Real-Time Search for Autonomous Agents and Multiagent Systems

TORU ISHIDA   ishida@i.kyoto-u.ac.jp
Department of Social Informatics, Kyoto University

Abstract. Since real-time search provides an attractive framework for resource-bounded problem solving, this paper extends the framework for autonomous agents and for a multiagent world. To adaptively control search processes, we propose *ε-search* which allows suboptimal solutions with *ε* error, and *δ-search* which balances the tradeoff between exploration and exploitation. We then consider search in uncertain situations, where the goal may change during the course of the search, and propose a *moving target search (MTS)* algorithm. We also investigate *real-time bidirectional search (RTBS)* algorithms, where two problem solvers cooperatively achieve a shared goal. Finally, we introduce a new problem solving paradigm, called *organizational problem solving*, for multiagent systems.

1. Introduction

Existing search algorithms can be divided into two classes: offline search such as A* [16], and real-time search such as Real-Time-A* (RTA*) and Learning Real-Time-A* (LRTA*) [29]. Offline search completely examines every possible path to the goal state before executing that path, while real-time search makes each decision in a constant time, and commits its decision to the physical world. The problem solver eventually reaches the goal by repeating the cycle of planning and execution. Real-time search cannot guarantee to find an optimal solution, but can interleave planning and execution. Various extensions of real-time search have been studied in recent years [3, 15, 26, 27, 33, 36, 39].

This paper focuses on extending real-time search algorithms for autonomous agents and for a multiagent world. Though 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; and the problem solver cannot utilize and improve previous experiments. Other problems are that though the algorithms interleave planning and execution, they cannot be directly applied to a multiagent world; the problem solver cannot adapt to the dynamically changing goals; and the problem solver cannot cooperatively solve problems with other problem solvers.

The way of controlling learning processes is described in Section 3. The capability of learning is one of the salient features of real-time search. The major impediment is, however, the instability of the solution quality during convergence.

Other problems are that (1) they try to find all optimal solutions even after obtaining fairly good solutions, and (2) they tend to move towards unexplored areas thus failing to balance exploration and exploitation. In this paper, we propose two new real-time search algorithms to overcome the above deficits: *ε-search* allows suboptimal solutions with ε error, and *δ-search* which balances the tradeoff between exploration and exploitation.

Section 4 considers the case of heuristic search where the goal may change during the course of the search. For example, the goal may be a target that actively avoids the problem solver. A *moving target search (MTS)* algorithm is thus presented to solve this problem. We prove that if the average speed of the target is slower than that of the problem solver, the problem solver is guaranteed to eventually reach the target.

Section 5 investigates *real-time bidirectional search (RTBS)* algorithms, where two problem solvers, starting from the initial and the goal states, physically move toward each other. To evaluate the RTBS performance, two kinds of algorithms are proposed and are compared to real-time unidirectional search. One is called *centralized RTBS* where a supervisor always selects the best action from all possible moves of the two problem solvers. The other is called *decoupled RTBS* where no supervisor exists and the two problem solvers independently select their next moves. Experiments show that RTBS is more efficient than real-time unidirectional search for *n*-puzzles but not for randomly generated mazes.

Section 6 introduces a new problem solving paradigm, called *organizational problem solving*, where multiple agents cooperatively achieve a shared goal. In this paradigm, during a distributed problem solving process, the reorganization of multiple agents is triggered when the agents observe undesired performance degradation. To understand the paradigm and to evaluate its various implementations, a simple research testbed called the *tower building problem* is given.

2. Real-time search algorithms

This section investigates available real-time search algorithms, e.g., *Real-Time-A* (RTA*)* and *Learning-Real-Time-A* (LRTA*)* [29]. We assume the problem space is represented as a connected graph. Real-Time search algorithms commit to individual moves in constant time, and interleave the computation of each move with its execution. The problem solver does not have a map of the problem space, but can utilize a *heuristic evaluation function* (or simply *heuristic function*) that returns an estimate of the cost to the goal. The heuristic function must be *admissible*, meaning that it must never overestimate the actual cost [32].

2.1. LRTA*

The LRTA* algorithm repeats the following steps until the problem solver reaches the goal state. It builds and updates a table containing heuristic estimates of the cost from each state in the problem space. Initially, the entries in the table come

from a heuristic evaluation function, or are set to zero if no function is available, and are assumed to be lower bounds of actual costs. Through repeated exploration of the space, however, more accurate values are learned until they eventually converge to the actual costs to the goal. In the following description, let x be the current position of the problem solver, and $h(x)$ be the current heuristic estimate of the cost from x to the goal.

[LRTA*]

1. Lookahead:

   Calculate $f(x') = k(x, x') + h(x')$ for each neighbor x' of the current state x, where $h(x')$ is the current lower bound of the actual cost from x' to the goal state, and $k(x, x')$ is the edge cost from x to x'.

2. Consistency maintenance:

   Update the lower bound of the state x as follows.

   $$h(x) \leftarrow \min_{x'} f(x')$$

3. Action selection:

   Move to neighbor x' that has the minimum $f(x')$ value. Ties are broken randomly.

   The following theorem is already known [29].

Theorem 1 *In a finite problem space with positive edge costs, in which there exists a path from every state to the goal state, and starting with non-negative admissible initial heuristic values, LRTA* is complete in the sense that it will eventually reach the goal state.*

   A sketch of the proof for completeness is given in the following. Let $h^*(x)$ be the cost of the shortest path between state x and the goal state, and let $h(x)$ be the heuristic value of x. First of all, for each state $x, h(x) \leq h^*(x)$ 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 $h^*(x) - h(x)$ over all states x. Define a positive quantity called *heuristic disparity*, as the sum of the heuristic error and the heuristic value $h(x)$ of the current state x 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.

Furthermore, since LRTA* never overestimates, it *learns* the optimal solutions through repeated trials.

Theorem 2 *If the initial heuristic values are admissible, then over repeated problem solving trials, the values learned by LRTA* will eventually converge to their actual costs along every optimal path to the goal state.*

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. We can show 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 $h(x)$ is correct if $h(x) = h^*(x)$. If the problem solver on the shortest path moves from state x to the neighboring state x' and $h(x')$ is correct, $h(x)$ 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, $h(x)$ will eventually converge to the true value $h^*(x)$. The details are given in [37].

2.2. RTA*

RTA* updates the value of $h(x)$ in a different way from LRTA*. In *Step* 2 of RTA*, instead of setting $h(x)$ to the smallest value of $f(x')$ for all neighbors x', the second smallest value is assigned to $h(x)$. 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.

[RTA*]

1. Lookahead:

   Calculate $f(x') = k(x, x') + h(x')$ for each neighbor x' of the current state x, where $h(x')$ is the current lower bound of the actual cost from x' to the goal state, and $k(x, x')$ is the edge cost from x to x'.

2. Consistency maintenance:

   Update the lower bound of the state x as follows.

   $$h(x) \leftarrow secondmin_{x'} f(x')$$

3. Action selection:

   Move to neighbor x' that has the minimum $f(x')$ value. Ties are broken randomly.

   Similar to LRTA*, the following theorem is known [29].

Theorem 3   *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*.

Theorem 4   *In a tree problem space, 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.

3.   Controlling learning processes

An important capability of real-time search is learning, that is, as in LRTA*, the solution path converges to an optimal path by repeating problem solving trials. In this section, we will focus not on the performance of the first problem solving trial, but on the learning process to converge to an optimal solution. This section is the first to point out that the following problems are incurred when repeatedly applying LRTA* to solve a problem.

—Searching all optimal solutions:

Even after obtaining a fairly good solution, the algorithm continues searching for an optimal solution. When more than one optimal solution exists, the algorithm does not stop until it finds all of them. Since only the lower bounds of actual costs are memorized, the algorithm cannot determine whether the obtained solution is optimal or not. In a real-time situation, though, it is seldom important to find a truly optimal solution (it is definitely not important to obtain all of them), but the algorithm is not satisfied with suboptimal solutions.

