# 4 Search Algorithms for Agents

**Makoto Yokoo and Toru Ishida**

## 4.1 Introduction

In this chapter, we introduce several search algorithms that are useful for problem solving by multiple agents. Search is an umbrella term for various problem solving techniques in AI. In search problems, the sequence of actions required for solving a problem cannot be known *a priori* but must be determined by a trial-and-error exploration of alternatives. Since virtually all AI problems require some sort of search, search has a long and distinguished history in AI.

The problems that have been addressed by search algorithms can be divided into three classes: path-finding problems, constraint satisfaction problems, and two-player games.

A typical example of the first class, i.e., path-finding problems, is a puzzle called the *n-puzzle*. Figure 4.1 shows the 8-puzzle, which consists of eight numbered tiles arranged on a $3 \times 3$ board (in a generalized case, there are $n = k^2 - 1$ tiles on a $k \times k$ board). The allowed moves are to slide any tile that is horizontally or vertically adjacent to the empty square into the position of the empty square. The objective is to transform the given initial configuration to the goal configuration by making allowed moves. Such a problem is called a path-finding problem, since the objective is to find a path (a sequence of moves) from the initial configuration to the goal configuration.

A constraint satisfaction problem (CSP) involves finding a goal configuration rather than finding a path to the goal configuration. A typical example of a CSP is a puzzle called 8-queens. The objective is to place eight queens on a chess board (8×8 squares) so that these queens will not threaten each other. This problem is called a constraint satisfaction problem since the objective is to find a configuration that satisfies the given conditions (constraints).

Another important class of search problems is two-player games, such as chess. Since two-player games deal with situations in which two *competitive* agents exist, it is obvious that these studies have a very close relation with DAI/multiagent systems where agents are competitive.

On the other hand, most algorithms for the other two classes (constraint satisfaction and path-finding) were originally developed for single-agent problem solving.

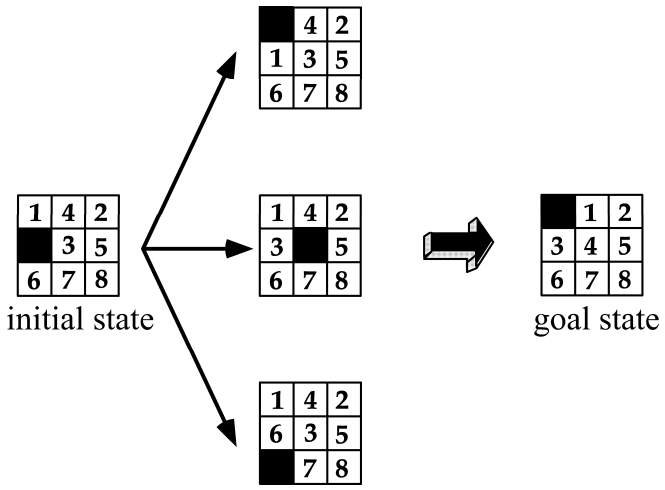

**Figure 4.1** Example of a path-finding problem (8-puzzle).

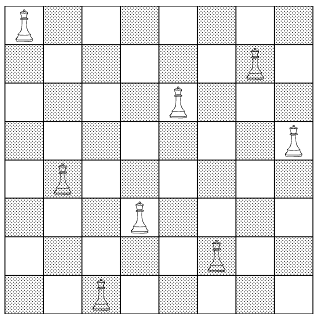

**Figure 4.2** Example of a constraint satisfaction problem (8-queens).

Among them, what kinds of algorithms would be useful for cooperative problem solving by multiple agents?

In general, an agent is assumed to have *limited rationality.* More specifically, the computational ability or the recognition ability of an agent is usually limited. Therefore, getting the complete picture of a given problem may be impossible. Even if the agent can manage to get complete information on the problem, dealing with the global information of the problem can be too expensive and beyond the computational capability of the agent. Therefore, the agent must do a limited amount of computations using only partial information on the problem and then take appropriate actions based on the available resources.

In most standard search algorithms (e.g., the A* algorithm [20] and backtracking algorithms [26]), each step is performed sequentially, and for each step, the global knowledge of the problem is required. For example, the A* algorithm extends the wavefront of explored states from the initial state and chooses the most promising state within the whole wavefront.

On the other hand, a search problem can be represented by using a graph, and there exist search algorithms with which a problem is solved by accumulating local computations for each node in the graph. The execution order of these local computations can be arbitrary or highly flexible, and can be executed asynchronously and concurrently. We call these algorithms *asynchronous search* algorithms.

When a problem is solved by multiple agents each with limited rationality, asynchronous search algorithms are appropriate based on the following reasons.

- We can assume that the computational and recognition abilities required to perform the local computations of each node will be small enough for the agents. On the other hand, if each step of the algorithm requires the global knowledge of the problem, it may be beyond the capability of an agent.
- If multiple agents are cooperatively solving a problem using the asynchronous search algorithm, the execution order of these agents can be highly flexible or arbitrary. Otherwise, we need to synchronize the computations of the agents, and the overhead for such control can be very high.

The importance of solving a problem by combining such local and asynchronous computations was first pointed out by Lesser [24], and this idea has been widely acknowledged in DAI studies.

In the following, we give a formal definition of a constraint satisfaction problem and a path-finding problem and introduce asynchronous search algorithms for solving these problems. Then, we show the formalization of and algorithms for two-player games.

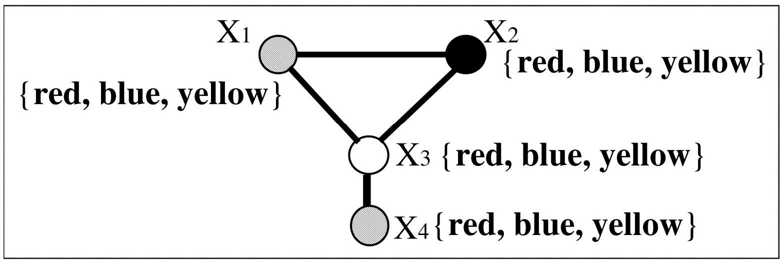

**Figure 4.3** Example of a constraint satisfaction problem (graph-coloring).

## 4.2 Constraint Satisfaction

### 4.2.1 Definition of a Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a problem to find a consistent value assignment of variables that take their values from finite, discrete domains. Formally, a CSP consists of $n$ variables $x_1, x_2, \ldots, x_n$, whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively, and a set of constraints on their values. A constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \ldots, x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied. Since constraint satisfaction is NP-complete in general, a trial-and-error exploration of alternatives is inevitable.

For example, 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 $\{1, 2, \ldots, 8\}$. A solution is a combination of values of these variables. The constraints that the queens will 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 (Figure 4.3). 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.

If all constraints are binary (i.e., between two variables), a CSP can be represented as a graph, in which a node represents a variable, and a link between nodes represents a constraint between the corresponding variables. Figure 4.4 shows a constraint graph 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 focus our attention on binary CSPs in

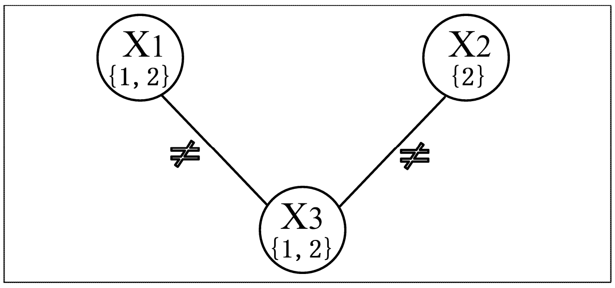

**Figure 4.4** Constraint graph.

the following chapter. However, the algorithms described in this chapter are also applicable to non-binary CSPs.

Then, how can the CSP formalization be related to DAI? Let us assume that the variables of a CSP are distributed among agents. Solving a CSP in which multiple agents are involved (such a problem is called a *distributed* CSP) can be considered as achieving coherence among the agents. Many application problems in DAI, e.g., interpretation problems, assignment problems, and multiagent truth maintenance tasks, can be formalized as distributed CSPs.

An interpretation problem can be viewed as a problem to find a compatible set of hypotheses that correspond to the possible interpretations of input data. An interpretation problem can be mapped into a CSP by viewing possible interpretations as possible variable values. If there exist multiple agents, and each of them is assigned a different part of the input data, such a problem can be formalized as a distributed CSP. The agents can eliminate the number of hypotheses by using the filtering algorithm or the hyper-resolution-based consistency algorithm, both of which are described in the following.

If the problem is to allocate tasks or resources to multiple 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. Furthermore, we can formalize multiagent truth maintenance tasks described in Chapter 2 as a distributed CSP, where each item of the uncertain data is represented as a variable whose value can be IN or OUT.

In the following, we describe asynchronous search algorithms in which each process corresponds to a variable, and the processes act asynchronously to solve a CSP.

We assume the following communication model.

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

Furthermore, we call the processes that have links to $x_i$ *neighbors* of $x_i$. We assume that a process knows the identifiers of its neighbors.

### 4.2.2 Filtering Algorithm

In the filtering algorithm [36], each process communicates its domain to its neighbors and then removes values that cannot satisfy constraints from its domain. More specifically, a process $x_i$ performs the following procedure **revise**$(x_i, x_j)$ for each neighboring process $x_j$.

```
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;
```

If some value of the domain is removed by performing the procedure **revise**, process $x_i$ sends the new domain to neighboring processes. If $x_i$ receives a new domain from a neighboring process $x_j$, the procedure **revise**$(x_i, x_j)$ is performed again. The execution order of these processes is arbitrary.

