The Trailblazer Search with a Hierarchical Abstract Map

Takahiro Sasaki* and **Fumihiko Chimura**† and **Mario Tokoro**‡
Department of Computer Science,
Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223, Japan

Abstract

We deal with the *moving target search problem* where the location of the goal may change during the search process. The *Trailblazer Search* (*TBS*)[Chimura and Tokoro, 1994] achieves a systematic and effective search by maintaining a map. The map stores path information about the region where the algorithm has already searched through. However, because of the growth of the map, there is a problem that the time to make decisions of search steps increases rapidly. We propose an algorithm, the *Trailblazer Search with an Abstract map*(*TBSA*), that reduces the cost of map maintenance, and hence improves the reactiveness of the problem solver. We partition the information about the problem space into *local maps*, and build an *abstract map* that controls maintenance of the local maps. In this way, the problem solver can systematically manage information about the problem space, and it can utilize the map with less cost. We evaluate the efficiency of our method, and show how significant cost reduction in map maintenance can be achieved by using a two-layered map.

1 Introduction

Heuristic search is the process of trial and error during an attempt to reach a goal state in a certain domain. We deal with search problems for agents, considering that the agents will take actual actions in the real world. Korf presented the *Learning Real-Time A** (*LRTA**)[1990] that interleaves constant time decision and actual execution of search steps. However, LRTA* is a search algorithm for stationary goals. When considering the dynamic property of the real world, goals may change locations while the agents are planning to accomplish them. Ishida and Korf presented *moving target search problem*[1991]. Representing a goal that changes locations as a moving target, and taking a uniformly weighted graph as the problem space, the aim of the problem is for a problem solver to reach a node where the target is located, namely, to capture the target.

Moving Target Search (*MTS*)[Ishida and Korf, 1991] extends LRTA* to tackle the search problem for moving targets. While exploring the problem space, starting from heuristic estimates, the problem solver tries to learn the exact distances between any two nodes in the problem space. Once the problem solver learns the set of exact distance values, the search task is then reduced to moving to the adjacent nodes that are closer to the target. However, since the problem solver cannot learn the exact distances at once, it may re-explore the same locations many times and require many search steps to solve a problem. This occurs if the problem space is complicated as for a maze and the problem solver is trapped in deep dead ends, where the initial heuristic estimates greatly differ from the true values.

In *Intelligent Moving Target Search* (*IMTS*) [Ishida, 1992], while overlooking the motion of the target, the problem solver conducts a *lookahead* search to find exits of dead ends. This method significantly reduces the number of search steps of MTS, since the problem solver can make decisions that widely reflect the structure of the search space.

In contrast, *Trailblazer Search* (*TBS*)[Chimura and Tokoro, 1994] records the information about locations where the problem solver has already explored. Due to this information, the problem solver can avoid exploring the same nodes repeatedly. Furthermore, from the stored information, the algorithm creates a *map* that enables the problem solver to chase the target, once the target moves on a direct path found in the map. In this way, TBS significantly reduces the number of search steps. However, as the search proceeds, TBS rapidly increases its cost of map maintenance, according to the growth of the map. As a result, TBS takes an increasingly longer time to make a decision for each step.

In this paper, we propose an algorithm, the *Trailblazer Search with an Abstract map* (*TBSA*), that achieves an effective map maintenance, so as to improve the reactiveness of the problem solver. We partition the information about the problem space into *local maps*, and build a global layer, called *abstract map*, which controls maintenance of the local maps. In this way, we can systematically manage information about the problem space, and

* *sasaki@mt.cs.keio.ac.jp*
† *chimura@mt.cs.keio.ac.jp*
‡ *mario@mt.cs.keio.ac.jp*. Also affiliated with Sony Computer Science Laboratory Inc.

limit map maintenance within smaller local maps, instead of the entire map. TBSA can thus make decisions in less time.

We describe the hierarchical map maintenance of TBSA. We formally analyze the method, and show empirical results indicating its effectiveness.

2 Search for Moving Targets and the Trailblazer Search

We represent the problem space as a connected and undirected graph with unit cost assigned to each edge. We represent a goal that changes location as a moving target. At any time period during the process, a problem solver and a target are assigned nodes in the graph. In alternate turns, they move to any node adjacent to their current locations. Starting from separated initial nodes, the search ends when the locations of the problem solver and the target coincide.

We make three assumptions to assure a solution for the search. Firstly, the problem solver always knows the location of the target. Concretely speaking, the problem solver must know the *name* of the node where the target is located. Secondly, the problem solver has an *admissible*[Pearl, 1984] heuristic function that returns an estimated distance between any two nodes in the problem space. Lastly, the problem solver moves faster than the target. In the particular problem space defined above, we implement this assumption by periodically skipping the target's turn.