—Instability of solution quality:

Every real-time search algorithm always moves toward a state with the smallest estimated cost. As the lower bounds increase with learning, the estimated costs of visited states become higher than those of unvisited states. As a result, the algorithm tends to explore unvisited states, and often moves along a more expensive path than the one obtained before. What is guaranteed by the learning capability is to eventually converge to an optimal solution, but not to improve the solution quality by each trial.

In this section, we propose two new real-time search algorithms to overcome the above deficits: *ε-search* (*weighted real-time search*) allows suboptimal solutions with ε error, and *δ-search* (*real-time search with upper bounds*) which balances the tradeoff between exploration and exploitation [23]. ε-search limits the exploration of new territories of a search space, and δ-search restrains the search in the

current trial from going too far away from the solution path found in the previous trial. The upper bounds of estimated costs become available after the problem is solved once, and gradually approach the actual costs by repeating a problem solving trial.

3.1. Introducing ε-lower and upper bounds

We introduce two new kinds of estimated costs (*ε-lower bounds* and *upper bounds*). Let $h^*(x)$ be the actual cost from state x to the goal, and $h(x)$ be its lower bound. We introduce ε-lower bound denoted by $h_\varepsilon(x)$, and upper bound denoted by $h_u(x)$. The initial ε-lower bound at each state is set to $(1 + \varepsilon)$ times the initial lower bound value given by the heuristic evaluation function. On the other hand, the initial upper bound is set to infinity, i.e., at each state $x, h_u(x) \leftarrow \infty$, while at the goal state $t, h_u(t) \leftarrow 0$. The following operations are performed to maintain the consistency of estimated costs.

$$h(x) \leftarrow \min_{x'} f(x')$$

$$= \min_{x'} \{k(x, x') + h(x')\} \tag{1}$$

$$h_\varepsilon(x) \leftarrow \max \begin{Bmatrix} \min_{x'} f_\varepsilon(x') \\ h_\varepsilon(x) \end{Bmatrix}$$

$$= \max \begin{Bmatrix} \min_{x'} \{k(x, x') + h\varepsilon(x')\} \\ h_\varepsilon(x) \end{Bmatrix} \tag{2}$$

$$h_u(x) \leftarrow \min \begin{Bmatrix} \min_{x'} f_u(x') \\ h_u(x) \end{Bmatrix}$$

$$= \min \begin{Bmatrix} \min_{x'} \{k(x, x') + h_u(x')\} \\ h_u(x) \end{Bmatrix} \tag{3}$$

To maintain the consistency of ε-lower bounds and upper bounds, we modify the lookahead and consistency maintenance steps of LRTA* as follows.

1. Lookahead:

   For all neighboring states x' of x, calculate $f(x') = k(x, x') + h(x')$, $f_\varepsilon(x') = k(x, x') + h_\varepsilon(x')$ and $f_u(x') = k(x, x') + h_u(x')$ for each neighbor x' of the current state x, where $h(x'), h_\varepsilon(x'), h_u(x')$ are the current lower, ε-lower and upper bounds of the actual cost from x' to the goal state, and $k(x, x')$ is the edge cost from x to x'.

2. Consistency maintenance:

   Update $h(x), h_\varepsilon(x)$ and $h_u(x)$ based on operations (1), (2) and (3).

3.2. Weighted real-time search (ε-search)

As LRTA* maintains lower bounds, ε-search requires us to maintain ε-lower bounds as described in the previous subsection. The ε-search algorithm also modifies the action selection step of LRTA* as follows.

3. Action selection:

Move to neighbor x' with minimum $f_\varepsilon(x')$ value. Ties are broken randomly.

When $\varepsilon = 0$, ε-search is exactly same as LRTA*. By repeatedly applying ε-search to the same problem, the algorithm converges to a suboptimal solution with $\varepsilon \cdot h^*(s)$ error. Note that the path P $(x = x_0, x_1, \ldots, x_{n-1}, x_n = t)$ from state x to goal state t is called *ε-optimal*, when the following condition is satisfied.

$$\sum_{i=0}^{n-1} k(x_i, x_{i+1}) \leq (1 + \varepsilon) h^*(x)$$

Theorem 5 *In a finite problem space with positive edge costs, in which there exists a path from every state to the goal state, and starting with non-negative admissible initial heuristic values, through over repeated problem solving trials of ε-search, a path from the initial state s to the goal state t along the minimum value of ε-lower bounds will eventually converge to ε-optimal.*

To evaluate the efficiency of ε-search, we use a rectangle problem space containing 10,000 states with 35% obstacles. The obstacles are randomly chosen grid positions. In the problem space, the initial and the goal states are positioned 100 units apart in terms of Manhattan distance. The actual solution length to the goal in this case is 122 units. Note that real-time search algorithms can be applied to any kind of problem spaces. Mazes are used in this evaluation, simply because the search behavior is easy to observe.

Figures 1 shows the solution length (the number of moves taken by the problem solver to reach the goal state) of repeatedly applying ε-search to the sample maze. Figures 1 was created by averaging 300 charts, each of which records a different convergence process. As ε increases, optimal solutions become hard to get. The converged solution length increases up to the factor of $(1 + \varepsilon)$. When $\varepsilon = 0.5$ for example, the algorithm does not converge to an optimal solution. Through repeated problem solving trials, the solution length decreases more rapidly as ε increases. However, in the case of $\varepsilon = 0.2$ for example, when the algorithm starts searching for an alternative solution, the solution length irregularly increases. This shows that, ε-search fails, by itself, to stabilize the solution quality during convergence.

The learning efficiency of real-time search can be improved significantly by allowing suboptimal solutions. The remaining problem is how to stabilize the solution quality during the convergence process.
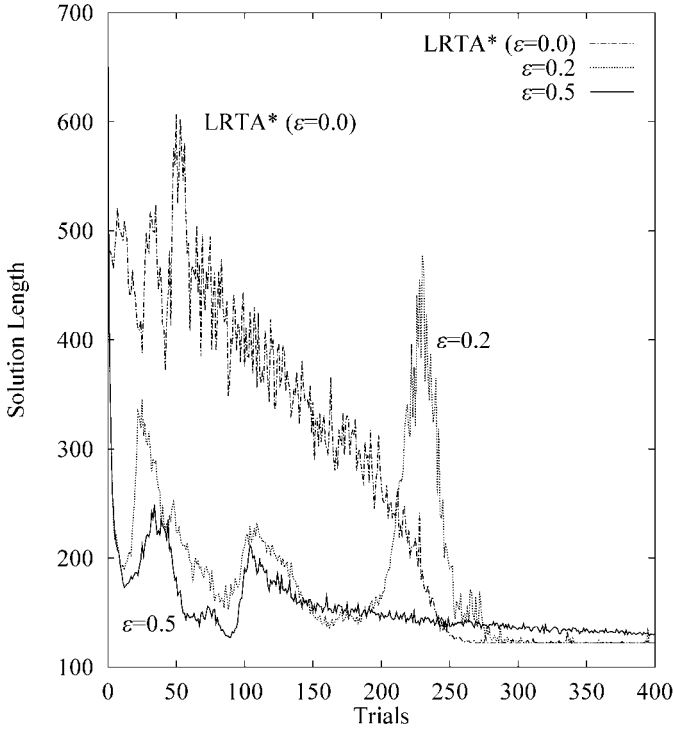
Figure 1. Solution length in ε-search.

3.3. Real-time search with upper bounds (δ-search)

As LRTA* maintains lower bounds, δ-search maintains upper bounds. The upper bounds of estimated costs can be useful for balancing exploration and exploitation. Our idea is to guarantee the worst case cost by using the upper bounds. For this purpose, we assume the graph is undirected, allowing motion of the problem solver along any edge in either direction, and the cost of motion does not depend on the direction.