We show an example of an algorithm execution in Figure 4.5. The example problem is a smaller version of the 8-queens problem (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 exchanging the domains (Figure 4.5 (a)), $x_1$ performs **revise**$(x_1, x_2)$ and removes 2 from its domain (if $x_1 = 2$, none of $x_2$'s values satisfies the constraint with $x_1$). Similarly, $x_2$ performs **revise**$(x_2, x_3)$, $x_3$ performs **revise**$(x_3, x_2)$, and each process removes 2 from its domain. After exchanging the new domains (Figure 4.5 (b)), $x_1$ performs **revise**$(x_1, x_3)$, and removes 1 and 3 from its domain. The domain of $x_1$ then becomes an empty set, so the process discovers that this problem has no solution.

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 4.6 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 process can remove a value from its domain. Furthermore, in the 8-queens problem (which has many solutions), no process can remove a value from its domain 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.

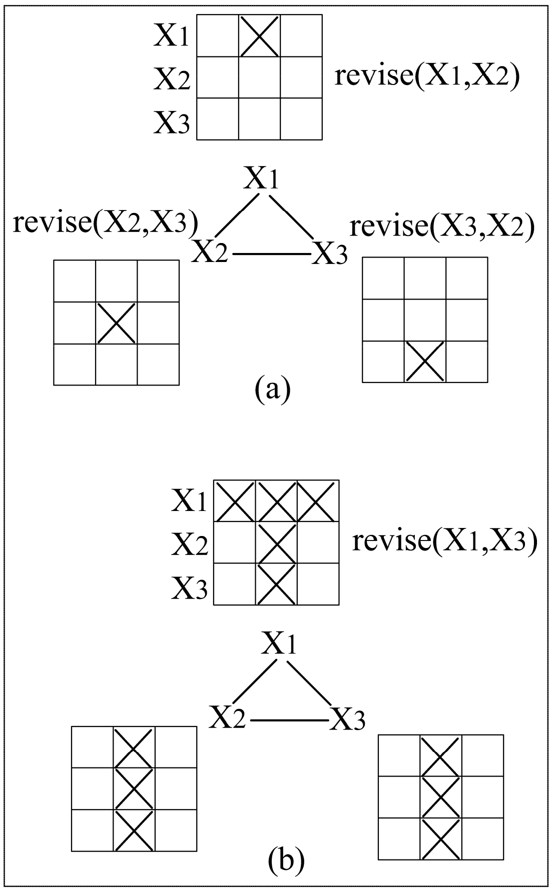

**Figure 4.5** Example of an algorithm execution (filtering).

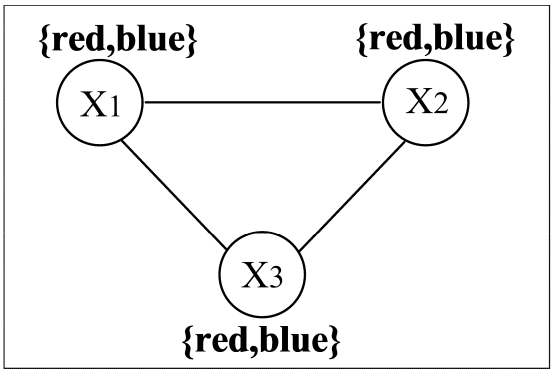

**Figure 4.6** Example that the filtering algorithm cannot solve.

### 4.2.3 Hyper-Resolution-Based Consistency Algorithm

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 [9]. A CSP is k-consistent iff the following condition is satisfied.

- Given any instantiation of any $k-1$ variables satisfying all the constraints among those variables, it is possible to find an instantiation of any $k$th 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 (having the same solutions as the original problem) k-consistent problem. If the problem is k-consistent and 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.

In the following, we describe a consistency algorithm using the the hyper-resolution rule [6]. In this algorithm, all constraints are represented as a nogood, which is a prohibited combination of variable values. For example, in Figure 4.6, 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 Figure 4.6, 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 $x_1 = 1$).

$$\begin{array}{l}
A_1 \vee A_2 \vee \ldots \vee A_m \\
\neg(A_1 \wedge A_{11} \ldots), \\
\neg(A_2 \wedge A_{21} \ldots), \\
\vdots \\
\neg(A_m \wedge A_{m1} \ldots) \\
\hline
\neg(A_{11} \wedge \ldots \wedge A_{21} \wedge \ldots \wedge A_{m1} \ldots)
\end{array}$$

In the hyper-resolution-based consistency algorithm, each process represents its constraints as nogoods. The process then generates new nogoods by combining the information about its domain and existing nogoods using the hyper-resolution rule. A newly obtained nogood is communicated to related processes. If a new nogood is communicated, the process tries to generate further new nogoods using the communicated nogood.

For example, in Figure 4.6, assume $x_1$ generates a new nogood $\{x_2 = red, x_3 = blue\}$ using nogood $\{x_1 = red, x_2 = red\}$ and nogood $\{x_1 = blue, x_3 = blue\}$. This nogood is communicated to $x_2$ and $x_3$. $x_2$ generates a new nogood $\{x_3 = blue\}$ using this communicated nogood and nogood $\{x_2 = blue, x_3 = blue\}$. Similarly, $x_1$ generates a new nogood $\{x_2 = blue, x_3 = red\}$ from $\{x_1 = blue, x_2 = blue\}$ and $\{x_1 = red, x_3 = red\}$. $x_2$ generates a new nogood $\{x_3 = red\}$ using this nogood and nogood $\{x_2 = red, x_3 = red\}$. Then, $x_3$ can generate $\{\}$ from nogood$\{x_3 = blue\}$ and $\{x_3 = red\}$, which is an empty set. 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 whose lengths (the length of a nogood is the number of variables that constitute the nogood) are less than $k$ are produced, the problem becomes strongly k-consistent.

### 4.2.4 Asynchronous Backtracking

The asynchronous backtracking algorithm [39] is an asynchronous version of a backtracking algorithm, which is a standard method for solving CSPs. In the asynchronous backtracking algorithm, the priority order of variables/processes is determined, and each process communicates its tentative value assignment to neighboring processes. The priority order is determined by alphabetical order of the variable identifiers, i.e., preceding variables in the alphabetical order have higher priority. A process changes its assignment if its current value assignment is not consistent with the assignments of higher priority processes. If there exists no value that is consistent with the higher priority processes, the process generates a new nogood, and communicates the nogood to a higher priority process; thus the higher priority process changes its value.

The generation procedure of a new nogood is basically identical to the hyper-resolution rule described in Section 4.2.3. However, in the consistency algorithm, all constraints (nogoods) are considered for generating new nogoods. On the other hand, the asynchronous backtracking algorithm generates only the constraints that are not satisfied in the current situation. In other words, a new nogood is generated only if the nogood actually occurs in the asynchronous backtracking.

Each process maintains the current value assignment of other processes from its viewpoint (local_view). It must be noted that since each process acts asynchronously and concurrently and processes communicate by sending messages, the local_view

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

when received (nogood, nogood) do — (ii)
  record nogood as a new constraint;
  when ($x_k$, $d_k$) where $x_k$ is not a neighbor do
    request $x_k$ to add $x_i$ to its neighbors;
    add $x_k$ to neighbors;
    add ($x_k$, $d_k$) to local_view; end do;
  check_local_view;
end do;

