as a CSP, in which there are eight variables $x_1, x_2, \ldots, x_8$, each of which corresponds to the position of a queen in each row. The domain of a variable is $\{1, 2, 3, 4, 5, 6, 7, 8\}$. 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 $(x_i \neq x_j) \wedge (\mid i - j \mid \neq \mid x_i - x_j \mid)$.

Another typical example problem is a graph-coloring problem (Fig. 1.2). The objective of a graph-coloring problem is to paint nodes in a graph so that any two nodes connected by a link do not have the same color. Each node has a finite number of possible colors. This problem can be formalized as a CSP by representing the color of each node as a variable, and the possible colors of the node as a domain of the variable.

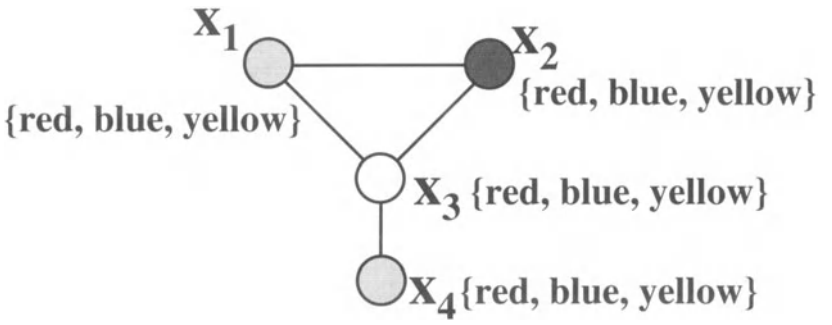

Fig. 1.2. Example of Constraint Satisfaction Problem (Graph-Coloring)

Another toy example problem is a crossword puzzle (Fig. 1.3). In this example, for simplicity, we assume that candidates of words are specified. Since a five-letter word must be placed in 1-across, possible candidates are {Hoses, Laser, Sails, Sheet, Steer}. If we choose 'Hoses' for 1-across, we can only choose a five-letter word that begins with the letter 's' for 2-down, i.e., 'Sails', 'Sheet', or 'Steer'. This problem can be formalized as a CSP by representing the word of each column/row as a variable, and a set of word candidates as a domain of the variable.

A satisfiability problem for propositional formulas in conjunctive normal form (SAT) is an important subclass of CSPs. This problem can be defined as follows. A boolean *variable* x_i is a variable that takes the value true or false. We call the value assignment of one variable a *literal*. A *clause* is a disjunction of literals, e.g., $x_1 \vee \overline{x_2} \vee x_4$. Given a set of clauses $C_1, C_2, \ldots C_m$ and variables $x_1, x_2, \ldots, x_n$, the satisfiability problem is to determine whether the formula

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

is satisfiable. That is, if there is an assignment of values to the variables such that the above formula is true. In particular, we call a SAT problem in which the number of literals in each clause is restricted to 3 a *3-SAT* problem.

Note that there is no restriction on 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*.

If all constraints are binary (i.e., between two variables), a CSP can be represented as a network, in which a node represents a variable, and a link between nodes represents a constraint between the corresponding variables. Figure 1.4 shows a constraint network representing a CSP with three variables x_1, x_2, x_3 and constraints $x_1 \neq x_3$, $x_2 \neq x_3$. For simplicity, we will mainly

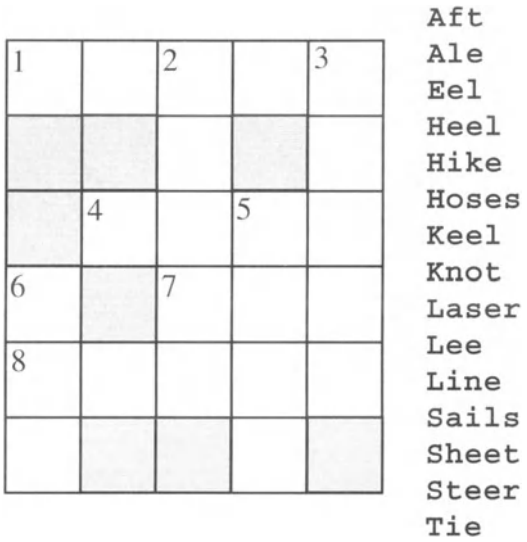

Fig. 1.3. Example of Constraint Satisfaction Problem (Crossword Puzzle)

focus our attention on binary CSPs. However, the algorithms described in this book are also applicable to non-binary CSPs.

Although the formalization of a CSP is very simple, this formalization can cover a surprisingly wide variety of application problems. For example, the scene labeling problem in computer vision was probably the first application problem of CSPs. After some preprocessing of a camera image, lines can be recognized and a scene like the one in Fig. 1.5 can be obtained.

To recognize this object as a cube, we must interpret the lines in the drawing. Lines in a scene can be categorized into three types, i.e., a convex edge, a concave edge, and an occluding edge (where one of the planes is hidden behind the other). For example, the edges in Fig. 1.5 are categorized in Fig. 1.6, in which '+' represents a convex edge, '−' represents a concave edge, and $\rightarrow$ and $\leftarrow$ represent occluding edges. The direction of these arrows indicates that as we move in the direction of the arrows, the surface lies to the right.

In (Huffman 1971), possible combinations of these labels at a junction are enumerated (Fig. 1.7). We can formalize a scene labeling problem as a CSP, in which an edge is a variable, the domain is $\{+, -, \rightarrow, \leftarrow\}$, and there exist junction constraints among these variables as specified in Fig. 1.7.

Another example of application problems of CSPs is a frequency assignment problem (also called a channel assignment problem) (Box 1978; Gamst 1986; Hale 1980; Pennotti and Boorstyn 1976; Sivarajan, McEliece, and Ketchum 1989). In a frequency assignment problem, there exists a set

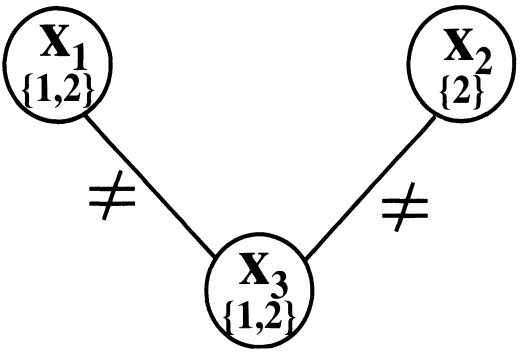

Fig. 1.4. Example of Constraint Network

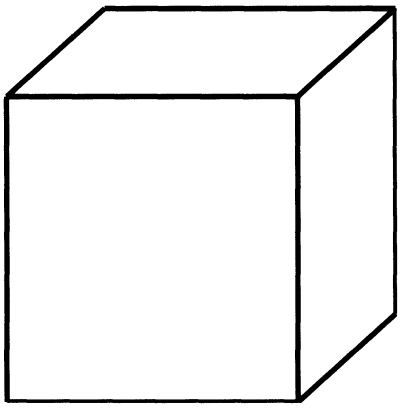

Fig. 1.5. Example of Scene

of geographically divided, typically hexagonal regions called cells (Fig. 1.8). Frequencies (channels) must be assigned to each cell according to the number of call requests. This problem has three types of electro-magnetic separation constraints.

- co-channel constraint: the same frequency cannot be assigned to pairs of cells that are geographically close to each other.
- adjacent channel constraint: similar frequencies cannot be simultaneously assigned to adjacent cells.
- co-site constraint: any pair of frequencies assigned to the same cell must have a certain separation.

The goal is to find a frequency assignment that satisfies the above constraints using a minimum number of frequencies (more precisely, using the minimum span of the frequencies).

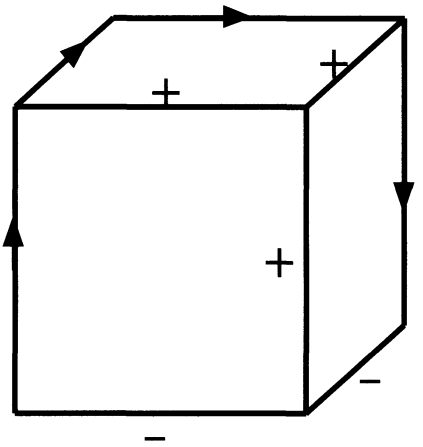

Fig. 1.6. Example of Scene with Labeled Edges

A frequency assignment problem is formalized as follows. We follow the formalization used in (Sivarajan, McEliece, and Ketchum 1989). Frequencies are represented by positive integers $1, 2, 3, \ldots$.

Given: N: the number of cells
d_i, $1 \leq i \leq N$: the number of requested calls (demands) in cell i
c_{ij}, $1 \leq i, j \leq N$: the frequency separation required between a call in cell i and a call in cell j

Find: f_{ik}, $1 \leq i \leq N$, $1 \leq k \leq d_i$: the frequency assigned to the kth call in cell i.
such that,
subject to the separation constraints,
$| f_{ik} - f_{jl} | \geq c_{ij}$, for all i, j, k, l except for $i = j$ and $k = l$,
minimize
$max f_{ik}$ for all i, k.

These constraints can be represented as an $N \times N$ symmetric compatibility matrix C. In addition, a set of requested calls can be represented by an N-element demand vector D. Let us show an example of a compatibility matrix and a demand vector:

$$C = \begin{pmatrix} 5 & 3 & 0 & 0 \\ 3 & 5 & 0 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 2 & 1 & 5 \end{pmatrix}, \quad D = \begin{pmatrix} 2 \\ 1 \\ 2 \\ 3 \end{pmatrix}.$$

In this example, there are four cells, and there exist constraints between cell 1 and cell 2, cell 2 and cell 4, cell 3 and cell 4, namely, the required minimum separations are 3, 2, and 1, respectively.

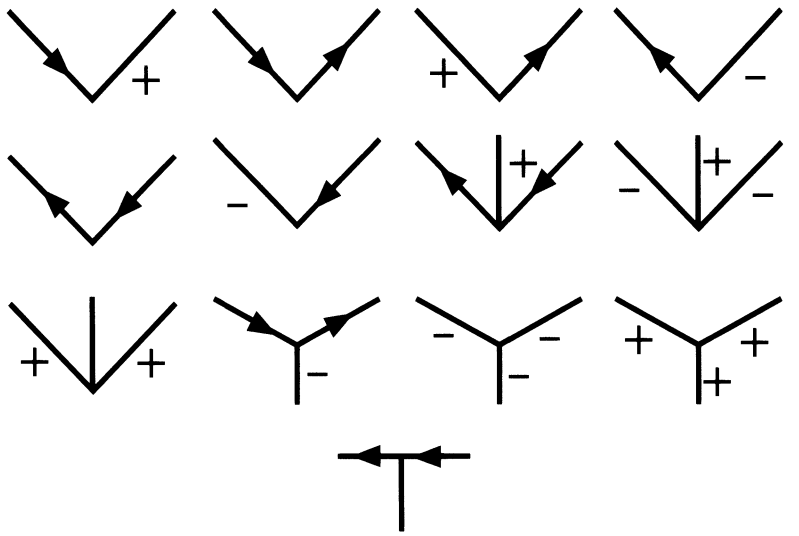

Fig. 1.7. Possible Labels for Junctions

Various AI techniques, including simulated annealing, neural networks, Tabu search, and genetic algorithms, have been applied to this problem (Carlsson and Grindal 1993; Funabiki and Takefuji 1992; Hao, Dorne, and Galinier 1998; Hurley, Smith, and Thiel 1997; Kim and Kim 1994; Kunz 1991; Smith, Hurley, and Thiel 1998; Walser 1996). The most straightforward way of solving such problems using constraint satisfaction techniques would be to represent each call as a variable (whose domain is available frequencies), and then to solve the problem as a generalized graph-coloring problem. Although this problem is an optimization problem, we can find an optimal solution by iteratively decreasing the possible upper-bound of available frequencies.

Other examples of application problems of CSPs include belief maintenance (Dechter and Dechter 1988), planning and scheduling (Kautz and Selman 1992; Fox 1987), diagnostic reasoning (Geffner and Pearl 1987).

1.3 Algorithms for Solving CSPs

Algorithms for solving CSPs can be divided into two groups, i.e., search algorithms for finding a solution of CSPs and preprocessing algorithms for reducing futile search, which are called *consistency algorithms*.

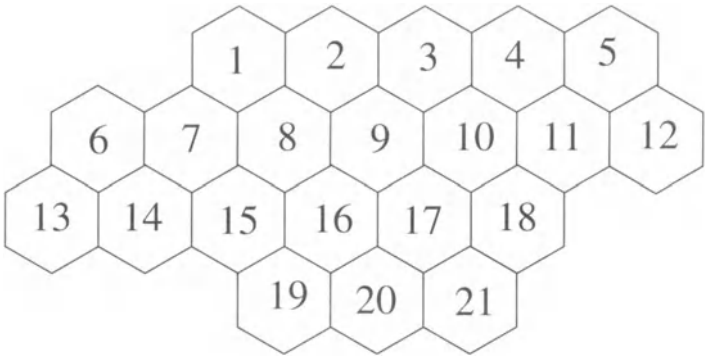

Fig. 1.8. Example of Cells

1.3.1 Backtracking

A backtracking algorithm is a basic, systematic search algorithm for solving CSPs. Let us represent the fact that the value of variable x_i is assigned to d_i as (x_i, d_i). In a backtracking algorithm, a value assignment to a subset of variables that satisfies all of the constraints within the subset is constructed. This subset is called a *partial solution*. Initially, a partial solution is an empty set. A partial solution is expanded by adding new variables one by one, until it becomes a complete solution. For example, if we are solving the 4-queens problem (Fig. 1.9), we can add $(x_1, 1)$ to the empty partial solution first (Fig. 1.10 (a)). then, we can add $(x_2, 3)$ to this partial solution. Note that we cannot select the values 1 or 2 for x_2, since these values violate the constraint with x_1 (Fig. 1.10 (b)).

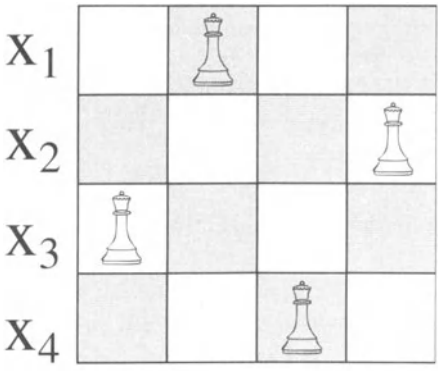

Fig. 1.9. 4-Queens Problem

When for one variable, no value satisfies all of the constraints with the variables in the partial solution, the value of the most recently added variable to the partial solution is changed. This operation is called backtracking. For example, if the current partial solution is $\{(x_1, 1), (x_2, 3)\}$, there exists no possible value for x_3 (Fig. 1.10 (c)). Therefore, the value assignment of x_2 is changed (Fig. 1.10 (d)).

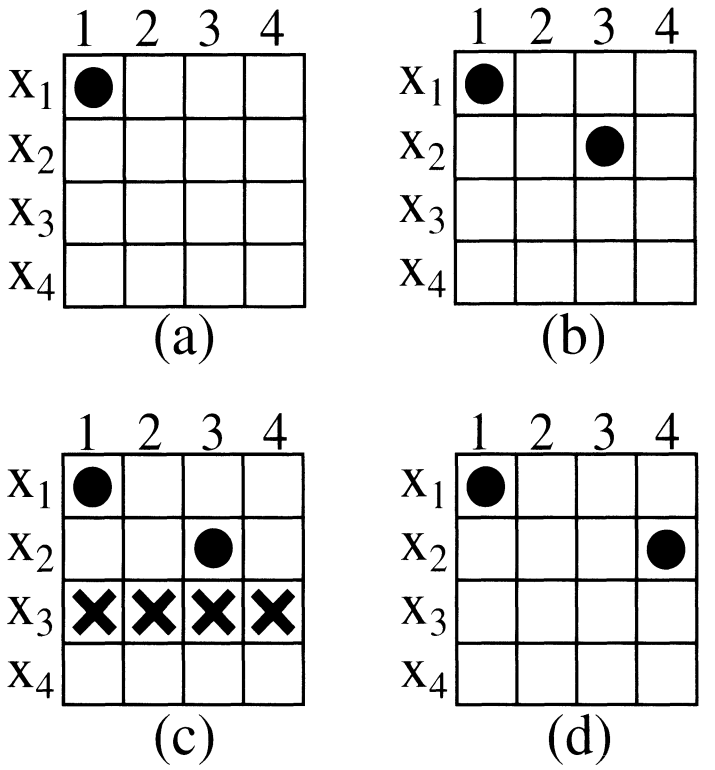

Fig. 1.10. Example of Algorithm Execution (Backtracking)

Min-Conflict Backtracking. Although backtracking is a simple depth-first tree search algorithm (Fig. 1.11), 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* (Minton, Johnston, Philips, and Laird 1992) has proven to be very effective 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

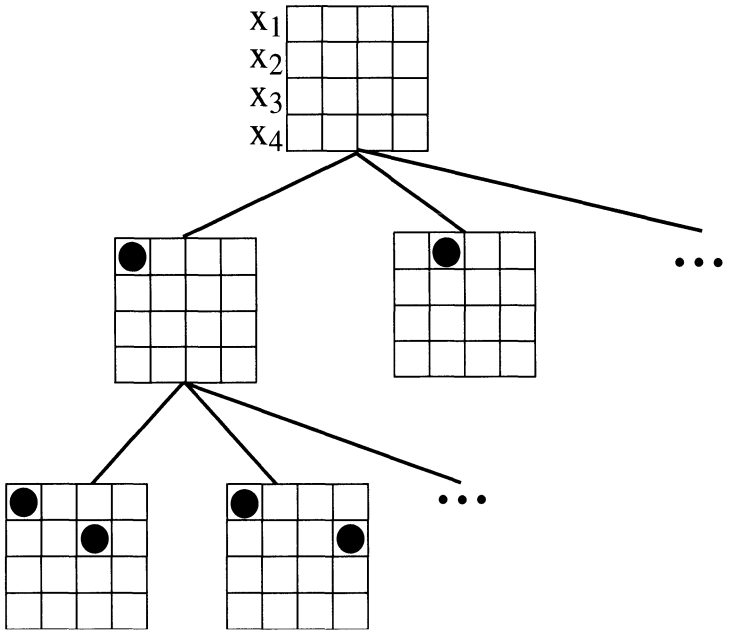

Fig. 1.11. Example of Search-Tree of 4-Queens Problem

a backtracking algorithm, the revised value must satisfy all of the constraints with 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.

For example, in Fig. 1.12, let us assume that the partial solution contains $\{(x_1, 1)\}$ only, and we are now selecting the value of x_4. We represent a queen in a partial-solution as a filled circle. Since the expanded partial solution should satisfy all constraints, we can choose 2 or 3 for x_4. The value 2 violates the constraint between x_2 and x_4, and the constraint between x_3 and x_4, while 3 satisfies both constraints. Therefore, we choose 3 instead of 2.

The min-conflict backtracking algorithm is illustrated in Fig. 1.13. Initially, *vars-left* is set to $\{(x_1, d_1), (x_2, d_2), \ldots, (x_n, d_n)\}$, where d_i is the tentative initial value of x_i. Also, *partial-solution* is assigned to an empty set. This algorithm moves variables from *vars-left* to *partial-solution* one by one. To simplify the description, we assume that when the algorithm finds that a partial solution cannot be further expanded, the algorithm records the partial solution as a nogood, i.e., the prohibited combination of variable values.

Forward-Checking. The order of selecting variables can also greatly affect the efficiency. Common wisdom for selecting a variable for adding to the partial solution is to select the variable that is most strongly constrained. This heuristic is called *first-fail* principle, since it suggests that the task that

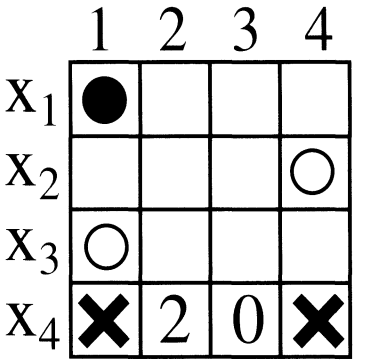

Fig. 1.12. Example of Algorithm Execution (Min-Conflict Backtracking)

is most likely to fail should be performed first. However, how can we measure the strength of constraints on a particular variable?

A very simple way to measure the strength of constraints on a particular variable is to use the number of constraint predicates related to the variable. For example, in a graph-coloring problem, we can use the number of links connected to a node (a variable), i.e., the larger the number of links, the more constrained the variable. However, this measure is rather weak since it is static, that is, it does not take into account the current state of a partial solution. A better way is to count the number of links connected to the nodes (variables) already in a partial solution. This measure is dynamic, i.e., the number dynamically changes during the search process. An even more precise measure is to count the number of values that are consistent with a partial solution. This method is called *forward-checking*. More specifically, during the backtracking search, we maintain a list for each variable. The list contains the values that are consistent with a partial solution. Initially, this list is identical to the domain of the variable, but the list becomes smaller as the partial solution is expanded. We show an example of the forward-checking procedure in Fig. 1.14 (a). In this case, a partial solution contains the variables x_1, x_2, and x_3. Since x_6 has only one possible value, we should determine the value of x_6 immediately. Another advantage of forward-checking is the early detection of failure. Assume we choose x_4 instead of x_6, and set its value to 2 (Fig. 1.14 (b)). In this case, there exists no possible value for x_6, so we can perform backtracking before determining the values of other variables. Forward checking can be considered as performing a weak consistency algorithm described in Section 1.3.3 during the search process.

Dependency-Directed Backtracking. Another method for improving the efficiency of backtracking algorithms is dependency-directed backtracking (Stallman and Sussman 1977). The idea is to identify the culprits causing

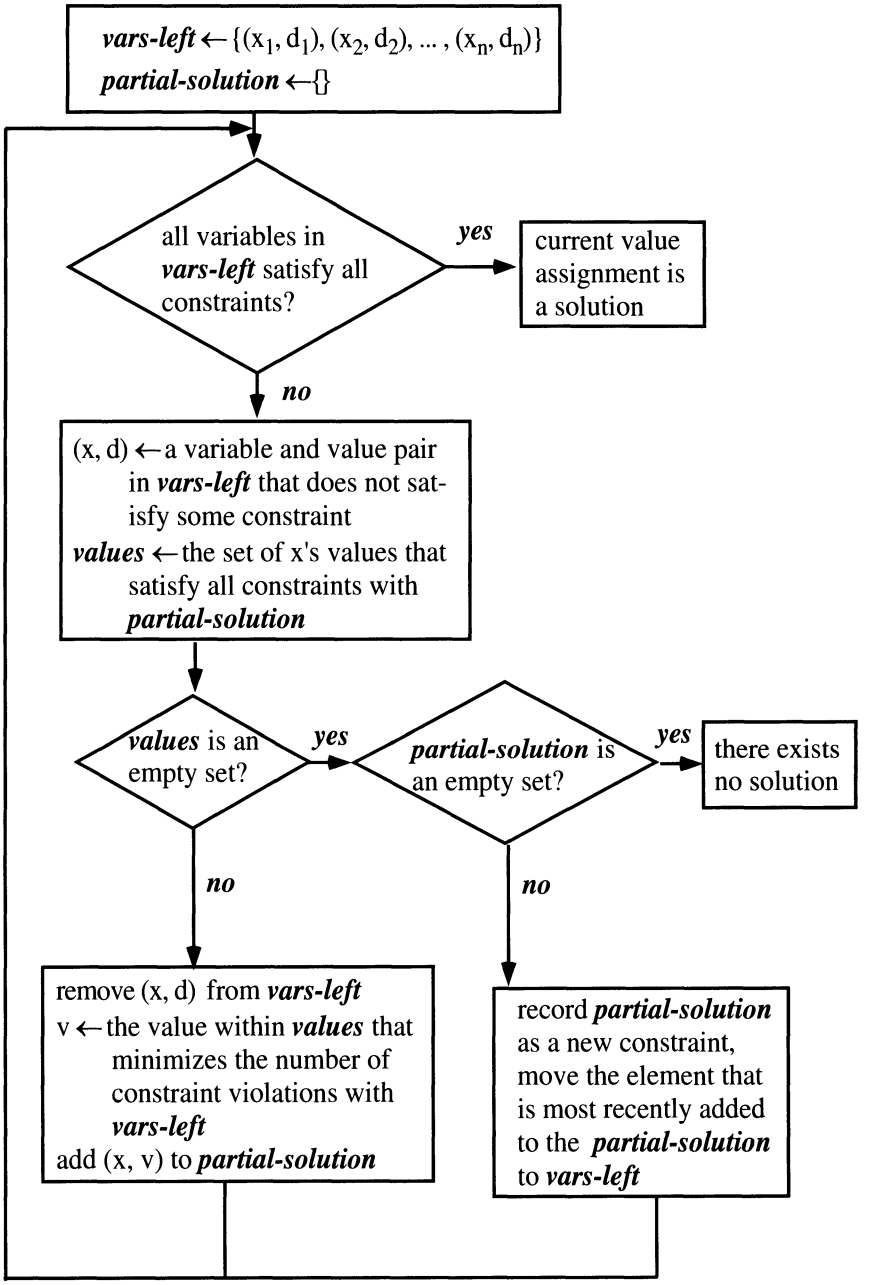

Fig. 1.13. Min-conflict Backtracking Algorithm

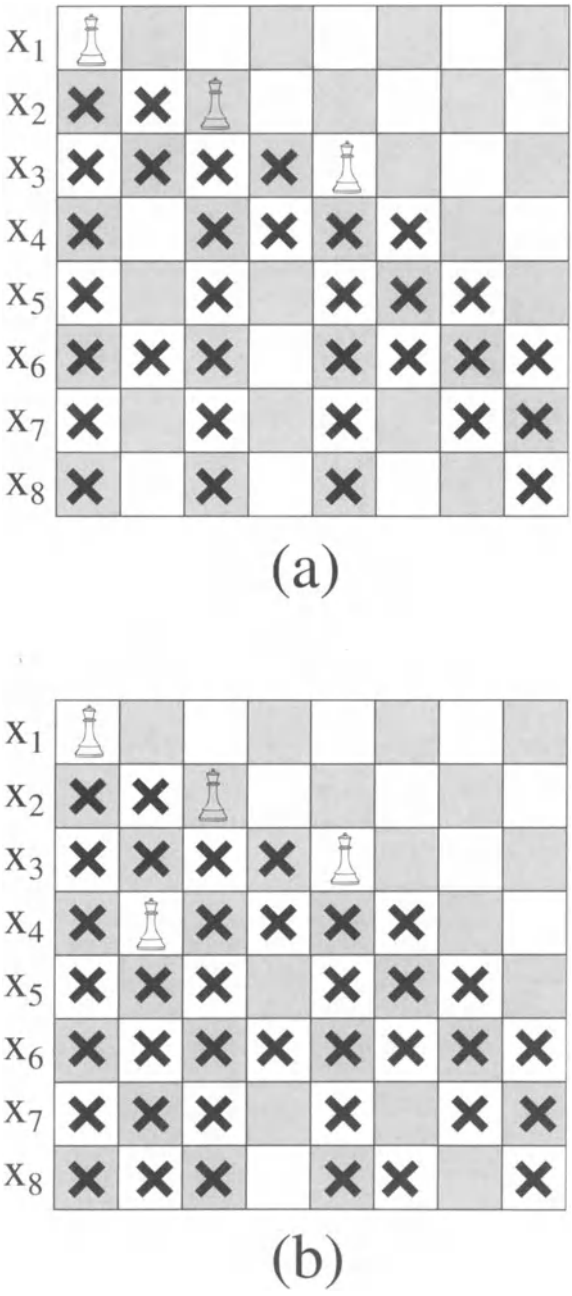

Fig. 1.14. Example of Forward-Checking

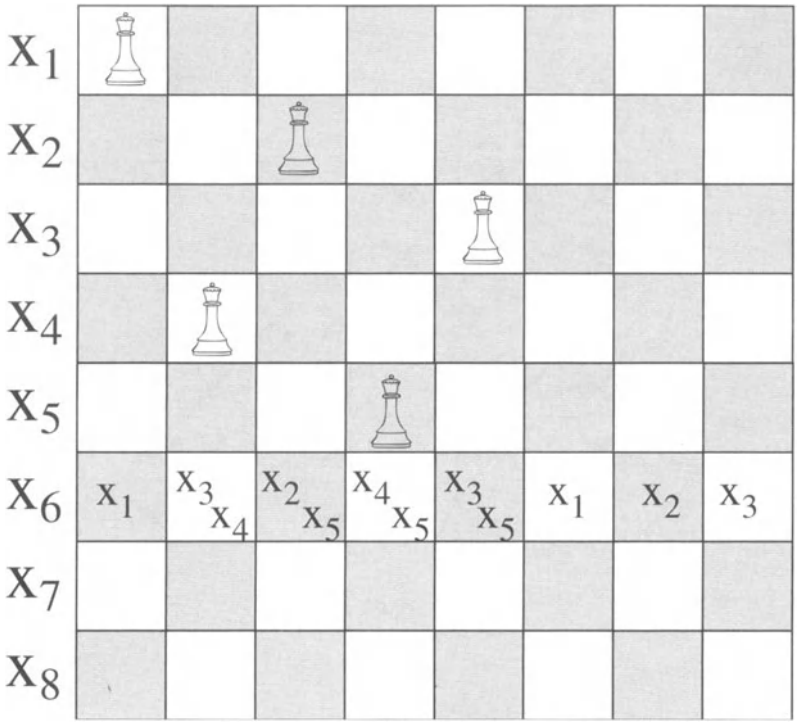

Fig. 1.15. Example of Dependency-Directed Backtracking

failure on backtracking, so that the algorithm can backtrack to the relevant decisions only.

Let us consider the situation in Fig. 1.15. This partial solution is constructed by adding variables in the order $x_1, x_2, \ldots, x_5$. Here, there exists no consistent value for x_6. In a normal backtracking algorithm (also called a *chronological backtracking* algorithm), the variable that is most recently added to the partial solution is changed. Therefore, we change the value of x_5. However, by closely examining the cause of the failure, namely, which variable forbids each value of x_6, we can see that changing the value of x_5 is useless, since even without x_5, all values of x_6 are forbidden. Therefore, the algorithm should backtrack to x_4 instead of x_5.

Identifying the precise cause of the failure could require many computational costs. Therefore, there exists a trade-off between the amount of reduced futile backtracking and the computational costs to identify the cause of the failure. A method called *graph-based backjumping* (Dechter and Pearl 1988) achieves a good compromise, i.e., it can find less precise causes of failure at relatively low computational costs.

1.3.2 Iterative Improvement

In iterative improvement algorithms (Selman, Levesque, and Mitchell 1992; Selman and Kautz 1993; Morris 1993), 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 a 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.

For example, assume that tentative initial values are assigned as shown in Fig. 1.16 (a). In this figure, a white circle indicates a queen that satisfies all constraints, and a gray circle indicates a queen that violates some constraint. By moving the position of x_1 from 1 to 2, x_1 satisfies all related constraints (Fig. 1.16 (b)). Then, by moving the position of x_4 from 1 to 3, all constraints are satisfied.

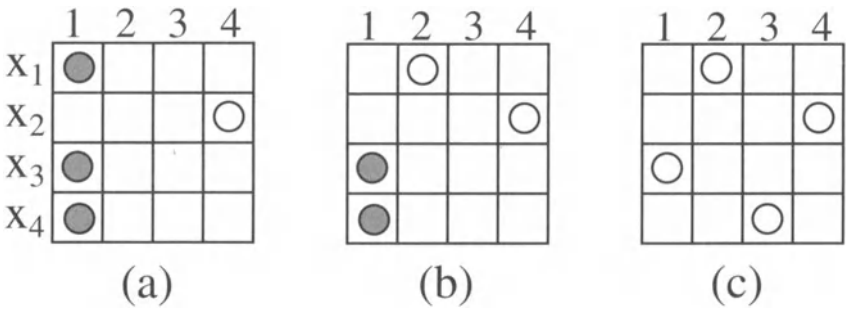

Fig. 1.16. Example of Algorithm Execution (Iterative Improvement)

```
procedure GSAT
    restart: set variable values to randomly selected initial values;
    for j:=1 to Max-flips do
        if current value assignment is a solution then return;
        else select a variable x_i such that a change in its truth assignment
                give the largest increase in the total number of clauses
                that are satisfied;
            reverse the value of x_i;
        end if;
    end do;
    goto restart;
```

Fig. 1.17. GSAT Algorithm

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 GSAT algorithm (Selman, Levesque, and Mitchell 1992), which is developed for solving SAT, when a solution cannot be found after certain iterations, the algorithm simply restarts from a new initial value assignment. An outline of the GSAT algorithm is shown in Fig. 1.17.