The action selection step of δ-search is described below. Let $c(j)$ be the total cost of j moves performed so far by δ-search, i.e., when the problem solver moves along a path ($s = x_0, x_1, \ldots, x_j = x$),

$$c(j) = \sum_{i=0}^{j-1} k(x_i, x_{i+1}).$$

Note that h_0 indicates the upper bound value of the initial state at the time when the current trial starts.

3. Action selection:

For each neighboring state x' of x, update the upper bound as follows.

$$h_u(x') \leftarrow \min \begin{Bmatrix} k(x', x) + h_u(x) \\ h_u(x') \end{Bmatrix} \tag{4}$$

Move to the neighbor x' that has the minimum $f(x')$ value among the states that satisfy the following condition.

$$c(j) + f_u(x') \leq (1 + \delta)h_0 \tag{5}$$

Ties are broken randomly.

When $\delta = \infty$, since condition (5) is always satisfied, δ-search is exactly the same as LRTA*. When $\delta \neq \infty$, δ-search ensures that the solution length will be less than $(1 + \delta)$ times the upper bound value of the initial state at the time when the current trial starts. The following theorem confirms the contribution of δ-search in stabilizing the solution quality.

Theorem 6   *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, allowing motion of the problem solver along any edge in either direction, and assuming the cost of motion does not depend on the direction, the solution length of δ-search cannot be greater than $(1 + \delta)h_0$, where $\delta \geq 0$ and h_0 is the upper bound value of the initial state s at the time when the current trial starts.*

δ-search can guarantee the worst case solution length. Note that, however, to take this advantage, the upper bound value of the initial state s must be updated reflecting the results of the previous trial. Therefore, in the following evaluation, each time the problem is solved, we back-propagate the upper bound value from the goal state t to the initial state s along the solution path.

Figure 2 shows the solution length of repeatedly applying δ-search to the sample maze. In the case of $\delta = 0$, the algorithm is satisfied with the solution yielded by the first trial, and thus exploration is not encouraged afterward. In the case of $\delta \geq 2$, the solution path converges to the optimal path. The decrease of learning amount does not always mean that convergence speed is increased. When $\delta = 2$, the convergence is slower than that when $\delta = \infty$. This is because δ-search restricts the amount of learning in each trial. As δ decreases, the solution length is dramatically stabilized. On the other hand, as δ decreases, it becomes hard to obtain optimal solutions. For example, when $\delta = 1.0$, the algorithm does not converge to the optimal solution, and the solution quality is worse than the case of $\delta = 0$. This is because the δ value is not small enough to inhibit exploration, and not large enough to find a better solution. Unlike ε-search, δ-search eventually converges to an optimal solution when an appropriate δ value is selected. To find a better solution than those already obtained, however, δ-search requires the round trip cost of the current best solution, i.e., δ should not be less than 2.
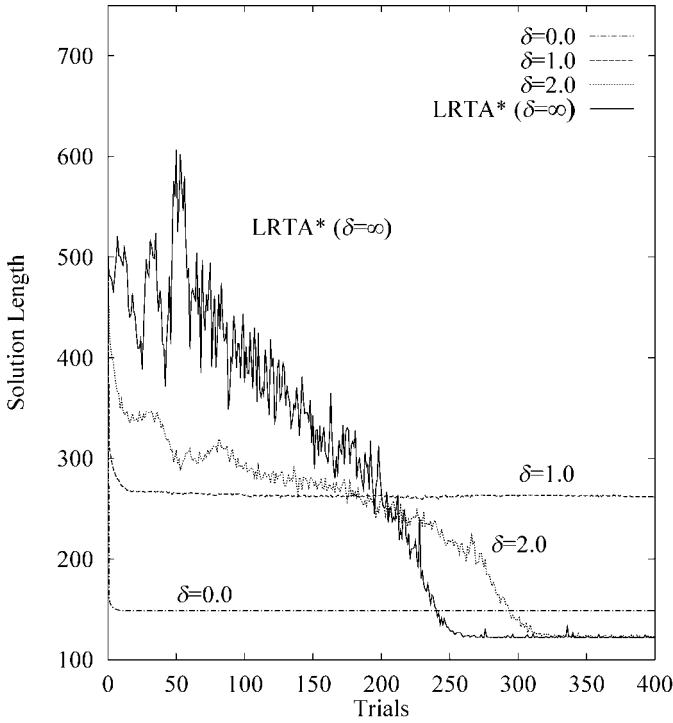
Figure 2.   Solution length in δ-search.

We can merge the ideas of ε- and δ-search. The combined algorithm is called *εδ-search*, which modifies δ-search to move not along the minimum lower bounds but along the minimum ε-lower bounds. εδ-search can reduce the amount of learning by ε-search, and stabilize the solution quality by δ-search.

4.   Adapting to changing goals

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 section, we relax this assumption, and allow the goal to change during the search [17, 18, 21] 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 propose a real-time *moving target search (MTS)* algorithm. We first implement MTS with the minimum operations necessary to guarantee completeness. The resulting algorithm consists of a pair of steps, which are repeatedly performed in alternation. The first step is incremental learning of the estimated distance between the problem solver and the target, and the second step moves the problem solver toward the target. As a result, MTS is *reactive*, i.e., capable of performing each move in constant time, but it is not very efficient. To improve its efficiency, we introduce ideas from the area of *resource-bounded planning* into MTS, including *commitment to goals*, and *deliberation for selecting plans*.

4.1. Moving target search (MTS)

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 state. In the description below, x is the current position of the problem solver, and y is the current position of the target. To simplify the following discussions, we assume that all edges in the graph have unit cost.

[MTS]

When the problem solver moves:

1. Calculate $h(x', y)$ for each neighbor x' of x.
2. Update the value of $h(x, y)$ as follows:

$$h(x, y) \leftarrow max \begin{cases} h(x, y) \\ min_{x'}\{h(x', y) + 1\} \end{cases}$$

3. Move to the neighbor x' with the minimum $h(x', y)$, i.e., assign the value of x' to x. Ties are broken randomly.

When the target moves:

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

$$h(x, y) \leftarrow max \begin{cases} h(x, y) \\ h(x, y') - 1 \end{cases}$$

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

We prove that a problem solver executing MTS is guaranteed to eventually reach the target.

Theorem 7   *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 the problem solver or the target along any edge in either direction with unit cost, a problem solver executing MTS will eventually reach the target, if the target periodically skips moves.*

4.2. Performance evaluation of MTS

We implemented MTS in a 100×100 rectangular grid problem space. We allow motion along the horizontal and vertical dimensions, but not along the diagonals. Interesting target behavior is obtained by allowing a human user to indirectly control the motion of the target. The user moves a cursor around the screen using a mouse, and the target always moves toward the current position of the cursor, using static heuristic values for guidance. Figure 3 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 3(a), the user's task is to avoid the problem solver, which is executing MTS, for as long as possible, while in Figure 3(b), the user's task is to meet the problem solver as quickly as possible. We 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. The pursuer then reaches the opposite side of obstacles, and moves back and forth.