procedure check_local_view
  when local_view and current_value are not consistent do
    if no value in $D_i$ is consistent with local_view
      then resolve a new nogood using hyper-resolution rule
        and send the nogood to the lowest priority process in the nogood;
        when an empty nogood is found do
          broadcast to other processes that there is no solution,
          terminate this algorithm; end do;
      else select $d \in D_i$ where local_view and $d$ are consistent;
        current_value ← $d$;
        send (ok?, ($x_i$, $d$)) to neighbors; end if; end do;
```

**Algorithm 4.1** Procedures for receiving messages (asynchronous backtracking).

may contain obsolete information. Even if $x_i$'s local_view says that $x_j$'s current assignment is 1, $x_j$ may already have changed its value. Therefore, if $x_i$ does not have a consistent value with the higher priority processes according to its local_view, we cannot use a simple control method such as $x_i$ orders a higher priority process to change its value, since the local_view may be obsolete. Therefore, each process needs to generate and communicate a new constraint (nogood), and the receiver of the new nogood must check whether the nogood is actually violated from its own local_view.

The main message types communicated among processes are *ok?* messages to communicate the current value, and *nogood* messages to communicate a new nogood. The procedures executed at process $x_i$ after receiving an *ok?* message and a *nogood* message are described in Algorithm 4.1 (i) and Algorithm 4.1 (ii), respectively.

We show an example of an algorithm execution in Figure 4.7. In Figure 4.7 (a), after receiving *ok?* messages from $x_1$ and $x_2$, the *local_view* of $x_3$ will be $\{(x_1, 1), (x_2, 2)\}$. Since there is no possible value for $x_3$ consistent with this *local_view*, a new nogood $\{(x_1, 1), (x_2, 2)\}$ is generated. $x_3$ chooses the lowest priority process in the nogood, i.e., $x_2$, and sends a *nogood* message. By receiving this *nogood* message, $x_2$ records this nogood. This nogood, $\{(x_1, 1), (x_2, 2)\}$, contains process $x_1$, which is not a neighbor $x_2$. Therefore, a new link must be added be-

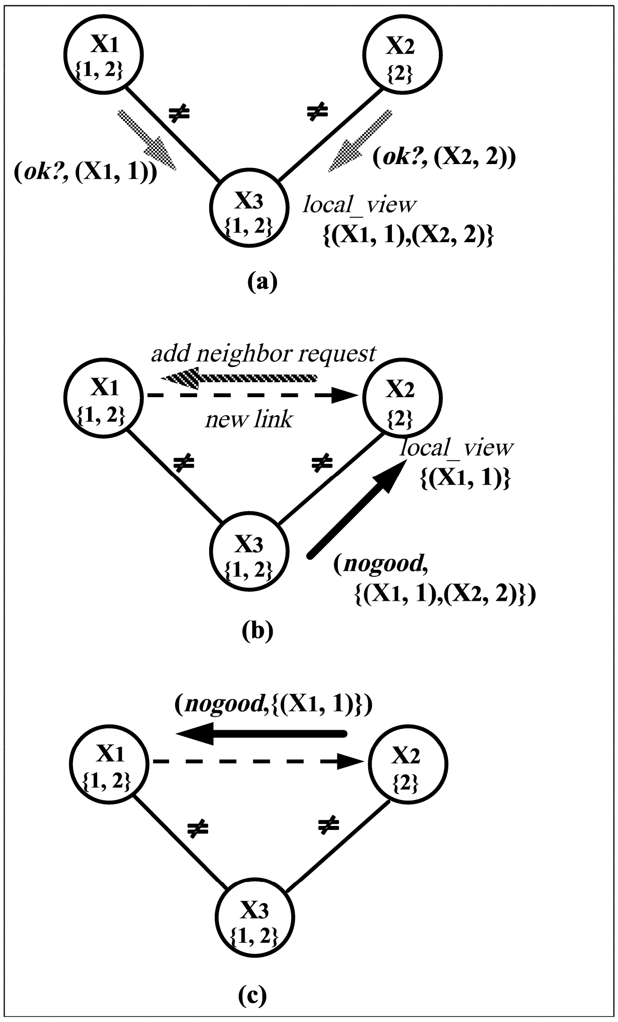

**Figure 4.7** Example of an algorithm execution (asynchronous backtracking).

tween $x_1$ and $x_2$. $x_2$ requests $x_1$ to send $x_1$'s value to $x_2$, and adds $(x_1, 1)$ to its *local_view* (Figure 4.7 (b)). $x_2$ checks whether its value is consistent with the local_view. The *local_view* $\{(x_1, 1)\}$ and the assignment $(x_2, 2)$ violate the received nogood $\{(x_1, 1), (x_2, 2)\}$. However, there is no other possible value for $x_2$. Therefore, $x_2$ generates a new nogood $\{(x_1, 1)\}$, and sends a *nogood* message to $x_1$ (Figure 4.7 (c)).

The completeness of the algorithm (always finds a solution if one exists, and terminates if no solution exists) is guaranteed. The outline of the proof is as follows.

We can show that this algorithm never falls into an infinite processing loop by induction. In the base case, assume that the process with the highest priority, $x_1$, is in an infinite loop. Because it has the highest priority, $x_1$ only receives *nogood* messages. When it proposes a possible value, $x_1$ either receives a *nogood* message back, or else gets no message back. If it receives *nogood* messages for all possible values of its variable, then it will generate an empty nogood (any choice leads to a constraint violation) and the algorithm will terminate. If it does not receive a nogood message for a proposed value, then it will not change that value. Either way, it cannot be in an infinite loop.

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

Since the algorithm does not fall in an infinite processing loop, the algorithm eventually reaches a solution if one exists, and if the problem is over-constrained, some process will eventually generate a nogood that is an empty set.

### 4.2.5 Asynchronous Weak-Commitment Search

One limitation of the asynchronous backtracking algorithm is that the process/variable ordering is statically determined. If the value selection of a higher priority process is bad, the lower priority processes need to perform an exhaustive search to revise the bad decision.

We can reduce the chance of a process making a bad decision by introducing value ordering heuristics, such as the *min-conflict* heuristic [27]. In this heuristic, when a variable value is to be selected, a value that minimizes the number of constraint violations with other variables is preferred. Although this heuristic has been found to be very effective [27], it cannot completely avoid bad decisions.

The asynchronous weak-commitment search algorithm[38] introduces a method for dynamically ordering processes so that a bad decision can be revised without an exhaustive search. More specifically, a *priority value* is determined for each variable, and the priority order among processes is determined using these priority values by the following rules.

- For each variable/process, a non-negative integer value representing the priority order of the variables/processes is defined. We call this value the *priority value.*
- The order is defined such that any variable/process with a larger priority value has higher priority.
- If the priority values of multiple processes are the same, the order is determined

by the alphabetical order of the identifiers.

- For each variable/process, 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 processes.

In the asynchronous weak-commitment search, as in the asynchronous backtracking, each process concurrently assigns a value to its variable, and sends the variable value to other processes. After that, processes wait for and respond to incoming messages. Although the following algorithm is described in a way that a process reacts to messages sequentially, a process can handle multiple messages concurrently, i.e., the process first revises the *local_view* and constraints according to the messages, and then performs **check_local_view** only once.

In Algorithm 4.2, the procedure executed at process $x_i$ by receiving an *ok?* message is described (the procedure for a *nogood* message is basically identical to that for the asynchronous backtracking algorithm). The differences between these procedures and the procedures for the asynchronous backtracking algorithm are as follows.

- The priority value, as well as the current value assignment, is communicated through the *ok?* message (Algorithm 4.2 (i)).
- The priority order is determined using the communicated priority values. If the current value is not *consistent* with the *local_view*, i.e., some constraint with variables of higher priority processes is not satisfied, the agent changes its value using the *min-conflict* heuristic, i.e., it selects a value that is not only consistent with the *local_view*, but also minimizes the number of constraint violations with variables of lower priority processes (Algorithm 4.2 (iii)).
- When $x_i$ cannot find a consistent value with its *local_view*, $x_i$ sends *nogood* messages to other processes, and increments its priority value. If $x_i$ cannot resolve a new nogood, $x_i$ will not change its priority value but will wait for the next message (Algorithm 4.2 (ii)). This procedure is needed to guarantee the completeness of the algorithm. In the asynchronous weak-commitment algorithm, processes try to avoid situations previously found to be nogoods. However, due to the delay of messages, a *local_view* of a process can occasionally be identical to a previously found nogood. In order to avoid reacting to such unstable situations, and performing unnecessary changes of priority values, each process records the nogoods that have been resolved. If no new nogood is found, the process will not change the priority value and waits for the next message.

We illustrate an execution of the algorithm using the distributed 4-queens problem, i.e., there exist four processes, each of which corresponds to a queen in one of the rows. The goal of the process is to find positions on a 4×4 chess board so that the queens do not threaten each other.

The initial values are shown in Figure 4.8 (a). Processes communicate these values with each other. The values within parentheses represent the priority values. The initial priority values are 0. Since the priority values are equal, the priority order is

```
when received (ok?, ($x_j$, $d_j$, priority)) do — (i)
    add ($x_j$, $d_j$, priority) to local_view;
    check_local_view;
end do;

procedure check_local_view
    when local_view and current_value are not consistent do
      if no value in $D_i$ is consistent with local_view
        then resolve a new nogood using hyper-resolution rule;
          when an empty nogood is found do
            broadcast to other processes that there is no solution,
              terminate this algorithm; end do;
          when a new nogood is found do — (ii)
            send the nogood to the processes in the nogood;
            current_priority ← $1 + p_{max}$,
              where $p_{max}$ is the maximal priority value of neighbors;
            select_best_value; end do;
        else select_best_value; end if; end do;