On the other hand, in the breakout algorithm (Morris 1993), we assume all constraints are represented as nogoods, i.e., a set of variable values that are forbidden, and a weight is defined for each constraint, where 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. We show the outline of the breakout algorithm in Fig. 1.18.

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 (i.e., always finding a solution if one exists, or terminating if no solution exists) cannot be guaranteed.

```
procedure breakout
    until current-state is solution do
    if current state is not a local-minimum
        then make any change of a variable value that reduces the total cost
        else increase weights of all current nogoods
    end if; end do;
```

Fig. 1.18. Breakout Algorithm

1.3.3 Consistency Algorithms

One drawback of standard backtracking algorithms is *thrashing*, i.e., the repeated exploration of assignments that are different only in inessential features, such as the assignment to a variable irrelevant to the failure. Various preprocessing procedures called *consistency algorithms* (Mackworth 1992; Tsang 1993) have been developed to reduce thrashing.

We show an example of consistency algorithms called the *Waltz filtering* algorithm (Waltz 1975). In this algorithm, the following procedure **revise**(x_i, x_j) is applied to each variable pair.

procedure **revise** (x_i, x_j)
  for all $v_i \in D_i$ **do**
    if there is no value $v_j \in D_j$ such that v_j is consistent with v_i
    then delete v_i from D_i; **end if; end do;**

We show an example of an algorithm execution in Fig. 1.19. The example problem is a 3-queens problem. There are three variables x_1, x_2, x_3, whose domains are {1,2,3}. Obviously, this problem is over-constrained and has no solution. After performing **revise**(x_1, x_2), 2 is removed from x_1's domain, since if $x_1 = 2$, none of x_2's values satisfies the constraint with x_1. Similarly, after performing **revise**(x_3, x_2), 2 is removed from x_3's domain. Then, by performing **revise**(x_1, x_3), all values are removed from x_1's domain.

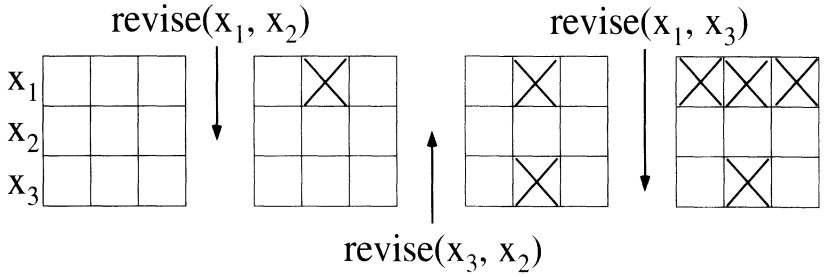

Fig. 1.19. Example of Algorithm Execution (Filtering)

By applying the filtering algorithm, if a domain of some variable becomes an empty set, the problem is over-constrained and has no solution. Also, if each domain has a unique value, then the combination of the remaining values becomes a solution. On the other hand, if there exist multiple values for some variable, we cannot tell whether the problem has a solution or not, and further trial-and-error search is required to find a solution.

Figure 1.20 shows a graph-coloring problem. Since there are three variables and the only possible colors of each variable are red or blue, this problem is over-constrained. However, in the filtering algorithm, no value can be removed from any variable domains. Similarly, in the 8-queens problem (which has many solutions), no value can be removed from any variable domains by using the filtering algorithm.

Since the filtering algorithm cannot solve a problem in general, it should be considered a preprocessing procedure that is invoked before the application of other search methods. Even though the filtering algorithm alone cannot

solve a problem, reducing the domains of variables for the following search procedure is worthwhile.

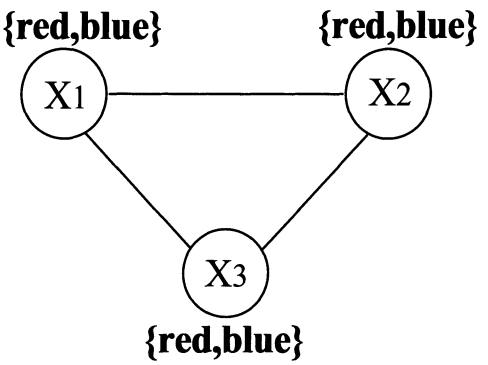

Fig. 1.20. Example in Which Filtering Algorithm Cannot Detect Failure

The filtering algorithm is one example of a general class of algorithms called *consistency algorithms.* Consistency algorithms can be classified by the notion of k-consistency (Freuder 1978). A CSP is k-consistent iff the following condition is satisfied.

- Given any instantiation of any $k-1$ variables satisfying all of the constraints among those variables, it is possible to find an instantiation of any kth variable such that these k variable values satisfy all the constraints among them.

The filtering algorithm achieves 2-consistency (also called arc-consistency), i.e., any variable value has at least one consistent value of another variable. A k-consistency algorithm transforms a given problem into an equivalent k-consistent problem (i.e., having the same solutions as the original problem).

If the problem is k-consistent, and also j-consistent for all $j < k$, the problem is called *strongly* k-consistent. If there are n variables in a CSP and the CSP is strongly n-consistent, then a solution can be obtained immediately without any trial-and-error exploration, since for any instantiation of $k-1$ variables, we can always find at least one consistent value for k-th variables.

Also, there exist certain kinds of CSPs for which k-consistency (where $k < n$) guarantees backtrack-free search (Freuder 1978). An *ordered* constraint network is a constraint network whose nodes are ordered linearly. For a node in an ordered constraint network, the *width* of a node is defined as the number of links from the node to preceding nodes in the order (Fig. 1.21). The width of an ordered constraint graph is the maximum of the width of all nodes. For example, in the ordered constraint graph described in Fig. 1.21, the width of the graph is 2. We can prove that the following property holds.

– If the width of an ordered constraint graph is w, and the constraint graph is strongly k-consistent (where $k > w$), then we can perform backtrack-free search of the constraint graph in that order.

The proof is rather straightforward. Since the width of the ordered constraint graph is w, and the constraint graph is strongly k-consistent, we can always find at least one variable that is consistent with a partial solution. If a constraint graph is tree, i.e., it has no cycle, we can find an ordering whose width is 1. Therefore, achieving arc-consistency is sufficient for guaranteeing backtrack-free search. Also, in (Dechter and Pearl 1988), consistency algorithms that take into account the search order called *directional consistency* algorithms are presented.

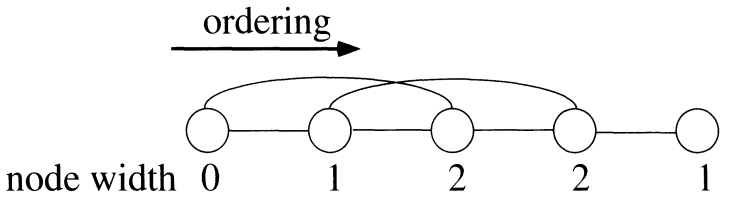

Fig. 1.21. Width of Ordered Constraint Graph

In the following, we describe a consistency algorithm using the hyper-resolution rule (de Kleer 1989). In this algorithm, all constraints are represented as a nogood, which is a prohibited combination of variable values. For example, in Fig. 1.20, a constraint between x_1 and x_2 can be represented as two nogoods $\{x_1 = red, x_2 = red\}$ and $\{x_1 = blue, x_2 = blue\}$.

A new nogood is generated from several existing nogoods by using the hyper-resolution rule. For example, in Fig. 1.20, there are nogoods such as $\{x_1 = red, x_2 = red\}$ and $\{x_1 = blue, x_3 = blue\}$. Furthermore, since the domain of x_1 is $\{red, blue\}$, $(x_1 = red) \vee (x_1 = blue)$ holds. The hyper-resolution rule combines nogoods and the condition that a variable takes one value from its domain, and generates a new nogood, e.g., $\{x_2 = red, x_3 = blue\}$. The meaning of this nogood is as follows. If x_2 is red, x_1 cannot be red. Also, if x_3 is blue, x_1 cannot be blue. Since x_1 is either red or blue, if x_2 is red and x_3 is blue, there is no possible value for x_1. Therefore, this combination cannot satisfy all constraints.

The hyper-resolution rule is described as follows (A_i is a proposition such as $x_1 = 1$).

$$A_1 \vee A_2 \vee \ldots \vee A_m$$
$$\neg(A_1 \wedge A_{11} \ldots),$$
$$\neg(A_2 \wedge A_{21} \ldots),$$

$$\frac{\begin{array}{l}\vdots \\ \neg(A_m \wedge A_{m1} \ldots) \end{array}}{\neg(A_{11} \wedge \ldots \wedge A_{21} \wedge \ldots \wedge A_{m1} \ldots)}$$

In the hyper-resolution-based consistency algorithm, new nogoods are generated using the hyper-resolution rule. For example, in Fig. 1.20, a new nogood $\{x_2 = red, x_3 = blue\}$ is generated using nogood $\{x_1 = red, x_2 = red\}$ and nogood $\{x_1 = blue, x_3 = blue\}$. Then, another nogood $\{x_3 = blue\}$ can be generated using this nogood and nogood $\{x_2 = blue, x_3 = blue\}$. Similarly, a new nogood $\{x_2 = blue, x_3 = red\}$ can be generated from $\{x_1 = blue, x_2 = blue\}$ and $\{x_1 = red, x_3 = red\}$. Then, a new nogood $\{x_3 = red\}$ can be generated using this nogood and nogood $\{x_2 = red, x_3 = red\}$. Then, an empty nogood $\{\}$ can be generated using nogood$\{x_3 = blue\}$ and $\{x_3 = red\}$. Recall that a nogood is a combination of variable values that is prohibited. Therefore, a superset of a nogood cannot be a solution. Since any set is a superset of an empty set, if an empty set becomes a nogood, the problem is over-constrained and has no solution.

The hyper-resolution rule can generate a very large number of nogoods. If we restrict the application of the rules so that only nogoods with lengths less than k are produced, the problem becomes strongly k-consistent (the length of a nogood is the number of variables that constitute the nogood).

1.4 Hybrid-Type Algorithm of Backtracking and Iterative Improvement

This section describes a hybrid-type algorithms of backtracking and iterative improvement called the weak-commitment search algorithm.

1.4.1 Weak-Commitment Search Algorithm

This algorithm 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.

The weak-commitment search algorithm is illustrated in Fig. 1.22. The essential difference between this algorithm and the min-conflict backtracking (Fig. 1.13) is the shaded part in Fig. 1.22. In the min-conflict backtracking, backtracking is performed at this part and the most-recently added variable is removed from the partial solution. In the weak-commitment search, the whole partial solution is abandoned, i.e., all elements of *partial-solution* are

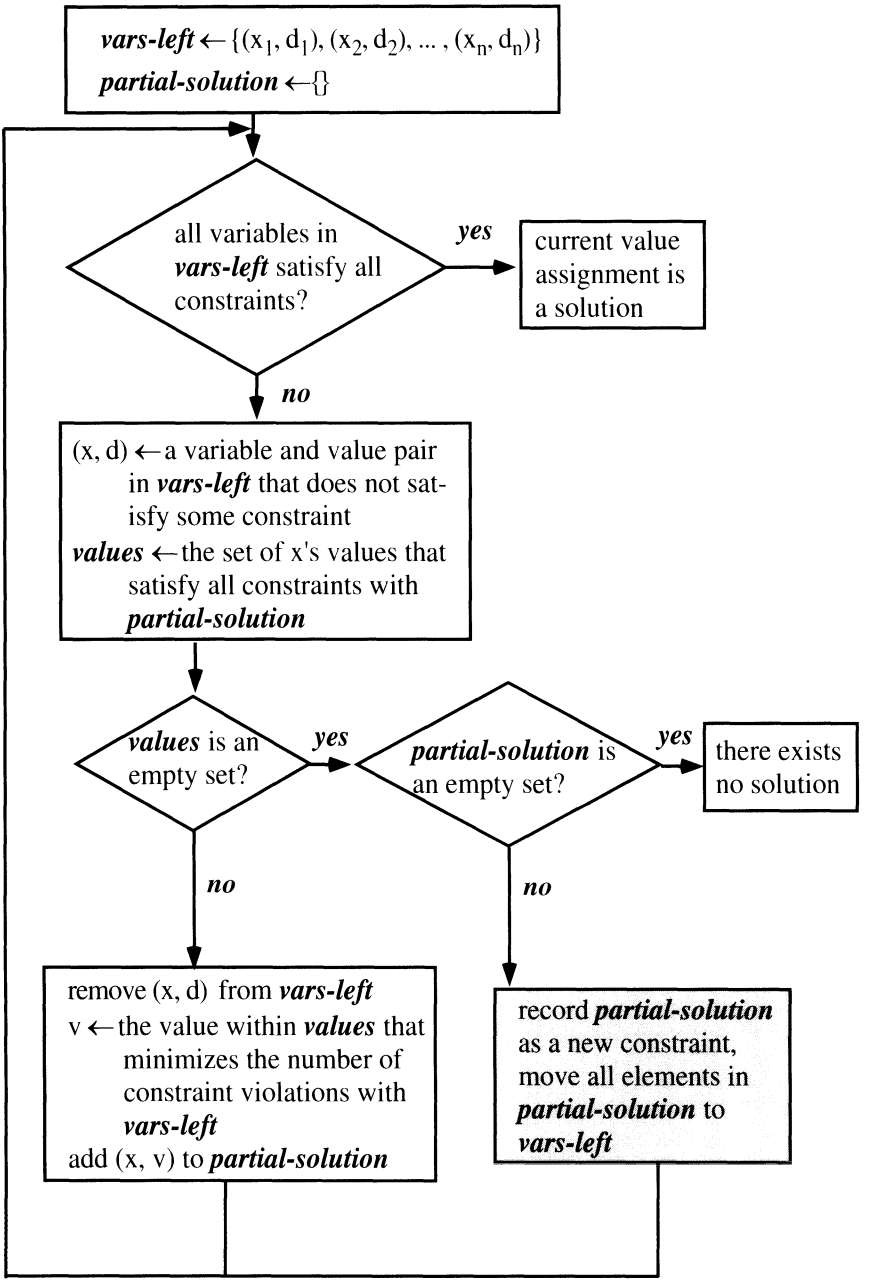

Fig. 1.22. Weak-Commitment Search Algorithm

moved to *vars-left*. Then, the search process is restarted using the current value assignment as new tentative initial values. It must be noted that not all variable values in the partial solution will be revised again. Since the algorithm revises only the constraint violating variables, the variables that already satisfy all constraints will not be revised again.

This algorithm records the abandoned partial solutions as new constraints, and avoids creating the same partial solution that has been created and abandoned before. Therefore, the completeness of the algorithm (i.e., always finding a solution if one exists, or terminating if no solution exists) is guaranteed.

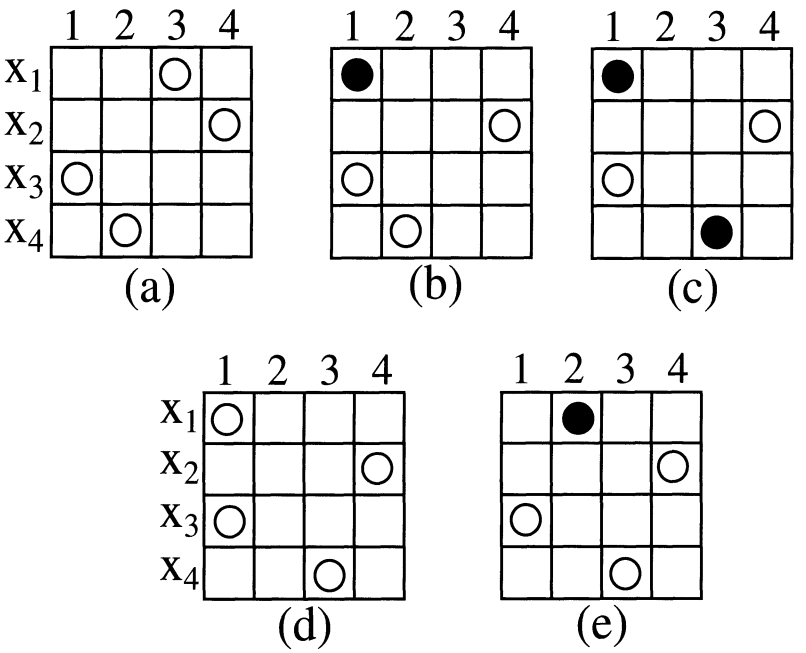

Fig. 1.23. Example of Algorithm Execution (Weak-Commitment Search)

1.4.2 Example of Algorithm Execution

We show an example of algorithm execution using the 4-queens problem. The initial state is illustrated in Fig. 1.23 (a). The algorithm first revises the position of the first queen (Fig. 1.23 (b)). In this case, for x_1, 1,2, and 4 are equally good (each of which violates one constraint). We assume that the algorithm randomly choose 1. A queen whose position is revised is added to *partial-solution*. We represent a queen in *partial-solution* as a filled circle.

Then, the algorithm revises the position of the fourth queen (Fig. 1.23 (c)). In this case, the algorithm can choose only 2 or 3 for the value of x_4, and the min-conflict heuristic prefers 3 to 2. In Fig. 1.23 (c), there exists no consistent position with *partial-solution* for the third queen. Therefore, the entire *partial-solution* is abandoned (Fig. 1.23 (d)). The algorithm then revises the position of the first queen again (Fig. 1.23 (e)). Consequently, all constraints are satisfied.

1.4.3 Evaluations

We compare the weak-commitment search, the min-conflict backtracking, and an iterative improvement algorithm by experiments on typical examples of CSPs (n-queens and graph-coloring problems). We use the breakout algorithm (Morris 1993) as the representative for iterative improvement algorithms. This algorithm has a notable feature in that it does not stop at local minima, and has been shown to be more efficient than other iterative improvement algorithms (Morris 1993). However, this algorithm is not complete, i.e., the algorithm may fall into an infinite processing loop.

We measure the number of required steps and the number of consistency checks. We count each change of one variable value, each backtracking, and each restart as one step. Also, one consistency check represents the check of one combination of variable values among which a constraint exists. The number of checks for newly added constraints (abandoned partial solutions) is also included. The number of consistency checks is widely used as a machine-independent measure for constraint satisfaction algorithms. For all three algorithms, in order to reduce unnecessary consistency checks, the result of consistency checks at the previous step is recorded and only the difference is calculated in each step.

In order to terminate the experiments within a reasonable amount of time, the maximum number of steps is limited to 5,000, and we interrupt any trial that exceeds this limit. For an interrupted trial, we count the number of required steps as 5,000, and use the number of consistency checks performed up to the interruption for the evaluation.

The first example problem is the n-queens problem. We show the ratio of successful trials (trials finished within the limit), the number of required steps, and the number of consistency checks for n=10, 50, and 100 in Table 1.1. We run 100 trials with different initial values and show the average. These initial values are generated by the greedy method described in (Minton, Johnston, Philips, and Laird 1992). In (Minton, Johnston, Philips, and Laird 1992), it is reported that the min-conflict backtracking can solve the 100-queens problem in around 25 steps. In our experiment, there exists one trial that exceeds 5,000 steps and the result of this trial becomes the dominant factor in the average. The average except this trial is almost identical to the result reported in (Minton, Johnston, Philips, and Laird 1992). For $n \geq 1{,}000$, the results for the min-conflict backtracking and weak-commitment search are

Table 1.1. Evaluation with n-Queens

Algorithm		n		
		10	50	100
weak-commitment	ratio	100%	100%	100%
	steps	29.7	23.9	27.1
	checks	2292.8	48593.5	236821.7
min-conflict BT	ratio	100%	97%	99%
	steps	240.7	264.5	76.4
	checks	15482.2	300175.0	912465.1
breakout	ratio	100%	100%	100%
	steps	41.7	37.9	38.9
	checks	7065.0	180393.5	777051.1

exactly the same, and almost identical to the result reported in (Minton, Johnston, Philips, and Laird 1992).

As shown in Table 1.1, for all cases, the weak-commitment search is more efficient than the min-conflict backtracking and the breakout algorithm. We can see that the breakout algorithm requires many more consistency checks for each step compared with the weak-commitment search. This fact can be explained as follows. When choosing a variable to change its value, the weak-commitment search (and the min-conflict backtracking) can choose any of the constraint violating variables. On the other hand, the breakout algorithm must choose a variable so that the number of constraint violations can be reduced by changing its value. Therefore, in the worst case (when the current assignment is a local minimum), the breakout algorithm has to check all of the values for all constraint violating variables. On the other hand, if the current assignment is not a local minimum, the breakout algorithm does not have to check all variables. For the trials without backtracking/restarting, the behaviors of the weak-commitment search and the min-conflict backtracking are exactly the same. We show the ratio of trials with backtracking/restarting, and the average number of steps for these trials in Table 1.2. We can see that the numbers of required steps for the min-conflict backtracking in trials with backtracking are very large and dominate the average of all trials.

Table 1.2. Required Steps for Trials with Backtracking/Restarting

n	ratio of trials with BT	weak-commitment	min-conflict BT
10	80%	35.9	299.7
50	17%	58.2	1473.4
100	2%	96.5	2563.5

Table 1.3. Evaluation with Graph-Coloring

Algorithm		n		
		120	180	240
weak-commitment	ratio	100%	100%	100%
	steps	28.9	41.3	71.9
	checks	2118.8	3178.9	5988.6
min-conflict BT	ratio	99%	99%	95%
	steps	78.1	98.5	443.6
	checks	2931.2	5548.5	42164.5
breakout	ratio	100%	100%	100%
	steps	198.4	352.3	601.2
	checks	14620.8	32139.3	66892.5

The graph-coloring problem involves painting nodes in a graph by k different colors so that any two nodes connected by an arc do not have the same color. We randomly generate a problem with n nodes and m arcs by the method described in (Minton, Johnston, Philips, and Laird 1992), so that the graph is connected and the problem has a solution. We evaluate the problem $n = 120, 180, 240$, where $m = n \times 2$ and k=3. This parameter setting corresponds to the "sparse" problems for which (Minton, Johnston, Philips, and Laird 1992) report poor performance of the min-conflict heuristic. We generate ten different problems, and for each problem, ten trials with different initial values are performed (100 trials in all). As in the n-queens problem, the initial values are set by the greedy method.

We introduce forward-checking and the first-fail principle described in Section 1.3.1 into the min-conflict backtracking and the weak-commitment search, i.e., for each variable, we keep the list of values consistent with the partial solution, and when selecting a variable to be added to the partial solution, we select the variable that has the least number of consistent values. Also, a variable that has only one consistent value is included into the partial solution immediately. Furthermore, before including a variable into the partial solution, we check whether each of remaining variables (variables in *vars-left*) has at least one consistent value with the partial solution, and avoid selecting a value that causes immediate failure. Table 1.3 shows evaluation results for the three algorithms.

Although (Minton, Johnston, Philips, and Laird 1992) report poor performance of the min-conflict backtracking for these problems, by introducing the two heuristics (forward-checking and the first-fail principle), the performance of the min-conflict backtracking becomes relatively good in our evaluation. However, there exist several trials in which a mistake in the value selection becomes fatal, and the min-conflict backtracking fails to find a solution within the limit. Therefore, the weak-commitment search is more efficient than the min-conflict backtracking. For these problems, forward-checking and first-fail principle are very efficient and the weak-commitment search is about 10

times more efficient than the breakout algorithm. We can see that the capability of accommodating such powerful heuristics is a great advantage of the weak-commitment search over iterative improvement algorithms.

Furthermore, to show theoretical evidence that the weak-commitment search is more efficient than the min-conflict backtracking, we use a simple probabilistic model as follows. Let us assume that the probability for finding a solution without any backtracking in the min-conflict backtracking is given by the constant value p, regardless of the initial values. Also, let us assume that the expected number of steps for trials without backtracking is given by n_s ($n_s \leq n$), the expected number of steps for trials with backtracking is given by B, and the expected number of steps up to the occurrence of the first backtracking is given by n_b ($n_b \leq n$). Then, the expected number of steps for the min-conflict backtracking can be represented as $n_s p + Bq$, where $q = 1 - p$ (Fig. 1.24).

On the other hand, in the weak-commitment search, a solution can be found without any restarting with the probability p, and the expected number of steps in this case is given by n_s. Also, the probability that a solution can be found after one restart is given by pq, and the expected number of steps is given by $n_s + n_b$. In the same way, the probability that a solution can be found after k restart is given by pq^k, and the expected number of steps is given by $n_s + kn_b$ (Fig. 1.24). This probability distribution of the number of restarts is identical to the well-known geometric distribution, and the expected number of restarts is given by q/p. Therefore, the expected number of steps can be given by $n_s + n_b q/p$.

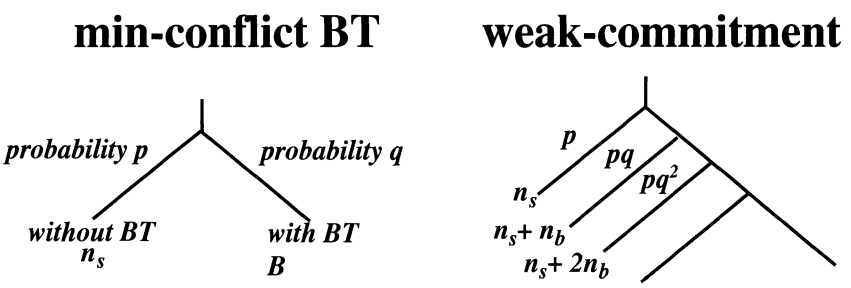

Fig. 1.24. Probabilistic Model (Min-conflict Backtracking and Weak-Commitment Search)

The condition that the weak-commitment search is more efficient than the min-conflict backtracking is $n_s p + Bq > n_s + n_b q/p$. By transforming this formula, we obtain $p > n_b/(B - n_s)$. Since we can assume that $B \gg n_s$, and $n_b \leq n$, we obtain the sufficient condition $p > n/B$, i.e., if the probability of

finding a solution without backtracking is larger than the ratio of the number of variables and the number of steps for the trials with backtracking, the weak-commitment search is more efficient than the min-conflict backtracking.

The experimental results in this section show that the min-conflict backtracking can find a solution efficiently without backtracking in many trials, but the number of required steps for trials with backtracking becomes very large. Therefore, we can assume that the condition $p > n/B$ is satisfied in many cases. For example, from the experimental results in Table 1.2, we can assume that B for the 50-queens problem is around 1,473.4, and p is around 0.83 (where q is 0.17). Therefore, n/B is around 0.034. This value is much smaller than the expected value of p (0.83).

In reality, the probability of finding a solution without backtracking is affected by the initial values. In the weak-commitment search, when restarting, the current value assignment is used as the new tentative initial values. Therefore, by repeating the restarts, we can expect the value assignment to become close to the final solution; thus, the probability of finding a solution without restarts increases. In such a case, even if the average probability of finding a solution without backtracking is very small and the condition $p > n/B$ is not satisfied, the weak-commitment search can be more efficient than the min-conflict backtracking.

1.4.4 Algorithm Complexity

The worst-case time complexity of the weak-commitment search becomes exponential in the number of variables n. This result seems inevitable since constraint satisfaction is NP-complete in general. The space complexity of the weak-commitment search is determined by the number of newly added nogoods (constraints), i.e., the number of restarts. In the worst case, this is also exponential in n. On the other hand, the space complexity of the backtracking algorithm is linear to n. This result seems inevitable since the weak-commitment search changes the search order flexibly while guaranteeing the completeness of the algorithm. This is also the case for the fill algorithm described in (Morris 1993), whose worst-case space complexity becomes exponential in n.

However, the number of restarts will never exceed the number of required steps. Therefore, we can assume that the space complexity would never become a problem in practice as long as the problem can be solved within a reasonable amount of time. Also, the nogood that is a superset of other nogoods is redundant and can be abandoned. Furthermore, we can restrict the number of recorded nogoods. In this case, the theoretical completeness cannot be guaranteed. However, in practice, the weak-commitment search algorithm can still find a solution for all example problems when the number of recorded nogoods is restricted so that only ten of the most recently found nogoods are recorded.

1.5 Analyzing Landscape of CSPs

1.5.1 Introduction

In this section, we describe which kinds of problem instances of CSPs are most difficult for complete search algorithms and hill-climbing algorithms. This topic has attracted a lot of attention, and rapid advances have been made in complete search algorithms (Cheeseman, Kanefsky, and Taylor 1991; Hogg, Huberman, and Williams 1996; Mitchell, Selman, and Levesque 1992). In brief, these works show that the problems in the region where the solvable probability is about 50% (phase transition region) tend to be most difficult.

More specifically, let us assume we have at least one parameter that affects the difficulty of a problem instance. For example, let us consider a SAT problem instance. If we randomly generate problem instances, we can choose the number of generated clauses. If the number of clauses is small, the problem instance should have many solutions. If the number of clauses becomes large, the problem instance probably has no solution. By changing the number of clauses, at some point, the probability that the generated problem instances have at least one solution becomes about 50%, and these problem instances tend to be most difficult (Fig. 1.25).

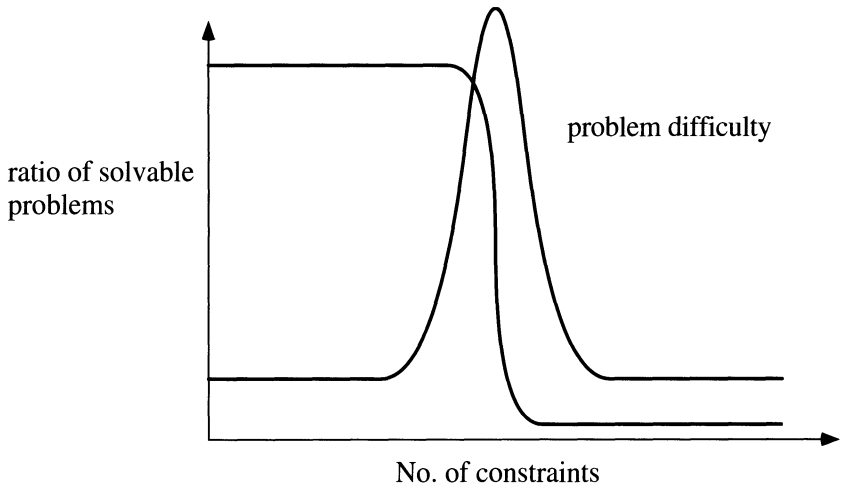

Fig. 1.25. Phase Transition

The above results are intuitively natural. If the problem is weakly constrained, a complete algorithm will easily find a solution. Also, if the problem is very strongly constrained, the problem becomes easy since the complete algorithm can prune most of the branches in the search tree. Therefore, the

most difficult problems will exist in the middle, i.e., the phase transition region. Interestingly, the change of the probability is very rapid, so we call it a phase transition. Furthermore, in many cases, we can very accurately predict the point where the phase transition likely occurs by using a small number of parameters. For example, in SAT problems, we can use the clause density, i.e., the ratio of the number of clauses to the number of variables for predicting the difficulty of a problem instance. Furthermore, it has been pointed out that phase transition usually occurs when the clause density is around 4.3 for 3-SAT problems (Cheeseman, Kanefsky, and Taylor 1991).

On the other hand, what kinds of problems are most difficult for incomplete hill-climbing algorithms, such as GSAT (Selman, Levesque, and Mitchell 1992), that do not perform an exhaustive search? Since these algorithms cannot discover the fact that a problem is unsolvable, trying to solve unsolvable problems by these algorithms is futile. Therefore, if we choose solvable problem instances and apply the algorithms to these problem instances, what kinds of problems are most difficult?

At first glance, we may think that a problem becomes more difficult for hill-climbing algorithms if we add more constraints, which provides the problem with fewer solutions. However, by actually solving problems using hill-climbing algorithms, the problems in the phase transition region are the most difficult, as with the complete algorithms, although these problems have more solutions than the problems beyond the phase transition region, i.e., more constrained problems. Such paradoxical results have been reported in (Clark, Frank, Gent, MacIntyre, Tomov, and Walsh 1996).

In the rest of this section , we clarify the cause of this paradoxical phenomenon by exhaustively analyzing the state-space landscape of CSPs. A state is one possible assignment to all variable values, and the evaluation value of the state is the number of its constraint violations. The landscape of the state-space is formed by the evaluation values of the states. Such analyses are important not only for clarifying the cause of this paradoxical phenomenon, but also for developing more efficient hill-climbing algorithms by utilizing the results of landscape analyses.

For path-planning problems, such landscape analyses have been reported in (Ishida 1996). On the other hand, as far as the author knows, there has not been much research done on analyzing the landscapes of CSPs, with the notable exceptions of (Hertz, Jaumard, and de Aragao 1994; Frank, Cheeseman, and Stutz 1997; Hogg 1996).