We then examined the performance of MTS more systematically. The motion of the target is automatically controlled by one of the following four response strategies. The response modes are: 1) the target actively avoids the problem solver
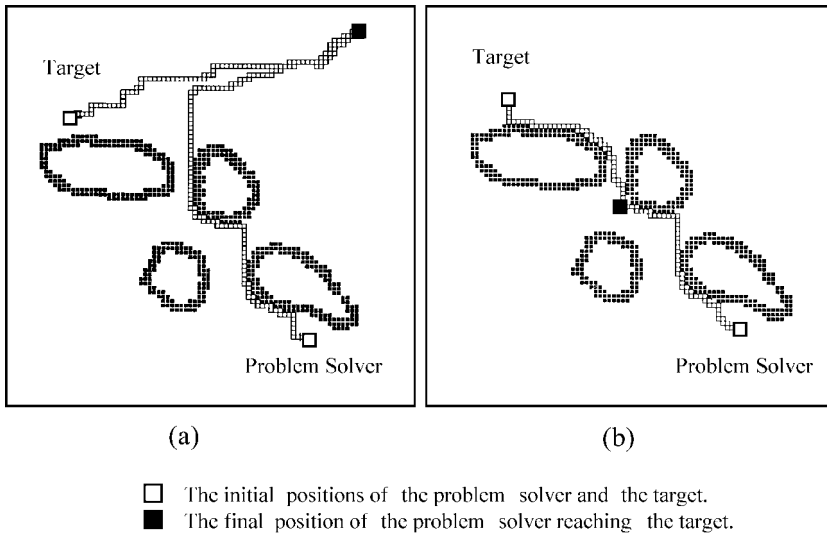
(a)   (b)

☐   The initial positions of the problem solver and the target.
■   The final position of the problem solver reaching the target.

Figure 3.   Sample tracks of MTS.

(*Avoid*), 2) the target moves randomly (*Random*), 3) the target moves coopera-
tively to try to meet the problem solver (*Meet*), and 4) the target remains stationary
(*Stationary*).

Figure 4 illustrates the search cost represented by the total number of moves of
the problem solver for various response strategies of the target. The x-axis
represents the obstacle ratio, and the y-axis represents the number of moves taken
by the problem solver to catch the target. Each data point in this graph are
obtained by averaging 100 trials.

With relatively few obstacles, the target that is easiest to catch is one that is
trying to *Meet* the problem solver, and the most difficult target to catch is one that
is trying to *Avoid* the problem solver, as one would expect. The experimental
results show that the performance degrades as the number of obstacles increases,
since the accuracy of the initial heuristic values decreases. Though this phenomena
is observed in all behavior modes of the target, the performance decreases more
rapidly when the target moves than when it remains *Stationary*. When the obstacles
become more numerous, it becomes harder to catch a target making *Random*
moves and one that is trying to *Meet* the problem solver, than a target trying to
Avoid the problem solver.

At first, this result seems counterintuitive. Recall, however, that in order to avoid
a faster pursuer, the best strategy is not to run away, but to hide behind obstacles.
Both *Meet* and *Random* approximate this strategy better than *Avoid*. In these
situations, the two agents tend to reach opposite sides of the same wall, and move
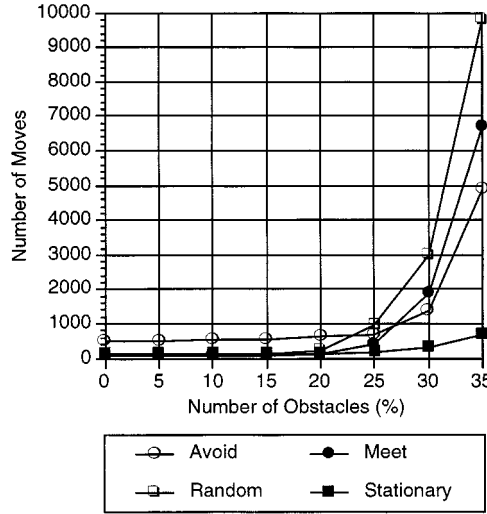back and forth in confusion.

Figure 4. Performance of MTS.

To explain the problem solver's behavior, we define a *heuristic depression* with respect to a single goal state. A heuristic depression is a set of states with heuristic values less than or equal to those of a set of immediately and completely surrounding states. Note that no depressions can exist in actual distance. However, as the situation becomes uncertain, heuristic values differ significantly from the actual distances, and so heuristic depressions tend to appear frequently in the problem space.

When in a heuristic depression, the problem solver faces the situation that there is no way to decrease the heuristic distance, and recognizes that its heuristic values are inaccurate. The problem solver cannot reach the target without "filling" the depression by repeatedly updating the heuristic values. Furthermore, in general the target moves during this learning process. Since MTS maintains different heuristic values for each goal location, the problem solver has to start the learning process over again for the target's new position. This is why the performance of MTS rapidly decreases in uncertain situations.

4.3. Introducing commitment and deliberation

To improve the efficiency of MTS, we introduce ideas from the planning literature. In this area, researchers have focused on dynamically changing environments, where agents are *resource-bounded* in constructing plans. Georgeff and Lansky [14] have studied agent architectures to cope with environmental changes. Cohen and

Levesque [4] have defined the notion of *commitment* as a persistent goal. Kinny and Georgeff [25] quantitatively evaluated how the *degree of commitment* affects agents' performance. Durfee and Lesser [10] performed a similar evaluation in multiagent environments. The role of *deliberation* has been investigated by Bratman *et al.* [2]. Pollack and Ringuette [35] proposed an experimental environment called *Tileworld* and have quantitatively evaluated the *tradeoff between deliberation and reactiveness*, which has been discussed in the planning community [8, 36]. These notions are introduced into MTS to improve its efficiency.

In the original MTS, the problem solver always knows the position of the target. The idea of *commitment to goals* is to ignore some of the target's moves, while concentrating on filling the current heuristic depression. Surprisingly, this turns out to improve performance in many situations. On the other hand, the idea of *deliberation* is to perform an offline search (or a *lookahead search*), in which the problem solver updates heuristic values without moving. With inaccurate heuristic values, real-time search is not always more efficient than offline search. Deliberative investigation using offline search, though it decreases the speed of the problem solver, can often improve overall performance in uncertain situations. In practice, introducing commitment and deliberation dramatically improves the efficiency of MTS. The evaluation results show that MTS performance is improved by 10 to 20 times in uncertain situations depending on the behavior of the target [18].

It is notable that the performance of MTS has been improved by introducing ideas from the area of resource bounded planning. Since only a few steps are added to the original MTS, the improved MTS has not lost its simplicity. However, the behaviors of the two algorithms, as observed on a workstation display, are substantially different. The improved MTS behaves like a predator: In some situations, the problem solver is always sensitive to the target's moves and reactively moves toward the target current position, while in other situations, the problem solver ignores the target's moves, commits to its current goal, and deliberates to find a promising direction to reach that goal.

5. Cooperating in uncertain situations

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 robot 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? These are the research issues of *real-time bidirectional search*, which will be investigated throughout this section [20, 22].

All previous research on bidirectional search focused on offline search [6, 7, 34]. The challenge of this section is to study *real-time bidirectional search (RTBS)*, and to investigate its performance. RTBS can be viewed as cooperative problem solving in uncertain and dynamic situations, and as a step towards organizational problem solving [9, 11, 12], viewing distributed artificial intelligence problems as distributed search [30]. In RTBS, 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, however, the moves of the two problem solvers may be inefficient.

This section proposes two kinds of RTBS algorithms and compares them to *real-time unidirectional search (RTUS)*. 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.

The evaluation results will 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, but increases it for randomly generated mazes. This section is not intended to stress the superiority of RTBS, even though RTBS efficiently solves *n*-puzzles. The motivation is rather to analyze RTBS performance to understand the theory behind the performance of cooperative problem solving.

5.1. Real-time bidirectional search (RTBS)

Pohl proposed the framework of offline bidirectional search: the *control strategy* first selects *forward or backward* search, and then performs the actual state expansion [34]. We here propose the framework of RTBS algorithms, which inherits the framework of offline bidirectional search. The difference from Pohl's framework is that, in RTBS, the forward and backward operations are not state expansions but physical moves of the problem solvers.

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: *centralized control* and *decoupled control*. Besides the type of control, the RTBS algorithms can be further classified from the information sharing point of view, i.e., how two problem solvers share heuristic distances: *shared heuristic information* and *distributed heuristic information*. In this section, only two extremes in four possible combinations will be investigated: *centralized control with shared heuristic information* and *decoupled control with distributed heuristic information*. We simply call the two extremes *centralized RTBS* and *decoupled RTBS* unless there is doubt. Two combinations that remain will not be discussed, because they can be generated rather straightforwardly, and their individual performance lies somewhere between the two extremes.