procedure select_best_value
    select $d \in D_i$ where local_view and $d$ are consistent, and $d$ minimizes
        the number of constraint violations with lower priority processes; — (iii)
    current_value ← $d$;
    send (ok?, ($x_i$, $d$, current_priority)) to neighbors; end do;
```

**Algorithm 4.2** Procedures for receiving messages (asynchronous weak-commitment search).

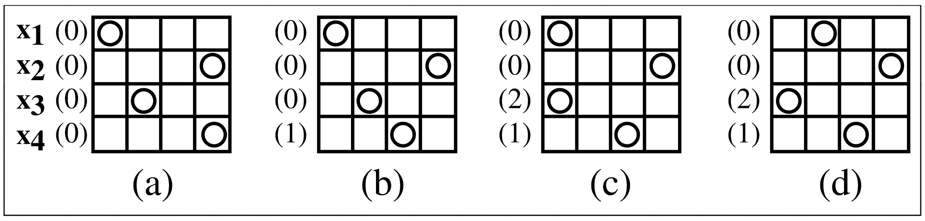

**Figure 4.8** Example of an algorithm execution (asynchronous weak-commitment search).

determined by the alphabetical order of the identifiers. Therefore, only the value of $x_4$ is not consistent with its *local_view*. Since there is no consistent value, $x_4$ 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 processes (Figure 4.8 (b)). Then, $x_3$ tries to change its value. Since there is no consistent value, $x_3$ 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 processes (Figure 4.8 (c)). After that, $x_1$ changes its value to 2, and a solution is obtained (Figure 4.8 (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 $x_1$ (assigning its value to 1) can be revised without an exhaustive search in the asynchronous weak-commitment search.

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

However, the completeness of the algorithm is guaranteed by the fact that the processes record all nogoods found so far. Handling a large number of nogoods is time/space consuming. We can restrict the number of recorded nogoods, i.e., each process 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 processes repeatedly find identical nogoods). Yet, when the number of recorded nogoods is reasonably large, such an infinite processing loop rarely occurs. Actually, when solving large-scale problems, the theoretical completeness has only theoretical importance.

## 4.3 Path-Finding Problem

### 4.3.1 Definition of a Path-Finding Problem

A path-finding problem consists of the following components: a set of nodes $N$, each representing a state, and a set of directed links $L$, each representing an operator available to a problem solving agent. We assume that there exists a unique node $s$ called the start node, representing the initial state. Also, there exists a set of nodes $G$, each of which represents a goal state. For each link, the weight of the link is defined, which represents the cost of applying the operator. We call the weight of the link between two nodes the *distance* between the nodes. We call the nodes that

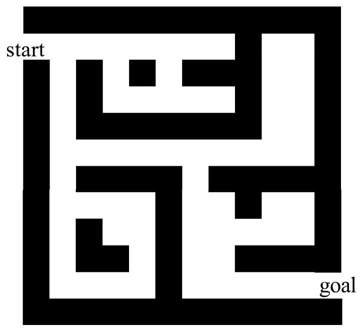

**Figure 4.9** Example of a path-finding problem (maze).

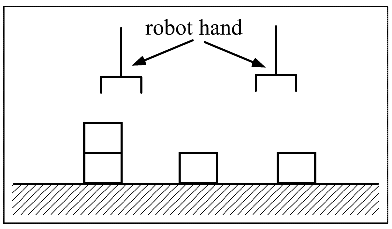

**Figure 4.10** Planning for multiple robot hands.

have directed links from node *i* *neighbors* of node *i*.

The 8-puzzle problem can be formalized as a path-finding problem by representing possible arrangements of tiles as nodes, and allowed moves as links. The arrangements that can be reached by sliding one tile are the neighbors of the original arrangement. In this problem, the weights of all links are 1, and for each link, there exists a link in the opposite direction.

Another example of a path-finding problem is a maze in a grid state space (Figure 4.9). There exists a grid state-space with obstacles. We allow moves along the horizontal and vertical dimensions, but not diagonal motions. The initial state is at the upper-left corner and the goal state is at the bottom-right corner.

Then, how can the path-finding problem formalization be related to DAI? Assume that multiple robots are exploring an unknown environment for finding a certain location. Such a problem can be formalized as a path-finding problem. Furthermore, the planning problem of multiple robot hands shown in Figure 4.10 can be represented as a path-finding problem.

In the following, we first introduce asynchronous dynamic programming as the basis of other algorithms. Then, we present the Learning Real-time A* algorithm, the Real-time A* algorithm, the Moving Target Search algorithm, Real-time Bidirectional Search algorithms, and real-time multiagent search algorithms, as special cases of asynchronous dynamic programming.

### 4.3.2 Asynchronous Dynamic Programming

In a path-finding problem, the *principle of optimality* holds. In short, the principle of optimality states that a path is optimal if and only if every segment of it is optimal. For example, if there exists an optimal (shortest) path from the start node to a goal node, and there exists an intermediate node $x$ on the path, the segment from the start node to node $x$ is actually the optimal path from the start node to node $x$. Similarly, the segment from node $x$ to the goal state is also the optimal path from node $x$ to the goal state.

Let us represent the shortest distance from node $i$ to goal nodes as $h^*(i)$. ¿From the principle of optimality, the shortest distance via a neighboring node $j$ is given by $f^*(j) = k(i,j) + h^*(j)$, where $k(i,j)$ is the cost of the link between $i, j$. If node $i$ is not a goal node, the path to a goal node must visit one of the neighboring nodes. Therefore, $h^*(i) = min_j f^*(j)$ holds.

If $h^*$ is given for each node, the optimal path can be obtained by repeating the following procedure.

- For each neighboring node $j$ of the current node $i$, compute $f^*(j) = k(i,j) + h^*(j)$. Then, move to the $j$ that gives $\min_j f^*(j)$.

Asynchronous dynamic programming [4] computes $h^*$ by repeating the local computations of each node.

Let us assume the following situation.

- For each node $i$, there exists a process corresponding to $i$.
- Each process records $h(i)$, which is the estimated value of $h^*(i)$. The initial value of $h(i)$ is arbitrary (e.g., $\infty$, 0) except for goal nodes.
- For each goal node $g$, $h(g)$ is 0.
- Each process can refer to $h$ values of neighboring nodes (via shared memory or message passing)

In this situation, each process updates $h(i)$ by the following procedure. The execution order of the processes is arbitrary.

- For each neighboring node $j$, compute $f(j) = k(i,j) + h(j)$, where $h(j)$ is the current estimated distance from $j$ to a goal node, and $k(i,j)$ is the cost of the link from $i$ to $j$. Then, update $h(i)$ as follows: $h(i) \leftarrow \min_j f(j)$.

We show an example of an algorithm execution in Figure 4.11. Assume that the initial value of $h$ is infinity except for the goal node (Figure 4.11 (i)). Then, $h$ values are changed at the nodes adjoining the goal node (Figure 4.11 (ii)). It must be noted that these values do not have to be the true values. For example, though the estimated cost from node $d$ is currently 3, there exists a path from node $d$ to the goal node via node $c$, and the cost of the path is 2.

However, $h$ values are further changed at the nodes that can be reached to the goal node (Figure 4.11 (iii)). Now, the $h$ value of $d$ is equal to the true value. We

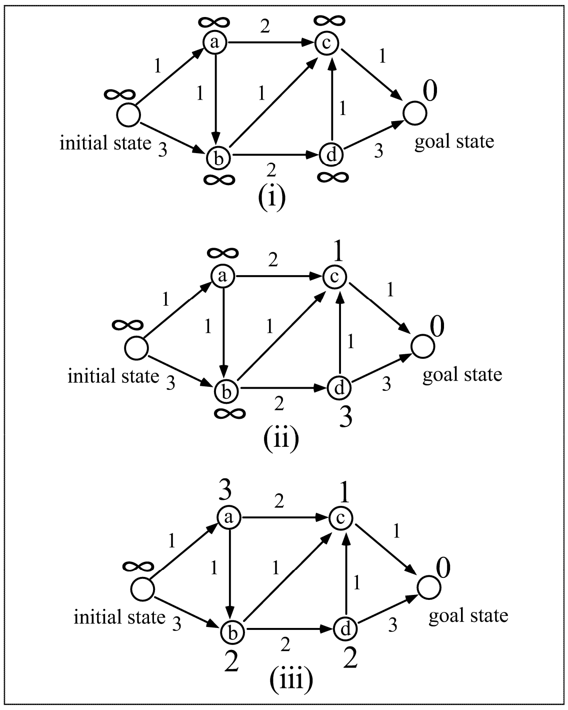

**Figure 4.11** Example of an algorithm execution (asynchronous dynamic programming).

can see that the $h$ values converge to the true values from the nodes that are close to the goal node. By repeating the local computations, it is proved that for each node $i$, $h(i)$ will eventually converge to the true value $h^*(i)$ if the costs of all links are positive.

In reality, we cannot use asynchronous dynamic programming for a reasonably large path-finding problem. In a path-finding problem, the number of nodes can be huge, and we cannot afford to have processes for all nodes. However, asynchronous dynamic programming can be considered a foundation for the other algorithms introduced in this section. In these algorithms, instead of allocating processes for all nodes, some kind of control is introduced for enabling the execution by a reasonable number of processes (or *agents*).

### 4.3.3 Learning Real-Time A*

When only one agent is solving a path-finding problem, it is not always possible to perform local computations for all nodes. For example, autonomous robots may not have enough time for planning and should interleave planning and execution. Therefore, the agent must selectively execute the computations for certain nodes. Given this requirement, which node should the agent choose? One intuitively natural way is to choose the current node where the agent is located. It is easily to imagine that the sensing area of an autonomous robot is always limited. First, the agent updates the $h$ value of the current node, and then moves to the best neighboring node. This procedure is repeated until the agent reaches a goal state. This method is called the Learning Real-time A* (LRTA*) algorithm [19].

More precisely, in the LRTA* algorithm, each agent repeats the following procedure (we assume that the current position of the agent is node $i$). As with asynchronous dynamic programming, the agent records the estimated distance $h(i)$ for each node.

1. Lookahead:
   Calculate $f(j) = k(i,j) + h(j)$ for each neighbor $j$ of the current node $i$, where $h(j)$ is the current estimate of the shortest distance from $j$ to goal nodes, and $k(i,j)$ is the link cost from $i$ to $j$.
2. Update:
   Update the estimate of node $i$ as follows.
   $$h(i) \leftarrow \min_j f(j)$$
3. Action selection:
   Move to the neighbor $j$ that has the minimum $f(j)$ value. Ties are broken randomly.

One characteristic of this algorithm is that the agent determines the next action in a constant time, and executes the action. Therefore, this algorithm is called an *on-line, real-time* search algorithm.

In the LRTA*, the initial value of $h$ must be optimistic, i.e., it must never overestimate the true value. Namely, the condition $h(i) \leq h^*(i)$ must be satisfied. If the initial values satisfy this condition, $h(i)$ will not be greater than the true value $h^*(i)$ by updating.