In this section, we first divide the states into two classes: states that are *reachable* to solutions, and states that are unreachable to solutions since they lead to local-minima. Then, we show that the problems beyond the phase transition region actually have more solution-reachable states than the problems in the phase transition region. Next, we show that the number of local-minima decreases by adding more constraints, thus the number of solution-reachable states increases. Furthermore, we examine the number and

widths of *basins*, each of which is a set of interconnected local-minima. We show that the number of local-minima decreases because a basin is divided into smaller regions by adding more constraints.

1.5.2 Hill-Climbing Algorithm

In a hill-climbing algorithm for solving a CSP, each variable has a tentative initial value, and the value of a variable is changed one by one so that the number of constraint violations decreases.

We introduce the following terms for explaining our algorithm and the analyses in the next section.

state: We call one possible assignment of all variables a *state*. In a SAT problem with n variables, the number of states is 2^n.

evaluation value: For a state s, we call the number of constraint violations of the state the *evaluation value* of the state (represented as $eval(s)$).

neighbor: For a state s, we say another state s' is a *neighbor* of s, if s' is obtained by changing one variable value of s. In a SAT problem with n variables, each state has n neighbors.

local-minimum: For a state s that is not a solution, if the evaluation values of all of its neighbors are larger than or equal to s's evaluation value, we say s is a *local-minimum*. Specifically, the following condition is satisfied: $\forall s'$ if s' is a neighbor of s, then $eval(s') \geq eval(s)$. For example, in the imaginary one-dimensional state-space shown in Fig. 1.26, states c, d, f, h, and i are local-minima.

strict local-minimum: For a state s that is not a solution, if the evaluation values of all of its neighbors are larger than s's evaluation value, we say s is a *strict local-minimum*. Specifically, the following condition is satisfied: $\forall s'$ if s' is a neighbor of s, then $eval(s') > eval(s)$. In Fig. 1.26, the state f is a strict local-minimum.

non-strict local-minimum: If a state s is a local-minimum, but not a strict local-minimum, we say s is a *non-strict local-minimum*. If s is a non-strict local-minimum, there exists at least one neighbor that has the same evaluation value. In Fig. 1.26, states c, d, h, and i are non-strict local-minima.

In this section, we use the algorithm described in Fig. 1.27 for our analyses. This algorithm is a greedy, unfair deterministic tie-breaking algorithm (Gent and Walsh 1993). As the basic GSAT, this algorithm moves from the current state to the neighboring state, which has the smallest evaluation value. Also, if the current state is a non-strict local-minimum, this algorithm can move to the neighboring state that has the same evaluation value (sideway move).

On the other hand, unlike GSAT, in which ties are broken randomly, if there exist multiple states that have the best (smallest) evaluation value, ties are broken in a fixed, deterministic, unfair way (e.g., each state has a unique identifier, and a tie is broken by using the alphabetical order of these

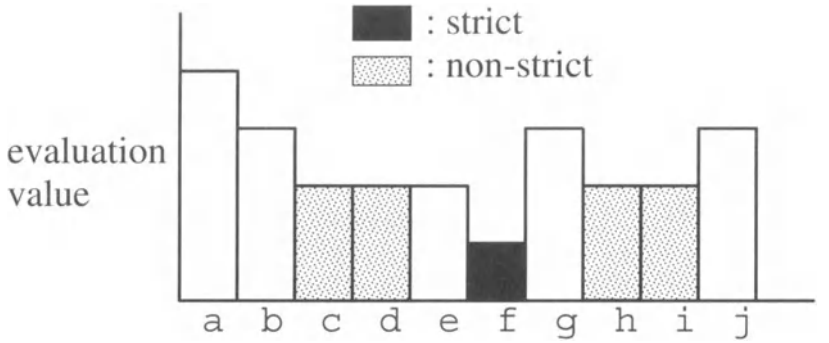

Fig. 1.26. One-Dimensional State-Space

identifiers). We use this method in order to simplify the following analyses. We discuss the effect of tie-breaking methods on algorithm efficiency in Section 1.5.4.

```
procedure hill_climbing
    restart: set s to a randomly selected initial state;
    for j:=1 to Max-flips
        if eval(s) = 0 then return s;
        else if s is a strict local-minimum then goto restart;
            else s:= a neighbor of s that has the smallest evaluation value;
            (ties are broken deterministically)
            end if;
        end if;
    end;
    goto restart;
```

Fig. 1.27. Hill-Climbing Algorithm

In the following, we mainly use randomly generated 3-SAT problems for our analyses. In a randomly generated 3-SAT problem, each clause is generated by randomly selecting three variables, and each of the variables is given the value true or false with a 50% probability. Figure 1.28 shows the ratio of solvable problems, in 100 randomly generated 3-SAT problem instances with 24 variables, by varying the clause density (number of clauses/number of variables). We can see that phase transition occurs[1] when the clause density is around 4.67.

[1] Since we use very small-scale problems, the phase transition is not as drastic as with large-scale problems.

Furthermore, we select 50 solvable problem instances from randomly generated problems for each clause density, and solve these instances with the algorithm described in Fig. 1.27. We show the average number of restarts (including the first trial) required to solve the problem instances in Fig. 1.29 (for each problem instance, 1,000 trials are performed using different initial states). Also, we show the average number of solutions to these solvable problem instances in Fig. 1.30.

From these figures, we can see that the average number of solutions at the clause density 4.67 (where the phase transition occurs) is 11.5, and the average number of solutions at the clause density 5.83 (beyond the phase transition) is 3.6 (less than 1/3 of the solutions in the phase transition region). However, the required number of restarts at the clause density 5.83 is 6.9 (about 50% fewer than for the phase transition region, i.e., 13.0). Consequently, the problems become easier.

Although we use very small-scale problem instances, we can see the paradoxical phenomenon that the problems with fewer solutions are actually easier than the problems with more solutions.

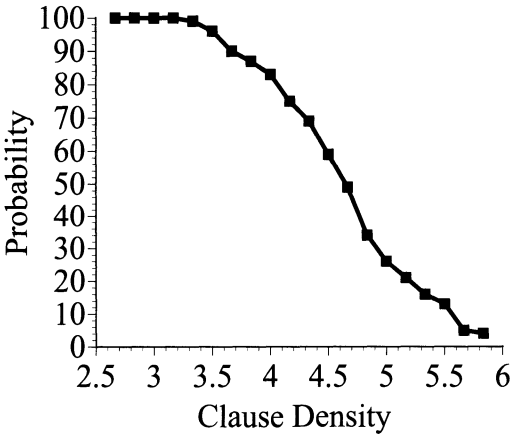

Fig. 1.28. Probability of Satisfiability (3-SAT Problems)

1.5.3 Analyzing State-Space

Analyzing Solution-Reachability. For each state, the neighboring state that the algorithm can move to is uniquely determined, since ties are broken in a fixed, deterministic method. The state-space of the problem can be described using a graph, in which a state is represented as a node, and a possible transition between states is represented as a directed link (Fig. 1.31). In Fig. 1.31, a symbol near a node represents the identifier of the state, and

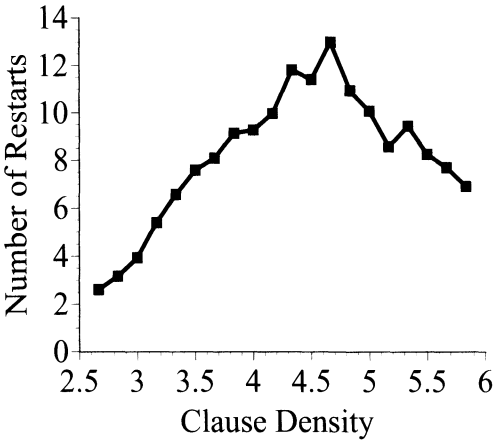

Fig. 1.29. Average Number of Restarts (3-SAT Problems)

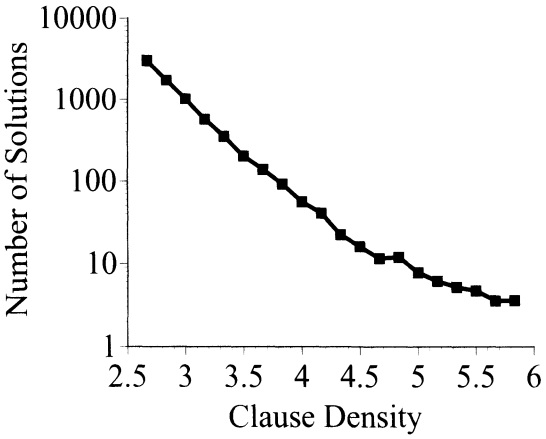

Fig. 1.30. Average Number of Solutions (3-SAT Problems)

a value within a node represents the evaluation value of the state. Also, a dotted link connects two neighboring states that do not have a directed link between them. For example, in Fig. 1.31, e and h are neighbors, but there is no directed link between them, since the algorithm moves to f from these states. In this example, we assume that ties are broken in the alphabetical order of identifiers.

Each node has exactly one outgoing link, except for a strict local-minimum (e.g., state g) and a solution (e.g., state f). Also, there can be a cycle that circulates around multiple non-strict local-minima (e.g., links between a and b).

By tracing directed links from a state s, the result will be either to reach a solution or not (i.e., to reach a strict local-minimum, or to go around non-strict local-minima). In the former case, we say s is *reachable* to solutions, and in the latter case, we say s is *unreachable* to solutions. For example, in Fig. 1.31, c, e, f and h are reachable and a, b, d and g are unreachable to solutions.

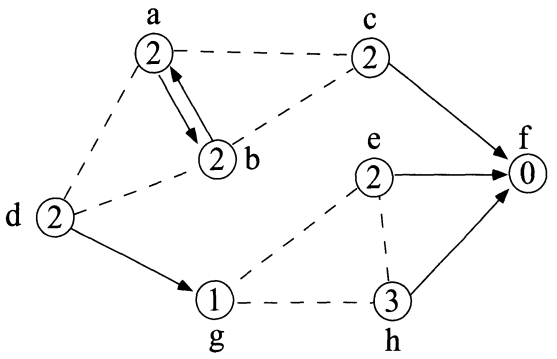

Fig. 1.31. Example of State-Space

In Fig. 1.32, we show the average ratio of the states that are solution-reachable. By adding more constrains, the average ratio of solution-reachable states actually increases beyond the phase transition region.

If the probability that a randomly selected state is solution-reachable is given by p, the estimated number of restarts required to find a solution will be $1/p$ (i.e., $p + 2p(1-p) + 3p(1-p)^2 + \ldots$). In Fig. 1.33, we show the estimated number of restarts[2] derived from Fig. 1.32. The values in Figure 1.33 are almost identical to the actual measurements in Fig. 1.29.

[2] Note that the average ratio at clause density 4.67 is 0.12, but this does not mean the estimated number of restarts will be 8.33=1/0.12. In general, the average of '$1/p$' is not equal to the reciprocal of the average of p. For example, the average of 1/5 and 1/2 is 7/20, while the reciprocal of the average of 5 and 2 is 2/7.

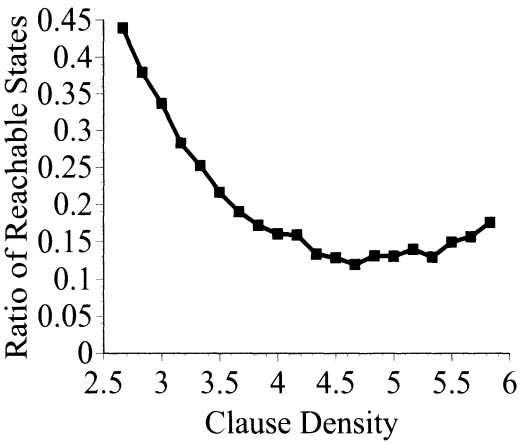

Fig. 1.32. Average Ratio of Solution-Reachable States (3-SAT Problems)

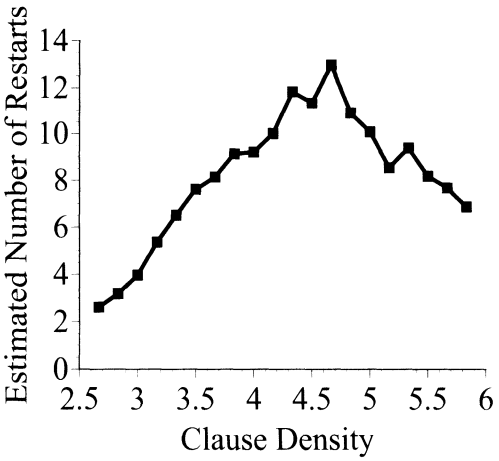

Fig. 1.33. Average Estimated Number of Restarts (3-SAT Problems)

Analyzing Local-Minima. In view of the above, why does adding more constraints increase the number of solution-reachable states beyond the phase transition, although it decreases the number of solutions? As shown in Fig. 1.31, tracing directed links from a state will either result in reaching a solution or reaching a local-minimum (strict or non-strict). Let us examine how the number of local-minima changes by adding constraints. Figure 1.34 shows the average number of local-minima. We can see that the number of local-minima decreases monotonically by adding constraints.

While adding constraints makes a problem harder for a hill-climbing algorithm by decreasing the number of solutions, it also makes the problem *easier* by decreasing the number of local-minima. As Figure 1.30 shows, the number of solutions decreases very rapidly around the phase transition region, and then decreases slowly beyond the phase transition region (note that the y-axis of Fig. 1.30 is log-scaled). On the other hand, the number of local minima decreases at a more constant rate (note that the y-axis of Fig. 1.34 is linear). Therefore, we can assume that by adding more constraints, the effect of making the problem easier will exceed the effect of making the problem harder beyond the phase transition region.

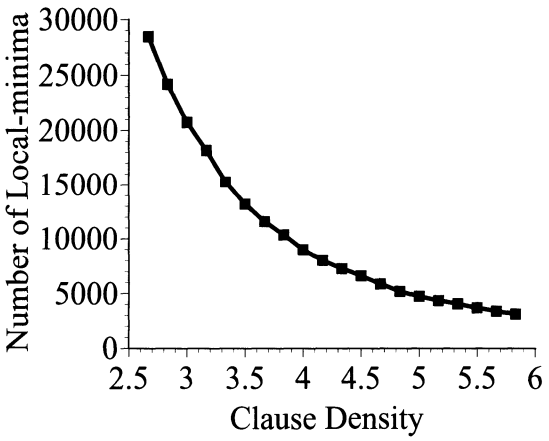

Fig. 1.34. Average Number of Local-Minima (3-SAT Problems)

Analyzing Basins. Considering the above, why does the number of local-minima decrease monotonically by adding more constraints? Adding constraints does not necessarily mean that the number of local-minima must decrease. By adding more constraints and increasing the evaluation values of states, some of these states may change to non local-minima, but some states that are neighbors of the increased states may become local-minima.

This question can be answered by analyzing the landscape of the state-space. By checking each local-minimum, we found that almost all local-

minima are non-strict local-minima (there are only a few strict local-minima for each problem), and many non-strict local-minima are interconnected by neighborhood relation to create a flat surface in the state-space. We call such parts *basins*.

The formal definition of a basin is as follows.

Definition: A basin is a set of connected[3] states that satisfies the following conditions.

- All states in the basin have the same evaluation value, which is larger than 0.
- For each state s in the basin, each neighboring state s' of s, which is not a member of the basin, has an evaluation value greater than or equal to the evaluation value of s, i.e., $eval(s') \geq eval(s)$.

We say a basin is *maximal* if the above conditions are violated by adding any neighboring state. Also, we call the number of states in a basin the *width* of the basin. For example, in Fig. 1.26, there are three maximal basins, i.e., $\{c, d\}$, $\{f\}$, and $\{h, i\}$. Also, in Fig. 1.31, there are two maximal basins, i.e., $\{a, b\}$ and $\{g\}$. From this definition, any state in a basin is a local-minimum, any local-minimum is a member of some maximal basin, and a state cannot be a member of two different maximal basins. Therefore, the summation of widths of maximal basins is equal to the number of local-minima. Also, a strict local-minimum forms a basin that has only one member.

Figure 1.35 shows the average width of maximal basins. We can see that the average width of basins decreases monotonically by adding constraints. Figure 1.36 shows the average number of maximal basins. We can see that the number of basins increases initially, then reaches a peak, and finally slowly decreases. When a problem is very weakly constrained, adding more constraints divides a basin into several regions. When the problem is strongly constrained, each basin is small and the basins tend to be eliminated by adding more constraints. It is obvious that if a basin is divided into several regions, the summation of widths of these parts would be smaller than the original width.

To summarize, most local-minima are interconnected with each other to create basins. By adding constraints, a basin is divided into smaller regions and the summation of widths of maximal basins (the number of local-minima) decreases.

1.5.4 Discussions

3-Coloring Problem. The analyses in the previous section give us a clear explanation of the paradoxical phenomenon. However, how general are these results?

[3] We mean *connected* by neighborhood relation in general, not only by directed links.

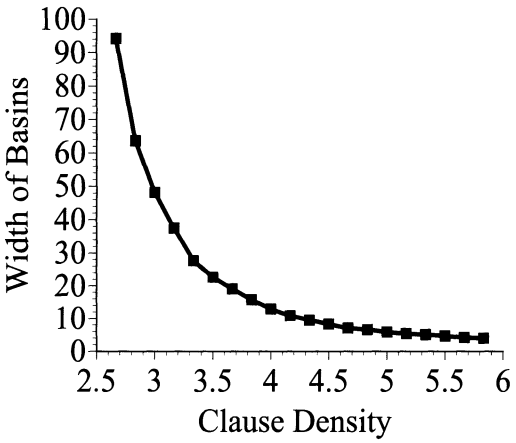

Fig. 1.35. Average Width of Basins (3-SAT Problems)

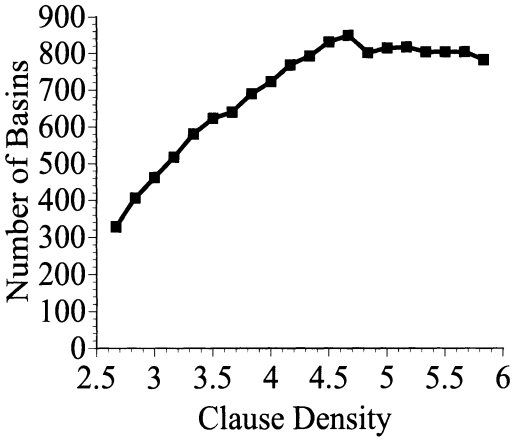

Fig. 1.36. Average Number of Basins (3-SAT Problems)

To reconfirm the results in other classes of CSPs, we show the evaluation results in another important class of CSPs, i.e., graph-coloring problems. In Fig. 1.37, we show the ratio of solvable problems in 100 randomly generated problem instances, each having 12 variable and three possible colors (3-coloring problems), by varying the link density (number of links/number of variables). We can see that phase transition occurs[4] when the link density is around 1.9.

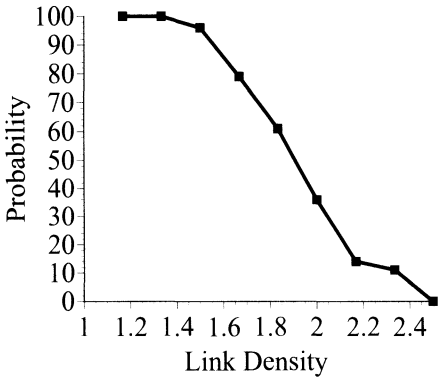

Fig. 1.37. Probability of Satisfiability (3-Coloring Problems)

Furthermore, we select 50 solvable problem instances from randomly generated problems for each link density and solve these instances. We show the average number of restarts required to solve the problem instances in Fig. 1.38, and the average number of solutions to these solvable problem instances in Fig. 1.39. As with the 3-SAT problem, we can see the paradoxical phenomenon in which problems with fewer solutions are easier than problems with more solutions.

Figure 1.40 shows the average ratio of the solution-reachable states. By adding more constrains, the average ratio of solution-reachable states increases beyond the phase transition region.

Figure 1.41 shows the average number of local-minima. As with the 3-SAT problems, the number of local-minima decreases monotonically by adding constraints, except for the part where the link density is very low.

Figure 1.42 shows the average width of maximal basins, and Figure 1.43 shows the average number of maximal basins. The results are very similar to those obtained for the 3-SAT problems, except that the number of basins increases monotonically.

[4] As in the case of the 3-SAT problems, the phase transition is not as drastic as with large-scale problems.

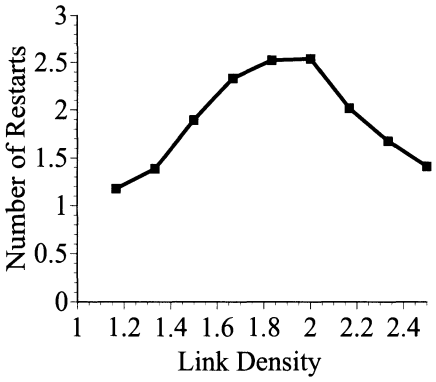

Fig. 1.38. Average Number of Restarts (3-Coloring Problems)

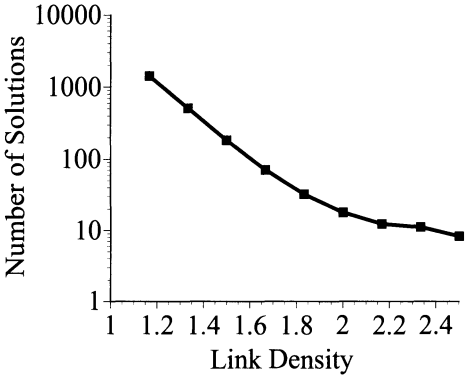

Fig. 1.39. Average Number of Solutions (3-Coloring Problems)

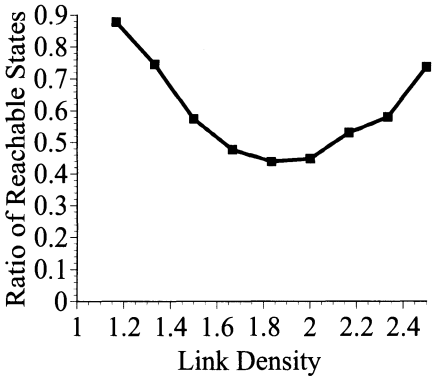

Fig. 1.40. Average Ratio of Solution-Reachable States (3-Coloring Problems)

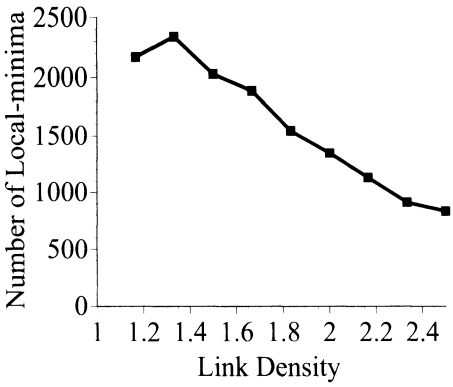

Fig. 1.41. Average Number of Local-Minima (3-Coloring Problems)

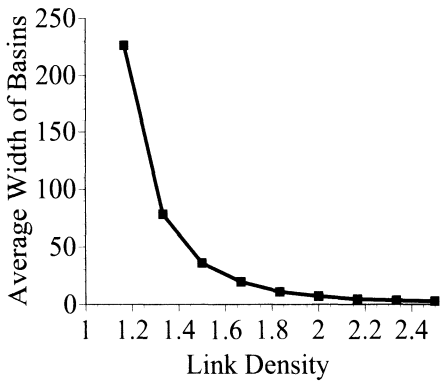

Fig. 1.42. Average Width of Basins (3-Coloring Problems)

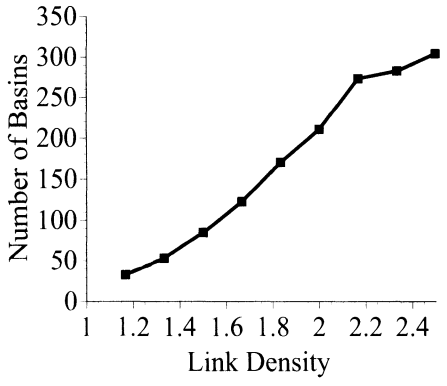

Fig. 1.43. Average Number of Basins (3-Coloring Problems)

Effects of Tie-Breaking. We use an algorithm with an unfair deterministic tie-breaking method. As discussed in (Gent and Walsh 1993), using an unfair tie-breaking method degrades the performance of the algorithm. In the example problems used in this section, we can obtain about a two-fold speedup by using a random tie-breaking method.

A random tie-breaking method complicates the solution-reachability analyses. This is because the same state can be either solution-reachable or unreachable, depending on the tie-breaking. For example, in Fig. 1.31, let us assume that the current state is a. If ties are broken in a fixed way as described in the figure, the algorithm circulates between a and b. On the other hand, if ties are broken randomly, the algorithm may fall into the strict local-minimum g or reach a solution via c.

However, it should be noted that the analyses of local-minima and basins are not affected by tie-breaking methods.

Problem Size. One drawback of our results is that only very small-scale problems were analyzed. To perform an analysis of solution-reachability or an analysis of basins within a reasonable amount of time, we need to explicitly construct a state-space within the physical memories of a computer. For example, if the number of variables of a 3-SAT problem is 30, the total number of states becomes 10^9. Even if we could manage to represent each state with 4 bytes, the required memory size would be 4 GB. Evaluating the number of local-minima could be done without explicitly constructing a state-space. However, we still need to exhaustively check each state. In the case of a 3-SAT problem with 40 variables, the total number of states is 10^{12}. If we could check 10^6 states per second, we would still need more than a week to analyze one problem instance.

However, although the analyses were done only for very small-scale problems, the observed phenomenon is very similar to that of larger-scale problems reported in (Clark, Frank, Gent, MacIntyre, Tomov, and Walsh 1996). Also, very similar results were obtained for two different classes of problems (3-SAT and 3-coloring). Therefore, we can assume that the obtained results would also be valid for large-scale problems.

1.6 Partial Constraint Satisfaction Problem

1.6.1 Introduction

When a problem designer tries to describe a real-life problem as a CSP, the resulting CSP is often over-constrained and has no solutions. For such an over-constrained CSP, almost all conventional CSP algorithms simply produce a result that says there is no solution. If we are interested in solutions for practical use, the designer has to go back to the design phase and find another description so that the CSP is not over-constrained. Freuder and Wallace (1992) extended the CSP framework and provided a *partial constraint*

satisfaction problem (partial CSP) as one approach to over-constrained CSPs (Descotte and Latombe 1985; Freeman-Benson, Maloney, and Borning 1990). In a partial CSP, we are required to find consistent assignments to an allowable relaxed problem. In the rest of this section, we will give a formal definition of partial CSPs and describe algorithms for solving them.

1.6.2 Formalization

A partial CSP is formally described as the following three components:

$$\langle (P, U), (PS, \leq), (M, (N, S)) \rangle,$$

where P is an original CSP, U is a set of 'universes', i.e., a set of potential values for each variable in P, $(PS, \leq)$ is a problem space, where PS is a set of CSPs (including P), and $\leq$ is a partial order over PS, M is a distance function over the problem space, and (N, S) are necessary and sufficient bounds on the distance between P and some solvable member of PS.

A *solution* to a partial CSP is a soluble problem P' from the problem space and its solution, where the distance between P and P' is less than N. Any solution will suffice if the distance between P and P' is not more than S, and all search can terminate when such a solution is found. An *optimal solution* to a partial CSP is a solution in which the distance between P and P' is minimal, and such a minimal distance is called the optimal distance.

This formalization is very general, and we can consider various special cases of partial CSPs. One important subclass of partial CSPs is maximal CSPs (Freuder and Wallace 1992). The goal of a maximal CSP is to find a solution that satisfies as many constraints as possible. In other words, the distance between a soluble problem and the original problem is measured by the number of removed constraints from the original problems. Also, maximal CSPs can be generalized to weighted CSPs (Freuder and Wallace 1992), in which a weight is defined for each constraint, and the goal is to maximize the weighted sum of satisfied constraints.

Another important subclass is hierarchical CSPs (Freuder and Wallace 1992). In a hierarchical CSP, constraints are divided into several groups, and these groups are ordered according to the importance value of constraints within a group. If a problem is over-constrained, we can discard constraints in a least important group. In this case, the distance between a soluble problem and the original problem is measured by the maximal importance value of discarded constraints.

1.6.3 Algorithms

Since partial CSPs are optimization problems, we can apply standard techniques for optimization problems, such as branch and bound search algorithms (Pearl 1984). In (Freuder and Wallace 1992), a branch and bound

search algorithm called P-BB was developed. This algorithm is a generalized version of the standard backtracking algorithm for solving normal CSPs. We show an overview of the P-BB algorithm in Fig. 1.44. In this algorithm, we have a sufficient and a necessary bound of the distance, S and N, respectively. We need to find a solution whose distance is less than N. Also, if a solution has a distance that is smaller than or equal to S, we can terminate the algorithm. Various extensions of this algorithm, such as introducing backmarking (Gasching 1977), forward-checking, and consistency algorithms have been developed (Freuder and Wallace 1992).

```
P-BB (Search-path, Distance, Variables, Values)
    if Variables={} — all variables have been assigned values in Search-path;
        then Best-solution ← Search-path;
            N ← Distance;
            if N ≤ S then return 'finished'; — Best-solution is sufficient;
            else return 'keep-searching';
    else if Values = {} then — tried all values for extending Search-path;
        return 'keep-searching'; — so will back up to see if can try another value
                for the last variable;
    else if Distance = N then — already extended Search-path to assign values
                for remaining variables without violating any additional constraints;
        return 'keep-searching'; — so will see if can do better by backing up;
    else — try to extend Search-path;
        Current-value ← first value in Values;
        New-distance ← Distance;
        try choices in Search-path, from first to last, as long as New-distance < N;
            when choice is inconsistent with Current-value do
                New-distance ← New-distance+1; end do;
        if New-distance < N and
            P-BB(Search-path plus Current-value, New-Distance,
                    Variables minus the first variable,
                    values of the second variable)='finished'
        then return 'finished; — Search-path was extended to sufficient solution;
        else — will see if can do better with another value;
            return P-BB(Search-path, Distance, Variables,
                    Values minus Current-value);
```

Fig. 1.44. Branch and Bound Algorithm

1.7 Summary

In this chapter, we presented a formal definition of CSPs and described algorithms for solving CSPs. Furthermore, we presented a hybrid-type algorithm of backtracking and iterative improvement algorithms called weak-

commitment search. Evaluation results showed the advantage of the weak-commitment search algorithm. Furthermore, we discussed which kinds of problem instances would be most difficult for complete search algorithms and hill-climbing search algorithms, and showed an analysis of the landscapes of CSPs. Finally, we described the formalization and algorithms of partial CSPs.

For further reading on CSPs, Tsang's textbook (Tsang 1993) on constraint satisfaction covers topics from basic concepts to recent research results. There are also several concise overviews of constraint satisfaction problems, such as (Mackworth 1992), (Dechter 1992), and (Kumar 1992).

2. Distributed Constraint Satisfaction Problem

2.1 Introduction

In this chapter, we give a formal definition of distributed CSPs (Section 2.2). 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 multi-agent systems (MAS). Therefore, distributed constraint satisfaction techniques can be considered as an important infrastructure for cooperation. As described in Section 2.3, various application problems in MAS can be formalized as distributed CSPs, by extracting the essential part of the problems. Once we formalize our application problem as a distributed CSP, we don't have to develop algorithms for solving it from scratch, since various algorithms for solving distributed CSPs have been developed. We show the classification of these algorithms in Section 2.4.

2.2 Problem Formalization

A distributed CSP is a CSP in which variables and constraints are distributed among automated agents. We assume the following communication model.

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

It must be noted that this model does not necessarily mean that the physical communication network must be fully connected (i.e., a complete graph). Unlike most parallel/distributed algorithm studies, in which the topology of the physical communication network plays an important role, we assume the existence of a reliable underlying communication structure among the agents and are not concerned about the implementation of the physical communication network. This is because our primary concern here is cooperation

M. Yokoo, *Distributed Constraint Satisfaction*

among intelligent agents rather than solving CSPs by certain multiprocessor architectures.

Each agent has some variables and tries to determine their values. However, there exist 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 $belongs(x_j, i)$). We can consider the case that several agents share a variable. However, such a case can be formalized as these agents having different variables, and there exist constraints that these variables must have the same value. Constraints are also distributed among agents. The fact that an agent l knows a constraint predicate p_k is represented as $known(p_k, l)$.

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

- $\forall\, i$, $\forall x_j$ where belongs(x_j, i), the value of x_j is assigned to d_j, and $\forall\, l$, $\forall p_k$ where known(p_k, l), p_k is true under the assignment $x_j = d_j$.

For example, if we assume there exists an agent that corresponds to each queen in Fig. 1.9, and these queens try to find their positions so that they do not kill each other, this problem can be formalized as a distributed CSP. We call this problem the distributed 4-queens problem (Fig. 2.1).