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). By repeating these steps, the algorithm eventually achieves the goal state. 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 the possible forward and backward moves to minimize the estimated distance to the goal state. We use two centralized RTBS algorithms called LRTA*/B and RTA*/B, 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. At this time, MTS is the only algorithm that can handle situations in which both the problem solver and the goal move. Therefore, decoupled RTBS should employ MTS for both forward and backward moves. The algorithm is called MTS/B. As described in Section 4, the original MTS algorithm assumes that the problem solver moves faster than the target. Note that the same assumption is even required in MTS/B, however, where the two problem solvers try to meet each other.

5.2. *Performance of RTBS*

Let us examine the performance of the RTBS algorithms. The computational complexity will be investigated first, and then the actual performance will be measured on typical example problems.

Figure 5(a) illustrates the search tree for real-time unidirectional search. Each node represents a position of the problem solver. An example path from the initial state to the goal state is represented by the wide line. Let B_f be the number of operators (branching factor) for the forward move, B_b be the number of operators
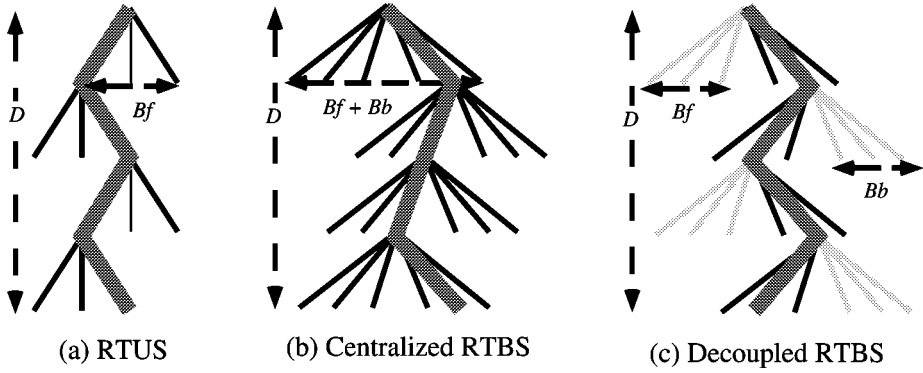
Figure 5. Computational complexity.

for the backward move, and D be the number of moves before reaching the goal state. Then, the number of generated states can be represented by $B_f \times D$.

The key to understanding the real-time bidirectional search performance is to view that RTBS algorithms solve a totally different problem from real-time 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 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. 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.

Figure 5(b) represents the combined problem space for centralized RTBS. At each move, the best action is selected from all possible $B_f + B_b$ moves of the two problem solvers. Let D be the sum of moves of the two problem solvers before meeting with each other. Then, the number of generated states can be represented by $(B_f + B_b) \times D$.

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. Figure 5(c) represents the search tree for decoupled RTBS. Each problem solver selects the best action from possible B_f or B_b moves represented by solid edges, but does not examine the moves of the other problem solver represented by dashed edges. The figure shows that the selected action might not be the

best among possible $B_f + B_b$ moves of the two problem solvers. Let D be the sum of moves of the two problem solvers. Since decoupled RTBS utilizes partial information, D increases (as opposed to centralized RTBS). On the other hand, the number of generated states can be represented by $\{(B_f + B_b)/2\} \times D$, which can decrease (as opposed to centralized RTBS).

5.3. Measurements on typical problems

The real-time unidirectional and bidirectional search algorithms (RTA*, LRTA*, RTA*/B, LRTA*/B and MTS/B) are applied to the two typical path finding problems: *rondamly generated mazes* and *n-puzzles*. Figures 6 to 7 display the experimental results. The *x*-axis represents the obstacle ratio in mazes and the problem ID in *n*-puzzles. The *y*-axis represents the total number of moves, generated states, and the CPU time taken by two problem solvers. All numbers in Figure 6 is obtained by averaging 100 trials, while the numbers in Figure 7 are obtained by averaging 30 trials due to the limitation of computing resources.

In both problems, the superiority or inferiority of centralized and decoupled RTBS depends on whether the situation is clear or uncertain, i.e., whether the heuristic function returns accurate values or not. In clear situations, decoupled RTBS performs better and in uncertain situations, centralized RTBS performs better. These results are supported by the discussion of computational complexity in Section 5.2.

By comparing real-time unidirectional and bidirectional search performances, interesting observations can be made as follows. In clear situations, both search algorithms have a similar performance. In uncertain situations, however, the superiority or inferiority of real-time unidirectional and bidirectional search heavily depends on the applied problems. RTBS can solve *n*-puzzles with less moves than unidirectional search: the number of moves for centralized RTBS is around $1/2$ in 15-puzzles and $1/6$ in 24-puzzles that for unidirectional search. In mazes, however, when the obstacle ratio becomes more than 10%, RTBS cannot perform better than real-time unidirectional search. As the obstacle ratio increases, this tendency becomes even more clear: the number of moves for RTBS is roughly doubled compared to real-time unidirectional search.