We call a function that gives the initial values of $h$ a *heuristic function*. For example, in the 8-puzzle, we can use the number of mismatched tiles, or the sum of the Manhattan distances (the sum of the horizontal and vertical distances) of the mismatched tiles, for the heuristic function (the latter is more accurate). In the maze problem, we can use the Manhattan distance to the goal as a heuristic function.

A heuristic function is called *admissible* if it never overestimates. The above examples satisfy this condition. If we cannot find any good heuristic function, we can satisfy this condition by simply setting all estimates to 0.

In asynchronous dynamic programming, the initial values are arbitrary and can be infinity. What makes this difference? In asynchronous dynamic programming, it is assumed that the updating procedures are performed in all nodes. Therefore, the $h$ value of a node eventually converges to the true value, regardless of its initial value. On the other hand, in LRTA*, the updating procedures are performed only for the nodes that the agent actually visits. Therefore, if the initial value of node $i$

is larger than the true value, it is possible that the agent never visits node $i$; thus, $h(i)$ will not be revised.

The following characteristic is known [19].

- In a finite number of nodes with positive link costs, in which there exists a path from every node to a goal node, and starting with non-negative admissible initial estimates, LRTA* is *complete*, i.e., it will eventually reach a goal node.

Furthermore, since LRTA* never overestimates, it *learns* the optimal solutions through repeated trials, i.e., if the initial estimates are admissible, then over repeated problem solving trials, the values learned by LRTA* will eventually converge to their actual distances along every optimal path to the goal node.

A sketch of the proof for completeness is given in the following. Let $h^*(i)$ be the cost of the shortest path between state $i$ and the goal state, and let $h(i)$ be the heuristic value of $i$. First of all, for each state $i$, $h(i) \leq h^*(i)$ always holds, since this condition is true in the initial situation where all $h$ values are admissible, meaning that they never overestimate the actual cost, and this condition will not be violated by updating. Define the *heuristic error* at a given point of the algorithm as the sum of $h^*(i) - h(i)$ over all states $i$. Define a positive quantity called *heuristic disparity*, as the sum of the heuristic error and the heuristic value $h(i)$ of the current state $i$ of the problem solver. It is easy to show that in any move of the problem solver, this quantity decreases. Since it cannot be negative, and if it ever reaches zero the problem is solved, the algorithm must eventually terminate successfully. This proof can be easily extended to cover the case where the goal is moving as well. See [11] for more details.

Now, the convergence of LRTA* is proven as follows. Define the *excess cost* at each trial as the difference between the cost of actual moves of the problem solver and the cost of moves along the shortest path. It can be shown that the sum of the excess costs over repeated trials never exceeds the initial heuristic error. Therefore, the problem solver eventually moves along the shortest path. It is said that $h(i)$ is correct if $h(i) = h^*(i)$. If the problem solver on the shortest path moves from state $i$ to the neighboring state $j$ and $h(j)$ is correct, $h(i)$ will be correct after updating. Since the $h$ values of goal states are always correct, and the problem solver eventually moves only along the shortest path, $h(i)$ will eventually converge to the true value $h^*(i)$. The details are given in [33].

### 4.3.4 Real-Time A*

Real-time A* (RTA*) updates the value of $h(i)$ in a different way from LRTA*. In the second step of RTA*, instead of setting $h(i)$ to the smallest value of $f(j)$ for all neighbors $j$, the second smallest value is assigned to $h(j)$. Thus, RTA* learns more efficiently than LRTA*, but can overestimate heuristic costs. The RTA* algorithm is shown below. Note that *secondmin* represents the function that returns the second smallest value.

1. Lookahead:
   Calculate $f(j) = k(i,j) + h(j)$ for each neighbor $j$ of the current state $i$, where $h(j)$ is the current lower bound of the actual cost from $j$ to the goal state, and $k(i,j)$ is the edge cost from $i$ to $j$.
2. Consistency maintenance:
   Update the lower bound of state $i$ as follows.
   $$h(i) \leftarrow \text{secondmin}_j f(j)$$
3. Action selection:
   Move to the neighbor $j$ that has the minimum $f(j)$ value. Ties are broken randomly.

Similar to LRTA*, the following characteristic is known [19].

- In a finite problem space with positive edge costs, in which there exists a path from every state to the goal, and starting with non-negative admissible initial heuristic values, RTA* is *complete* in the sense that it will eventually reach the goal.

Since the second smallest values are always maintained, RTA* can make *locally optimal decisions* in a tree problem space, i.e., each move made by RTA* is along a path whose estimated cost toward the goal is minimum based on the already-obtained information. However, this result cannot be extended to cover general graphs with cycles.

### 4.3.5 Moving Target Search

Heuristic search algorithms assume that the goal state is fixed and does not change during the course of the search. For example, in the problem of a robot navigating from its current location to a desired goal location, it is assumed that the goal location remains stationary. In this subsection, we relax this assumption, and allow the goal to change during the search. In the robot example, instead of moving to a particular fixed location, the robot's task may be to reach another robot which is in fact moving as well. The target robot may cooperatively try to reach the problem solving robot, actively avoid the problem solving robot, or independently move around. There is no assumption that the target robot will eventually stop, but the goal is achieved when the position of the problem solving robot and the position of the target robot coincide. In order to guarantee success in this task, the problem solver must be able to move faster than the target. Otherwise, the target could evade the problem solver indefinitely, even in a finite problem space, merely by avoiding being trapped in a dead-end path.

We now present the *Moving Target Search* (MTS) algorithm, which is a generalization of LRTA* to the case where the target can move. MTS must acquire heuristic information for each target location. Thus, MTS maintains a matrix of heuristic values, representing the function $h(x, y)$ for all pairs of states $x$ and $y$.

Conceptually, all heuristic values are read from this matrix, which is initialized to the values returned by the static evaluation function. Over the course of the search, these heuristic values are updated to improve their accuracy. In practice, however, we only store those values that differ from their static values. Thus, even though the complete matrix may be very large, it is typically quite sparse.

There are two different events that occur in the algorithm: a move of the problem solver, and a move of the target, each of which may be accompanied by the updating of a heuristic value. We assume that the problem solver and the target move alternately, and can each traverse at most one edge in a single move. The problem solver has no control over the movements of the target, and no knowledge to allow it to predict, even probabilistically, the motion of the target. The task is accomplished when the problem solver and the target occupy the same node. In the description below, $x_i$ and $x_j$ are the current and neighboring positions of the problem solver, and $y_i$ and $y_j$ are the current and neighboring positions of the target. To simplify the following discussions, we assume that all edges in the graph have unit cost.

*When the problem solver moves:*

1. Calculate $h(x_j, y_i)$ for each neighbor $x_j$ of $x_i$.
2. Update the value of $h(x_i, y_i)$ as follows:

$$h(x_i, y_i) \leftarrow max \left\{ \begin{array}{l} h(x_i, y_i) \\ min_{x'}\{h(x_j, y_i) + 1\} \end{array} \right\}$$

3. Move to the neighbor $x_j$ with the minimum $h(x_j, y_i)$, i.e., assign the value of $x_j$ to $x_i$. Ties are broken randomly.

*When the target moves:*

1. Calculate $h(x_i, y_j)$ for the target's new position $y_j$.
2. Update the value of $h(x_i, y_i)$ as follows:

$$h(x_i, y_i) \leftarrow max \left\{ \begin{array}{l} h(x_i, y_i) \\ h(x_i, y_j) - 1 \end{array} \right\}$$

3. Reflect the target's new position as the new goal of the problem solver, i.e., assign the value of $y_j$ to $y_i$.

A problem solver executing MTS is guaranteed to eventually reach the target. The following characteristic is known [11]. The proof is obtained by extending the one for LRTA*.

- In a finite problem space with positive edge costs, in which there exists a path from every state to the goal state, starting with non-negative admissible initial heuristic values, and allowing motion of either a problem solver or the target along any edge in either direction with unit cost, the problem solver executing MTS will eventually reach the target, if the target periodically skips moves.

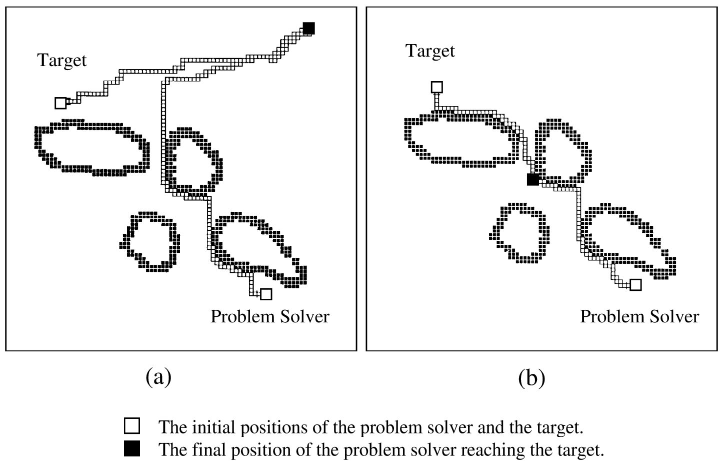
Target
Problem Solver
(a)
Target
Problem Solver
(b)
The initial positions of the problem solver and the target.
The final position of the problem solver reaching the target.

**Figure 4.12** Sample Tracks of MTS.

An interesting target behavior is obtained by allowing a human user to indirectly control the motion of the target. Figure 4.12 shows the experimental setup along with sample tracks of the target (controlled by a human user) and problem solver (controlled by MTS) with manually placed obstacles. The initial positions of the problem solver and the target are represented by white rectangles, while their final positions are denoted by black rectangles. In Figure 4.12 (a), the user's task is to avoid the problem solver, which is executing MTS, for as long as possible, while in Figure 4.12 (b), the user's task is to meet the problem solver as quickly as possible. We can observe that if one is trying to avoid a faster pursuer as long as possible, the best strategy is not to run away, but to hide behind obstacles. The pursuer then reaches the opposite side of obstacles, and moves back and forth in confusion.