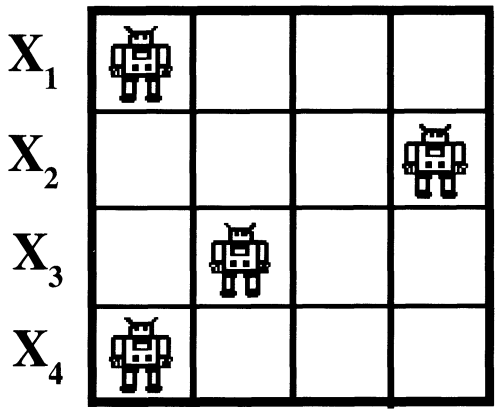

Fig. 2.1. Distributed 4-Queens Problem

It must be noted that although algorithms for solving distributed CSPs seem to be similar to parallel/distributed processing methods for solving CSPs (Collin, Dechter, and Katz 1991; Zhang and Mackworth 1991), research motivations are fundamentally different. The primary concern in parallel/distributed processing is efficiency, and we can choose any type of parallel/distributed computer architecture for solving a given problem efficiently. In contrast, in a distributed CSP, there already exists a situation where

knowledge about the problem (i.e., variables and constraints) is distributed among automated agents. 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 to be solved. Therefore, the main research issue is how to reach a solution from this given situation.

If all knowledge about the problem can be gathered into one agent, this agent could solve the problem alone using normal centralized constraint satisfaction algorithms. However, collecting all information about a problem requires not only communication costs but also the costs of translating one's knowledge into an exchangeable format. For example, a constraint might be stored as a very complicated specialized internal function within an agent. In order to communicate the knowledge of this constraint to another agent, which might be implemented on different computer architecture, the agent would have to translate the knowledge into an exchangeable format, such as a table of allowed (or not allowed) combinations of variable values. These costs of centralizing all information to one agent could be prohibitively high.

Furthermore, in some application problems, such as software agents in which each agent acts as a secretary of an individual, gathering all information to one agent is undesirable or impossible for security/privacy reasons. In such cases, multiple agents have to solve the problem without centralizing all information.

2.3 Application Problems

In this section, we show how various application problems in MAS can be mapped into distributed CSPs.

2.3.1 Recognition Problem

A recognition problem can be viewed as a problem in finding a compatible set of hypotheses that corresponds to the possible interpretations of input data. A recognition problem can be mapped into a CSP by viewing possible interpretations as possible variable values. Several examples are described below. For example, as described in Chapter 1, a scene labeling problem (Waltz 1975) can be formalized as a CSP. Therefore, a problem for finding the compatible interpretations of agents, each of which is assigned a different part of a scene, can be formalized as a distributed CSP (Fig. 2.2).

In (Mason and Johnson 1989), a distributed assumption-based truth maintenance system (distributed ATMS) is used for solving recognition problems. In this framework, each agent finds a compatible combination of the interpretations of its input sensor data. Using ATMS-based consistency algo-

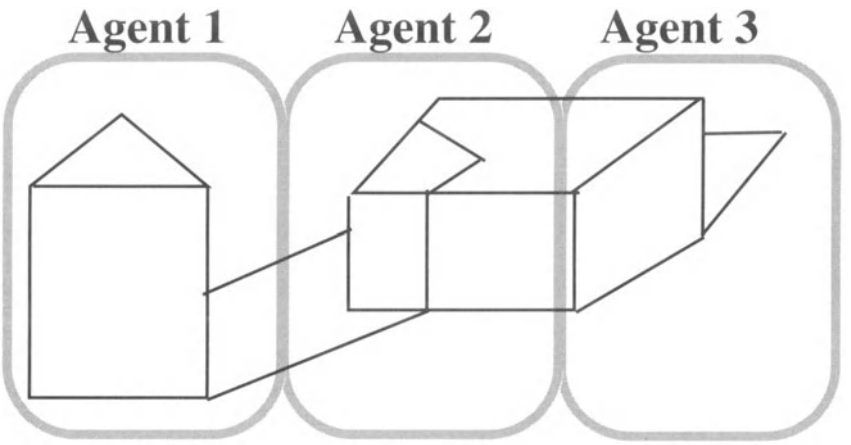

Fig. 2.2. Example of Distributed Recognition Problem (Scene Labeling)

rithms described in Chapter 6, each agent can eliminate the interpretations that are not compatible with the interpretations of other agents [1].

The Distributed Vehicle Monitoring Testbed (DVMT) (Lesser 1991) utilizes a method called functionally accurate/cooperative (FA/C), where each agent solves a sub-problem and eliminates the possibilities by exchanging the intermediate results (result sharing), and finally reaches the mutually consistent solution. The method for solving a distributed CSP, where each agent first finds possible solutions to its sub-problem, exchanges these solutions, and eliminates the possibilities by consistency algorithms, can be regarded as a kind of FA/C.

2.3.2 Allocation Problem

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 (Conry, Kuwabara, Lesser, and Meyer 1991) 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 that enables all tasks to be executed simultaneously. In (Conry, Kuwabara, Lesser, and Meyer 1991), a problem formalization for a class of allocation problems using goals, plans, and plan fragments is described. This class of problems is a subset of distributed CSPs.

[1] In the application problem of (Mason and Johnson 1989), however, although it is preferable that the interpretations of different agents are compatible, these interpretations are possibly incompatible if the agents have different opinions.

We show an example problem of a communication network used in the multi-stage negotiation protocol in Fig. 2.3. 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. Table 2.1 shows the plan fragments for the goal of connecting A-1 and D-1 (goal-1), and that for the goal of connecting A-2 and E-1 (goal-2). Also, Table 2.2 shows the global plans that are composed of these plan fragments.

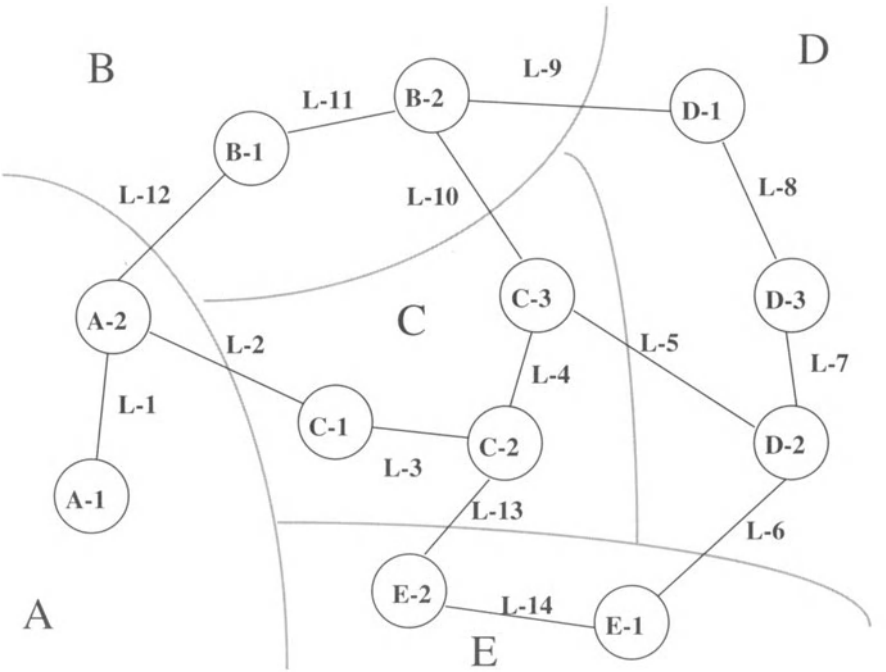

Fig. 2.3. Example of Distributed Resource Allocation Problem

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. The variables and values of agents are described in Table 2.3. Each agent has one variable that corresponds to one goal, and the values of a variable corresponds to plan fragments. Since an agent does

Table 2.1. Plan Fragments

agent	goal	plan fragment	resource used
A	goal-1	1A	L-1, L-2
		2A	L-1, L-12
	goal-2	3A	L-12
B	goal-1	1B	L-10, L-11, L-12
	goal-2	2B	L-10, L-11, L-12
C	goal-1	1C	L-2, L-3, L-4, L-5
		2C	L-5, L-10
	goal-2	3C	L-5, L-10
		4C	L-4, L-10, L-13
D	goal-1	1D	L-5, L-7, L-8
	goal-2	2D	L-5, L-6
E	goal-2	1E	L-6
		2E	L-13, L-14

Table 2.2. Global Plans

goal	plan	plan fragments
goal-1	plan-11	1A, 1C, 1D
	plan-12	2A, 1B, 2C, 1D
goal-2	plan-21	3A, 2B, 3C, 2D, 1E
	plan-22	3A, 2B, 4C, 2E

not have to contribute to all goals, a variable has a value 'idle' that means that the agent does not contribute to the goal.

Table 2.3. Variables and Domains of Agents

agent	variable	value
A	A-goal1	1A, 2A
	A-goal2	3A
B	B-goal1	1B, idle
	B-goal2	2B, idle
C	C-goal1	1C, 2C, idle
	C-goal2	3C, 4C, idle
D	D-goal1	1D
	D-goal2	2D, idle
E	E-goal2	1E, 2E

If we assume one communication link can handle only one communication channel, plan fragments that use the same communication link cannot be achieved simultaneously. For example, 1B and 2B cannot be executed simultaneously, i.e., {(B-goal1, 1B), (B-goal2, 2B)} is a nogood. Also, if agent A executes plan fragment 3A, agent B must executes plan fragment 2B. Therefore, {(A-goal2, 3A), (B-goal2, idle)} is a nogood.

2.3.3 Multi-agent Truth Maintenance

A multi-agent truth maintenance system (Huhns and Bridgeland 1991) is a distributed version of a truth maintenance system (Doyle 1979). In this system, there exist multiple agents, each of which has its own truth maintenance system (Fig. 2.4). Each agent has an uncertain fact that can be IN or OUT, i.e., believed or not believed, and each shares some facts with other agents. Each agent must consistently determine the label of its uncertain facts according to the dependencies among facts. Also, a shared fact must have the same label. The multi-agent truth maintenance task can be formalized as a distributed CSP, where a fact is a variable whose value can be IN or OUT, and dependencies are constraints. For example, in Fig. 2.4, 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.

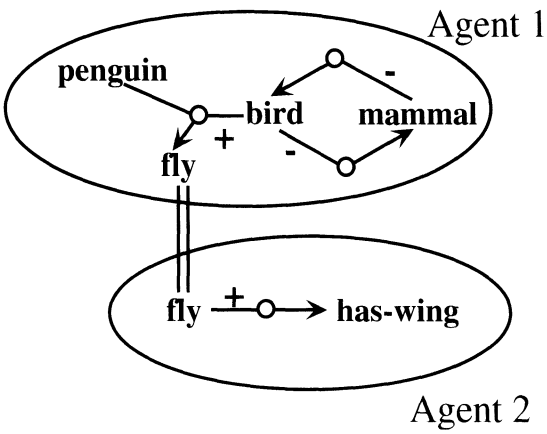

Fig. 2.4. Multi-Agent Truth Maintenance System

2.3.4 Time-Tabling/Scheduling Tasks

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 (Solotorevsky and Gudes 1996) involves assigning nurses to shifts in each department of a hospital. Since the time-table of each department is basically independent, it can be handled by different agents. However, there exist inter-agent constraints involving the transportation of nurses. In this work, a real-life problem with ten departments, 20 nurses in each department, and 100 weekly assignments, was solved using distributed CSP techniques.

Table 2.4. Algorithms for Solving Distributed CSPs

	single	multiple	partial
backtracking	asynchronous backtracking (Chapter 3)		asynchronous incremental relaxation (Chapter 8)
Iterative improvement	distributed breakout (Chapter 5)		incremental distributed breakout (Chapter 8)
hybrid	asynchronous weak-commitment (Chapter 4)	AWC+AP multi-AWC (Chapter 7)	
consistency algorithm	distributed consistency algorithm (Chapter 6)		

Also, in a job shop scheduling problem (Fox 1987; Fox, Sadeh, and Baykan 1989), a set of jobs has to be performed on a set of machines. Each job is composed of a set of sequential operations, called subtasks. For each subtask, start time must be determined so that various constraints are satisfied. In (Liu and Sycara 1996; Sycara, Roth, Sadeh, and Fox 1991), an agent is assigned to each machine, and these agents cooperatively satisfy given constraints.

2.4 Classification of Algorithms for Solving Distributed CSPs

In the rest of this book, we are going to describe series of algorithms for solving distributed CSPs. We show the classification of these algorithms in Table 2.4. These algorithms are classified using two dimensions, i.e., the basic algorithm that is based on (backtracking, iterative improvement, hybrid, and consistency algorithms), and the type of the problems it mainly deals with (distributed CSPs with a single local variable, multiple local variables, and distributed partial CSPs).

2.5 Summary

In this chapter, we presented the formalization of distributed CSPs and described how various application problems in MAS can be mapped into this formalization. Although these problems have been studied independently by different researchers, they can be mapped into a unified framework. Therefore, the various algorithms for solving distributed CSPs described in this book can be applied to these application problems.

3. Asynchronous Backtracking

3.1 Introduction

In this chapter, we describe the basic backtracking algorithm for solving distributed CSPs called the *asynchronous backtracking* algorithm. In this algorithm, agents can act concurrently and asynchronously without any global control, while the completeness of the algorithm is guaranteed.

We first show several assumptions for simplicity (Section 3.2) and two trivial algorithms for solving distributed CSPs (Section 3.3). Then, we describe the asynchronous backtracking algorithm in detail (Section 3.4), including an example of the algorithm execution (Section 3.4.3) and the proof that this algorithm is sound and complete (Section 3.4.4). Finally, we present empirical results that compare the efficiency of the presented algorithms (Section 3.5).

3.2 Assumptions

In most chapters of this book, we make the following assumptions for simplicity.

- Each agent has exactly one variable.
- All constraints are binary.
- Each agent knows all constraint predicates relevant to its variable.

Relaxing the second and the third assumptions to general cases is relatively straightforward. Actually, in the asynchronous backtracking algorithm described in this chapter, even if all originally given constraints are binary, newly derived constraints (nogoods) can be among more than two variables, and the algorithm can handle such non-binary constraints. We show how to deal with the case where an agent has multiple local variables in Chapter 7. 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, *neighbors* of x_i.

We represent a distributed CSP in which all constraints are binary as a network, where variables are nodes and constraints are links between nodes.

M. Yokoo, *Distributed Constraint Satisfaction*

It must be emphasized that this constraint network has nothing to do with the physical communication network. The link in the constraint network is not a physical communication link but a logical relation between agents. Since each agent has exactly one variable, a node also represents an agent. We use the same identifier for representing an agent and its variable. For example, in Fig. 3.1 there are three agents, x_1, x_2, x_3, with variable domains $\{1,2\}, \{2\}, \{1,2\}$, respectively, and constraints $x_1 \neq x_3$ and $x_2 \neq x_3$.

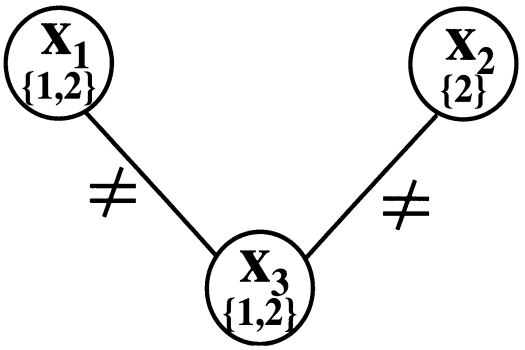

Fig. 3.1. Example of Constraint Network

3.3 Simple Algorithms

We can consider two simple algorithms for solving distributed CSPs. These algorithms are straightforward extensions of algorithms for solving normal, centralized CSPs.

3.3.1 Centralized Method

The most trivial algorithm for solving a distributed CSP selects a leader agent among all agents, and gathers all the information about the variables, their domains, and their constraints into the leader agent. The leader then solves the CSP alone by normal centralized constraint satisfaction algorithms. However, as discussed in Chapter 2, the cost of collecting all the information about a problem can be prohibitively high. Furthermore, in some application problems, such as software agents in which each agent acts as a secretary of an individual, gathering all the information to one agent is undesirable or impossible for security/privacy reasons.

In the asynchronous backtracking algorithm (Chapter 3), the asynchronous weak-commitment search algorithm (Chapter 4), and the distributed break-out algorithm (Chapter 5), agents communicate current value assignments

and nogoods. By observing the value assignments of agent x_i, other agents can gradually accumulate the information about the domain of x_i. However, other agents cannot tell whether the obtained information of x_i's domain is complete or not. There might be other values of x_i, which are not selected because they violate some constraints with higher priority agents. Furthermore, agent x_i never directly reveals information about its constraints. A nogood message sent from x_i is a highly summarized piece of information about its constraints and nogoods sent from other agents. Therefore, we can see that the amount of information revealed by these algorithms is much smaller than that of the centralized methods, where agents must declare precise information about their variable domains and constraints.

3.3.2 Synchronous Backtracking

The standard backtracking algorithm for CSPs can be modified to yield the *synchronous backtracking* algorithm for distributed CSPs. Assume the agents agree on an instantiation order for their variables (such as agent x_1 goes first, then agent x_2, and so on). Each agent, receiving a partial solution (the instantiations of the preceding variables) from the previous agent, instantiates its variable based on the constraints that it knows. If it finds such a value, it appends this to the partial solution and passes it on to the next agent. If no instantiation of its variable can satisfy the constraints, then it sends a *backtracking* message to the previous agent (Fig. 3.2).

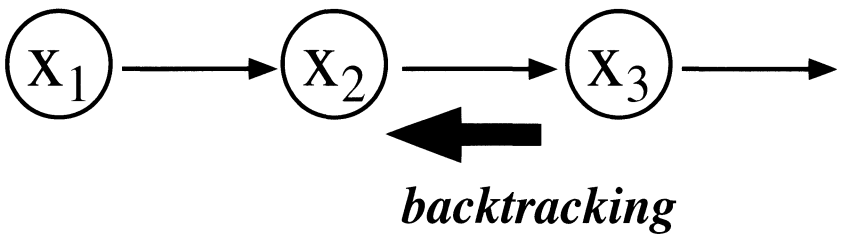

Fig. 3.2. Example of Algorithm Execution (Synchronous Backtracking)

While this algorithm does not suffer from the same communication overhead as the centralized method, determining the instantiation order still requires certain communication costs. Furthermore, this algorithm cannot take advantage of parallelism, because at any given time only one agent is receiving the partial solution and acting on it, and so the problem is solved sequentially. In (Collin, Dechter, and Katz 1991), a variation of the synchronous backtracking algorithm called the *Network Consistency Protocol* is presented. In this algorithm, agents construct a depth-first search tree. Agents act syn-

chronously by passing *privilege*, but the agents that have the same parent in the search tree can act concurrently.

3.4 Asynchronous Backtracking Algorithm

3.4.1 Overview

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

We assume that every link (constraint) is *directed*. In other words, one of the two agents involved in a constraint is assigned that constraint, and receives the other agent's value. A link is directed from the value-sending agent to the constraint-evaluating agent (Fig. 3.3).

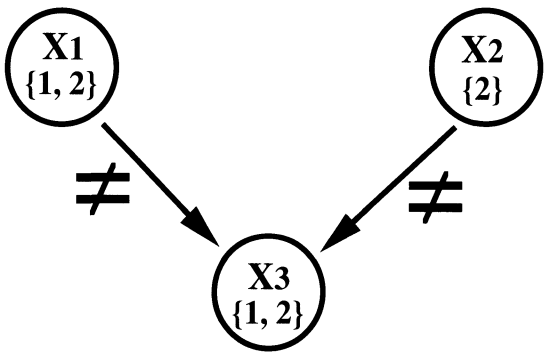

Fig. 3.3. Example of Constraint Network with Directed Links

Each agent instantiates its variable concurrently and sends the value to the agents that are connected to it by outgoing links. After that, the agents wait for and respond to messages. Figure 3.4 describes procedures executed by agent x_i for receiving two kinds of messages. One kind is an *ok?* message, which a constraint-evaluating agent receives from a value-sending agent asking whether the value chosen is acceptable (Fig. 3.4 (i)). The second kind is a *nogood* message, which a value-sending agent receives indicating that the constraint-evaluating agent has found a constraint violation (Fig. 3.4 (ii)). Although the following algorithm is described in such a way that an agent reacts to messages sequentially, an agent can in fact handle multiple messages concurrently, i.e., the agent first revises the *agent_view* and *nogood_list* according to the messages, and performs **check_agent_view** only once.

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

when received (nogood, x_j, nogood) do — (ii)
  add nogood to nogood_list;
  when (x_k,d_k) where x_k is not connected is contained in nogood do
    request x_k to add a link from x_k to x_i;
    add (x_k, d_k) to agent_view; end do;
  old_value ← current_value; check_agent_view; — (ii-a)
  when old_value = current_value do
      send (ok?, (x_j, current_value)) to x_j; end do; end do;

procedure check_agent_view
  when agent_view and current_value are not consistent do
    if no value in D_i is consistent with agent_view then backtrack; — (iii)
    else select d ∈ D_i where agent_view and d are consistent;
      current_value ← d;
      send (ok?, (x_i, d)) to outgoing links; end if; end do;