Why is RTBS effective for *n*-puzzles, but not for randomly generated mazes? The above fact shows that two problem solvers (one from the entrance and the other from the exit) could not cooperatively solve a maze. The most efficient way is allowing only one problem solver to move. The results cannot be directly explained by the previous discussion of computational complexity. Though we do not have enough theory to support the results, we investigated the mechanism behind the effectiveness of RTBS. Recall that the p-state is composed by a combination of two states. In the combined problem space, since the space is widened, the numbers of states in heuristic depressions also increase. This leads to a problem that is difficult to solve. Contrarily, since various paths are created, heuristic depressions in the combined problem space become shallow. This leads to a problem that is easy to solve. These effects are combined and appear differently in different problems.
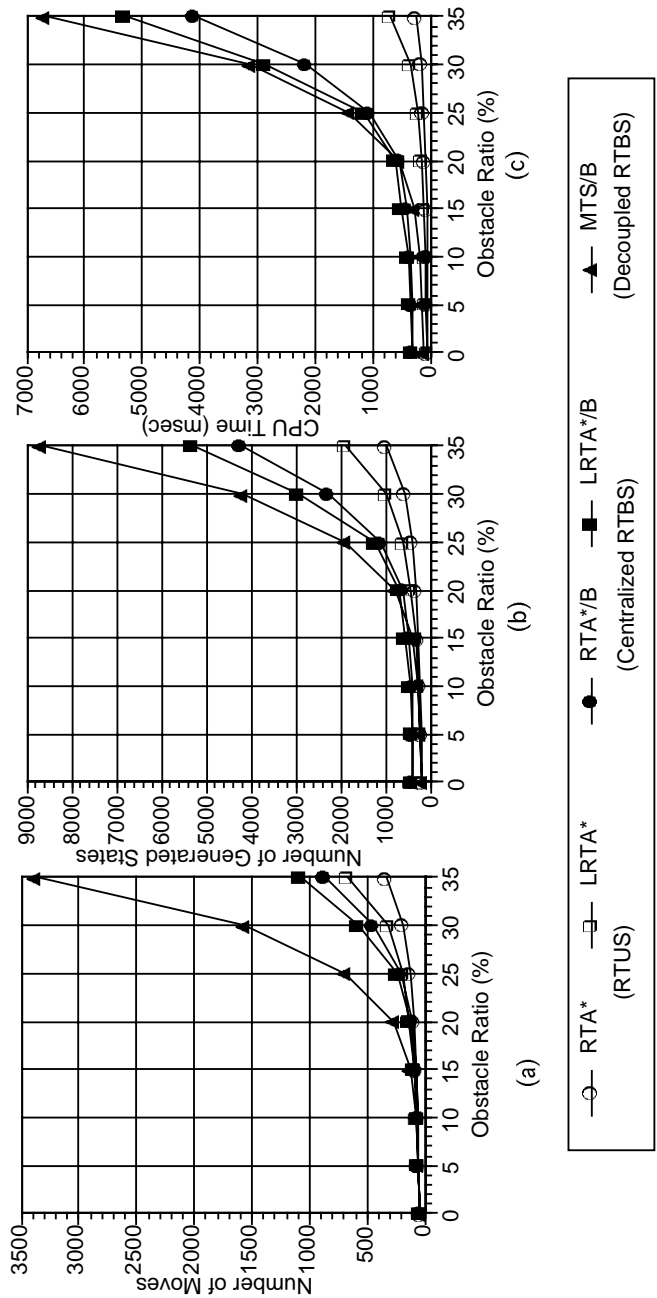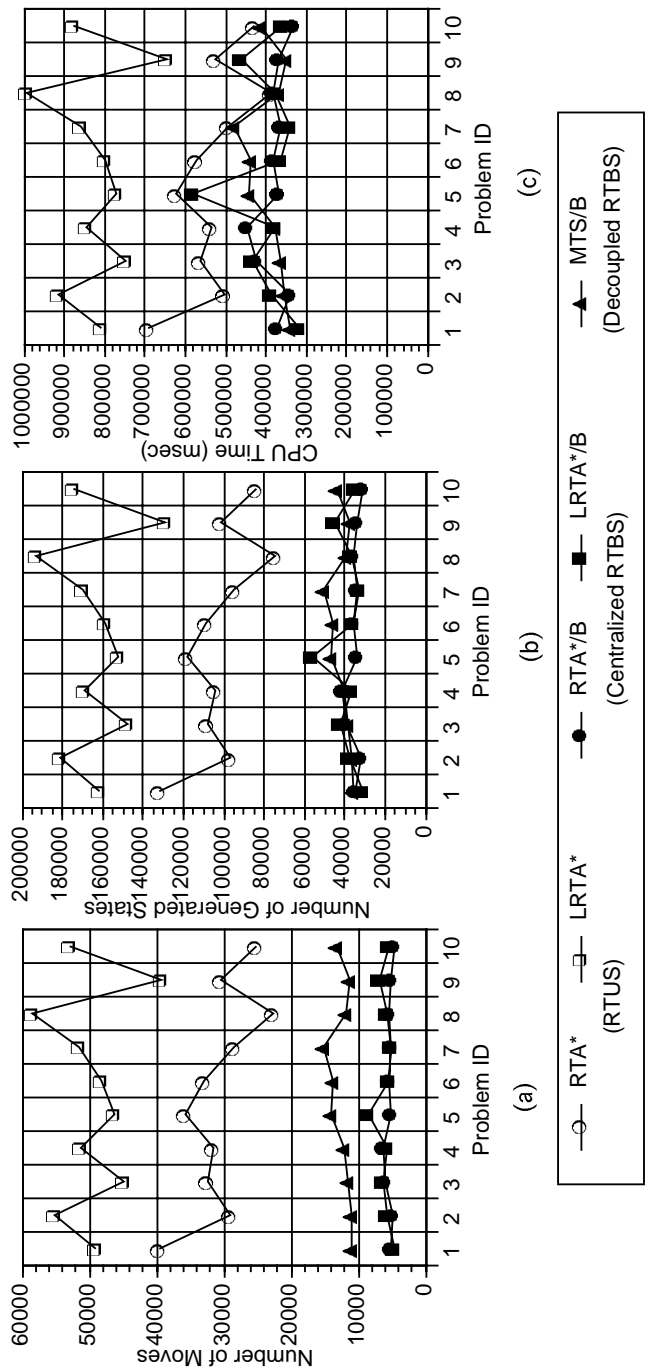
Figure 6. Performance for mazes.

Figure 7. Performance for 24-puzzles.

This work can be considered as a step toward cooperative problem solving. RTBS is not the only way for organizing two problem solvers. Another possible way is to have both problem solvers start from the initial state and move toward the goal state. In this case, the problem space can be viewed differently from the case of RTBS. 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 problem solvers hardly overcome the deficit.

Let us revisit the example at the beginning of this section. The two robots first make decisions independently to move toward each other. However, the problem is hardly solved. To overcome this inefficiency, the robots then introduce centralized decision making to choose the appropriate robot to move next. They think that two is better than one, because the 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.

6.   Forming problem solving organizations

Various problem solving organizations were first investigated by Corkill [5]. The load balancing of spatially distributed agents, for example, was studied to efficiently track vehicles crossing over areas with multiple agents. However, no mechanisms for organization self-design have yet been implemented, and therefore the dynamic behavior of problem solving organizations has not actually been observed.

This section proposes a paradigm called *organizational problem solving* [19], where multiple agents perform organization self-design through problem decomposition and load balancing to efficiently achieve a shared goal. As a result, subgoals and knowledge are distributed to each agent so that the agents can independently create plans and perform actions. In distributed problem solving, however, undesired phenomena (e.g. conflicts among agents) can occur, and thus the performance of the organization can significantly degrade. In such a situation, the agents are then triggered to autonomously reorganize themselves, leading to a new organizational structure. This *reflective feedback between distributed problem solving and organization self-design* characterizes organizational problem solving, and enables the organization to maintain its efficiency even though the environment changes dynamically.

To understand the proposed paradigm and to evaluate the performance of various implementation methods, a simple new problem called the *tower building problem* is introduced. There already exist several typical example problems for problem solving in dynamic environments, such as *Tileworld* [35] and the *pursuit game* [1], however, they can neither reveal the necessity of organization self-design nor evaluate its various implementations.

6.1. Tower building problem

The tower building problem is designed to require agents to flexibly reorganize themselves in dynamically changing environments.

Tower building problem:

>   In a two-dimensional grid space, sequentially numbered blocks are scattered. The base of the tower is located at the center of the grid. The goal of this problem is to build a tower on the base by stacking the blocks starting from number 1.

Let us define the details of this problem. Blocks are randomly distributed over the grid space. It is assumed that, except for the base, only one block at a time can occupy a node, and that agents cannot share a node with any other agent. Each agent can move to any neighboring node in unit time. When an agent and a block are placed on the same node, the agent can hold the block and can bring it afterward. The agent can release the block at any time. Under the above conditions, agents are requested to complete the tower as efficiently as possible.

The design policies of the tower building problem are as follows:

—*To simulate problem solving in human organizations:*

>   The problem contains a good amount of parallelism, but also requires sequential planning. The problem is not like making an *ant hill*, but is similar to constructing a building.

—*To be a testbed for organizational problem solving:*

>   The problem can easily be solved by a single agent, but becomes more complex as the number of agents increases. The problem aims to quantitatively evaluate the superiority or inferiority of various implementations of organizational problem solving.

Figure 8 represents the tower building problem with a single agent. The agent first approaches block number 1. In this process, the agent has to avoid the other blocks, but this can be solved with real-time search. After reaching the first block, the agent brings the block toward the base, places it on the base, and then approaches block number 2. By repeating this process, the tower will eventually be completed.