### 4.3.6 Real-Time Bidirectional Search

Moving target search enables problem solvers to adapt to changing goals. This allows us to investigate various organizations for problem solving agents. Suppose there are two robots trying to meet in a fairly complex maze: one is starting from the entrance and the other from the exit. Each of the robots always knows its current location in the maze, and can communicate with the other robot; thus, each robot always knows its goal location. Even though the robots do not have a map of the maze, they can gather information around them through various sensors.

For further sensing, however, the robots are required to physically move (as opposed to state expansion): planning and execution must be interleaved. In such a situation, how should the robots behave to efficiently meet with each other? Should they negotiate their actions, or make decisions independently? Is the two-robot organization really superior to a single robot one?

All previous research on bidirectional search focused on offline search [29] [5].

In RTBS, however, two problem solvers starting from the initial and goal states physically move toward each other. As a result, unlike the offline bidirectional search, the coordination cost is expected to be limited within some constant time. Since the planning time is also limited, the moves of the two problem solvers may be inefficient.

In RTBS, the following steps are repeatedly executed until the two problem solvers meet in the problem space.

1. *Control strategy*:
   Select a forward (*Step2*) or backward move (*Step3*).
2. *Forward move*:
   The problem solver starting from the initial state (i.e., the *forward problem solver*) moves toward the problem solver starting from the goal state.
3. *Backward move*:
   The problem solver starting from the goal state (i.e., the *backward problem solver*) moves toward the problem solver starting from the initial state.

RTBS algorithms can be classified into the following two categories depending on the autonomy of the problem solvers. One is called *centralized RTBS* where the best action is selected from among all possible moves of the two problem solvers, and the other is called *decoupled RTBS* where the two problem solvers independently make their own decisions. Let us take an $n$-puzzle example. The real-time unidirectional search algorithm utilizes a single game board, and interleaves both planning and execution; it evaluates all possible actions at a current puzzle state and physically performs the best action (slides one of the movable tiles). On the other hand, the RTBS algorithm utilizes two game boards. At the beginning, one board indicates the initial state and the other indicates the goal state. What is pursued in this case is to equalize the two puzzle states. Centralized RTBS behaves as if one person operates both game boards, while decoupled RTBS behaves as if each of two people operates his/her own game board independently.

In centralized RTBS, the control strategy selects the best action from among all of the possible forward and backward moves to minimize the estimated distance to the goal state. Two centralized RTBS algorithms can be implemented, which are based on LRTA* and RTA*, respectively. In decoupled RTBS, the control strategy merely selects the forward or backward problem solver alternately. As a result, each problem solver independently makes decisions based on its own heuristic information. MTS can be used for both forward and backward moves for implementing decoupled RTBS.

The evaluation results show that, in clear situations, (i.e., heuristic functions return accurate values), decoupled RTBS performs better than centralized RTBS, while in uncertain situations (i.e., heuristic functions return inaccurate values), the latter becomes more efficient. Surprisingly enough, compared to real-time unidirectional search, RTBS dramatically reduces the number of moves for 15- and 24-puzzles, and even solves larger games such as 35- 48- and 63- puzzles. On the other hand, it increases the number of moves for randomly generated mazes: the number of moves for centralized RTBS is around 1/2 in 15-puzzles and 1/6 in 24-puzzles that for real-time unidirectional search; In mazes, however, as the number of obstacles increases, the number of moves for RTBS is roughly double that for unidirectional search [12].

Why is RTBS efficient for $n$-puzzles but not for mazes? The key to understanding the real-time bidirectional search performance is to view that RTBS algorithms solve a totally different problem from unidirectional search, i.e., the difference between real-time unidirectional search and bidirectional search is not the number of problem solvers, but their problem spaces. Let $x$ and $y$ be the locations of two problem solvers. We call a pair of locations $(x, y)$ a *p-state*, and the problem space consisting of p-states a *combined problem space*. When the number of states in the original problem space is $n$, the number of p-states in the combined problem space becomes $n^2$. Let $i$ and $g$ be the initial and goal states; then $(i, g)$ becomes the initial p-state in the combined problem space. The goal p-state requires both problem solvers to share the same location. Thus, the goal p-state in the combined problem space is not unique, i.e., when there are $n$ locations, there are $n$ goal p-states. Each state transition in the combined problem space corresponds to a move by one of the problem solvers. Thus, the branching factor in the combined problem space is the sum of the branching factors of the two problem solvers.

Centralized RTBS can be naturally explained by using a combined problem space. In decoupled RTBS, two problem solvers independently make their own decisions and alternately move toward the other problem solver. We can view, however, that even in decoupled RTBS, the two problem solvers move in a combined problem space. Each problem solver selects the best action from possible moves, but does not examine the moves of the other problem solver. Thus, the selected action might not be the best among the possible moves of the two problem solvers.

The performance of real-time search is sensitive to the topography of the problem space, especially to *heuristic depressions*, i.e., a set of connected states with heuristic values less than or equal to those of the set of immediate and completely surrounding states. This is because, in real-time search, erroneous decisions seriously affect the consequent problem solving behavior. Heuristic depressions in the original problem space have been observed to become *large* and *shallow* in the combined problem space. If the original heuristic depressions are deep, they become large and that makes the problem harder to solve. If the original depressions are shallow, they become very shallow and this makes the problem easier to solve. Based on the above observation, we now have a better understanding of real-time bidirectional search: in $n$-puzzles, where heuristic depressions are shallow, the performance increases significantly, while in mazes, where deep heuristic depressions exist, the performance seriously decreases.

Let us revisit the example at the beginning of this section. The two robots first make decisions independently to move toward each other. However, this method hardly solves the problem. To overcome this inefficiency, the robots then introduce centralized decision making to choose the appropriate robot to move next. They are going to believe that two is better than one, because a two-robot organization has more freedom for selecting actions; better actions can be selected through sufficient coordination. However, the result appears miserable. The robots are not aware of the changes that have occurred in their problem space.

### 4.3.7 Real-Time Multiagent Search

Even if the number of agents is two, RTBS is not the only way for organizing problem solvers. Another possible way is to have both problem solvers start from the initial state and move toward the goal state. In the latter case, it is natural to adopt the original problem space. This means that the selection of the problem solving organization is the selection of the problem space, which determines the baseline of the organizational efficiency; once a difficult problem space is selected, the local coordination among the problem solvers hardly overcomes the deficit.

If there exist multiple agents, how can these agents cooperatively solve a problem? Again, the key issue is to select an appropriate organization for the agents. Since the number of possible organizations is quite large, we start with the most simple organization: the multiple agents share the same problem space with a single fixed goal. Each agent executes the LRTA* algorithm independently, but they share the updated $h$ values (this algorithm is called multiagent LRTA*). In this case, when one of the agents reaches the goal, the objective of the agents as a whole is satisfied. How efficient is this particular organization? Two different effects are observed as follows:

1. Effects of sharing experiences among agents:
   As the execution order of the local computations of processes is arbitrary in asynchronous dynamic programming, the LRTA* algorithm inherits this property. Although the agents start from the same initial node, since ties are broken randomly, the current nodes of the agents are gradually dispersed even though the agents share $h$ values. This algorithm is complete and the $h$ values will eventually converge to the true values, in the same way as the LRTA*.
2. Effects of autonomous decision making:
   If there exists a critical choice in the problem, solving the problem with multiple agents becomes a great advantage. Assume the maze problem shown in Figure 4.13. If an agent decides to go down at the first branching point, the problem can be solved straightforwardly. On the other hand, if the agent goes right, it will take a very long time before the agent returns to this point.
   If the problem is solved by one agent, since ties are broken randomly, the probability that the agent makes a correct decision is 1/2, so the problem can be solved efficiently with the probability 0.5, but it may take a very long time with the probability of 0.5. If the problem is solved by two agents, if one of the agents goes down, the problem can be solved efficiently. The probability that a solution can be obtained straightforwardly becomes 3/4 (i.e., 1-1/4, where the probability that both agents go right is 1/4). If there exist $k$ agents, the

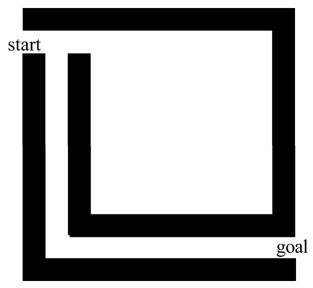

**Figure 4.13** Example of a critical choice.

probability that a solution can be obtained straightforwardly becomes $1-1/2^k$.

By solving a problem with multiple agents concurrently, we can increase both the efficiency and robustness. For further study on problem solving organizations, there exist several typical example problems such as *Tileworld* [30] and the *Pursuit Game* [2]. There are several techniques to create various organizations: explicitly break down the goal into multiple subgoals which may change during the course of problem solving; dynamically assign multiple subgoals to multiple agents; or assign problem solving skills by allocating relevant operators to multiple agents. Real-time search techniques will provide a solid basis for further study on problem solving organizations in dynamic uncertain multiagent environments.

## 4.4 Two-Player Games

### 4.4.1 Formalization of Two-Player Games

For games like chess or checkers, we can describe the sequence of possible moves using a tree. We call such a tree a *game tree.* Figure 4.14 shows a part of a game tree for tic-tac-toe (noughts and crosses). There are two players; we call the player who plays first the *MAX player*, and his opponent the *MIN player.* We assume MAX marks crosses ($\times$) and MIN marks circles ($\bigcirc$). This game tree is described from the viewpoint of MAX. We call a node that shows MAX's turn a MAX node, and a node for MIN's turn a MIN node. There is a unique node called a *root node*, representing the initial state of the game. If a node $n'$ can be obtained by a single move from a node $n$, we say $n'$ is a child node of $x$, and $n$ is a parent of $n'$. Furthermore, if a node $n''$ is obtained by a sequence of moves from a node $n$, we call $n$ an ancestor of $n''$.

If we can generate a complete game tree, we can find a winning strategy, i.e., a strategy that guarantees a win for MAX regardless of how MIN plays, if such a strategy exists. However, generating a complete game tree for a reasonably complicated game is impossible. Therefore, instead of generating a complete game

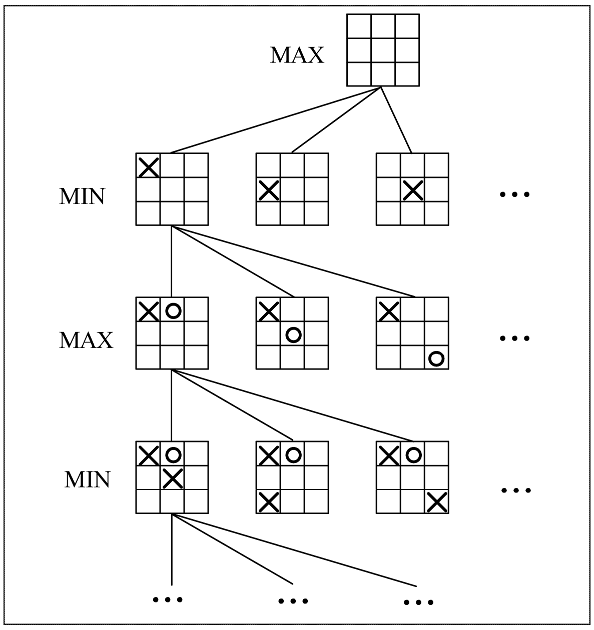

**Figure 4.14** Example of a game tree.

tree, we need to find out a good move by creating only a reasonable portion of a game tree.

### 4.4.2 Minimax Procedure

In the minimax procedure, we first generate a part of the game tree, evaluate the merit of the nodes on the search frontier using a static evaluation function, then use these values to estimate the merit of ancestor nodes. An evaluation function returns a value for each node, where a node favorable to MAX has a large evaluation value, while a node favorable to MIN has a small evaluation value. Therefore, we can assume that MAX will choose the move that leads to the node with the maximum evaluation value, while MIN will choose the move that leads to the node with the minimum evaluation value. By using these assumptions, we can define the evaluation value of each node recursively as follows.

- The evaluation value of a MAX node is equal to the maximum value of any of its child nodes.
- The evaluation value of a MIN node is equal to the minimum value of any of its child nodes.

By backing up the evaluation values from frontier nodes to the root node, we can obtain the evaluation value of the root node. MAX should choose a move that gives the maximum evaluation value.

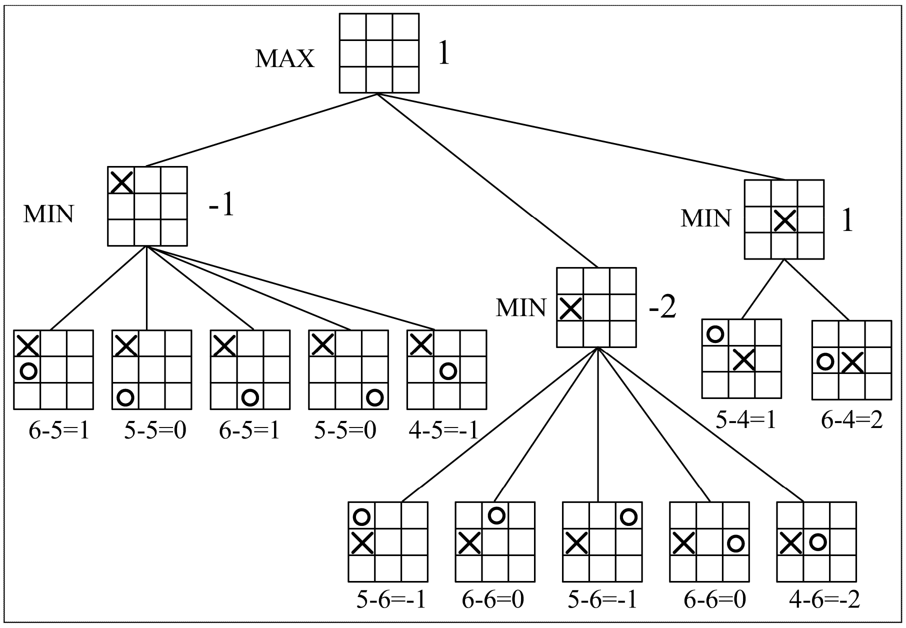

**Figure 4.15** Example of evaluation values obtained by the minimax procedure.

Figure 4.15 shows the evaluation values obtained using the minimax algorithm, where nodes are generated by a search to depth 2 (symmetries are used to reduce the number of nodes). We use the following evaluation function for frontier nodes: (the number of complete rows, columns, or diagonals that are still open for MAX) – (the number of complete rows, columns, or diagonals that are still open for MIN). In this case, MAX chooses to place a × in the center.

### 4.4.3 Alpha-Beta Pruning

The alpha-beta pruning method is commonly used to speed up the minimax procedure without any loss of information. This algorithm can prune a part of a tree that cannot influence the evaluation value of the root node. More specifically, for each node, the following value is recorded and updated.

$\alpha$ value: represents the lower bound of the evaluation value of a MAX node.

$\beta$ value: represents the upper bound of the evaluation value of a MIN node.

While visiting nodes in a game tree from the root node by a depth-first order to a certain depth, these values are updated by the following rules.

- The $\alpha$ value of a MAX node is the maximum value of any of its child nodes visited so far.

- The $\beta$ value of a MIN node is the minimum value of any of its child nodes visited so far.

We can prune a part of the tree if one of the following conditions is satisfied.

**$\alpha$-cut:** If the $\beta$ value of a MIN node is smaller than or equal to the maximum $\alpha$ value of its ancestor MAX nodes, we can use the current $\beta$ value as the evaluation value of the MIN node, and can prune a part of the search tree under the MIN node. In other words, the MAX player never chooses a move that leads to the MIN node, since there exists a better move for the MAX player.

**$\beta$-cut:** If the $\alpha$ value of a MAX node is larger than or equal to the minimum $\beta$ value of its ancestor MIN nodes, we can use the current $\alpha$ value as the evaluation value of the MAX node, and can prune a part of the search tree under the MAX node. In other words, the MIN player never chooses a move that leads to the MAX node, since there exists a better move for the MIN player.

Figure 4.16 shows examples of these pruning actions. In this figure, a square shows a MAX node, and a circle shows a MIN node. A number placed near each node represents an $\alpha$ or $\beta$ value. Also, $\times$ shows a pruning action. A pruning action under a MAX node represents an $\alpha$-cut, and that under a MIN node represents a $\beta$-cut.

The effect of the alpha-beta pruning depends on the order in which the child nodes are visited. If the algorithm first examines the nodes that will likely be chosen (i.e., MAX nodes with large $\alpha$ values, and MIN nodes with small $\beta$ values), the effect of the pruning becomes great. One popular approach for obtaining a good ordering is to do an iterative deepening search, and use the backed-up values from one iteration to determine the ordering of child nodes in the next iteration.

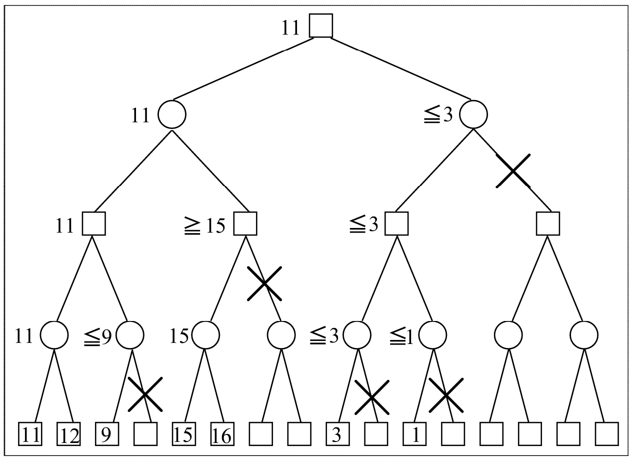

**Figure 4.16** Example of alpha-beta pruning.

## 4.5 Conclusions

In this chapter, we presented several search algorithms that will be useful for problem solving by multiple agents. For constraint satisfaction problems, we presented the filtering algorithm, the hyper-resolution-based consistency algorithm, the asynchronous backtracking algorithm, and the weak-commitment search algorithm. For path-finding problems, we introduced asynchronous dynamic programming as the basis for other algorithms; we then described the LRTA* algorithm, the RTA* algorithm, the MTS algorithm, RTBS algorithms, and real-time multiagent search algorithms as special cases of asynchronous dynamic programming. For two-player games, we presented the basic minimax procedure, and alpha-beta pruning to speed up the minimax procedure.

There are many articles on constraint satisfaction, path-finding, two-player games, and search in general. Pearl's book [28] is a good textbook for path-finding and two-player games. Tsang's textbook [35] on constraint satisfaction covers topics from basic concepts to recent research results. Concise overviews of path-finding can be found in [18, 20], and one for constraint satisfaction is in [26].

The first application problem of CSPs was a line labeling problem in vision research. The filtering algorithm [36] was developed to solve this problem. The notion of k-consistency was introduced by Freuder [9]. The hyper-resolution-based consistency algorithm [6] was developed during the research of an assumption-based truth maintenance system (ATMS). Forbus and de Kleer's textbook [8] covers ATMS and truth maintenance systems in general. Distributed CSPs and the asynchronous backtracking algorithm were introduced in [39], and the asynchronous weak-commitment search algorithm was described in [38]. An iterative improvement search algorithm for distributed CSPs was presented in [40].

Dynamic programming and the principle of optimality were proposed by Bellman [3], and have been widely used in the area of combinatorial optimization and control. Asynchronous dynamic programming [4] was initially developed for distributed/parallel processing in dynamic programming. The Learning Real-time A* algorithm and its variant Real-time A* algorithm were presented in [19]. Barto *et al.* [1] later clarified the relationship between asynchronous dynamic programming and various learning algorithms such as the Learning Real-time A* algorithm and Q-learning [37]. The multiagent real-time A* algorithm was proposed in [16], where a path-finding problem is solved by multiple agents, each of which uses the Real-time A* algorithm. Methods for improving the multiagent Real-time A* algorithm by organizing these agents was presented in [15, 41].