procedure backtrack
  nogoods ← {V | V= inconsistent subset of agent_view}; — (iii-a)
  when an empty set is an element of nogoods do
      broadcast to other agents that there is no solution,
          terminate this algorithm; end do;
  for each V ∈ nogoods do;
      select (x_j, d_j) where x_j has the lowest priority in V; — (iii-b)
      send (nogood, x_i, V) to x_j;
    remove (x_j, d_j) from agent_view; end do;
  check_agent_view;
```

Fig. 3.4. Procedures for Receiving Messages (Asynchronous Backtracking)

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

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

with the *agent_view*, agent x_i tries to change the *current_value* so that it will be consistent with the *agent_view*.

A subset of an *agent_view* is called a *nogood* if the agent is not able to find any consistent value with the subset. For example, in Fig. 3.5 (a), if agents x_1 and x_2 instantiate their variables to 1 and 2, the *agent_view* of x_3 will be $\{(x_1, 1), (x_2, 2)\}$. Since there is no possible value for x_3 that is consistent with this *agent_view*, this *agent_view* is a nogood. If an agent finds that a subset of its *agent_view* is a nogood, the assignments of other agents must be changed. Therefore, the agent causes a *backtrack* (Fig. 3.4 (iii)) and sends a *nogood* message to one of the other agents. Ideally, the nogoods detected in Fig. 3.4 (iii-a) should be *minimal*, i.e., no subset of them should be a nogood. However, since finding all minimal nogoods requires certain computation costs, an agent can make do with non-minimal nogoods. In the simplest case, it can use the whole *agent_view* as a nogood.

3.4.2 Characteristics of the Asynchronous Backtracking Algorithm

Avoiding Infinite Processing Loops. If agents change their values again and again and never reach a stable state, they are in an infinite processing loop. An infinite processing loop can occur if there exists a value changing loop of agents, such as if a change in x_1 causes x_2 to change, then this change in x_2 causes x_3 to change, which then causes x_1 to change again, and so on. In the network representation, such a loop is represented by a cycle of directed links (Fig. 3.6 (a)).

One way to avoid cycles in a network is to use a total order relationship among nodes. If each node has a unique identifier, we can define a priority order among agents by using the alphabetical order of these identifiers (the preceding agent in the alphabetical order has higher priority). If a link is directed by using this priority order (from the higher priority agent to the lower priority agent), there will be no cycle in the network (Fig. 3.6 (b)). This means that for each constraint, the lower priority agent will be an evaluator, and the higher priority agent will send an *ok?* message to the evaluator. Furthermore, if a nogood is found, a *nogood* message is sent to the lowest priority agent in the nogood (Fig. 3.4 (iii-b)), the lowest priority agent within the nogood becomes an evaluator, and links are added between each of the non-evaluator agents in the nogood and the evaluator. Similar techniques to this unique identifier method are used for avoiding deadlock in distributed database systems (Rosenkrantz, Stearns, and Lewis 1978).

The knowledge each agent requires for this unique identifier method is much more local than that needed for the synchronous backtracking algorithm. In the synchronous backtracking algorithm, agents must act in a predefined sequential order. Such a sequential order cannot be easily obtained just by simply giving a unique identifier to each agent. Each agent must know the previous and next agent, which means polling all of the other agents to

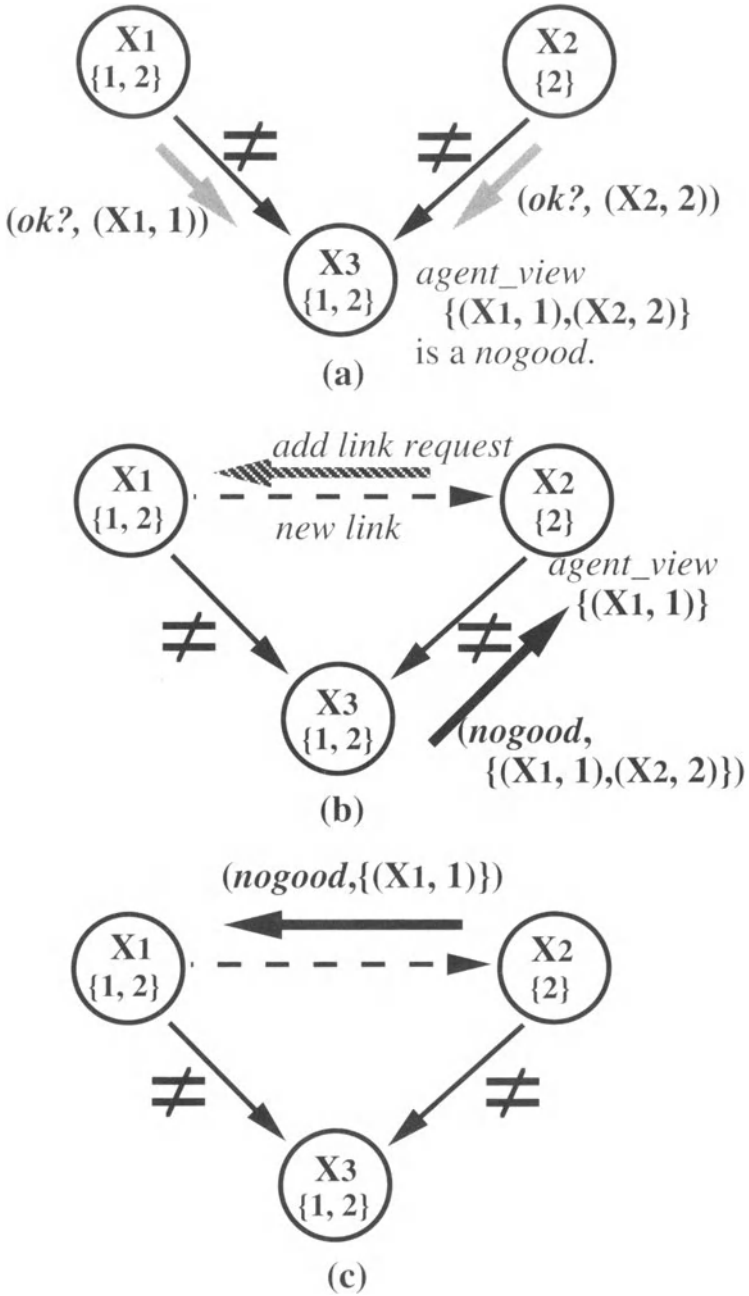

Fig. 3.5. Example of Algorithm Execution (Asynchronous Backtracking)

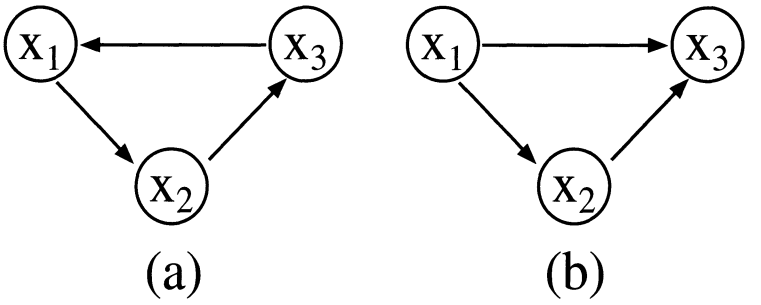

Fig. 3.6. Determining Variable Priority Using Identifiers

find the closest identifiers above and below it. On the other hand, in the unique identifier method for asynchronous backtracking, each agent has to know only the identifiers of an agent with which it must establish a constraint in order to direct the constraint. As for CSPs, the order of the variables greatly affects the search efficiency. Further research is needed in order to introduce variable ordering heuristics into asynchronous backtracking.

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

We introduce the use of *context attachment* to deal with these potential inconsistencies. In context attachment, an agent couples its message with the nogood that triggered it. This nogood is the context of backtracking. After receiving this message, the recipient only changes its value if the nogood is *compatible* with its current *agent_view* and its own assignment (Fig. 3.4 (ii-a)). Since the nogood attached to a *nogood* message indicates the cause of the failure, asynchronous backtracking includes the function of dependency-directed backtracking in CSPs described in Section 1.3.1.

A nogood can be viewed as a new constraint derived from the original constraints. By incorporating such a new constraint, agents can avoid repeating the same mistake. For example, in Fig. 3.5 (b), the nogood $\{(x_1, 1), (x_2, 2)\}$ represents a constraint between x_1 and x_2. Since there is originally no link between x_1 and x_2, in order to incorporate this new constraint, a new link must be added between them. Since a link in the constraint network represents a logical relation between agents, adding a link does not mean adding a new physical communication path between agents. Therefore, after receiving the nogood message, agent x_2 asks x_1 to add a link between them. In general, even if all the original constraints are binary, newly derived constraints can

be among more than two variables. In such a case, the agent that has the lowest priority in the constraint will be an evaluator, and links will be added between each of the non-evaluator agents and the evaluator. We will discuss the worst-case space complexity of this algorithm in Section 3.4.4.

3.4.3 Example of Algorithm Execution

In Fig. 3.5 (a), by receiving *ok?* messages from x_1 and x_2, the *agent_view* of x_3 will be $\{(x_1, 1), (x_2, 2)\}$. Since there is no possible value for x_3 consistent with this *agent_view*, this *agent_view* is a nogood. Agent x_3 chooses the lowest priority agent in the *agent_view*, i.e., agent x_2, and sends a *nogood* message with the nogood. By receiving this *nogood* message, agent x_2 records this nogood. This nogood, $\{(x_1, 1), (x_2, 2)\}$ contains agent x_1, which is not connected to x_2 by a link. Therefore, a new link must be added between x_1 and x_2. Agent x_2 requests x_1 to send x_1's value to x_2, and adds $(x_1, 1)$ to its *agent_view* (Fig. 3.5 (b)). Agent x_2 checks whether its value is consistent with the *agent_view*. Since the nogood received from agent x_3 is compatible with its assignment $(x_2, 2)$ and its *agent_view* $\{(x_1, 1)\}$, the assignment $(x_2, 2)$ is inconsistent with the *agent_view*. The *agent_view* $\{(x_1, 1)\}$ is a nogood because x_2 has no other possible values. There is only one agent in this nogood, i.e., agent x_1, so agent x_2 sends a *nogood* message to agent x_1 (Fig. 3.5 (c)).

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

The initial values are shown in Fig. 3.7 (a). Agents communicate these values to each other. The priority order is determined by the alphabetical order of identifiers. Each agent except x_1 changes its value so that the new value is consistent with its *agent_view* (Fig. 3.7 (b)), i.e., agent x_2 changes its value to 3, which is consistent with x_1's value; agent x_3 changes its value to 4, which is consistent with x_1 and x_2's value (since x_2 changes its value, x_3's value is no longer consistent with the new value). Since there is no consistent value for agent x_4, it sends a *nogood* message to x_3, and changes its value so that the value is consistent with its *agent_view*, except for the value of x_3. Note that x_3 will ignore this nogood message since it has changed its value before it receives this message. The agents send *ok?* messages to other agents. Then, x_3 does not satisfy constraints with x_2, and there is no value that is consistent with x_1 and x_2, while other agents' values are consistent with their *agent_view*. Therefore, x_3 sends a nogood message to x_2. After receiving this nogood message, x_2 changes its value to 4 (Fig. 3.7 (c)). Then, x_3 changes its value to 2. There is no consistent value for agent x_4, so it sends a *nogood* message to x_3, and changes its value so that the value is consistent with

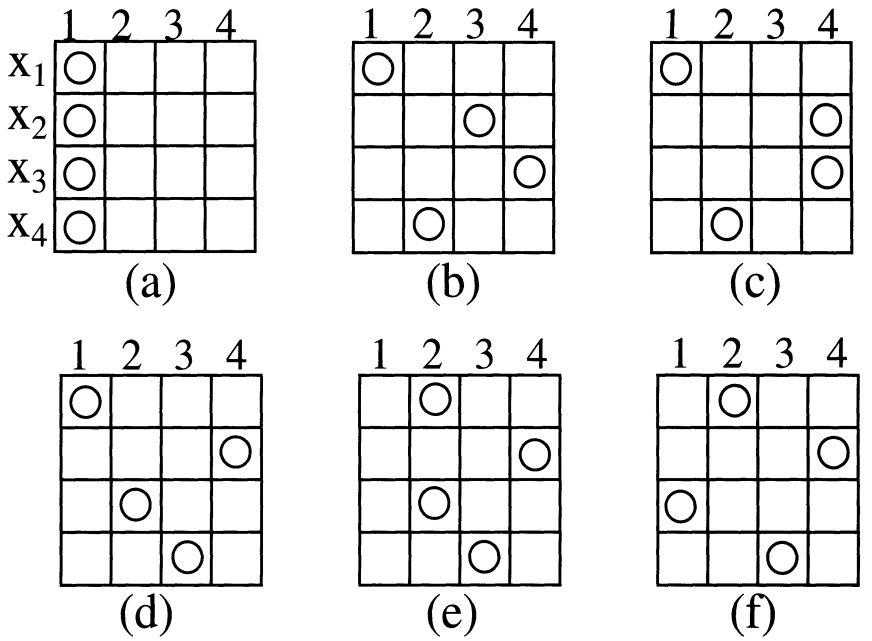

Fig. 3.7. Example of Algorithm Execution (Asynchronous Backtracking, Distributed 4-Queens)

its *agent_view*, except for the value of x_3 (Fig. 3.7 (d)). Again, this nogood message is ignored. There is no consistent value for agent x_4, so it sends a *nogood* message to x_3. After receiving this message, x_3 has no other consistent value, so x_3 sends a nogood message to x_2. After receiving this message, x_2 also has no other consistent value, so x_2 sends a nogood message to x_1. Then, x_1 changes its value to 2 (Fig. 3.7 (e)). Then, x_3 changes its value to 1. There is no consistent value for agent x_4, so it sends a *nogood* message to x_3, and changes its value so that the value is consistent with its *agent_view*, except for the value of x_3. Again, this nogood message is ignored, and a solution is found (Fig. 3.7 (f)).

We should mention that the way to determine that agents as a whole have reached a stable state is not contained in this algorithm. To detect the stable state, distributed termination detection algorithms such as (Chandy and Lamport 1985) are needed.

3.4.4 Algorithm Soundness and Completeness

We will show that if there exists a solution, this algorithm reaches a stable state where all the variable values satisfy all the constraints and all agents are waiting for an incoming message. Also, we will show that if no solution

exists, this algorithm discovers this fact and terminates. For the agents to reach a stable state, all of their variable values must perforce satisfy all the constraints. Thus, the soundness of the algorithm is clear. Furthermore, the algorithm is complete, in that it finds a solution if one exists and terminates with failure when there is no solution.

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

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

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

Now, assume that agents x_1 to x_{k-1} ($k > 2$) are in a stable state, and agent x_k is in an infinite processing loop. In this case, the only messages agent x_k receives are *nogood* messages from agents whose priorities are lower than k, and these *nogood* messages contain only the agents x_1 to x_k. Since agents x_1 to x_{k-1} are in a stable state, the *nogoods* agent x_k receives must be compatible with its *agent_view*, and so x_k will change the instantiation of its variable with a different value. Because its variable's domain is finite, x_k will either eventually generate a value that does not cause it to receive a nogood (which contradicts the assumption that x_k is in an infinite loop) or exhaust the possible values and send a nogood to one of $x_1 \ldots x_{k-1}$. However, this nogood would cause an agent that we assumed to be in a stable state, not to be in a stable state. Thus, by contradiction, x_k cannot be in an infinite processing loop.

Since constraint satisfaction is NP-complete in general, the worst-case time complexity of the asynchronous backtracking algorithm becomes exponential in the number of variables n. The worst-case space complexity of the algorithm is determined by the number of recorded nogoods. In the asyn-

chronous backtracking algorithm, an agent can forget old nogoods after it creates a new nogood from them. Also, an agent does not need to keep the nogoods that are not compatible with the *agent_view*. Therefore, each agent x_i needs to record at most $|D_i|$ nogoods, where $|D_i|$ is the number of possible values of x_i.

3.5 Evaluations

In this book, 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 drawback of this model is that it does not take into account the costs of communication. However, introducing communication costs into the model is difficult since we don't have any standard way to compare communication costs with computation costs. One cycle corresponds to a series of agent actions in which an agent recognizes the state of the world (the value assignments of other agents), then decides its response to that state (its own value assignment), and communicates its decisions.

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

To verify these expectations, we performed experimental evaluations on the distributed n-queens problem explained in the previous chapter. Each agent corresponds to each queen in a row. Therefore, the distributed n-queens problem is solved by n agents. In the distributed n-queens problem, constraints among agents become weak as n increases. The results are summarized in the graph shown in Fig. 3.8. To make the comparisons fair, we included dependency-directed backtracking in the synchronous backtracking. Each agent randomly selects a value among the consistent values with higher priority agents. The graph shows the average of 100 trials. In this evaluation,

we did not include the cost of determining the sequential order in the synchronous backtracking algorithm, nor the cost of the termination detection in the asynchronous backtracking algorithm.

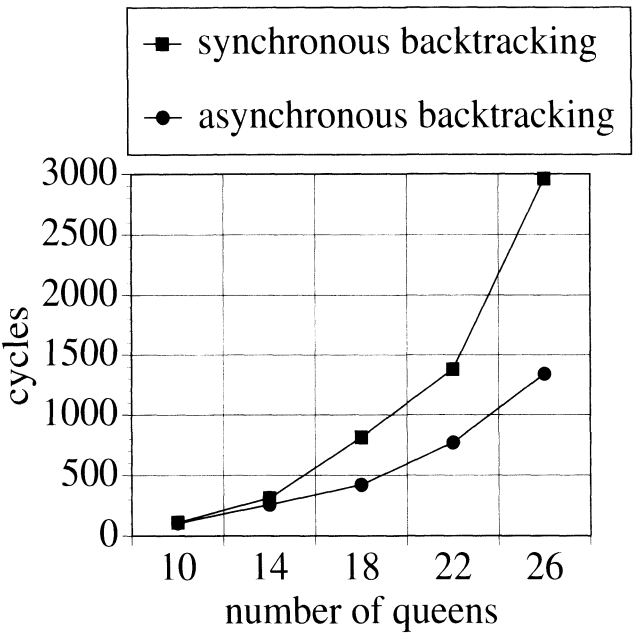

Fig. 3.8. Comparison between Synchronous and Asynchronous Backtracking (Distributed n-Queens)

In the distributed n-queens problem, there exist constraints between any pair of agents. Therefore, the synchronous backtracking algorithm is basically equivalent to the Network Consistency Protocol described in (Collin, Dechter, and Katz 1991). As we expected, the obtained parallelism of the asynchronous backtracking algorithm becomes larger as n increases. When $n > 18$, the asynchronous backtracking algorithm is approximately twice as fast as the synchronous backtracking algorithm[2].

Traditionally, distributed artificial intelligence applications involve having agents work on nearly-independent, loosely-coupled sub-problems (Durfee, Lesser, and Corkill 1992). These results confirm that, if the local subproblems are loosely-coupled, solving the problem asynchronously by multiple agents is worthwhile.

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

3.6 Summary

In this chapter, we introduced a basic backtracking algorithm for solving distributed CSPs called asynchronous backtracking. In this algorithm, agents can act concurrently and asynchronously without any global control. This algorithm is guaranteed to be complete. Experimental evaluations showed that this algorithm is more efficient than the synchronous backtracking algorithm because agents can act concurrently in the asynchronous backtracking algorithm.

4. Asynchronous Weak-Commitment Search

4.1 Introduction

In this chapter, we describe how the asynchronous backtracking algorithm can be modified into a more efficient algorithm called the *asynchronous weak-commitment search* algorithm, which is inspired by the weak-commitment search algorithm for solving CSPs described in Chapter 1. The main characteristic of this algorithm is as follows.

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

In the asynchronous backtracking algorithm, the priority order of agents is determined, and each agent tries to find a value satisfying the constraints with the variables of higher priority agents. When an agent sets a variable value, the agent is strongly committed to the selected value, i.e., the selected value will not be changed unless an exhaustive search is performed by lower priority agents. Therefore, in large-scale problems, a single mistake in the selection of values becomes fatal since such an exhaustive search is virtually impossible. This drawback is common to all backtracking algorithms.

In the asynchronous weak-commitment search algorithm, when an agent cannot find a value consistent with the higher priority agents, the priority order is changed so that the agent has the highest priority. As a result, when an agent makes a mistake in selecting a value, the priority of another agent becomes higher; thus the agent that made the mistake will not commit to the bad decision, and the selected value is changed.

We first show the basic ideas of the asynchronous weak-commitment search algorithm (Section 4.2). Then, we describe the algorithm in detail (Section 4.3) and show an example of algorithm execution by using the distributed 4-queens problem (Section 4.4). Furthermore, we discuss the algorithm complexity (Section 4.5), and show the evaluation results to illustrate the efficiency of the asynchronous weak-commitment search algorithm (Section 4.6).

M. Yokoo, *Distributed Constraint Satisfaction*

4.2 Basic Ideas

The main characteristics of the weak-commitment search algorithm described in Chapter 1 are as follows.

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

Introducing the first characteristic into the asynchronous backtracking algorithm is relatively straightforward. When selecting a variable value, if there exist multiple values consistent with the *agent_view* (those that satisfy the constraints with variables of higher priority agents), the agent prefers the value that minimizes the number of constraint violations with variables of lower priority agents. In contrast, introducing the second characteristic into the asynchronous backtracking is not straightforward, since agents act concurrently and asynchronously, and no agent has exact information on the partial solution. Furthermore, multiple agents may try to restart the search process simultaneously.

In the following, we show that a distributed constraint satisfaction algorithm that weakly commits to the partial solution can be constructed by dynamically changing the priority order. We define the method of establishing the priority order by introducing *priority values*, which are changed by the following rules.

- For each variable/agent, a non-negative integer value representing the priority order of the variable/agent is defined. We call this value the *priority value*.
- The order is defined such that any variable/agent with a larger priority value has higher priority.
- If the priority values of multiple agents are the same, the order is determined by the alphabetical order of the identifiers.
- For each variable/agent, the initial priority value is 0.
- If there exists no consistent value for x_i, the priority value of x_i is changed to $k + 1$, where k is the largest priority value of related agents.

It must be noted that the asynchronous weak-commitment search algorithm is fundamentally different from backtracking algorithms with dynamic variable ordering (e.g., dynamic backtracking (Ginsberg 1993), dependency-directed backtracking (Mackworth 1992)). In backtracking algorithms, a partial solution is never modified unless it is sure that the partial solution cannot be a part of any complete solution (dynamic backtracking and dependency-backtracking are ways of finding out the true cause of the failure/backtracking). In the asynchronous weak-commitment search algorithm,

a partial solution is not modified but completely abandoned after one failure/backtracking.

By following these rules, when a backtracking occurs, the priority order will be changed so that the agent that had highest priority before backtracking has to satisfy the constraints with the agent that causes backtracking and now has larger priority value.

Furthermore, in the asynchronous backtracking algorithm, agents try to avoid situations previously found to be nogoods. However, due to the delay of messages, an *agent_view* of an agent can occasionally be superset of a previously found nogood. In order to avoid reacting to such unstable situations and performing unnecessary changes of priority values, each agent performs the following procedure.

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

4.3 Details of Algorithm

In the asynchronous weak-commitment search algorithm, each agent assigns a value to its variable, and sends the variable value to other agents. After that, agents wait for and respond to incoming messages[1]. In Fig. 4.1, the procedures executed at agent x_i by receiving an *ok?* message and a *nogood* message are described[2].

The differences between these procedures and the procedures for the asynchronous backtracking algorithm are as follows.

- In the asynchronous backtracking algorithm, each agent sends its variable value only to related lower priority agents, while in the asynchronous weak-commitment search algorithm, each agent sends its variable value to both lower and higher priority agents connected by constraints. We call these related agents *neighbors*.
- The priority value, as well as the current value assignment, is communicated through the *ok?* message (Fig. 4.1 (i)).
- The priority order is determined using the communicated priority values. If the current value is not *consistent* with the *agent_view*, i.e., some constraint with variables of higher priority agents is not satisfied, the agent changes its value so that the value is consistent with the *agent_view*, and also the

[1] As in the asynchronous backtracking algorithm, although the following algorithm is described in a way that an agent reacts to messages sequentially, an agent can handle multiple messages concurrently, i.e., the agent first revises the *agent_view* and *nogood_list* according to the messages and performs **check_agent_view** only once.

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

value minimizes the number of constraint violations with variables of lower priority agents (Fig. 4.1 (ii)).
- When x_i cannot find a consistent value with its *agent_view*, x_i sends *nogood* messages to other agents, and increments its priority value. If x_i has already sent an identical nogood, x_i will not change its priority value but will wait for the next message (Fig. 4.1 (iii)).

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

when received (**nogood**, x_j, *nogood*) **do**
  add *nogood* to *nogood_list*;
  when (x_k,d_k,*priority*) where x_k is not in *neighbors*
      is contained in *nogood* **do**
    add x_k to *neighbors*, add (x_k,d_k,*priority*) to *agent_view*; **end do**;
  check_agent_view; **end do**;

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

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

Fig. 4.1. Procedures for Receiving Messages (Asynchronous Weak-Commitment Search)

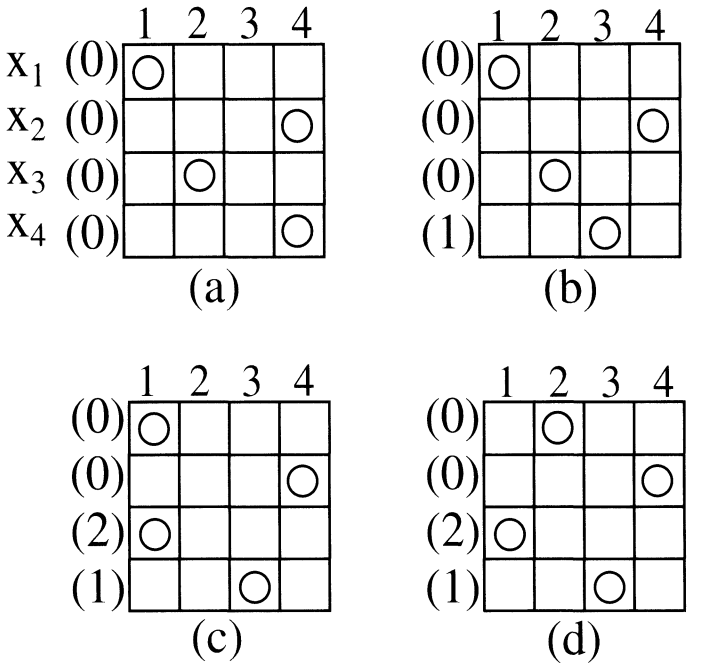

Fig. 4.2. Example of Algorithm Execution (Asynchronous Weak-Commitment Search)

4.4 Example of Algorithm Execution

We will illustrate the execution of the algorithm using the distributed 4-queens problem. The initial values are shown in Fig. 4.2 (a). Agents communicate these values with each other. The values within parentheses represent the priority values, and the initial priority values are 0. Since the priority values are equal, the priority order is determined by the alphabetical order of identifiers. Therefore, only the value of x_4 is not consistent with its *agent_view*. Since there is no consistent value, agent x_4 sends *nogood* messages and increments its priority value. In this case, the value minimizing the number of constraint violations is 3, since it conflicts with x_3 only. Therefore, x_4 selects 3 and sends *ok?* messages to the other agents (Fig. 4.2 (b)).

Then, x_3 tries to change its value. Since there is no consistent value, agent x_3 sends *nogood* messages and increments its priority value. In this case, the value that minimizes the number of constraint violations is 1 or 2. In this example, x_3 selects 1 and sends *ok?* messages to the other agents (Fig. 4.2 (c)). After that, x_1 changes its value to 2, and a solution is obtained (Fig. 4.2 (d)).

In the distributed 4-queens problem, there exists no solution when x_1's value is 1. We can see that the bad decision of x_1 (setting its value to 1) can be

revised without an exhaustive search in the asynchronous weak-commitment search algorithm.

4.5 Algorithm Completeness

The priority values are changed if and only if a new nogood is found. Since the number of possible nogoods is finite[3], the priority values cannot be changed infinitely. Therefore, after a certain time point, the priority values will be stable. We can show that the situations described below will not occur when the priority values are stable.

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

If situation (i) occurs, there exist at least two agents that do not satisfy the constraint between them. Let us assume that the agent ranking k-th in the priority order does not satisfy the constraint with the agent ranking j-th (where $j < k$), and that all the agents ranking higher than k-th satisfy all constraints within them. The only case where the k-th agent waits for incoming messages even though the agent does not satisfy the constraint with the j-th agent is when the k-th agent has sent nogood messages to higher priority agents. This fact contradicts the assumption that higher priority agents satisfy constraints within them. Therefore, situation (i) will not occur. Also, if the priority values are stable, the asynchronous weak-commitment search algorithm is basically identical to the asynchronous backtracking algorithm. Since the asynchronous backtracking algorithm is guaranteed not to fall into an infinite processing loop, situation (ii) will not occur.

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

Since constraint satisfaction is NP-complete in general, the worst-case time complexity of the asynchronous weak-commitment search algorithm becomes exponential in the number of variables n. Furthermore, the worst-case space complexity is exponential in n. This result seems inevitable since this algorithm changes the search order flexibly while guaranteeing its completeness. We can restrict the number of recorded nogoods in the asynchronous weak-commitment search algorithm, i.e., each agent records only a fixed number of the most recently found nogoods. In this case, however, the theoretical completeness cannot be guaranteed (the algorithm may fall into an infinite processing loop in which agents repeatedly find identical nogoods). However, when the number of recorded nogoods is reasonably large, such an infinite

[3] The number of possible nogoods is exponential in the number of variables n.

processing loop rarely occurs. Actually, the asynchronous weak-commitment search algorithm can still find solutions for all example problems when the number of recorded nogoods is restricted to 10.

4.6 Evaluations

We compare the following three kinds of algorithms: (a) asynchronous backtracking, in which a variable value is selected randomly from consistent values, and the priority order is determined by alphabetical order, (b) asynchronous backtracking with *min-conflict* heuristic, in which the *min-conflict* heuristic is introduced, but the priority order is statically determined by alphabetical order, and (c) asynchronous weak-commitment search. The amounts of communication overhead of these algorithms are almost equivalent. The amounts of local computation performed in each cycle for (b) and (c) are equivalent. The amount of local computation for (a) can be smaller because it does not use the *min-conflict heuristic*, but for the lowest priority agent, the amounts of local computation of these algorithms are equivalent.

We evaluate the efficiency of algorithms by a discrete event simulation described in Section 3.5. 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.

We first applied these three algorithms to the distributed n-queens problem described in the previous chapter, varying n from 10 to 1,000. The results are summarized in Table 4.1. Since the the min-conflict heuristic is very effective when n is very large (Minton, Johnston, Philips, and Laird 1992) and problems become very easy, we did not include the results for $n > 1{,}000$. For each n, we generated 100 problems, each of which had different randomly generated initial values, and averaged the results for these problems. For each problem, in order to conduct the experiments within a reasonable amount of time, we set the limit for the number of cycles at 1,000, and terminated the algorithm if this limit was exceeded; in such a case, 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 4.1.

The second example problem is the distributed graph-coloring problem. This is a graph-coloring problem in which each node corresponds to an agent. The graph-coloring problem involves painting nodes in a graph by k different colors so that any two nodes connected by an arc do not have the same color. We randomly generated a problem with n nodes/agents and m arcs by the method described in (Minton, Johnston, Philips, and Laird 1992), so that the graph is connected and the problem has a solution. We evaluated the problem

Table 4.1. Comparison between Asynchronous Backtracking and Asynchronous Weak-Commitment Search (Distributed n-Queens)

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

for $n = 60$, 90, and 120, where $m = n \times 2$ and k=3. This parameter setting corresponds to the "sparse" problems for which (Minton, Johnston, Philips, and Laird 1992) reported poor performance of the min-conflict heuristic. We generated 10 different problems, and for each problem, 10 trials with different initial values were performed (100 trials in all). As in the distributed n-queens problem, the initial values were set randomly. The results are summarized in Table 4.2.

Table 4.2. Comparison between Asynchronous Backtracking and Asynchronous Weak-Commitment Search (Distributed Graph-Coloring Problem)

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

Then, in order to examine the applicability of the asynchronous weak-commitment search algorithm to real-life problems rather than artificial random problems, we applied these algorithms to the distributed resource allocation problem in a communication network described in (Nishibe, Kuwabara, Ishida, and Yokoo 1994). In this problem, there exist requests for allocating circuits between switching nodes of NTT's communication network in Japan (Fig. 4.3). For each request, there exists an agent assigned to handle it, and the candidates for the circuits are given. The goal is to find a set of circuits that satisfies the resource constraints. This problem can be formalized as a distributed CSP by representing each request as a variable and each candidate as a possible value for the variable. We generated problems based on data from a 400 Mbps backbone network extracted from the network config-

uration management database developed at NTT Optical Network Systems Laboratories (Yamaguchi, Fujii, Yamanaka, and Yoda 1989).

In each problem, there exist 10 randomly generated circuit allocation requests, and for each request, 50 candidates are given. These candidates represent reasonably short circuits for satisfying the request. We assume that these candidates are calculated beforehand. The constraints between requests are that they do not assign the same circuits. We generated 10 different sets of randomly generated initial values for 10 different problems (100 trials in all), and averaged the results. As in the previous problems, the limit for the required number of cycles was set at 1,000. The results are summarized in Table 4.3. We can see the following facts from these results.

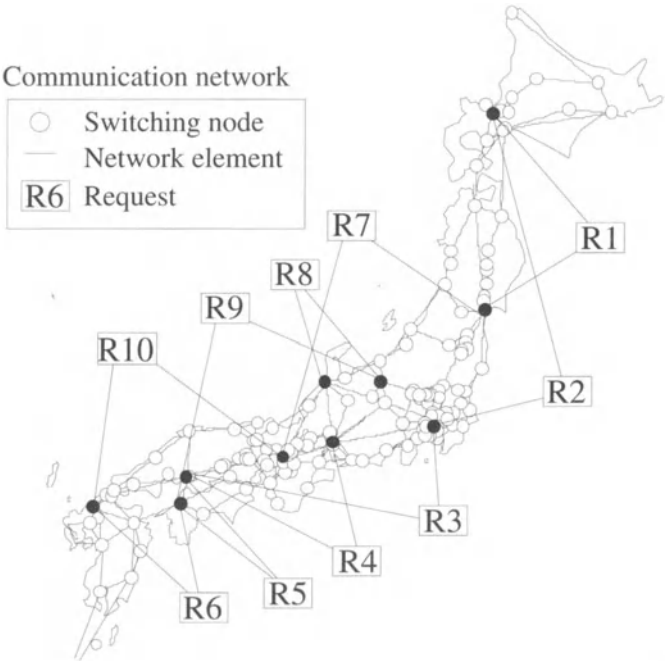

Fig. 4.3. Example of Network Resource Allocation Problem

- The asynchronous weak-commitment search algorithm can solve problems that cannot be solved within a reasonable amount of computation time by asynchronous backtracking algorithms. By using only the *min-conflict* heuristic, although a certain amount of speed-up can be obtained, the algorithm fails to solve many problem instances.
- When the priority order is static, the efficiency of the algorithm is highly dependent on the selection of initial values, and the distribution of required

Table 4.3. Comparison between Asynchronous Backtracking and Asynchronous Weak-Commitment Search (Network Resource Allocation Problem)

Algorithm	Result	
asynchronous backtracking	ratio	32%
	cycles	984.8
asynchronous backtracking with min-conflict heuristic	ratio	63%
	cycles	428.4
asynchronous weak-commitment	ratio	100%
	cycles	17.3

cycles is quite large. For example, in the network resource allocation problem, when only the *min-conflict* heuristic is used, the average number of required cycles for 63 successfully completed trials is only 92.8. However, the number of required cycles for 37 failed trials is more than 1,000. When the initial values of higher priority agents are good, the solution can easily be found. If some of these values are bad, however, an exhaustive search is required to revise these values; this tends to make the number of required cycles exceed the limit. On the other hand, in the asynchronous weak-commitment search algorithm, the initial values are less critical, and a solution can be found even if the initial values are far from the final solution, since the variable values gradually come close to the final solution.

- We can assume that the priority order represents a hierarchy of agent authority, i.e., the priority order of decision making. If this hierarchy is static, the misjudgments (bad value selections) of agents with higher priority are fatal to all agents. On the other hand, by dynamically changing the priority order and cooperatively selecting values, the misjudgments of specific agents do not have fatal effects, since bad decisions are weeded out, and only good decisions survive. These results are intuitively natural, since they imply that a flexible agent organization performs better than a static and rigid organization.

4.7 Summary

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

a flexible agent organization performs better than a static and rigid organization.

5. Distributed Breakout

5.1 Introduction

In this chapter, we describe a distributed iterative improvement algorithm called the *distributed breakout* algorithm, which is inspired by the breakout algorithm described in Chapter 1. The main characteristics of this algorithm are as follows.

- 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.
- Instead of detecting the fact that agents as a whole are trapped in a local-minimum, each agent detects the fact that it is in a quasi-local-minimum, which is a weaker condition than a local-minimum, and changes the weights of constraint violations.

In the remainder of this chapter, we briefly describe the breakout algorithm (Section 5.2). Then, we show the basic ideas (Section 5.3), and details of the distributed breakout algorithm (Section 5.4). Then, we show an example of the algorithm execution (Section 5.5). Furthermore, we show empirical results that illustrate the efficiency of this algorithm (Section 5.6). Finally, we compare characteristics of the distributed breakout algorithm and the asynchronous weak-commitment search algorithm (Section 5.7).

5.2 Breakout Algorithm

As discussed in Section 1.3.2, the breakout algorithm (Morris 1993) is one kind of iterative improvement algorithms (Minton, Johnston, Philips, and Laird 1992; Selman, Levesque, and Mitchell 1992). In these algorithms, a *flawed* solution containing some constraint violations is revised by local changes until all constraints are satisfied. In the breakout algorithm, a weight is defined for each pair of variable values that does not satisfy constraints (the initial weight is 1), and the summation of the weights of constraint violating pairs is used to evaluate the flawed solution. In the initial state, the summation is equal to the number of constraint violations. In the breakout

M. Yokoo, *Distributed Constraint Satisfaction*

algorithm, a variable value is changed to decrease the evaluation value (i.e., the number of constraint violations).

If the evaluation value cannot be decreased by changing the value of any variable, the current state is called a local-minimum. When trapped in a local-minimum, the breakout algorithm increases the weights of constraint violating pairs in the current state by 1 so that the evaluation value of the current state becomes larger than the neighboring states; thus the algorithm can escape from a local-minimum. We show an outline of the breakout algorithm in Fig. 5.1. Although the breakout algorithm is very simple, in (Morris 1993), it is shown that the breakout algorithm outperforms other iterative improvement algorithms (Minton, Johnston, Philips, and Laird 1992; Selman, Levesque, and Mitchell 1992).

```
procedure breakout
    until current-state is solution do
    if current state is not a local-minimum
        then make any change of a variable value that reduces the total cost
        else increase weights of all current nogoods
    end if; end do;
```

Fig. 5.1. Breakout Algorithm

5.3 Basic Ideas

For an agent x_i, we call a set of agents, each of which is directly connected to x_i by a link, *neighbors* of x_i. Also, a *distance* between two agents is defined as the number of links of the shortest path connecting these two agents. For example, Fig. 5.2 shows one instance of a distributed graph-coloring problem, where 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). The neighbors of x_1 are $\{x_2, x_6\}$, and the distance between x_1 and x_4 is 3.

In applying the breakout algorithm to distributed CSPs, we encounter the following difficulties.

- If we allow only one agent to change its value at a time, we cannot take advantage of parallelism. On the other hand, if two neighboring agents are allowed to change their values at the same time, the evaluation value may not be improved, and an oscillation may occur (i.e., the agents continue the same actions repeatedly).
- To detect the fact that agents as a whole are trapped in a local-minimum, the agents have to globally exchange information among themselves.

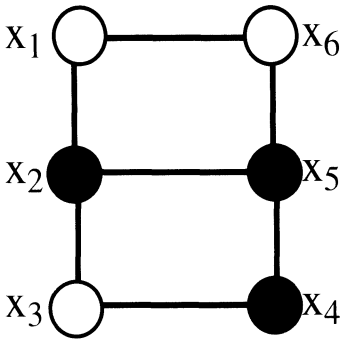

Fig. 5.2. Example of Constraint Network

In order to solve these difficulties, we introduce the following ideas.

- 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, 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 *ok?* message is used to exchange the current value assignment of the agent, and the *improve* message is used to communicate the possible improvement of the evaluation value by a change in the agent's value. By exchanging the *improve* messages among neighbors and giving the right to change its value only to the agent that can maximally improve the evaluation value, the neighbors will not change their values concurrently, while non-neighbors can.

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.

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, in Fig. 5.2, 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.

Increasing the weights in quasi-local-minima that are not real local-minima may adversely affect the performance if the evaluation function is modified too much. We will evaluate the effect of increasing the weights in quasi-local-minima rather than real local-minima in Section 5.6.

5.4 Details of Algorithm

In this algorithm, each agent randomly determines its initial value, and then sends *ok?* messages to its neighbors. After receiving *ok?* messages from all of its neighbors, it calculates its current evaluation value (the summation of the weights of constraint violating pairs related to its variable) and the possible improvement of the evaluation value, and sends *improve* messages to its neighbors.

The procedures executed at agent x_i when receiving *ok?* and *improve* messages are shown in Fig. 5.3 and Fig. 5.4. The agent alternates between the wait_ok? mode (Fig. 5.3) and the wait_improve mode (Fig. 5.4). In the wait_ok? mode, x_i records the value assignment of a neighbor in its *agent_view*. After receiving *ok?* messages from all neighbors, it calculates its current evaluation value and a possible improvement, sends *improve* messages to its neighbors, and enters the wait_improve mode.

The meanings of the state variables used in this algorithm are as follows:

can_move: represents whether x_i has the right to change its value. If the possible improvement of a neighbor x_j is larger than the improvement of x_i, or the possible improvements are equal and x_j precedes x_i in alphabetical order, *can_move* is changed to *false*.

quasi_local_minimum: represents whether x_i is in a quasi-local-minimum. If the possible improvement of a neighbor x_j is positive, *quasi_local_minimum* is changed to *false*.

my_termination_counter: is used for the termination detection of the algorithm. If the value of *my_termination_counter* is d, every agent whose distance from x_i is within d satisfies all of its constraints. We assume that x_i knows the maximal distance to other agents or an appropriate upper-bound *max_distance*. If the value of *my_termination_counter* becomes equal to *max_distance*, x_i can confirm that all agents satisfy their constraints.

The correctness of this termination detection procedure can be inductively proven using the fact that the *my_termination_counter* of x_i is increased from d to $d+1$ iff each neighbor of x_i satisfies all of its constraints and the *my_termination_counter* of each neighbor is equal to or larger than d.

After receiving *improve* messages from all neighbors, x_i changes the weights of constraint violations if the state variable *quasi_local_minimum* is *true*. Since each agent independently records the weights, neighboring agents do not need to negotiate about increasing the weights. If the state variable *can_move* is *true*, x_i's value is changed; otherwise, the value remains the same. The agent sends *ok?* messages and enters the wait_ok? mode.

Due to the delay of messages or differences in the processing speeds of agents, x_i may receive an *improve* message even though it is in the wait_ok?