6.2. Experiments and evaluation

Let us solve the problem with several agents. A 49×49 grid space with randomly distributed blocks is used for the experiment. The number of agents is gradually increased from 1 to 35. Note that each agent has the capability of solving the given
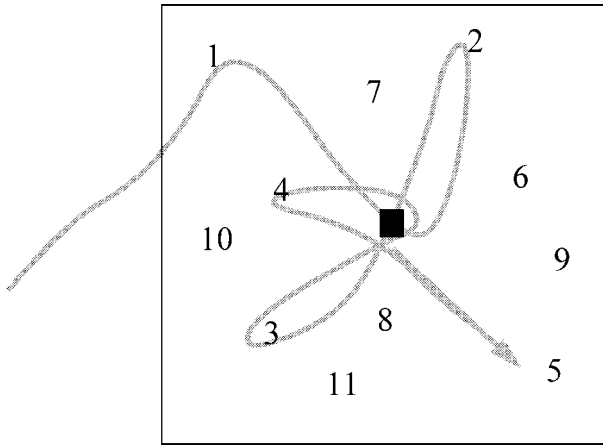
Figure 8.   Tower building problem with a single agent.

problem: each agent performs LRTA* and moves from one grid node to another. To adapt to the dynamically changing environment, the size of each agent's memory is limited so that the agents can forget an old map. To simplify the following discussion, it is assumed that agents can sense the whole grid space without any cost, and that the processing time required by agents is negligible compared to the time for moving blocks and agents. Three different organizations are examined as follows.

Basic agents

In this organization, all agents perform planning independently, and first approach block number 1 simultaneously. When one agent reaches block 1, the other agents remove the goal of "stacking block 1" from their individual plan. Then, these agents set a new goal of "stacking block 2," and move toward block 2. This process continues until the tower is completed.

   Figure 9 includes the performance of basic agents with 100 blocks. The x-axis represents the number of agents in an organization, and the y-axis represents the number of moves per each agent to complete the tower. The result shows that the organization of basic agents cannot effectively utilize more than five agents.

Coordinating agents

When the number of agents further increases, does coordination always increase parallelism? In this experiment, we introduced a simple mechanism called *goal announcement*: every agent broadcasts its goal, e.g., "stacking block 1." After
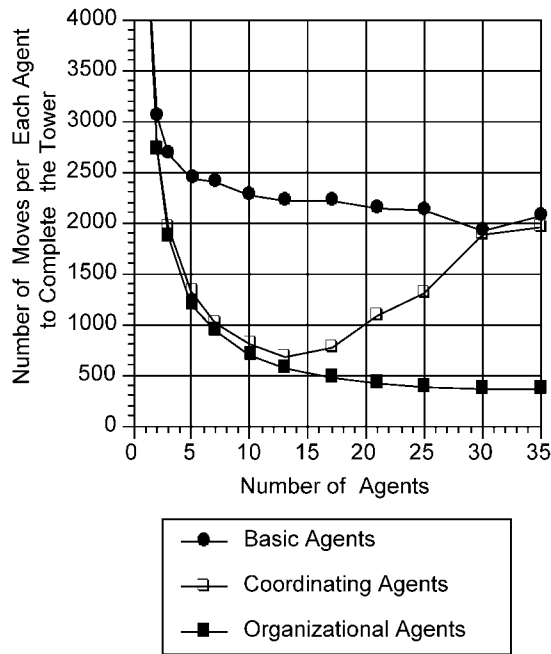
Figure 9.   Performance of different organizations.

broadcasting, each agent measures the distance between block 1 and itself. If there are other agents who have the same goal and are closer to the block, the agent simply removes the goal, and approaches the block with the next smallest number. When the number of agents is relatively small, as in Figure 9, the goal announcement naturally allocates subgoals into multiple agents, and thus can create an efficient organization.

As the number of agents increases, however, this technique becomes ineffective. This is because the goal announcement also increases conflicts among agents especially near the base. Since the blocks are randomly placed, there is no guarantee that the lowest numbered block always reaches the base first. As a result, agents with higher numbered blocks are often forced to wait for lower numbered blocks. Contrarily, agents with lower numbered blocks are often blocked from reaching the base by agents with higher numbered blocks. When the number of agents increases to 20 or 30, however, the situation becomes more difficult. Resolving conflicts among agents near the base become a time consuming problem. Figure 9 represents that as the number of agents increases to more than 15, the performance of this organization decreases considerably.

Organizational agents

Experiments on coordinating agents show that in inappropriate organizations, even if the agents pay the cost of coordination, the performance may not always be improved. In such a case, reorganization must be triggered to increase the organizational performance. It might be more effective for agents with higher numbered blocks to leave their blocks near the base so that they can start looking for new blocks. However, the blocks left near the base can form walls with various shapes. Thus, without a better organization, the agents cannot work effectively because of the conflicts they themselves create.

There are two approaches to form efficient organizations dynamically. The *implicit design approach* is to design utility functions to emerge organizations. The utility function of each agent is the weighted sum of *local* and *global contributions*. Local contribution consists of *individual efficiency*, such as "move any block toward the base," and *conflict avoidance*, such as "avoid being blocked by other agents." On the other hand, global contribution consists of *group efficiency*, such as "move a block to be stacked next," and *conflict avoidance*, such as "avoid blocking other agents."

Another approach of solving the bottleneck can be called as the *explicit design approach*, which is to design constraints to create a pattern of actions. One way is to divide the agents into two groups: one group works at a distance from the base, and the other group works near the base. The former group brings blocks near (but not too near) the base thus allowing the latter group to stack blocks on the base without difficulty. The subgoaling technique is employed for dividing goals by introducing intermediate points. Though it is not difficult to program this particular constraints, the real problem is how to design such an appropriate constraint in general.

The preliminary experiments on the implicit approach are represented in Figure 10 and its performance in Figure 9. Though explicit roles are not assigned to agents, the same effect of the explicit design approach is obtained. Figure 10 represents such an organization emerged by selfish agents each of which behaves as an utility maximizer. This figure shows that blocks are placed by agents near but not too near the base to avoid conflicts. Compared with other organizations, more than 20 agents can participate in problem solving actively. Though we do not have enough theoretical information on how to implement the organizational agents, the experiments clearly show the necessity of such techniques.

Conclusion

Most of work on real-time search focuses on the problem solving performance of the first trial, and did not pay much attention to the learning process. We first propose and analyze two new real-time search algorithms: *ε-search* allows suboptimal solutions with ε error to reduce the total amount of learning performed. *δ-search* utilizes the upper bounds of estimated costs, and can better control the tradeoff between exploration and exploitation.
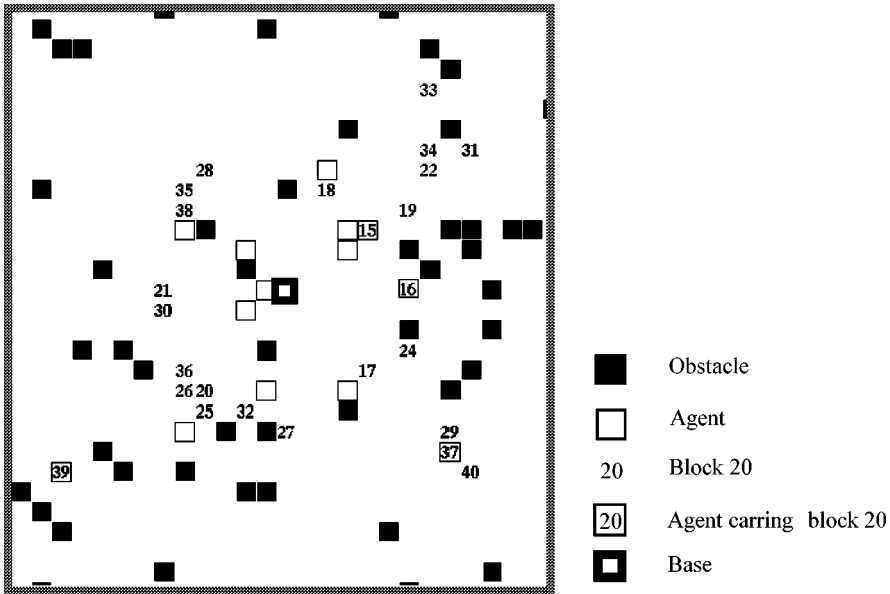
Figure 10. Organization emerged by selfish agents.