Although real-time search provides an attractive framework for resource-bounded problem solving, the behavior of the problem solver is not rational enough for autonomous agents: the problem solver tends to perform superfluous actions before attaining the goal; the problem solver cannot utilize and improve previous experiments; the problem solver cannot adapt to the dynamically changing goals; and the problem solver cannot cooperatively solve problems with other problem solvers.

Various extensions of real-time search, including Moving Target Search and Real-time Bidirectional Search, have been studied in recent years [31, 13, 14].

The idea of the minimax procedure using a static evaluation function was proposed in [32]. The alpha-beta pruning method was discovered independently by many of the early AI researchers [17]. Another approach for improving the efficiency of the minimax procedure is to control the search procedure in a best-first fashion [21]. Best-first minimax procedure always expands the leaf node which determines the $\alpha$ value of the root node.

There are other DAI works that are concerned with search, which were not covered in this chapter due to space limitations. Lesser [23] formalized various aspects of cooperative problem solving as a search problem. Attempts to formalize the negotiations among agents in real-life application problems were presented in [7, 22, 34].

## 4.6 Exercises

1. *[Level 1]* Implement the A* and LRTA* algorithms to solve the 8-puzzle problem. Compare the number of states expanded by each algorithm. Use the sum of the Manhattan distance of each misplaced tile as the heuristic function.
2. *[Level 1]* Implement the filtering algorithm to solve graph-coloring problems. Consider a graph structure in which the filtering algorithm can always tell whether the problem has a solution or not without further trial-and-error search.
3. *[Level 1]* Implement a game-tree search algorithm for tic-tac-toe, which introduces the alpha-beta pruning method. Use the static evaluation function described in this chapter. Increase the search depth and see how the strategy of the MAX player changes.
4. *[Level 2]* Implement the asynchronous backtracking algorithm to solve the n-queens problem. If you are not familiar with programming using multiprocess and inter-process communications, you may use shared memories, and assume that agents act sequentially in a round-robin order.
5. *[Level 2]* Implement the asynchronous weak-commitment algorithm to solve the n-queens problem. Increase $n$ and see how large you can make it to solve the problem in a reasonable amount of time.
6. *[Level 2]* In Moving Target Search, it has been observed that if one is trying to avoid a faster pursuer as long as possible, the best strategy is not to run away, but to hide behind obstacles. Explain how this phenomenon comes about.
7. *[Level 3]* When solving mazes by two problem solvers, there are at least two possible organizations: One way is to have the two problem solvers start from the initial and the goal states and meet in the middle of the problem space; Another way is to have both problem solvers start from the initial state and

move toward the goal state. Make a small maze and compare the efficiency of the two organizations. Try to create original organizations that differ from the given two organizations.

8. *[Level 3]* In the multiagent LRTA* algorithm, each agent chooses its action independently without considering the actions nor the current states of other agents. Improve the efficiency of the multiagent LRTA* algorithm by introducing coordination among the agents, i.e., agents coordinate their actions by considering the actions and current states of other agents.
9. *[Level 4]* When a real-life problem is formalized as a CSP, it is often the case that the problem is over-constrained. In such a case, we hope that the algorithm will find an incomplete solution that satisfies most of the important constraints, while violating some less important constraints [10]. One way for representing the subjective importance of constraints is to introduce a hierarchy of constraints, i.e., constraints are divided into several groups, such as $C_1, C_2, \ldots, C_k$. If all constraints cannot be satisfied, we will give up on satisfying the constraints in $C_k$. If there exists no solution that satisfies all constraints in $C_1, C_2, \ldots, C_{k-1}$, we will further give up on satisfying the constraints in $C_{k-1}$, and so on. Develop an asynchronous search algorithm that can find the best incomplete solution of a distributed CSP when a hierarchy of constraints is defined.
10. *[Level 4]* The formalization of a two-player game can be generalized to an *n-player game* [25], i.e., there exist $n$ players, each of which takes turns alternately. Rewrite the minimax procedure so that it works for n-player games. Consider what kinds of pruning techniques can be applied.

## 4.7 References

1. A. Barto, S. J. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.
2. M. Benda, V. Jagannathan and R. Dodhiawalla, On optimal cooperation of knowledge sources. *Technical Report BCS-G2010-28*, Boeing AI Center, 1985.
3. R. Bellman. *Dynamic programming.* Princeton University Press, Princeton, NJ, 1957.
4. D. P. Bertsekas. Distributed dynamic programming. *IEEE Trans. Automatic Control*, AC-27(3):610–616, 1982.
5. D. de Champeaux and L. Sint. An improved bidirectional heuristic search algorithm. *Journal of ACM*, 24(2), 177–191, 1977.
6. J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 290–296, 1989.
7. E. Durfee and T. Montgomery. Coordination as distributed search in a hierarchical behavior space. *IEEE Transactions on Systems, Man and Cybernetics*,

21(6):1363–1378, 1991.

8. K. D. Forbus and J. de Kleer. *Building Problem Solvers.* MIT Press, 1993.
9. E. C. Freuder. Synthesizing constraint expressions. *Communications ACM*, 21(11):958–966, 1978.
10. E. C. Freuder and R. J. Wallance. Partial constraint satisfaction. *Artificial Intelligence*, 58(1–3):21–70, 1992.
11. T. Ishida and R. E. Korf, A moving target search: A real-time search for changing goals. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 17(6): 609–619, 1995.
12. T. Ishida. Real-time bidirectional search: Coordinated problem solving in uncertain situations. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 18(6): 617–628, 1996.
13. T. Ishida and M. Shimbo. Improving the learning efficiencies of realtime search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 305–310, 1996.
14. T. Ishida. *Real-time search for learning autonomous agents.* Kluwer Academic Publishers, 1997.
15. Y. Kitamura, K. Teranishi, and S. Tatsumi. Organizational strategies for multiagent real-time search. In *Proceedings of the Second International Conference on Multi-Agent Systems.* MIT Press, 1996.
16. K. Knight. Are many reactive agents better than a few deliberative ones? In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 432–437, 1993.
17. D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
18. R. E. Korf. Search in AI: A survey of recent results. In H. Shrobe, editor, *Exploring Artificial Intelligence.* Morgan-Kaufmann, 1988.
19. R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211, 1990.
20. R. E. Korf. Search. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence,* pages 1460–1467. Wiley-Interscience Publication, New York, 1992.
21. R. E. Korf and D. M. Chickering. Best-first minimax search. *Artificial Intelligence,* 84:299–337, 1996.
22. S. E. Lander and V. R. Lesser. Understanding the role of negotiation in distributed search among heterogeneous agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 438–444, 1993.
23. V. R. Lesser. A retrospective view of FA/C distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1347–1362, 1991.
24. V. R. Lesser and D. D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):81–96, 1981.
25. C. A. Luchhardt and K. B. Irani. An algorithmic solution of n-person games. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 99–111, 1986.
26. A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley-Interscience Publication, New York, 1992.

27. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1–3):161–205, 1992.
28. J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.
29. I. Pohl. Bi-directional search. *Machine Intelligence*, 6, 127–140, 1971.
30. M. E. Pollack and M. Ringuette. *Introducing the Tileworld: Experimentally evaluating agent architectures. Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183-189, 1990.
31. S. Russell and E. Wefald. *Do the Right Thing.* The MIT Press, 1991.
32. C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine (series 7)*, 41:256–275, 1950.
33. M. Shimbo and T. Ishida. On the convergence of realtime search. *Journal of Japanese Society for Artificial Intelligence*, 1998.
34. K. R. Sycara. Multiagent compromise via negotiation. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence, Volume II*, pages 245–258. Morgan Kaufmann, 1989.
35. E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
36. D. Waltz. Understanding line drawing of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
37. C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3/4), 1992.
38. M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science 976)*, pages 88–102. Springer-Verlag, 1995.
39. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the Twelfth IEEE International Conference on Distributed Computing Systems*, pages 614–621, 1992.
40. M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 401–408. MIT Press, 1996.
41. M. Yokoo and Y. Kitamura. Multiagent Real-time-A* with selection: Introducing competition in cooperative search. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 409–416. MIT Press, 1996.