```
wait_ok? mode
when received (ok?, (x_j, d_j)) do
  counter ← counter + 1;
  add (x_j, d_j) to agent_view;
  when counter = number_of_neighbors do
      send_improve; counter ← 0;
      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 that gives the maximal improvement;
  if current_eval = 0 then consistent ← true;
    else consistent ← false;
      my_termination_counter ← 0; end if;
  if my_improve > 0
    then can_move ← true; quasi_local_minimum ← false;
    else can_move ← false;
      quasi_local_minimum ← true; end if;
  send (improve, x_i, my_improve, current_eval,
          my_termination_counter) to neighbors;
```

Fig. 5.3. Distributed Breakout Algorithm (wait_ok? mode)

mode, or vice versa. In such a case, x_i postpones the processing of the message, and waits for the next message. The postponed message is processed after x_i changes its mode. On the other hand, since an agent cannot send *ok?* messages unless it receives *improve* messages from all neighbors, x_i in the wait_ok? mode will never receive an *ok?* message that should be processed in the next or previous cycle. The same is true for *improve* messages.

5.5 Example of Algorithm Execution

We show an example of algorithm execution in Fig. 5.5. This problem is an instance of a distributed graph-coloring problem, where the possible colors of agents are black and white. We assume that initial values are chosen as in Fig. 5.5 (a). Each agent communicates this initial value via *ok?* messages. After receiving *ok?* messages from all of its neighbors, each agent calculates *current_evaluation* and *my_improvement*, 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 constraint violating pairs (x_1=white and x_6=white, x_2=black and x_5=black, and x_3=white and x_4=white) are increased by 1 (Fig. 5.5 (b)).

```
wait_improve? mode
when received (improve, x_j, improve,
                        eval, termination_counter) do
  counter ← counter + 1;
  my_termination_counter ←
      min(termination_counter, my_termination_counter)
  when improve > my_improve do
    can_move ← false;
    quasi_local_minimum ← false; end do;
  when improve = my_improve and x_j precedes x_i do
    can_move ← false; end do;
  when eval > 0 do
    consistent ← false; end do;
  when counter = number_of_neighbors do
    send_ok; counter ← 0; clear agent_view;
    goto wait_ok mode; end do;
  goto wait_improve mode; end do;

procedure send_ok
  when consistent = true do
    my_termination_counter ← my_termination_counter +1;
      when my_termination_counter = max_distance do
        notify neighbors that a solution has been found;
            terminate the algorithm;
      end do; end do;
  when quasi_local_minimum = true do
    increase the weights of constraint violations; end do;
  when can_move = true do
    current_value ← new_value; end do;
  send (ok?, (x_i, current_value)) to neighbors;
```

Fig. 5.4. Distributed Breakout Algorithm (wait_improve mode)

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 the alphabetical order within its own neighborhood). They each change their value from white to black (Fig. 5.5 (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 (Fig. 5.5 (d)). After that, agents repeat exchanging *ok?* messages and *improve* messages. When the *my_termination_counter* of some agent becomes equal to the *max_distance*, the agent can confirm that all constraints are satisfied, and agents terminate the execution of the algorithm.

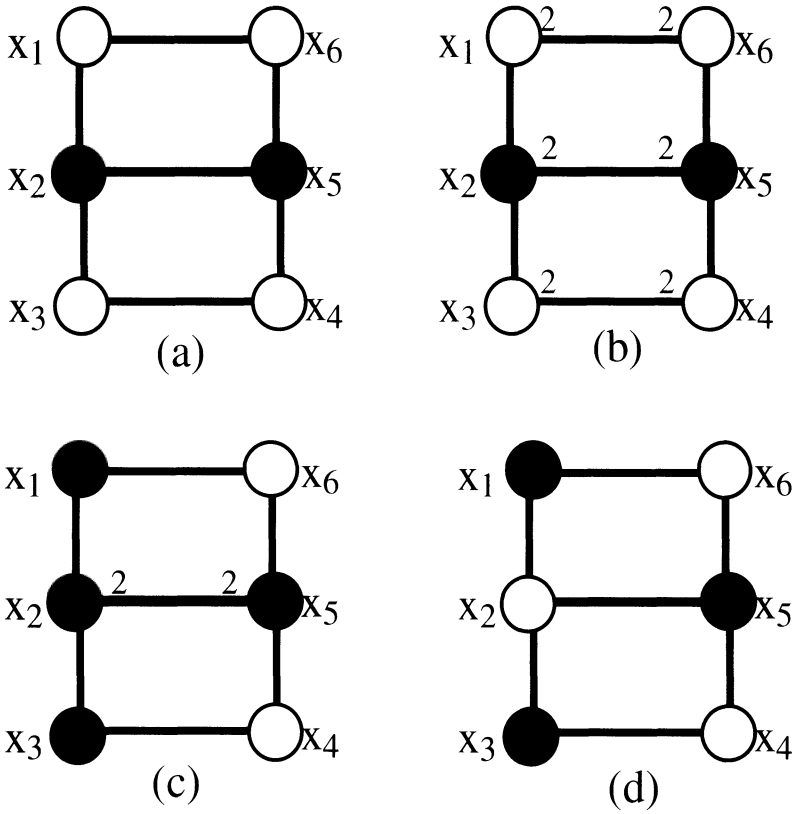

Fig. 5.5. Example of Algorithm Execution (Distributed Breakout)

5.6 Evaluations

We evaluate the efficiency of algorithms by a discrete event simulation described in Section 3.5. 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 of algorithms 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. In the distributed breakout algorithm, each mode (wait_ok? or wait_improve) requires one cycle. Therefore, each agent can change its value at most once in two cycles.

Here, we compare the distributed breakout algorithm (DB), the asynchronous weak-commitment search algorithm (AWC) described in Chapter 4, and the iterative improvement algorithm presented in (Hirayama and Toyoda 1995). We call the latter algorithm *hill-climbing+coalition* (HCC). In HCC, we use the altruistic strategy (Hirayama and Toyoda 1995) in coalitions, and break up all coalitions broken up to restart with new initial values if one of the coalitions grows to a state where it includes over five agents. Furthermore, to examine the effect of changing weights in quasi-local-minima rather than real local-minima, we show the result of an algorithm in which each agent broadcasts messages not only to its neighbors but to all agents, and increases the weights only in a real local-minimum. We call this algorithm *distributed breakout with broadcasting* (DB+BC). In this algorithm, messages from non-neighbors are used only for detecting local-minima, and other procedures are equivalent to the original distributed breakout algorithm.

We use the distributed graph-coloring problem for our evaluations. This problem can represent various application problems such as channel allocation problems in mobile communication systems, in which adjoining cells (regions) cannot use the same channels to avoid interference. A distributed graph-coloring problem can be characterized by three parameters, i.e., the number of agents/variables n, the number of possible colors of each agent k, and the number of links between agents m. We randomly generated a problem with n agents/variables and m arcs by the method described in (Minton, Johnston, Philips, and Laird 1992) so that the graph is connected and the problem has a solution.

First, we evaluated the problem for $n = 90$, 120, and 150, where $m = n \times 2$, k=3. This parameter setting corresponds to the "sparse" problems for which (Minton, Johnston, Philips, and Laird 1992) reported poor performance of the min-conflict heuristic. We generated 10 different problems, and for each problem, 10 trials with different initial values were performed (100 trials in all). The initial values were set randomly. The results are summarized in Table 5.1. For each trial, in order to conduct the experiments within a reasonable amount of time, we set the limit for the number of cycles at 10,000, and terminated the algorithm if this limit was exceeded; in such a case, we counted the result as 10,000. We show the ratio of trials completed successfully to the total number of trials in Table 5.1.

Furthermore, we show results where the number of links $m = n \times 2.7$ and $m = n \times (n-1)/4$ in Table 5.2 and Table 5.3, respectively. The setting where $k = 3$ and $m = n \times 2.7$ has been identified as a critical setting that produces particularly difficult, phase-transition problems in (Cheeseman, Kanefsky, and Taylor 1991). The setting where $m = n \times (n-1)/4$ represents the situation in which the constraints among agents are dense.

Finally, we evaluated the problem for $n = 60$, 90, and 120, where $m = n \times 4.7$ and k=4. This parameter setting has also been identified as a critical setting that produces particularly difficult problems in (Cheese-

Table 5.1. Evaluation with "Sparse" Problems ($k = 3, m = n \times 2$)

Algorithm		n		
		90	120	150
DB	ratio	100%	100%	100%
	cycles	150.8	210.1	278.8
DB+BC	ratio	100%	100%	100%
	cycles	230.4	253.4	344.5
AWC	ratio	100%	100%	100%
	cycles	70.1	106.4	159.2
HCC	ratio	98%	100%	99%
	cycles	533.5	538.4	1074.2

Table 5.2. Evaluation with "Critical" Problems ($k = 3, m = n \times 2.7$)

Algorithm		n		
		90	120	150
DB	ratio	100%	100%	100%
	cycles	517.1	866.4	1175.5
DB+BC	ratio	100%	100%	100%
	cycles	397.1	693.0	687.7
AWC	ratio	97%	65%	29%
	cycles	1869.6	6428.4	8249.5
HCC	ratio	66%	37%	19%
	cycles	5305.7	7788.2	8874.0

Table 5.3. Evaluation with "Dense" Problems ($k = 3, m = n \times (n-1)/4$)

Algorithm		n		
		90	120	150
DB	ratio	100%	100%	100%
	cycles	31.2	34.6	34.9
DB+BC	ratio	100%	100%	100%
	cycles	31.2	34.4	34.9
AWC	ratio	100%	100%	100%
	cycles	9.9	9.3	9.6
HCC	ratio	100%	100%	100%
	cycles	65.6	70.2	70.7

man, Kanefsky, and Taylor 1991). The results are summarized in Fig. 5.4. The limit of the cycles was set to 40,000 in this setting.

Table 5.4. Evaluation with "Critical" Problems ($k = 4, m = n \times 4.7$)

Algorithm		n		
		60	90	120
DB	ratio	100%	100%	100%
	cycles	591.3	1175.8	2218.1
DB+BC	ratio	100%	100%	100%
	cycles	497.1	691.8	1616.7
AWC	ratio	100%	83%	25%
	cycles	1733.6	14897.3	34771.6
HCC	ratio	61%	26%	9%
	cycles	19953.8	32923.9	38028.1

We can derive the following conclusions from these results.

- For "sparse" and "dense" problems, the asynchronous weak-commitment search algorithm is most efficient.
  Since these problems are relatively easy, the overhead for controlling concurrent actions among neighbors does not pay. Note that in the distributed breakout algorithm, an agent can change its value at most once in two cycles, and in the hill-climbing algorithm, an agent can change its value once in three cycles (i.e., sending negotiate, sending reply, and sending state), while in the asynchronous backtracking algorithm, an agent can change its value in every cycle.
- For critically difficult problem instances, the distributed breakout algorithm is most efficient.
  On the other hand, in centralized CSPs, as shown in Chapter 1, even for critically difficult problems, the weak-commitment search algorithm (the centralized version of the asynchronous weak-commitment search algorithm) is much more efficient than the breakout algorithm. What causes this difference?
  First, in the (centralized) weak-commitment search algorithm, various variable ordering heuristics such as forward-checking and the first-fail principle (Haralick and Elliot 1980) are introduced, and by performing a look-ahead search before determining a variable value, the algorithm can avoid choosing a value that leads to an immediate failure. Although it is possible to introduce variable/agent ordering heuristics in the asynchronous weak-commitment search algorithm, it is difficult to perform a look-ahead search in distributed CSPs since the knowledge about the problem is distributed among multiple agents.
  Furthermore, the breakout algorithm requires more computations in each cycle (i.e., changing one variable value or changing weights) than the weak-

commitment search algorithm. Especially, for detecting a local-minimum, the breakout algorithm has to check all values for all constraint violating variables. More specifically, when selecting a variable to modify its value, the weak-commitment search algorithm can choose any of the constraint violating variables, while the breakout algorithm must choose a variable so that the number of constraint violations is reduced. Therefore, in the worst case (when the current state is a local-minimum), the breakout algorithm has to check all the values for all constraint violating variables.
On the other hand, in the distributed breakout algorithm, the computations for each variable are performed concurrently by multiple agents. Therefore, the required computations of an agent for each cycle in the distributed breakout algorithm are basically equivalent to those in the asynchronous weak-commitment search algorithm. In other words, the potential parallelism in the breakout algorithm is greater than that in the weak-commitment search algorithm.

- Increasing the weights in quasi-local-minima that are not real local-minima does not adversely affect the performance in "sparse" and "dense" problems. The performance is even improved in "sparse" problems.
  This result can be explained as follows. Assume x_i is in a quasi-local-minimum, while the current state is not a real local-minimum since x_j (which is a non-neighbor of x_i) can improve the evaluation value. If the constraints among agents are sparse, the effect of changing x_j's value to x_i would be relatively small. Therefore, the chance that x_i will be not in a quasi-local-minimum after x_j changes its value is very slim, i.e., eventually the state would be a real local-minimum. When the constraints are dense, the chance of being trapped in a quasi or real local-minimum is very small.
- In critically difficult problems, increasing the weights in quasi-local-minima that are not real local-minima does adversely affect the performance.
  For example, in the case where $n = 120, k = 4$, and $m = n \times 4.7$, the distributed breakout algorithm requires 40% more steps than the distributed breakout algorithm with broadcasting. However, the broadcasting algorithm requires many more (more than ten times as many) messages than the original distributed breakout algorithm. Considering the cost of sending/receiving these additional messages, the performance degradation by increasing weights in quasi-local-minima seems to be acceptable. On the other hand, increasing the weights in quasi-local-minima does not adversely affect the performance in "sparse" and "dense" problems.
- The hill-climbing+coalition algorithm is not very efficient for all problems, especially for "critical" problems.
  One reason is that the current bound of the coalition size (i.e., 5) is too small for difficult problems, where coalitions tend to be very large. However, a larger coalition consumes a great amount of constraint checks and degrades the overall performance.

5.7 Discussions

One drawback of the distributed breakout algorithm is that the completeness of the algorithm cannot be guaranteed since it may fall into an infinite processing loop. We say that an algorithm is complete if it is guaranteed to eventually find one solution when solutions exist; and when there exists no solution, the algorithm is guaranteed to find this out and terminate. The distributed breakout algorithm may fall into an infinite processing loop, so it cannot guarantee that it will find a solution even if a solution does exist. On the other hand, the asynchronous weak-commitment search algorithm and the hill-climbing+coalition algorithm are guaranteed to be complete.

One advantage of the distributed breakout algorithm is that the termination detection procedure is embedded in the algorithm, while the other two algorithms must run separate procedures such as (Chandy and Lamport 1985) for termination detection.

In (Davenport, Tsang, Wang, and Zhu 1994), an algorithm similar to the breakout algorithm is implemented by connectionist architecture. In this algorithm, concurrent changes of neighboring clusters, which correspond to agents in distributed CSPs, are not prohibited. Therefore, there is a chance that the network of clusters will oscillate. If we assume the communications among clusters is fast, such an oscillation would not be very frequent. However, if we assume that communications are slow compared with the local computation, there is a chance that the network of clusters will oscillate even though each cluster runs asynchronously. Furthermore, a local-minimum is detected by the fact the network as a whole is not changed within a certain time period. Such a global control would be difficult to introduce into distributed CSPs.

5.8 Summary

In this chapter, we described an algorithm for solving distributed CSPs called the distributed breakout algorithm, which is inspired by the breakout algorithm for solving centralized CSPs. In this algorithm, each agent tries to minimize the number of constraint violations by exchanging the current value assignment and the amount of its possible improvement among neighboring agents. Instead of detecting a local-minimum, each agent detects a quasi-local-minimum, and changes the weights of constraint violations to escape from the quasi-local-minimum. Experimental evaluations showed that this algorithm is much more efficient than the asynchronous weak-commitment search algorithm for particularly difficult problem instances.

6. Distributed Consistency Algorithm

6.1 Introduction

In this chapter, we describe distributed consistency algorithms for solving distributed CSPs. Achieving 2-consistency (arc-consistency) by multiple agents is relatively straightforward, since the algorithm can be achieved by the iteration of local processes. In (Prosser, Conway, and Muller 1992), a distributed system that achieves arc-consistency for resource allocation tasks was developed. This system also maintains 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.

On the other hand, achieving a higher degree of consistency is not straightforward, since most higher degree consistency algorithms, in which sets of variable values that satisfy all constraints among them are generated, require global synchronization, i.e., 2-consistency must be achieved before going into 3-consistency, and so on. On the other hand, the ATMS-based consistency algorithm described in Section 1.3.3 generates forbidden sets, i.e., nogoods. In this case, the generation order is irrelevant to the final result. Therefore, this algorithm can be straightforwardly applied to distributed CSPs.

In this chapter, we first describe a distributed problem-solving model called *distributed ATMS*, in which each agent has its own ATMS, and these agents communicate hypothetical inference results and nogoods among themselves (Section 6.2). Then, we show a way to implement distributed consistency algorithms by using a distributed ATMS (Section 6.3). Furthermore, we show an example of the algorithm execution (Section 6.4), and evaluation results (Section 6.5).

6.2 Overview of Distributed ATMS

6.2.1 ATMS

An ATMS is a kind of database that maintains consistency among data. The characteristics of an ATMS are as follows.

M. Yokoo, *Distributed Constraint Satisfaction*

An ATMS can handle multiple contexts. An ATMS manages multiple environments. An environment is represented as a set of assumptions, and these environments form a lattice based on the superset-subset relationship of environments (Fig. 6.2). When a fact α is derived from a set of assumptions $\{H_1, H_2, \ldots, H_n\}$, which corresponds to an environment E, we say that α holds in E. One environment corresponds to one possible world where the set of assumptions holds. If α is true in E, it should also hold in all environments that are supersets of E. Therefore, the environments where α holds can be represented by a set of *minimal* environments. For each fact, an ATMS associates a set of minimal environments where the fact holds. This set is called a *label.* A label may contain multiple environments since a fact can be derived from different sets of assumptions. When a fact α is derived from a set of facts $\beta_1, \beta_2, \ldots, \beta_n$, the label of α can be calculated from the labels of $\beta_1, \beta_2, \ldots, \beta_n$, i.e., the environments where α holds are determined by the environments where the preconditions hold. This dependency $\beta_1, \beta_2, \ldots, \beta_n \rightarrow \alpha$ is called a *justification.*

An ATMS can efficiently avoid contradictions. When an contradiction is derived in an environment E, E is recorded as a nogood. Furthermore, all environments that are supersets of E also become nogoods. The environments that become nogoods are removed from the labels of all facts.

6.2.2 Distributed ATMS

In a distributed ATMS, each agent has its own ATMS, and communicates the following information (Fig. 6.1).

- an inference result and its label
  An inference result is communicated with its label. When an inference result is received and its label contains an unknown environment for the receiver, it creates a new environment. For example, in Fig. 6.2, assume that the fact that an animal is an ungulate is received. If the receiver does not have an environment $\{H_1, H_4\}$, it creates a new environment where the fact holds.
- a nogood
  When a nogood is communicated from another agent, e.g., nogood $\{H_3\}$ is received in Fig. 6.2, all environments that are supersets of this nogood, i.e., $\{H_1, H_3\}$, $\{H_2, H_3\}$, and $\{H_1, H_2, H_3\}$, become nogoods.

By using a distributed ATMS, an agent can share hypothetical inference results.

6.3 Distributed Consistency Algorithm Using Distributed ATMS

Agents can achieve strong k-consistency by the following procedure.

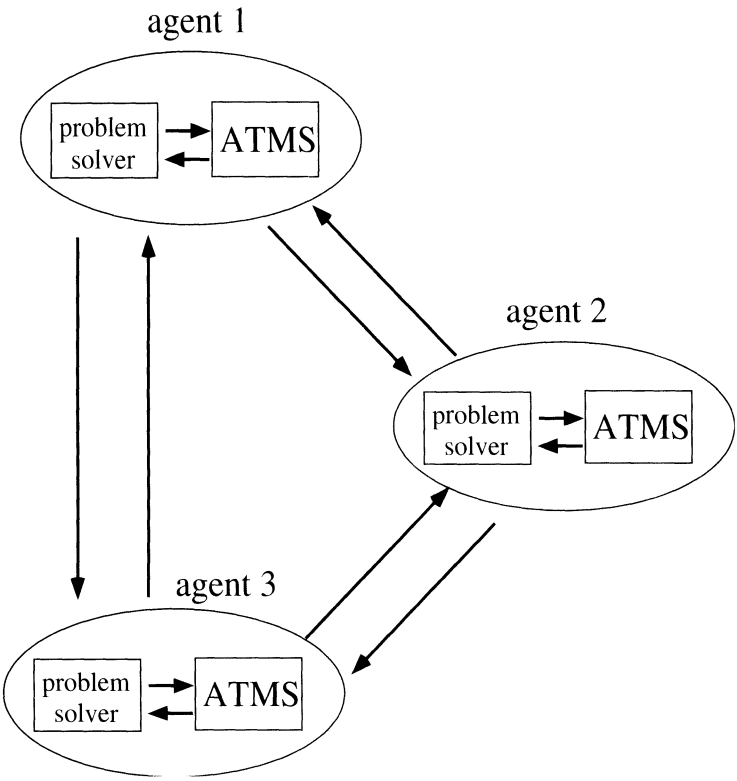

Fig. 6.1. Distributed ATMS

1. Each agent represents variable values as assumptions.
2. Agents exchange assumptions (i.e., domains of variables).
3. Agents generate nogoods using constraints, then generate new nogoods whose lengths are less than k using hyper-resolution rules, and communicate the new nogoods to relevant agents.
4. Agents generate new nogoods from communicated nogoods by using hyper-resolution rules. This procedure is repeated until no new nogood can be generated.

This procedure can be modified by restricting the application of hyper-resolution rules so that an identifier of agents in newly generated nogoods must be smaller than its own identifier. The obtained result is equivalent to the directional k-consistency algorithm (Dechter and Pearl 1988).

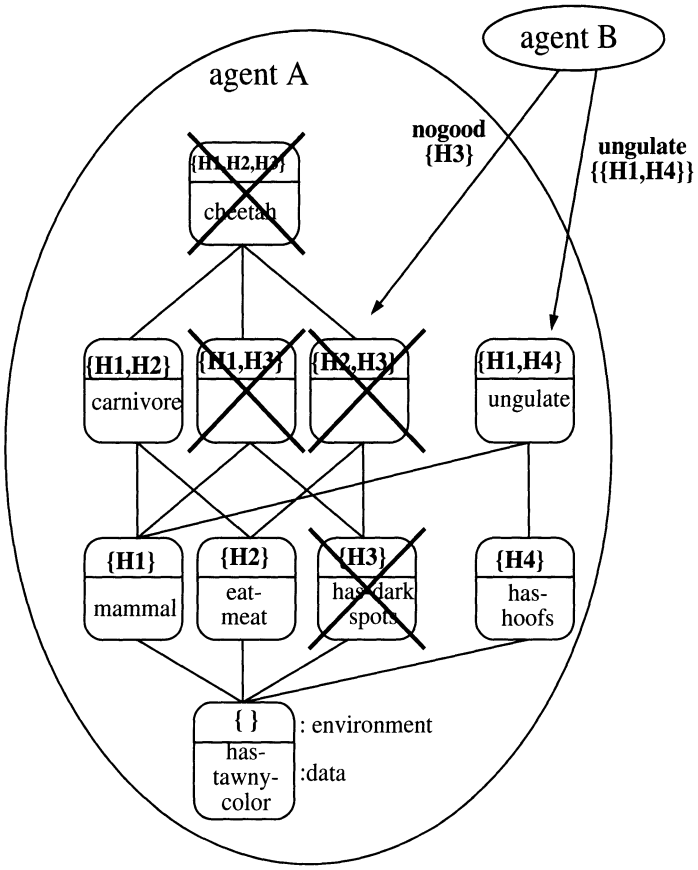

Fig. 6.2. Environment Lattice and Communication among Agents

6.4 Example of Algorithm Execution

Here, we show how distributed consistency algorithms can be applied to the network resource allocation problem described in Chapter 2. Figure 6.3 shows the distributed communication network, There are two goals, one is to connect A-1 and D-1 (goal-1), and the other is to connect A-2 and E-1 (goal-2). Table 6.1 shows the plan fragments of agents, and Table 6.2 shows global plans.

The variables and values of agents are described in Table 6.3. Each agent has one variable that corresponds to one goal, and the values of a variable corresponds to plan fragments. Since an agent does not have to contribute to all goals, a variable has a value 'idle' that means that the agent does not contribute to the goal. Since one communication link can handle only one communication channel, plan fragments that use the same communication link cannot be achieved simultaneously. For example, 1B and 2B cannot be

executed simultaneously, i.e., {(B-goal1, 1B), (B-goal2, 2B)} is a nogood. Also, if agent A executes plan fragment 3A, agent B must execute plan fragment 2B. Therefore, {(A-goal2, 3A), (B-goal2, idle)} is a nogood.

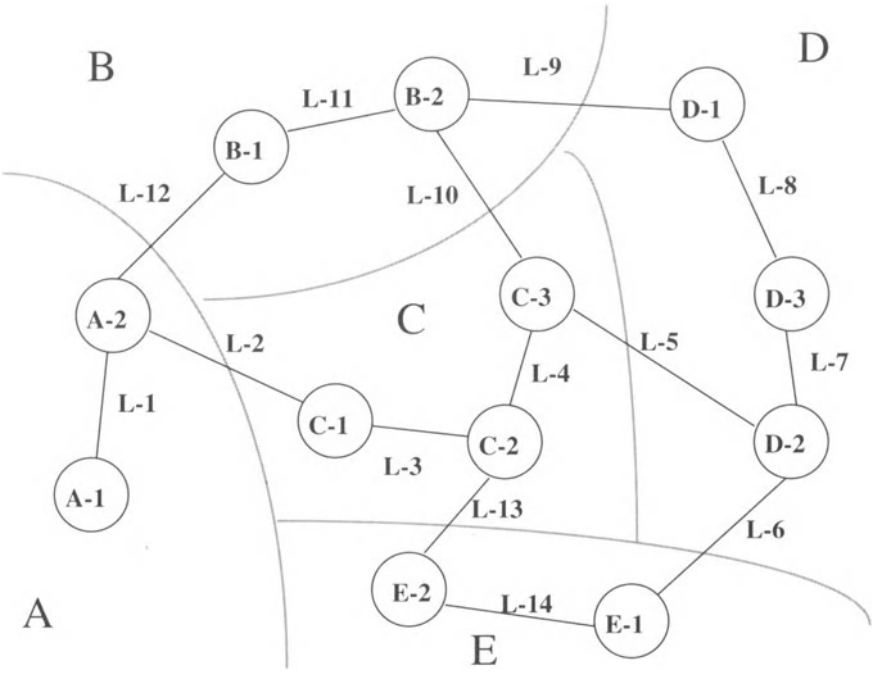

Fig. 6.3. Example of Distributed Resource Allocation Problem

When agents communicate nogoods and achieve 2-consistency, since variable A-goal2 has only one possible value 3A, nogood {(B-goal2, idle)} is obtained from nogood {(A-goal2, 3A), (B-goal2, idle)}. From this nogood and nogood {(B-goal1,1B), (B-goal2, 2B)}. a new nogood {(B-goal1, 1B)} is obtained. Therefore, variable B-goal1 has only one possible value 'idle'. Similarly, 2A, 2C, 1D, 3C, 2D, and 1E become nogoods, and all variable values are determined uniquely after achieving 2-consistency.

6.5 Evaluations

As in centralized CSPs, finding an appropriate combination of consistency algorithms and backtracking is an important research issue in distributed CSPs. For example, in the distributed 8-queens problem, weak consistency

Table 6.1. Plan Fragments

agent	goal	plan fragment	resource used
A	goal-1	1A	L-1, L-2
		2A	L-1, L-12
	goal-2	3A	L-12
B	goal-1	1B	L-10, L-11, L-12
	goal-2	2B	L-10, L-11, L-12
C	goal-1	1C	L-2, L-3, L-4, L-5
		2C	L-5, L-10
	goal-2	3C	L-5, L-10
		4C	L-4, L-10, L-13
D	goal-1	1D	L-5, L-7, L-8
	goal-2	2D	L-5, L-6
E	goal-2	1E	L-6
		2E	L-13, L-14

Table 6.2. Global Plans

goal	plan	plan fragments
goal-1	plan-11	1A, 1C, 1D
	plan-12	2A, 1B, 2C, 1D
goal-2	plan-21	3A, 2B, 3C, 2D, 1E
	plan-22	3A, 2B, 4C, 2E

Table 6.3. Variables and Domains of Agents

agent	variable	value
A	A-goal1	1A, 2A
	A-goal2	3A
B	B-goal1	1B, idle
	B-goal2	2B, idle
C	C-goal1	1C, 2C, idle
	C-goal2	3C, 4C, idle
D	D-goal1	1D
	D-goal2	2D, idle
E	E-goal2	1E, 2E

algorithms (e.g., 2 or 3-consistency algorithms) are totally useless since no new nogood is generated. Also, more powerful consistency algorithms are inefficient because too many nogoods are generated. On the other hand, in line drawing recognition problems, the filtering algorithm eliminates all futile backtracking. Also, in the resource allocation problems in communication networks described in (Conry, Kuwabara, Lesser, and Meyer 1991), where no solution exists since constraints are too tight, the fact that there is no solution is obtained by achieving 2 or 3-consistency.

In the experiments on many randomly generated problems, the following results were obtained.

- The effect of the distributed 2-consistency algorithm (i.e., the reduced number of required cycles in backtracking) is almost identical to the cost of achieving 2-consistency. The cost of achieving k-consistency where $k > 2$ exceeds the effect of reducing futile backtracking.

We show one example in Fig. 6.4. We evaluate the efficiency of algorithms by a discrete event simulation described in Section 3.5. 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. Figure 6.4 shows the total number of cycles for the asynchronous backtracking algorithm and the synchronous backtracking algorithm, with/without achieving 2-consistency, and the required cycles for the distributed 2-consistency algorithm. In this experiment, we used randomly generated distributed CSPs, where the number of variables/agents is 10, the number of values of a variable is 5, and the number of constraints is 15, and the strength of a constraint, i.e., the probability that a given value pair satisfies the constraint, is 0.4. Figure 6.4 shows the average of 100 problem instances.

According to (Dechter and Meiri 1989), the 2-consistency algorithm is usually effective for reducing futile backtracking in centralized CSPs, and k-consistency algorithms where $k > 2$ require too much preprocessing costs. However, in distributed CSPs, the distributed 2-consistency algorithm is not effective in reducing the required cycles for the asynchronous backtracking algorithm.

In the synchronous or centralized backtracking algorithm, variable values are determined sequentially. If there exists a strongly constrained variable, propagating the constraints from that variable by preprocessing is very effective for reducing useless backtracking, which occurs before determining the value of the strongly constrained variable. On the other hand, in the asynchronous backtracking algorithm, constraints are propagated immediately from the strongly constrained variable. Therefore, the distributed 2-consistency algorithm is not as effective for the asynchronous backtracking algorithm as for the synchronous backtracking algorithm.

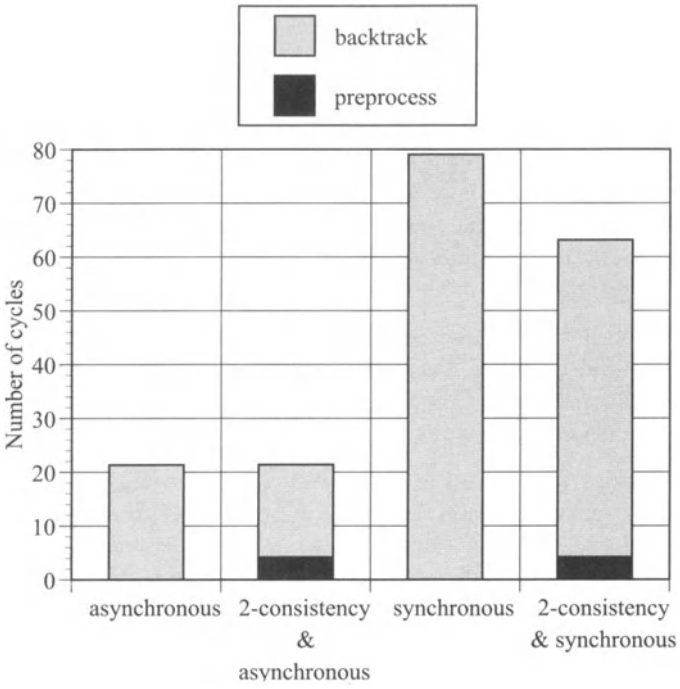

Fig. 6.4. Effect of Distributed 2-Consistency Algorithm for Synchronous and Asynchronous Backtracking

6.6 Summary

In this chapter, we showed a distributed problem-solving model called distributed ATMS, in which each agent has its own ATMS and communicates hypothetical inference results. We also described a way to implement distributed consistency algorithms by using a distributed ATMS. Furthermore, we presented an example of the algorithm execution and evaluation results. Experimental evaluations showed that distributed consistency algorithms are less effective for distributed CSPs when agents can act concurrently in distributed search algorithms.

7. Handling Multiple Local Variables

7.1 Introduction

One limitation of the algorithms described so far is that they assume each agent has only one local variable. This assumption cannot be satisfied when the local problem of each agent becomes large and complex. Although these 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 of 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 and not to distinguish the communications between virtual agents within a single real agent and the communications between real agents.

In (Armstrong and Durfee 1997), the prioritization among agents is introduced for handling multiple local variables. 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 have been examined (Armstrong and Durfee 1997).

M. Yokoo, *Distributed Constraint Satisfaction*

One limitation of this approach is that if a higher priority agent selects a bad local solution (i.e., a local solution that cannot be a part of a global solution), a lower priority agent must exhaustively search its local problem in order to change the bad decision made by the higher priority agent. When a local problem becomes large and complex, conducting such an exhaustive search becomes impossible. This approach is similar to that in Method 1 described above, except that each agent searches for its local solutions only as required instead of finding all solutions in advance. However, if the local solution selected by a higher priority agent is bad, a lower priority agent is forced to exhaustively search its local problem after all.

In this chapter, we develop a new algorithm that is similar to that in Method 2. However, in this algorithm, an agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution that satisfies all local constraints. Experimental evaluations using example problems show that this algorithm is far more efficient than an algorithm that employs prioritization among agents, or than a simple extension of the asynchronous weak-commitment search algorithm for the case of a single local variable.

In the following, we first describe an algorithm that uses the prioritization among agents (Section 7.2). Then, we present the basic ideas and details of the asynchronous weak-commitment search algorithm for the case of multiple local variables, and show an example of the algorithm execution (Section 7.3). Finally, we show empirical results that illustrate the efficiency of the newly developed algorithm (Section 7.4).

7.2 Agent-Prioritization Approach

In this section, we briefly describe the algorithm based on agent-prioritization (Armstrong and Durfee 1997). We assume each agent has multiple variables, and each variable has a unique identifier. An agent maintains a list called *current_assignments*, which contains the information of the current value assignment of its variables. Figure 7.1 shows the procedure executed at agent i by receiving an *ok?* message. Agent i checks *current_assignments* and changes the value of a variable that violates a constraint with higher priority agents. This algorithm is similar to the asynchronous backtracking algorithm, but if an agent finds that it cannot find a consistent value for its highest priority variable (i.e., it cannot find a consistent local solution with higher priority agents), it performs backtracking or changes the prioritization among agents.