We then presented the *moving target search (MTS)* algorithm for reaching a target that changes position over time. The algorithm has been implemented and tested in the cases of pursuing a fleeing target, reaching a randomly moving target, and meeting a cooperatively moving target. The MTS algorithm was constructed using the fewest operations necessary to guarantee its *completeness*, and hence is not very efficient. To improve its efficiency, ideas are introduced from the area of resource-bounded planning, including *commitment to goals*, and *deliberation for selecting plans*.

The two kinds of *real-time bidirectional search (RTBS)* have been proposed: the centralized RTBS algorithm which perform efficiently in uncertain situations, and the decoupled RTBS algorithm which is superior in clear situations. Experiments on mazes and *n*-puzzles show that in clear situations decoupled RTBS performs better, while in uncertain situations, centralized RTBS becomes more efficient, and that RTBS is more efficient than real-time unidirectional search for *n*-puzzles but not for randomly generated mazes. It is shown that the selection of the problem solving organization equals the selection of the problem space, which determines the baseline of the organizational efficiency; once a difficult problem space is selected, local coordination among the problem solvers rarely overcomes the deficit.

Finally, we introduce a new problem solving paradigm, called *organizational problem solving*, for multiagent systems. A simple problem called the *tower building problem* is introduced as a testbed of organizational problem solving. By using this, various techniques for implementing organizational problem solving can be quantitatively evaluated.

The primary motivation behind the series of research is to study problem solving agents for a dynamic, uncertain, and unpredictable domain. One approach to research in this area is to introduce these elements to the classical problem solving framework one at a time in a carefully controlled manner. Extending real-time search is an initial step in a research plan toward this general research issues. More detailed description of the algorithms presented in this paper can be found in [24].

Acknowledgment

References

1. M. Benda, V. Jagannathan and R. Dodhiawalla, "On Optimal Cooperation of Knowledge Sources," Technical Report BCS-G2010-28, Boeing AI Center, 1985.
2. M. E. Bratman, D. J. Israel and M. Pollack, "Plans and Resource Bounded Practical Reasoning," *Computational Intelligence*, vol. 4−4, pp. 349−355, 1988.
3. F. Chimura and M. Tokoro, "The Trailblazer Search: A New Method for Searching and Capturing Moving Targets," *AAAI-94*, pp. 1347−1352, 1994.
4. P. R. Cohen and H. J. Levesque, "Intention is Choice with Commitment," *Artificial Intelligence*, vol. 42−3, 1990.
5. D. D. Corkill, *A Framework for Organizational Self-Design in Distributed Problem Solving Networks*, Ph.D. Dissertation, COINS-TR-82-33: University of Massachusetts, 1982.
6. D. de Champeaux and L. Sint, "An Improved Bidirectional Heuristic Search Algorithm," *J. ACM*, vol. 24−2, pp. 177−191, 1977.
7. D. de Champeaux, "Bidirectional Heuristic Search Again," *J. ACM*, vol. 30−1, pp. 22−32, 1983.
8. T. Dean and M. Boddy, "An Analysis of Time-dependent Planning," *AAAI-88*, pp. 49−54, 1988.
9. E. H. Durfee, V. R. Lesser, and D. D. Corkill, "Coherent Cooperation among Communicating Problem Solvers," *IEEE Trans. on Computers*, vol C-36, pp. 1275−1291, 1987.
10. E. H. Durfee and V. R. Lesser, "Predictability versus Responsiveness: Coordinating Problem Solvers in Dynamic Domains," *AAAI-88*, pp. 66−71, 1988.
11. M. S. Fox, "An Organizational View of Distributed Systems," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 11−1, pp. 70−80, 1981.
12. L. Gasser, N. Rouquette, R. Hill and J. Lieb, "Representing and Using Organizational Knowledge in DAI Systems," in L. Gasser and M. N. Huhns, Editors, *Distributed Artificial Intelligence Volume II*, Pitman, pp. 55−78, 1989.
13. M. P. Georgeff, "Communication and Interaction in Multi-Agent Planning," *AAAI-83*, pp. 125−129, 1983.
14. M. P. Georgeff and A. L. Lansky, "Reactive Reasoning and Planning," *AAAI-87*, pp. 677−682, 1987.
15. B. Hamidzadeh and S. Shekhar, "Deadline Compliance, Predictability, and On-line Optimization in Real-Time Problem Solving," *IJCAI-95*, pp. 220−226, 1995.
16. P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on Systems Science and Cybernetics*, vol. 4−2, pp.100−107, 1968.
17. T. Ishida and R. E. Korf, "Moving Target Search," *IJCAI-91*, pp. 204−210, 1991.
18. T. Ishida, "Moving Target Search with Intelligence," *AAAI-92*, pp. 525−532, 1992.

19. T. Ishida, "Towards Organizational Problem Solving," *IEEE International Conference on Robotics and Automation*, pp. 839–845, 1993.

20. T. Ishida, "Two is not Always Better than One: Experiences in Real-Time Bidirectional Search," *International Conference on Multi-Agent Systems* (*ICMAS-95*), pp. 185–192, 1995.

21. T. Ishida and R. E. Korf, "A Moving Target Search: A Real-Time Search for Changing Goals," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 17–6, pp. 609–619, 1995.

22. T. Ishida, "Real-Time Bidirectional Search: Coordinated Problem Solving in Uncertain Situations," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 18–6, pp. 617–628, 1996.

23 T. Ishida and M. Shimbo, "Improving the Learning Efficiencies of Real-Time Search," *AAAI-96*, pp. 305–310, 1996.

24. T. Ishida, *Real-Time Search for Learning Autonomous Agents*, Kluwer Academic Publishers, 1997.

25. D. N. Kinny and M. P. Georgeff, "Commitment and Effectiveness of Situated Agents," *IJCAI–91*, pp. 82–88, 1991.

26. Y. Kitamura, K. Teranishi and S. Tatsumi, "Organizational Strategies for Multiagent Real-Time Search," *International Conference on Multi-Agent Systems (ICMAS-96)*, pp. 150–156, 1996.

27. K. Knight, "Are Many Reactive Agents Better than a Few Deliberative Ones?" *IJCAI-93*, pp. 432–437, 1993.

28. R. E. Korf, "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27–1, pp. 97–109, 1985.

29. R. E. Korf, "Real-Time Heuristic Search," *Artificial Intelligence*, vol. 42–2/3, pp. 189–211. 1990.

30. V. R. Lesser, "An Overview of DAI: Viewing Distributed AI as Distributed Search," *Journal of Japanese Society for Artificial Intelligence*, vol. 5–4, pp. 392–400, 1990.

31. S. Matsubara and T. Ishida, "Real-Time Planning by Interleaving Real-Time Search with Subgoaling, *International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp. 122–127, 1994.

32. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley: Reading, MA, 1984.

33. J. Pemberton and R. E. Korf, "Making Locally Optimal Decisions on Graphs with Cycles," *Technical Report*, *Computer Science Department*, *University of California at Los Angeles*, 1992.

34. I. Pohl, "Bi-directional Search," *Machine Intelligence*, vol. 6, pp. 127–140, 1971.

35. M. E. Pollack and M. Ringuette, "Introducing the Tileworld: Experimentally Evaluating Agent Architectures," *AAAI-90*, pp. 183–189, 1990.

36. S. Russell and E. Wefald, *Do the Right Thing*, The MIT Press, 1991.

37. M. Shimbo and T. Ishida, "On the Convergence of Real-Time Search," *Journal of Japanese Society for Artificial Intelligence*, pp. 631–638, 1998.

38. M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara, "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving," *International Conference on Distributed Computing Systems (ICDCS-92)*, pp. 614–621, 1992.

39. M. Yokoo and Y. Kitamura, "Multiagent Real-Time A* with Selection: Introducing Competition in Cooperative Search," *International Conference on Multi-Agent Systems (ICMAS-96)*, pp. 409–416, 1996.