```
when received (ok?, (sender_id, variable_id, variable_value)) do
  add (sender_id, variable_id, variable_value) to agent_view;
  check_agent_view; end do;

procedure check_agent_view
  when agent_view and current_assignments are not consistent do
    select a variable x_k in current_assignments,
        which has the lowest priority within inconsistent variables;
    check_agent_view_one (x_k); end do;

procedure check_agent_view_one(x_k)
  if no value in D_k is consistent with
        agent_view and current_assignments then
    backtrack (x_k); check_agent_view;
  else select d ∈ D_k where d is consistent
        with agent_view and current_assignments;
    set x_k to d;
    send (ok?, (i, x_k, d)) to other agents;
    add (i, x_k, d) to current_assignments; check_agent_view; end if;

procedure backtrack(x_k)
  if x_k has the highest priority within i's variables
  then perform backtracking or change agent priority;
  else find V, which is a subset of agent_view ∪ current_assignments
      where x_k has no consistent value;
    select (i, x_l, d_l) where x_l has the lowest priority in V;
    check_agent_view_one(x_l);
  end if;
```

Fig. 7.1. Procedures for Receiving Messages (Agent-Prioritization Approach)

7.3 Asynchronous Weak-Commitment Search with Multiple Local Variables

7.3.1 Basic Ideas

We modify the asynchronous weak-commitment search algorithm for the case of a single local variable in the following ways.

- 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 agents. 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.

7.3.2 Details of Algorithm

In the asynchronous weak-commitment search algorithm for the case of multiple local variables, each agent assigns values to its variables, and sends the values and the priority values to related agents. After that, agents wait for and respond to incoming messages[1]. In Fig. 7.2, the procedures executed by agent i in receiving an *ok?* message are described[2].

In order to guarantee the completeness of the algorithm, the agent needs to record and communicate nogoods. Agents try to avoid situations previously found to be nogoods. However, due to the delay of messages, an *agent_view* of an agent can occasionally be a superset of a previously found nogood. In order to avoid reacting to unstable situations and performing unnecessary changes to priority values, if an agent identifies an identical nogood it has already sent, the agent will not change the priority value but wait for the next message. By these procedures, the completeness of the algorithm is guaranteed, since the priority value of a variable is changed only when a new nogood is created.

7.3.3 Example of Algorithm Execution

We show an example of algorithm execution in Fig. 7.3. This problem is an instance of a distributed graph-coloring problem, where the goal is to assign a color to each node so that the nodes connected by a link have different colors. The possible colors for each node are black, white, or gray. There are two agents, i.e., agent1 and agent2, each of which has three variables.

We assume that the initial values are chosen as in Fig. 7.3 (a). Each agent communicates these initial values via *ok?* messages. In the initial state, priority values of all variables are 0. Each agent checks whether the current value

[1] As in the asynchronous weak-commitment search algorithm, although the following algorithm is described in a way that an agent reacts to messages sequentially, an agent can handle multiple messages concurrently, i.e., the agent first revises *agent_view* according to the messages, and performs **check_agent_view** only once.

[2] As in the asynchronous weak-commitment search algorithm, the way to determine that agents as a whole have reached a stable state is not contained in this algorithm.

```
when received (ok?, (sender_id, variable_id, variable_value, priority)) do
  add (sender_id, variable_id, variable_value, priority) to agent_view;
  when agent_view and current_assignments are not consistent do
    check_agent_view; end do;

procedure check_agent_view
  if agent_view and current_assignments are consistent then
    communicate changes to related agents;
  else select x_k, which has the highest priority and
      violating some constraint with higher priority variables;
    if no value in D_k is consistent with agent_view
        and current_assignments then
      record and communicate a nogood, i.e., the subset of agent_view
        and current_assignments, where x_k has no consistent value;
      when the obtained nogood is new do
        set x_k's priority value to the highest priority
          value of related variables + 1;
        select d ∈ D_k where d minimizes the number of
          constraint violations with lower priority variables;
        set the value of x_k to d;
        check_agent_view; end do;
    else select d ∈ D_k where d is consistent
        with agent_view and current_assignments, and minimizes
        the number of constraint violations with lower priority variables;
      set the value of x_k to d;
      check_agent_view; end if; end if;
```

Fig. 7.2. Procedure for Handling *ok?* Messages (Asynchronous Weak-Commitment Search for the Case of Multiple Local Variables)

assignments are consistent with higher priority variables. Since the priority values are all equal, the priority order is determined by the alphabetical order of variable identifiers. Therefore, all variables of agent1 are ranked higher than those of agent2, so agent1 does not need to change the values of its variables.

On the other hand, for agent2, while x_4, which has the highest priority within agent2, satisfies all constraints with higher priority variables, x_5 does not satisfy the constraint with x_2. Therefore, agent2 changes x_5's value to gray, which satisfies the constraints between x_2 and x_4. By this change, the constraint between x_5 and x_6 is violated. Agent2 tries to change x_6's value, but there exists no value that satisfies all constraints since all colors are taken by higher priority variables (x_3 is black, x_4 is white, and x_5 is gray). Therefore, agent2 increases x_6's priority value to 1.

It changes x_6's value so that it satisfies as many constraints between lower priority variables as possible. In this case, each color violates one constraint, so agent2 randomly selects x_6's color (black is selected in this case). Also, agent2 records and communicates a nogood {(x_3, black), (x_4, white), (x_5,

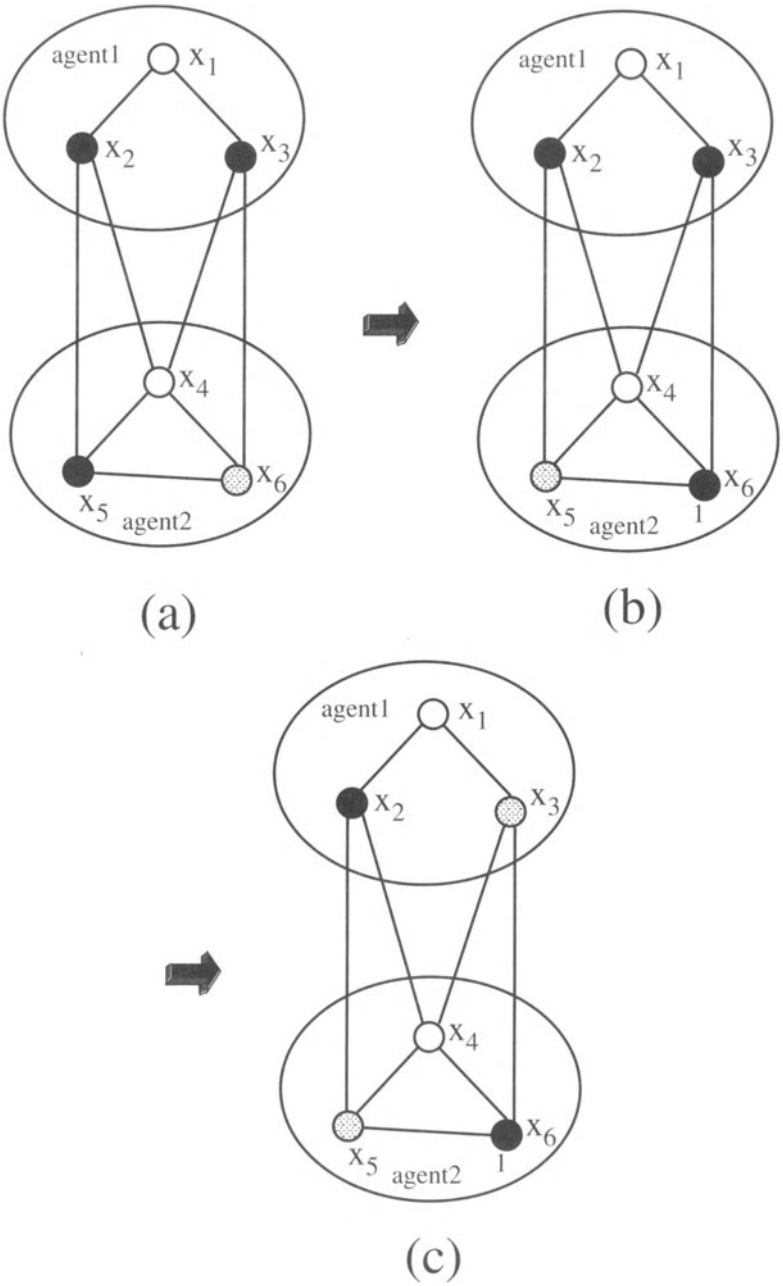

Fig. 7.3. Example of Algorithm Execution (Multiple Local Variables)

gray)}, if the completeness of the algorithm is required. As a result, all variables of agent2 satisfy all constraints with higher priority variables, so it communicates the changes to agent1 (Fig. 7.3 (b)). Then, for agent1, while x_1 and x_2 satisfy constraints with higher priority variables, x_3 violates a constraint with x_6, which has a priority value of 1. Therefore, agent1 changes x_3's value to gray, and a globally consistent solution is obtained (Fig. 7.3 (c)).

Actually, there exists no local solution for agent2 that is consistent with agent1's initial local solution. Therefore, if we use the prioritization among agents, agent2 needs to exhaustively search its local problem. Conversely, in this algorithm, since a priority value is associated to each variable, and agents change priority values dynamically, a bad local solution can be modified without exhaustively searching a local problem.

7.4 Evaluations

We evaluate the efficiency of algorithms by a discrete event simulation described in Section 3.5. 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.

We compare the efficiency of the following three algorithms: 1) multiple local variables asynchronous weak-commitment search, multi-AWC, 2) an algorithm that employs prioritization among agents, where the priorities are changed when one agent finds that there exists no consistent local solution with higher priority agents (asynchronous weak-commitment search with agent priority, AWC+AP), and 3) an algorithm in which each agent simulates the activities of multiple virtual agents (single variable asynchronous weak-commitment search, single-AWC). AWC+AP is basically identical to the algorithm using the *decaying nogoods* heuristic described in (Armstrong and Durfee 1997). However, in (Armstrong and Durfee 1997), agents are assumed to act in a sequential order. To make the comparison fair, we let agents act concurrently in AWC+AP. Also, each agent performs *min-conflict backtracking* (Minton, Johnston, Philips, and Laird 1992) in AWC+AP.

We use a distributed graph-coloring problem for our evaluations. This problem can represent various application problems such as channel allocation problems in mobile communication systems, in which adjoining cells (regions) cannot use the same channels to avoid interference. A graph-coloring problem can be characterized by three parameters, i.e., the number of nodes/variables n, the number of possible colors for each of the nodes k, and the number of links between nodes l. A parameter called link density (l/n) affects the difficulty of a problem instance, and when $k = 3$, the

setting $l/n = 2.7$ has been identified as a critical setting that produces particularly difficult, phase-transition problem instances (Cheeseman, Kanefsky, and Taylor 1991).

In Table 7.1, we show the results where the number of agents m is 10, the number of possible colors k is 3, and the number of links l is set to $n \times 2.7$, varying the number of variables for each agent n/m. Each data point is the average of the trials for 100 randomly generated problem instances. Also, in Table 7.2, we show the results obtained by varying the number of agents m, while setting the number of variables for each agent to 10.

If we simply generate links at random, the number of links within an agent becomes very small. For example, if there exist 10 agents, each of which has 10 variables, although there are 270 links in all, only less than 10% are local constraints. For each agent, there exist only two or three local constraints. Since a local problem of each agent should be a meaningful cluster of variables, it is natural to assume that local constraints should be at least as tight as inter-agent constraints. Therefore, we are going to assign half of the links to local constraints, and the other half to inter-agent constraints. We randomly generate a problem with these parameter settings by the method described in (Minton, Johnston, Philips, and Laird 1992) so that the graph is connected and the problem has a solution. The initial value of each variable is determined randomly.

To conduct the experiments within a reasonable amount of time, we limited the number of cycles to 10,000 for each trial, and terminated the algorithm if this limit was exceeded; in such a case, we counted the result as 10,000. We show the ratio of trials successfully completed to the total number of trials in Table 7.2. Furthermore, to obtain an idea of how much local computation is performed, we measure the number of consistency checks. Namely, for each cycle, we select an agent that performs the most consistency checks (a bottleneck agent for each cycle), and show the summation of consistency checks for these bottleneck agents.

Table 7.1. Evaluation by Varying the Number of Variables per Agent n/m ($k = 3, l = 2.7 \times n, m = 10$)

Algorithm		n/m			
		5	10	15	20
multi-AWC	ratio	100%	100%	100%	100%
	cycles	26.9	89.5	189.5	488.1
	checks	2989.6	22481.2	87688.8	320312.6
AWC+AP	ratio	100%	100%	79%	37%
	cycles	35.9	577.7	3951.8	7529.6
	checks	3617.6	155026.1	1978801.9	6691615.5
single-AWC	ratio	100%	79%	14%	0%
	cycles	323.0	4713.0	9083.4	—
	checks	13630.7	369195.0	1031475.1	—

Table 7.2. Evaluation by Varying the Number of Agents m ($k = 3.l = 2.7 \times n$, $n/m = 10$)

Algorithm		m		
		10	15	20
multi-AWC	ratio	100%	100%	100%
	cycles	89.5	214.7	615.6
	checks	22481.2	62049.3	190718.2
AWC+AP	ratio	100%	90%	54%
	cycles	577.7	3039.2	6568.1
	checks	155026.1	888422.6	2083577.1
single-AWC	ratio	79%	10%	0%
	cycles	4713.0	9573.6	—
	checks	369195.0	779022.9	—

The following conclusions can be derived from these results.

- For all parameter settings in our experiments, multi-AWC outperforms single-AWC and AWC+AP (both in the number of required cycles and the number of consistency checks), and this becomes greater as the size of local problems or the number of agents increases.
- The number of required cycles for multi-AWC is smaller than that for single-AWC, since in multi-AWC, agents communicate only when they find consistent local solutions. On the other hand, in single-AWC, each agent simply simulates the activities of multiple virtual agents, and agents communicate even if the values of the virtual agents within a single real agent are not consistent with the others. Although the number of consistency checks for each cycle in single-AWC is smaller than that in multiple-AWC, this advantage never compensates for the increase in the number of required cycles.
- The number of required cycles for multi-AWC is smaller than for AWC+AP. At first glance, this result seems somewhat surprising, since in AWC+AP, each agent is so diligent that it exhaustively searches for its local problem to find a local solution consistent with higher priority agents, while in multi-AWC, each agent is rather lazy and tries to increase the priority values of its variables instead of trying to satisfy constraints with higher priority variables. However, in reality, diligently trying to find a consistent local solution with higher priority agents is not necessarily good for agents as a whole. While the consistent local solution satisfies all constraints with higher priority agents, it may violate many constraints with lower priority agents. Therefore, the convergence to a global solution can be slower than with multi-AWC, where each agent simply increases the priority values of their variables, and then tries to minimize the number of constraint violations as a whole.

  Figure 7.4 shows the trace of the number of constraint violations when solving one problem instance with 10 agents and 10 variables. We can see

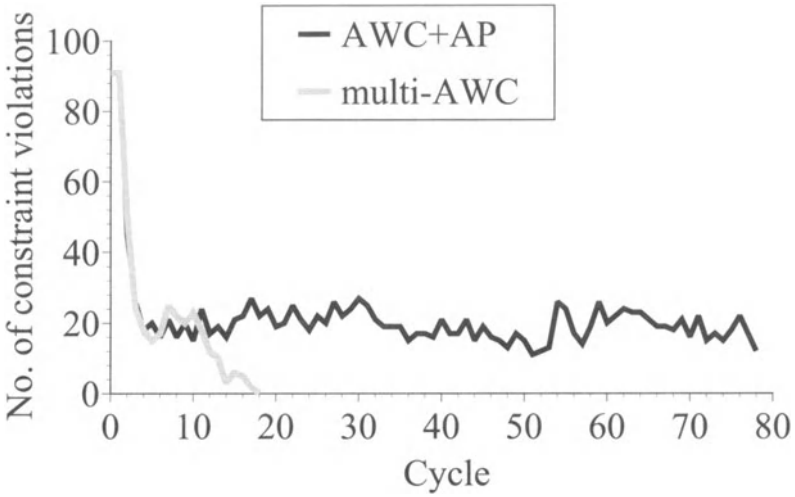

Fig. 7.4. Traces of the Number of Constraint Violations

that reducing the total number of constraint violations becomes rather difficult if agents devote too much energy to satisfying constraints with higher priority agents.

If each agent tries to find a consistent local solution that not only satisfies all constraints with higher priority agents, but also minimizes the number of constraint violations with lower priority agents, the convergence to a global solution can be hastened. However, it requires too many computations for an agent. Note that although each agent uses the min-conflict backtracking in AWC+AP, there is no guarantee that the obtained local solution minimizes the number of constraint violations with lower priority agents.

- In (Armstrong and Durfee 1997), more sophisticated heuristics for prioritization among agents were presented. However, the evaluation results in (Armstrong and Durfee 1997) show that the speedup obtained by employing these heuristics are not very drastic (at most two-fold) compared with the simple *decaying nogoods* heuristic used in AWC+AP. Therefore, we cannot assume that AWC+AP will outperform multi-AWC by employing more sophisticated prioritization heuristics when local problems are large.

7.5 Summary

In this chapter, we developed a new algorithm that can efficiently solve a distributed CSP, in which each agent has multiple local variables. This algorithm is based on the asynchronous weak-commitment search algorithm for the case

of a single local variable, but an agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution that satisfies all local constraints. By using this algorithm, a bad local solution can be modified without forcing other agents to exhaustively search their local solutions, and the number of interactions among agents can be decreased. Experimental evaluations showed that this algorithm is far more efficient than an algorithm that employs the prioritization among agents and a simple extension of the asynchronous weak-commitment search algorithm for the case of a single local variable.

8. Handling Over-Constrained Situations

8.1 Introduction

As discussed in Section 1.6, many real-life problems are often over-constrained. Similarly, various application problems in multi-agent systems can be over-constrained. For example, in a distributed interpretation problem (Lesser and Corkill 1981), each agent is assigned a task to interpret a part of sensor data. These agents produce possible interpretations and try to build a globally consistent interpretation. If an agent makes incorrect interpretations because of errors in the process—for example, noise on the sensor data— there may be a situation where no globally consistent interpretation exists.

Also, in the distributed resource allocation problem described in Section 2.3.2, each agent has a goal (variable) and possible plans to achieve the goal (domain of the variable). There exist resource conflicts between plans (constraints). The goal of this problem is to find a combination of plans that achieves the goals of all agents simultaneously. It is likely that all of the goals cannot be achieved without violating some constraints if enough resources are not available.

In this chapter, we give a formalization of a *distributed partial CSP*, where agents settle for a solution to a relaxed problem of over-constrained distributed CSP (Section 8.2). Also, we provide two classes of problems in this model, i.e., *distributed maximal CSPs* and *distributed hierarchical CSPs*, and describe algorithms for solving them (Section 8.3 and Section 8.4).

8.2 Problem Formalization

A distributed partial CSP consists of:

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

where, for each agent i, P_i is an original CSP, U_i is a set of universes, $(PS_i, \leq)$ is a problem space, and M_i is a distance function. G is a global distance function over distributed problem spaces and (N, S) are necessary and sufficient

M. Yokoo, *Distributed Constraint Satisfaction*

bounds on the global distance between an 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 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, and all search can terminate when such a solution is found. 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, and such a minimal global distance is called an optimal global distance.

The model for a distributed partial CSP can be specialized in various ways. In this chapter, we show two classes of problems: a *distributed maximal CSP* and a *distributed hierarchical CSP*.

8.3 Distributed Maximal CSPs

8.3.1 Problem Formalization

A distributed maximal CSP is a problem where each agent tries to find 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 $\max_i d_i$.

Figure 8.1 shows a distributed graph-coloring problem to illustrate a distributed maximal CSP. In this problem, there exist six agents. Each agent tries to determine its color so that neighbors do not have the same color (possible colors are white and black). An agent knows only the constraints that are relevant to its variable. For example, agent 1 (which has x_1) knows only $\{a, c, d\}$. The original CSP for agent 1 (i.e., P_1) consists of one variable: $\{x_1\}$ with a domain of {black, white} and constraints: $\{a, c, d\}$.

In this figure, the current distance of agent 1 is one because it only violates the constraint d; and for other agents, the distances are: one for agent 2, agent 5 and agent 6, and zero for agent 3 and agent 4. From these distances, we can see the global distance is one because it is measured as the maximal distance over agents. Since the goal of a distributed maximal CSP is to find a set of value assignments with the minimal global distance, agents try to reduce the global distance by changing their values. For the problem

instance in Fig. 8.1, since there is no set of value assignments that makes the maximal number of violated constraints over agents less than one, the value assignments shown in the figure gives an optimal solution.

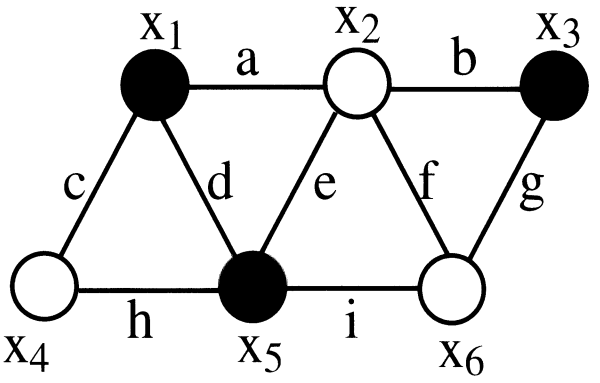

Fig. 8.1. Distributed 2-Coloring Problem

Two algorithms for solving distributed maximal CSPs have been presented in (Hirayama and Yokoo 1997). One is the *synchronous branch and bound algorithm*, and the other is the *iterative distributed breakout algorithm.*

8.3.2 Algorithms

In this section, we describe algorithms that find the optimal solution of a distributed maximal CSP.

The simplest algorithms for distributed maximal CSPs belong to the class called *centralized algorithms.* One of the centralized algorithms follows this procedure: agents run some leader election algorithm to elect one leader; send all distributed partial CSPs to the leader; the leader solves those gathered partial CSPs using some maximal constraint satisfaction algorithm (Freuder and Wallace 1992), while others are idle. If we were interested only in efficiency, the centralized algorithms might outperform other algorithms because they can make better use of the global knowledge of the entire problem. However, we believe such algorithms are not suitable for a distributed environment from a privacy and security standpoint (who on earth wants to expose an individual's schedule or private information to others?). We have therefore developed algorithms on the assumption that each agent's knowledge of the entire problem should remain limited throughout the execution of the algorithms. The algorithms we present in this section are the *Synchronous Branch and Bound* algorithm (SBB) and the *Iterative Distributed Breakout* algorithm (IDB).

Synchronous Branch and Bound. The Synchronous Branch and Bound algorithm (SBB) is a simple algorithm that simulates the branch and bound method for Max-CSPs (Freuder and Wallace 1992) described in Section 1.6. In the SBB, variable/agent and value ordering are fixed in advance, and a *path*, i.e., partial assignments for all variables, is exchanged among agents to be extended to a complete path. This extension process runs sequentially. To be concrete:

- the first agent in the ordering initiates the algorithm by sending a path that contains only its first value to the second agent;
- when receiving a path from the previous agent in the agent ordering, an agent evaluates the path and the first value of its domain in value ordering, and then sends the path plus the value as a new path to the next agent if its evaluation value is less than the current upper bound, or continues to try next values if the evaluation value is not less than the bound. If values are exhausted, it backtracks to the previous agent by returning the path;
- when receiving a path from the next agent in the agent ordering, an agent changes its assignment to the next value in its value ordering, reevaluates the new path, and sends it to the next if its evaluation value is less than the bound, or if not, continues to try next values. Another backtrack takes place if values are exhausted.

An element of a path actually consists of a variable, a value for the variable, and the number of constraint violations caused by the value. We measure the evaluation value of a path as the maximal number of constraint violations over the variables on the path, and the upper bound as the minimum evaluation value over those of complete paths found so far. Details of the SBB are shown in Fig. 8.2 and Fig. 8.3.

Since the SBB simply simulates the branch and bound method in a distributed environment, it appears obvious that the SBB is correct. Soundness is guaranteed since the SBB terminates iff it finds a complete path whose evaluation value is zero or it finds that no such complete path exists. With sequential control over agents and fixed variable/value orderings, the SBB enables agents to do an exhaustive search in distributed search spaces. This ensures that the SBB is complete, i.e., it eventually finds a sufficient solution or finds that there exists no such solution and terminates.

On the other hand, the SBB does not allow agents to assign or change their variable values in parallel, and thus the SBB cannot take advantage of parallelism.

Iterative Distributed Breakout. The Iterative Distributed Breakout algorithm (IDB) is a method for distributed maximal CSPs in which a variant of the distributed breakout algorithm described in Chapter 5 is repetitively applied to a distributed maximal CSP. The operation of the IDB is as follows: set a uniform constant value ub to each agent's necessary bound N_i and run the distributed breakout; if the distances of all agents become less than N_i,

```
procedure initiate /* done only by the first agent for starting the algorithm */
  d_i ← the first value in domain;
  n_i ← known upper bound; previous_path ← nil;
  send (token, [[x_i, d_i, 0]], n_i) to the next agent;

when i received (token, current_path, ub) from the previous agent do
  previous_path ← current_path; n_i ← ub;
  next ← get_next(domain);
  send_token; end do;

when i received (token, current_path, ub) from the next agent do
  [x_i, d_i, nv_i] ← the element related to x_i in current_path; n_i ← ub;
  next ← get_next(domain minus all elements up to d_i);
  send_token; end do;

procedure send_token
  if next ≠ nil then
    if i = the last agent then
      next_to_next ← next;
      while next_to_next ≠ nil do
        when max nv_j in new_path < n_i do
          n_i ← max nv_j in new_path;
          best_path ← new_path; end do;
        when n_i = 0 do
          terminate the algorithm; end do;
        next_to_next ← get_next(domain minus all elements up to next_to_next);
      end do;
      send (token, previous_path, n_i) to the previous agent;
    else
      send (token, new_path, n_i) to the next agent; end if;
  else
    if i = the first agent then
      terminate the algorithm;
    else
      send (token, previous_path, n_i) to the previous agent; end if; end if;
```

Fig. 8.2. Synchronous Branch and Bound (i)

the agent that detects this fact sets its N_i to $ub - 1$ and propagates its value to make N_i for all agents $ub - 1$. This process is continued until some agent detects that a solution that satisfies all constraints is found.

The IDB is very similar to the distributed breakout algorithm. It does, however, introduce an extension for handling necessary bounds on distance. The bounds are exchanged by *ok?* and *improve* messages, both of which are also used in the distributed breakout algorithm. We focus on the part that handles the necessary bounds and leaves details about the other parts, which are the same as in the distributed breakout algorithm described in Chapter 5.

```
procedure get_next(value_list)
  if value_list = nil then
    return nil;
  else
    d_i ← the first value in value_list; new_path ← nil; counter ← 0;
    if check(previous_path) then
      return d_i;
    else
      return get_next(value_list minus d_i); end if; end if;

procedure check(path)
  if path = nil then
    add [x_i, d_i, counter] to new_path;
    return true;
  else
    [x_j, d_j, nv_j] ← the first element in path;
    if [x_i, d_i] and [x_j, d_j] are not consistent then
      counter ← counter + 1;
      if counter ≥ n_i or nv_j + 1 ≥ n_i then
        return false;
      else
        add [x_j, d_j, nv_j + 1] to new_path;
        return check(path minus the first element); end if;
    else
      add [x_j, d_j, nv_j] to new_path;
      return check(path minus the first element); end if; end if;
```

Fig. 8.3. Synchronous Branch and Bound (ii)

- Before starting the IDB, an agent i assigns a uniform value, say ub, to its necessary bound N_i.
- When receiving *ok?* messages from all neighbors, an agent i counts the number of violated constraints and then sets zero as the evaluation value of its current assignment if the number is less than N_i or, if not, the agent proceeds as in the distributed breakout. The IDB thus permits an agent to have an assignment with the number of violated constraints less than N_i.
- For the distributed breakout algorithm, it is guaranteed that each agent is satisfied when some agent's *termination_counter* exceeds *diameter* (a diameter of graph). It is also guaranteed for the IDB that each agent finds a solution to its individual partial CSP with N_i when some agent's *termination_counter* exceeds *diameter*. The agent that finds this fact decreases its N_i by one and sends the new value with *improve* messages.
- An agent in the distributed breakout algorithm sets true to a state variable *consistent* iff the numbers of violated constraints in itself and all of its neighbors are zero. An agent in the IDB, on the other hand, sets true to *consistent* iff the numbers of violated constraints in itself and all of its neighbors are less than the necessary bounds.

```
procedure initiate /* done by every agent for starting the algorithm */
  current_value ← the value randomly chosen from domain;
  n_i ← known upper bound; my_termination_counter ← 0; counter ← 0;
  send (ok?, x_i, current_value) to neighbors;
  goto wait_ok? mode;

wait_ok? mode
when i received (ok?, x_j, d_j) do
  counter ← counter + 1; add (x_j, d_j) to agent_view;
  if counter = # of neighbors then
    send_improve; counter ← 0;
    goto wait_improve mode;
  else
    goto wait_ok? mode; end if; end do;

procedure send_improve
  if # of currently violated constraints < n_i then
    current_eval ← 0;
  else
    current_eval ← weighted sum of violated constraints; end if;
  my_improve ← possible maximal improvement;
  new_value ← the value that gives the maximal improvement;
  if current_eval = 0 then
    consistent ← true;
  else
    consistent ← false; my_termination_counter ← 0; end if;
  if my_improve > 0 then
    can_move ← true; quasi_local_minimum ← false;
  else
    can_move ← false; quasi_local_minimum ← true; end if;
  send (improve, x_i, my_improve, current_eval,
        my_termination_counter, n_i)
to neighbors;
```

Fig. 8.4. Iterative Distributed Breakout (wait_ok? mode)

The details of the IDB are shown in Fig. 8.4 and Fig. 8.5.

We can prove inductively that the termination detection of each iteration of the IDB is correct by the following fact: some agent i with $N_i = ub$ increases its *termination_counter* from d to $d+1$ iff each of i's neighbors has ub as the value of its necessary bound, has an assignment with the number of violated constraints less than ub, and has a *termination_counter* value of d or more.

While the SBB is sequential in terms of value assignments, the IDB enables parallel value assignments. However, the IDB is not complete, i.e., it may fail to get an optimal solution to a distributed maximal CSP. Furthermore, it cannot decide whether a solution is optimal even if it actually gets an optimal solution.

```
wait_improve mode
when i received
    (improve, x_j, improve, eval, termination_counter, ub) do
  counter ← counter + 1;
  my_termination_counter ←
      min(termination_counter, my_termination_counter);
  when n_i ≠ ub do
    n_i ← min(n_i, ub); consistent ← false; end do;
  when improve > my_improve do
    can_move ← false; quasi_local_minimum ← false; end do;
  when improve = my_improve and x_j precedes x_i do
    can_move ← false; end do;
  when eval > 0 do
    consistent ← false; end do;
  if counter = # of neighbors then
    send_ok; counter ← 0; clear agent_view;
    goto wait_ok? mode;
  else
    goto wait_improve mode; end if; end do;

procedure send_ok
  when consistent = true do
    my_termination_counter ← my_termination_counter + 1;
    when my_termination_counter = diameter of the constraint graph do
      n_i ← current global distance;
      if n_i = 0 then notify neighbors
        that a solution has been found; terminate the algorithm;
      else
        my_termination_counter ← 0; end if; end do; end do;
  when quasi_local_minimum = true do
    increase the weights of violated constraints; end do;
  when can_move = true do
    current_value ← new_value; end do;
  send (ok?, x_i, current_value) to neighbors;
```

Fig. 8.5. Iterative Distributed Breakout (wait_improve mode)

8.3.3 Evaluations

Here, we show an experimental evaluation of the SBB and the IDB. We tested both methods on *random binary distributed CSPs*, which are described as $\langle n, m, p_1, p_2 \rangle$. One problem instance was generated by distributing variables and constraints of an instance of *random binary CSPs* with those 4 parameters. We distributed them such that each agent has exactly one variable and constraints relevant to the variable. The parameters of random binary CSPs are: n is the number of variables; m is the number of values for each variable; p_1 is the proportion of variable pairs that are constrained; and p_2 is the proportion of prohibited value pairs between two constrained variables. When generating an instance of random binary CSPs with $\langle n, m, p_1, p_2 \rangle$, we

randomly selected $n(n-1)p_1/2$ pairs of variables, and for each variable pair we set up a constraint such that randomly selected m^2p_2 pairs of values are prohibited.

In the experiments, we chose classes of random binary CSPs with $n = m = 10$, p_1 taking values from $\{18/45, 27/45, 36/45, 45/45\}$, and p_2 from $\{0.8, 0.9\}$. These classes of problems are known to be relatively hard ones for Max-CSPs (Larrosa and Meseguer 1996). Accordingly, we believe that they are suitable as problems for evaluating the methods.

We evaluate the efficiency of algorithms by a discrete event simulation described in Section 3.5. 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.

Table 8.1. Median Cycles for the Synchronous Branch and Bound to Find Optimal Solution

problem class	median cycles	mean optimal distance
$\langle 10, 10, 18/45, 0.8\rangle$	3500	1.0
$\langle 10, 10, 18/45, 0.9\rangle$	18262	2.0
$\langle 10, 10, 27/45, 0.8\rangle$	46247	2.4
$\langle 10, 10, 27/45, 0.9\rangle$	499841	3.4
$\langle 10, 10, 36/45, 0.8\rangle$	336416	3.6
$\langle 10, 10, 36/45, 0.9\rangle$	1985700	5.0
$\langle 10, 10, 45/45, 0.8\rangle$	3435984	4.9
$\langle 10, 10, 45/45, 0.9\rangle$	21834077	6.0

Cost of Finding an Optimal Solution. Since it is guaranteed that the SBB finds an optimal solution, we can measure the SBB's cost of finding an optimal solution as cycles to be consumed until the SBB finds it. In the experiments, we applied the SBB to each of 25 instances randomly generated for each class of problems. Note that we used *conjunctive width heuristics* (width/domain-size) described in (Wallace and Freuder 1993) for variable ordering and lexical order for value ordering. Also note that the initial value of $\forall i N_i$ was set to the value of the maximum degree of a constraint graph minus one. Table 8.1 shows the median cycles for finding an optimal solution and the mean optimal distance over 25 instances for each class. The cost of finding an optimal solution by the SBB clearly seems to be very high.

On the other hand, the IDB may fail to get an optimal solution, as stated above, and thus we cannot measure the cost in terms of cycles. However, we conducted an experiment to determine how often the IDB fails to get

an optimal solution. In this experiment, we ran 10 trials of the IDB with randomly chosen initial assignments for each of 25 instances for the class of $n = m = 10, p_1 = 27/45, p_2 = 0.8$ (250 trials in total). Note that the initial values of $\forall i N_i$ for IDB were the same as those for the SBB. As a result, the IDB required fewer cycles than the SBB to obtain optimal solutions in 30 trials.

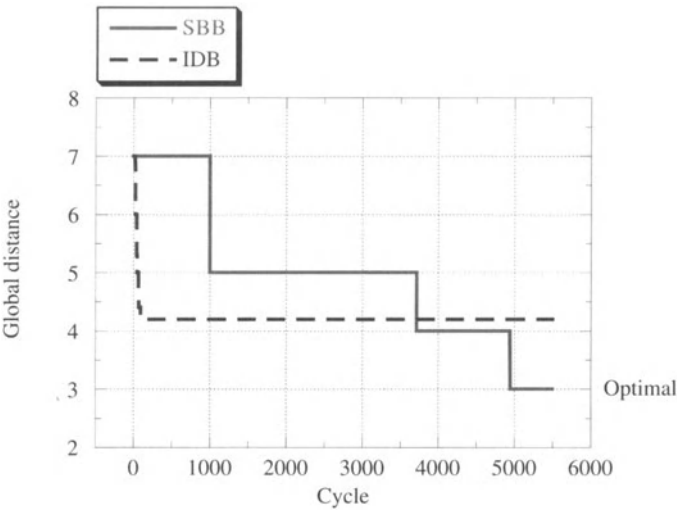

Fig. 8.6. Anytime Curve for Instance of $\langle 10, 10, 27/45, 0.9 \rangle$

Anytime Curves. Next, we compared the IDB with the the SBB in terms of *anytime curves*. An anytime curve illustrates how the global distance (maximum number of constraint violations over agents) of the best solution found so far improves as time proceeds. We show an anytime curve for each algorithm on the x-y plane, with the x axis being the number of cycles passed by and the y axis being the global distance of the best solution found so far.

The thick line in Fig. 8.6 shows an anytime curve using the SBB for an instance of a class of $n = m = 10, p_1 = 27/45, p_2 = 0.9$. For this instance, the SBB finds an optimal solution with the minimum cycles. The dotted line shows an anytime curve for the IDB with the same instance. For the IDB, the global distance at a certain cycle is averaged over the results of 10 trials with the same instance.

As shown in Fig. 8.6, while the curve of the SBB eventually converges to the optimal distance, it declines relatively slowly. The IDB, on the other hand, has a rapid drop at the beginning, and after that keeps steady at a *nearly optimal distance*. That is not peculiar to this instance but can be seen in other instances of this class or other classes. We conducted the same

experiment with other classes and measured the number of cycles the IDB consumes to reach a nearly optimal distance. We also measured the number of cycles the SBB consumes to outperform the nearly optimal distance. Table 8.2 shows for each class the measured number of cycles for the nearly optimal distance with the real optimal distance in parentheses. We can see that the IDB reaches the nearly optimal distance much sooner than does the SBB for all classes.

Table 8.2. Cycles to Find Nearly-Optimal Solutions

problem class	nearly-optimal (optimal)	cycle for IDB	cycle for SBB
$\langle 10, 10, 18/45, 0.8\rangle$	2.2 (1)	100	417
$\langle 10, 10, 18/45, 0.9\rangle$	3.6 (2)	46	585
$\langle 10, 10, 27/45, 0.8\rangle$	3.6 (2)	508	2052
$\langle 10, 10, 27/45, 0.9\rangle$	4.2 (3)	196	3716
$\langle 10, 10, 36/45, 0.8\rangle$	4.6 (3)	3416	6360
$\langle 10, 10, 36/45, 0.9\rangle$	6.0 (4)	344	200145
$\langle 10, 10, 45/45, 0.8\rangle$	5.8 (4)	438	258753
$\langle 10, 10, 45/45, 0.9\rangle$	7.3 (6)	90	45696

8.4 Distributed Hierarchical CSPs

8.4.1 Problem Formalization

A distributed hierarchical CSP is a problem where each agent tries to find variable values that minimize the maximal degree of the importance of violated constraints over agents. In this problem, each constraint is labeled a non-negative value, called *importance value*, that represents a degree of importance of the constraint. As the importance value of a constraint becomes larger, the constraint is more important.

A distributed hierarchical CSP corresponds to finding an optimal solution to the following distributed partial CSP. We assume possible importance values of constraints are represented as $\alpha_0, \alpha_1, \ldots$, where $\alpha_i < \alpha_j$ holds for all $i < j$, and $\alpha_0 = 0$.

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

We show an example problem of a hierarchical distributed CSP in Fig. 8.7. This problem is to place three queens in a 3×3 chess board so that these

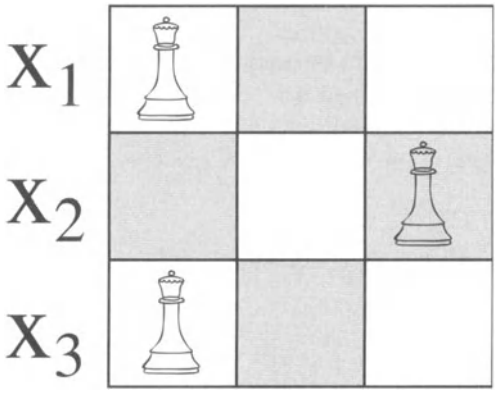

Fig. 8.7. 3-Queens Problem

queens do not threaten each other. This problem can be formalized as a distributed CSP, where there exist three agents, each of which has a variable x_1, x_2, and x_3, respectively, and the domain of each variable is $\{1, 2, 3\}$. 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 Fig. 8.7 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.

8.4.2 Asynchronous Incremental Relaxation

Basic Asynchronous Incremental Relaxation. The asynchronous backtracking algorithm described in Chapter 3 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[1], then try to find a solution that satisfies all constraints whose importance values are greater than the new *threshold*, and so on.

By this method, 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., $p_k(x_i) \equiv x_i \leq k$, where $k \in \{1000, 2000, 3000, \ldots, 10000\}$ and the importance value of p_k is k. These additional constraints represent our preference 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.

An alternative algorithm is 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.

Asynchronous Incremental Relaxation with 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 Fig. 8.7, 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 cannot 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 α_k. The importance value of a nogood represents the dependencies between this nogood and constraints that contribute to this

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

nogood, i.e., when all constraints whose importance values are equal to α_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, respectively, are relaxed. Therefore, the importance value of a nogood $\{(x_1, 1), (x_2, 3)\}$ is 1/4. By using the importance values 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 α_k, agents can tell that the new *threshold* should be α_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 α_k (i.e., relaxing constraints whose importance values are less than α_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 Fig. 8.7, 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 a 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 Fig. 8.7, 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 importance 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 different from the dependencies used in dependency-directed backtracking (Mackworth 1992). 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 dynamically changed by constraint relaxation.

In this algorithm, agents act concurrently and asynchronously rather than in a predefined sequential order. We show the procedures that are invoked at agent x_i by the reception of two kinds of messages (*ok?*, *revise_threshold*) in Fig. 8.8 (i) and (ii), respectively. The procedure that is invoked by the reception of a *nogood* message is basically identical to that for the asynchronous backtracking algorithm.

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. After that, the agent waits for and responds to messages.
- Agent x_i sends its value assignment (*current_value*) to outgoing links.
- On the other hand, agent x_i receives *ok?* messages from incoming links. Agent x_i puts these values in the *agent_view* (Fig. 8.8 (i)). It then tries to find a value consistent with its *agent_view*.
- 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*. (Fig. 8.8 (iv-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* (Fig. 8.8 (iii-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 nogood 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 received (**ok?**, (x_j, d_j)) **do** — (i)
  add (x_j, d_j) to *agent_view*;
  check_agent_view; **end do**;

when received (**revise_threshold**, x_j,*new_threshold*) **do** — (ii)
  when *new_threshold* > *threshold* **do**
    threshold ← *new_threshold*;
    send (**revise_threshold**, x_i,*new_threshold*) to other agents except x_j;
  end do; **end do**;

procedure **check_agent_view** — (iii)
  when F_i(*agent_view* ∪ {(x_i, *current_value*)}) > *threshold* **do**
    if there exists $d \in D_i$ where F_i(*agent_view* ∪ { (x_i, d)}) ≤ *threshold* **then**
      current_value ← d; — (iii - a)
      send (**ok?**, (x_i, d)) to outgoing links;— (iii - b)
    else backtrack; **check_agent_view**; **end if**; **end do**;

procedure **backtrack** — (iv)
  V ← a subset of *agent_view* where $\min_{d \in D_i} F_i(V \cup \{(x_i, d)) >$ *threshold*;
  new_importance_value ← $\min_{d \in D_i} F_i(V \cup \{(x_i, d)\})$ — (iv - a)
  if V={} **then** *threshold* ← *new_importance_value*;
      send (**revise_threshold**, x_i, *threshold*) to other agents; — (iv - b)
  else select (x_j, v_j) from V where j has the lowest priority;
      send (**nogood**, x_i, V, *new_importance_value*) to x_j;— (iv - c)
      remove (x_j, v_j) from *agent_view*; **end if**;

Fig. 8.8. Procedures for Receiving Messages (Asynchronous Incremental Relaxation)

- A *nogood* message is coupled with its importance value. The importance value of a new nogood is obtained by the minimal value of F_i among possible values (Fig. 8.8 (iv-a)).
- When an empty nogood is found, agents exchange *revise_threshold* messages and revise the *threshold* (Fig. 8.8 (iv-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 wait for messages.

8.4.3 Example of Algorithm Execution

We show an example of an algorithm execution using the distributed three-queens problem. In Fig. 8.9, 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 as 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 (Fig. 8.9 (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 (Fig. 8.9 (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 (Fig. 8.9 (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 (Fig. 8.9 (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*.).

8.4.4 Algorithm Completeness

By the completeness of the asynchronous backtracking algorithm, 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.

8.4.5 Evaluations

We experimentally examine the effect of introducing importance values of nogoods. 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 variables $i = 1,\ldots,n/2$, and $\{n/2+1,\ldots,n\}$ for variables

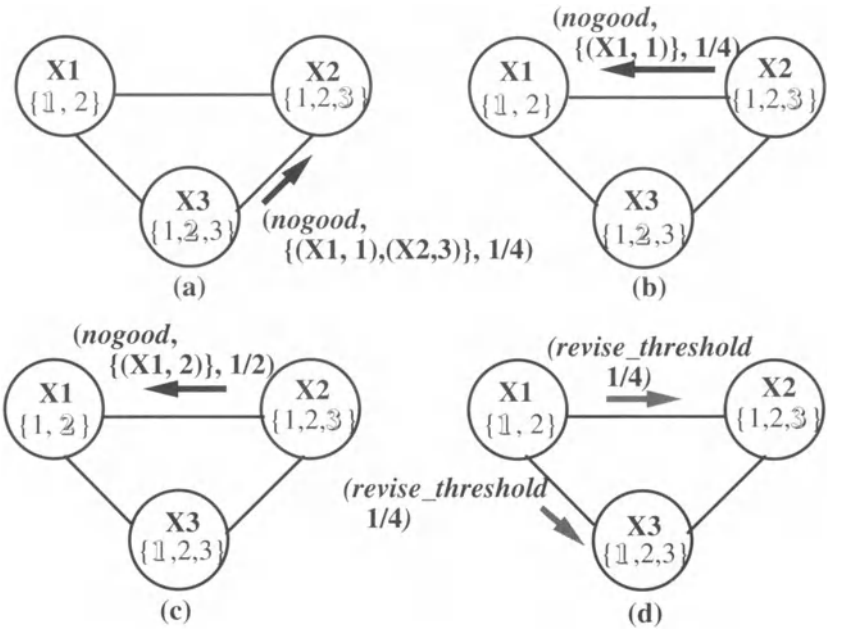

Fig. 8.9. Example of Algorithm Execution (Asynchronous Incremental Relaxation)

$i = n/2 + 1, \ldots, n$ (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 Fig. 8.7. Since this problem is over-constrained, some constraints must be relaxed.

Figure 8.10 shows the required cycles 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 throw away all nogoods after each iteration. This figure shows that the required cycles for the AIR with nogood dependency is about 1/5 of the cycles required for the basic AIR. Since the overhead of maintaining nogood dependency is very small, we can expect that the reduction in computation time will be proportional to this reduction in required cycles.

Figure 8.11 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 $1/72 \approx 0.014$.

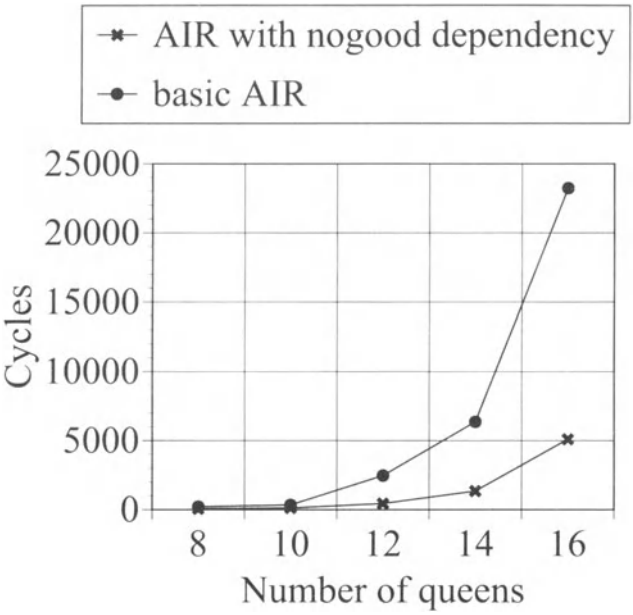

Fig. 8.10. Required Cycles for Over-Constrained Distributed n-Queens Problems

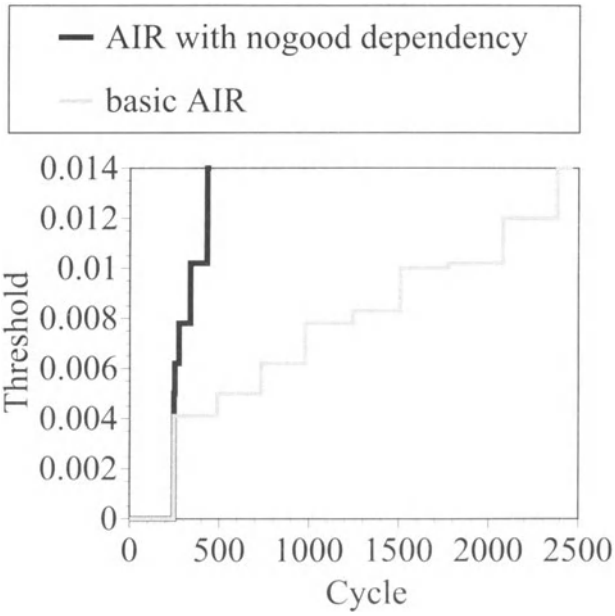

Fig. 8.11. Threshold Changes in Over-Constrained Distributed 12-Queens Problem

This figure shows that the AIR with nogood dependency avoids wasteful *threshold* values compared with the basic AIR. Furthermore, the required number of cycles 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 cycles for the search at *threshold*=$1/162 \approx 0.0062$, while the basic AIR requires 249 cycles.

8.5 Summary

In this chapter, we showed a formalization for a distributed partial CSP, where agents settle for a solution to a relaxed problem of an over-constrained distributed CSP. We also provided two classes of problems in this model, i.e., distributed maximal CSPs and distributed hierarchical CSPs, and gave outlines of algorithms for solving them. We described the synchronous branch and bound algorithm, and the iterative distributed breakout algorithm for solving distributed maximal CSPs. Finally, we described an asynchronous incremental relaxation algorithm for solving distributed hierarchical CSPs.

9. Summary and Future Issues

In this book, we gave an overview of the research on distributed CSPs. In Chapter 1, we showed the problem definition of normal, centralized CSPs and described various algorithms for solving CSPs, including complete, systematic search algorithms called backtracking algorithms, hill-climbing algorithms called iterative improvement algorithms, and consistency algorithms. In particular, we described the weak-commitment search algorithm, which is a hybrid-type algorithm of backtracking and iterative improvement, and investigated which kinds of problem instances are most difficult for hill-climbing algorithms. Also, we showed an extension of the basic CSP formalization called partial CSPs.

In Chapter 2, we showed the problem definition of distributed CSPs and described various MAS application problems that can be mapped into the distributed CSP formalization. These problems include recognition problems, resource allocation problems, multi-agent truth maintenance tasks, and scheduling/time-tabling tasks.

From Chapter 3 to Chapter 5, we presented distributed search algorithms for solving distributed CSPs. In Chapter 3, we presented the asynchronous backtracking algorithm, which is a basic backtracking algorithm for solving distributed CSPs. This algorithm allows agents to act concurrently and asynchronously without any global control, while guaranteeing the completeness of the algorithm.

In Chapter 4, we presented the asynchronous weak-commitment search algorithm, which is a distributed version of the weak-commitment search algorithm. In this algorithm, when an agent cannot find a value consistent with the higher priority agents, the priority order is changed so that the agent has the highest priority. As a result, when an agent makes a mistake in selecting a value, the priority of another agent becomes higher; thus the agent that made the mistake will not commit to the bad decision, and the selected value is changed. Experimental evaluations showed that this algorithm can solve large-scale problems that cannot be solved by the asynchronous backtracking algorithm within a reasonable amount of time.

In Chapter 5, we presented the distributed breakout algorithm, which is an iterative improvement algorithm for solving distributed CSPs. In this algorithm, neighboring agents exchange values of possible improvements so

M. Yokoo, *Distributed Constraint Satisfaction*

that only the agent that can maximally improve the evaluation value can change its variable value, and agents detect quasi-local-minima instead of real local-minima. Experimental evaluations showed that the distributed breakout algorithm is more efficient than the asynchronous weak-commitment search algorithm when problem instances are particularly difficult.

In Chapter 6, we described distributed consistency algorithms that are implemented by using a distributed ATMS. In a distributed ATMS, each agent has its own ATMS, and these agents communicate hypothetical inference results and nogoods among themselves. Experimental evaluations showed that distributed consistency algorithms are less effective for distributed CSPs when agents can act concurrently in the search algorithms for solving distributed CSPs.

In Chapter 7, we investigated the case where each agent has multiple local variables. We presented a modified version of the asynchronous weak-commitment search algorithm, in which each agent sequentially performs the computation for each variable, and communicates with other agents only when it can find a local solution that satisfies all local constraints. Experimental evaluations using example problems showed that this algorithm is far more efficient than an algorithm that employs the prioritization among agents, or than a simple extension of the asynchronous weak-commitment search algorithm.

In Chapter 8, we extended the formalization of distributed CSPs so that we can handle the case where constraints are too tight so that no solution satisfies all constraints completely. This formalization is called a distributed partial CSP, where agents settle for a solution to a relaxed problem. Also, we showed two special cases of this general model, i.e., distributed maximal CSPs and distributed hierarchical CSPs. We showed that agents can find an optimal solution with a given criterion by extending algorithms for solving distributed CSPs.

There are many remaining research issues concerning distributed CSPs. First, more work is needed to develop better algorithms, especially for the cases of multiple local variables and distributed partial CSPs. Also, we cannot expect that a single algorithm can efficiently solve all types of problems. More theoretical/experimental evaluations are needed to clarify which algorithm is most appropriate to which problem instances.

For centralized CSPs, as discussed in Chapter 1, it is well known that the most difficult problem instances are in the phase-transition region, where the provability that a problem instance has a solution is about 0.5. On the other hand, we do not have a clear idea about which 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.

Furthermore, 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 (Russel and Wefald 1991; Dechter and Dechter 1988; Verfaillie and Schiex 1994; Bessière 1991; Prosser, Conway, and Muller 1992) should be incorporated into the distributed CSP formalization.

References

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.

Bessière, C. (1991). Arc-consistency in dynamic constraint satisfaction problem. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 221–226.

Box, F. (1978). A heuristic technique for assignment frequencies to mobile radio nets. *IEEE Transactions on Vehicular Technology 27*(2), 57–64.

Carlsson, M. and M. Grindal (1993). Automatic frequency assignment for cellular telephones using constraint satisfaction techniques. In *Proceedings of the Tenth International Conference on Logic Programming*, pp. 647–663.

Chandy, K. and L. Lamport (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems 3*(1), 63–75.

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.

Clark, D. A., J. Frank, I. P. Gent, E. MacIntyre, N. Tomov, and T. Walsh (1996). Local search and the number of solutions. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP-96)*, pp. 119–133. Springer-Verlag. Lecture Notes in Computer Science 1118.

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.

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.

Cook, S. (1971). The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation*, pp. 151–158.

Davenport, A., E. Tsang, C. J. Wang, and K. Zhu (1994). Genet: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 325–330.

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.

Dechter, R. (1992). Constraint networks. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence*, pp. 276–285. New York: Wiley-Interscience Publication.

Dechter, R. and A. Dechter (1988). Belief maintenance in dynamic constraint satisfaction problem. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 37–42.

Dechter, R. and I. Meiri (1989). Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 271–277.

Dechter, R. and J. Pearl (1988). Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence 34*(1), 1–38.

Descotte, Y. and J. Latombe (1985). Making compromises among antagonistic constraints in planner. *Artificial Intelligence 27*(1), 183–217.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence 12*, 231–272.

Durfee, E. H., V. R. Lesser, and D. D. Corkill (1992). Distributed problem solving. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence*, pp. 379–388. New York: Wiley-Interscience Publication.

Fox, M. S. (1987). *Constraint-directed search: a case study of job-shop scheduling.* Morgan Kaufman.

Fox, M. S., N. Sadeh, and C. Baykan (1989). Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 309–321.

Frank, J., P. Cheeseman, and J. Stutz (1997). When gravity fails: Local search topology. *Journal of Artificial Intelligence Research 7*, 249–281.

Freeman-Benson, B., J. Maloney, and A. Borning (1990). An incremental constraint solver. *Communications of the ACM 33*(1), 54–62.

Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications ACM 21*(11), 958–966.

Freuder, E. C. and R. J. Wallace (1992). Partial constraint satisfaction. *Artificial Intelligence 58*(1–3), 21–70.

Funabiki, N. and Y. Takefuji (1992). A neural network parallel algorithm for channel assignment problems in cellular radio networks. *IEEE Transactions on Vehicular Technology 41*(4), 430–437.

Gamst, A. (1986). Some lower bounds for a class of frequency assignment problems. *IEEE Transactions on Vehicular Technology 35*(1), 8–14.

Gasching, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 457.

Geffner, H. and J. Pearl (1987). An improved constraint-propagation algorithm for diagnosis. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1105–1111.

Gent, I. P. and T. Walsh (1993). Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 28–33.

Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research 1*, 25–46.

Hale, W. K. (1980). Frequency assignment: Theory and application. *Proceedings of the IEEE 68*(12), 1497–1513.

Hao, J. K., R. Dorne, and P. Galinier (1998). Tabu search for frequency assignment in mobile radio networks. *Journal of Heuristics 4*, 47–62.

Haralick, R. and G. L. Elliot (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, 263–313.

Hertz, A., B. Jaumard, and M. P. de Aragao (1994). Local optima topology for the k-coloring problem. *Discrete Applied Mathematics 49*, 257–280.

Hirayama, K. and J. Toyoda (1995). Forming coalitions for breaking deadlocks. In *Proceedings of the First international Conference on Multi-agent Systems*, pp. 155–162. MIT Press.

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. Springer-Verlag. Lecture Notes in Computer Science 1330.

Hogg, T. (1996). Quantum computing and phase transitions in combinatorial search. *Journal of Artificial Intelligence Research 4*, 91–128.

Hogg, T., B. A. Huberman, and C. P. Williams (1996). Phase transitions and the search problem. *Artificial Intelligence 81*(1–2), 1–16.

Huffman, D. A. (1971). Impossible objects as nonsense sentences. In R. Meltzer and D. Michie (Eds.), *Machine Intelligence 6*, pp. 295–323. Elsevier.

Huhns, M. N. and D. M. Bridgeland (1991). Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics 21*(6), 1437–1445.

Hurley, S., D. H. Smith, and S. U. Thiel (1997). FASoft: A system for discrete channel frequency assignment. *Radio Science 32*(5), 1921–1939.

Ishida, T. (1996). Real-time bidirectional search: Coordinated problem solving in uncertain situations. *IEEE Transactions on Pattern Analysis and Machine Intelligence 18*(6), 617–628.

Kautz, H. and B. Selman (1992). Planning as satisfiability. In *Proceedings of Tenth European Conference on Artificial Intelligence*, pp. 360–363.

Kim, S. and S. L. Kim (1994). A two-phase algorithm for frequency assignment in cellular mobile systems. *IEEE Transactions on Vehicular Technology 43*(3), 542–548.

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine Spring*, 32–44.

Kunz, D. (1991). Channel assignment for cellular radio using neural networks. *IEEE Transactions on Vehicular Technology 40*(1), 188–193.

Larrosa, J. and P. Meseguer (1996). Phase transition in MAX-CSP. In *Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96)*, pp. 190–194.

Lesser, V. R. (1991). A retrospective view of FA/C distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics 21*(6), 1347–1362.

Lesser, V. R. and D. D. Corkill (1981). Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man and Cybernetics 11*(1), 81–96.

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. MIT Press.

Mackworth, A. K. (1992). Constraint satisfaction. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence*, pp. 285–293. New York: Wiley-Interscience Publication.

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*, pp. 293–318. Morgan Kaufmann.

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.

Mitchell, D., B. Selman, and H. Levesque (1992). Hard and easy distributions of SAT problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 459–465.

Morris, P. (1993). The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 40–45.

Nishibe, Y., K. Kuwabara, T. Ishida, and M. Yokoo (1994). Speed-up of distributed constraint satisfaction and its application to communication network path assignments. *Systems and Computers in Japan 25*(12), 54 – 67.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

Pennotti, R. J. and R. R. Boorstyn (1976). Channel assignments for cellular mobile telecommunications systems. In *Proceedings of National Telecommunications Conference*, pp. 16:5–1–16:5–5.

Prosser, P., C. Conway, and C. Muller (1992). A constraint maintenance system for the distributed allocation problem. *Intelligent Systems Engineering 1*, 76–83.

Rosenkrantz, D., R. Stearns, and P. Lewis (1978). System level concurrency control for distributed database systems. *ACM Trans. on Database Systems 3*(2), 178–198.

Russel, S. and E. Wefald (1991). *Do the Right Thing.* MIT Press.

Selman, B. and H. Kautz (1993). Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 290–295.

Selman, B., H. Levesque, and D. Mitchell (1992). A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 440–446.

Sivarajan, K. N., R. J. McEliece, and J. W. Ketchum (1989). Channel assignment in cellular radio. In *Proceedings of 39th IEEE Vehicular Technology Society Conference*, pp. 846–850.

Smith, D. H., S. Hurley, and S. U. Thiel (1998). Improving heuristics for the frequency assignment problem. *European Journal of Operational Research 107*, 76–86.

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.

Stallman, R. and G. J. Sussman (1977). Forward reasoning and dependency-directed backtracking. *Artificial Intelligence 9*(2), 135–196.

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.

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press.

Verfaillie, G. and T. Schiex (1994). Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 307–312.

Wallace, R. J. and E. C. Freuder (1993). Conjunctive width heuristics for maximal constraint satisfaction. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 762–768.

Walser, J. P. (1996). Feasible cellular frequency assignment using constraint programming abstractions. In *Proceedings of the CP'96 Workshop on Constraint Programming Applications*, pp. 105–114.

Waltz, D. (1975). Understanding line drawing of scenes with shadows. In P. Winston (Ed.), *The Psychology of Computer Vision*, pp. 19–91. McGraw-Hill.

Yamaguchi, H., H. Fujii, Y. Yamanaka, and I. Yoda (1989). Network configuration management database. *NTT R & D 38*(12), 1509–1518.

Yokoo, M. (1993). Constraint relaxation in distributed constraint satisfaction problem. In *5th International Conference on Tools with Artificial Intelligence*, pp. 56–63.

Yokoo, M. (1994). Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 313–318.

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. Springer-Verlag. Lecture Notes in Computer Science 976.

Yokoo, M. (1997). Why adding more constraints makes a problem easier for hill-climbing algorithms: Analyzing landscapes of CSPs. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP-97)*, pp. 356–370. Springer-Verlag. Lecture Notes in Computer Science 1330.

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.

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.

Yokoo, M. and K. Hirayama (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pp. 401–408. MIT Press.

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*. IEEE Computer Society Press.

Yokoo, M., T. Ishida, and K. Kuwabara (1990). Distributed constraint satisfaction for DAI problems. In *10th International Workshop on Distributed Artificial Intelligence*.

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.