Chimura and Tokoro presented the *Trailblazer Search* (*TBS*), an algorithm for moving targets. The basic idea of TBS is to store path information about the region the problem solver has previously explored, and to exploit this information for accomplishing the task. The problem solver records every step of itself and the target. This is done by recording an undirected edge connecting the departure and arrival nodes of the step. The problem solver organizes these records into a graph called the *trail*. From the trail, the problem solver calculates the routing table called the *map*, using *Dijkstra's shortest path algorithm* or *Floyd's algorithm*[Aho *et al.*, 1974]. The map holds a minimum cost path from the current location of the problem solver to any node it has explored.

TBS separates the capturing process of the target into two distinct phases: (1)the *search phase*, and (2)the *chase phase*. In both phases, the problem solver exploits the information of the map in order to achieve an effective process. The process starts from the search phase. By using the map in the search phase, the problem solver can distinguish an unexplored region from an explored region, and can thus conduct a systematic process by eliminating unnecessary re-exploration of the same nodes. Once the target crosses the trail of the problem solver (or vice versa), and hence the trails of the problem solver and the target overlap, the problem solver finds a path in the map leading to the target. Then, TBS enters the chase phase. The chase phase is deterministic since the values referred from the map are accurate. The problem solver simply needs to move to reduce the distance between itself and the target.

In the search phase, either the problem solver must find the target or the trail of the problem solver and the target must overlap before exploring the whole problem space. Hence, the algorithm eventually enters the chase phase. In the chase phase, according to the assumption that the problem solver moves faster than the target, the problem solver can always capture the target. Thus TBS is a complete algorithm, i.e., it terminates. In the following, we formally describe the TBS algorithm.

Let x and y be the locations of the problem solver and the target respectively. They take values from a set of integers that identify the actual nodes in the problem space. Let $h(x,y)$ be the heuristic estimate of the distance between x and y, and $C(x,y)$ be the distance between x and y derived from the map. The value of $C(x,y)$ is ∞ if there is no path found between x and y on the map. For each explored node, we assume that the problem solver records the *parent node* from which it has arrived to the node for the first time.

- Procedures of the problem solver when it is its own turn to move
  1. For each node x' adjacent to x, read $C(x',y)$ to find if there is any path from x' to y. If there is no path, enter the search phase, otherwise enter the chase phase.
     a. *In the search phase*
        For each unexplored node x' adjacent to x, calculate $h(x',y)$, move to the node x' with minimum $h(x',y)$. Ties are broken randomly. If all of the adjacent nodes have been explored, move to the node x' that is the parent node of x.
     b. *In the chase phase*
        Move to the adjacent node x' with minimum $C(x',y)$. Ties are broken randomly.
  2. Record the move of the problem solver by adding an undirected edge connecting x and x' to the trail. Assign the value of x' to x as the new location of the problem solver.
  3. Update the routing table.
- Procedures of the problem solver when it is the target's turn to move
  1. Record the move of the target by adding an undirected edge connecting y and y' to the trail, where y' is the new location of the target. Assign the value of y' to y as the new location of the target.
  2. Update the routing table.

With the map information, even if the problem space is complicated as for a maze, a problem solver executing TBS can capture the target in fewer steps than one executing Ishida's MTS. However, TBS requires map maintenance, whose cost increases rapidly as the search proceeds, and as trails of the problem solver and the target become more complicated. The problem solver uses the map to decide which direction to proceed next. As a result, the problem solver takes an increasingly longer time to make decisions for each step, and it becomes increasingly difficult for the problem solver to behave

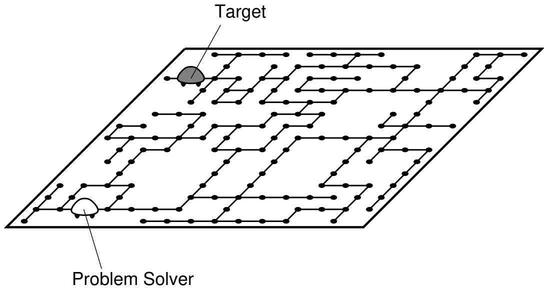

Figure 1: The detailed map of the entire problem space

reactively. If we consider real-world applications, such as the problem for an autonomous robot to capture a target robot, the cost of map maintenance is tolerable to some extent. The reason is that map maintenance takes computational cost, while search steps take physical cost such as the cost of the robot's move. However, for TBS, the increase in the cost of map maintenance is so extreme that it cannot be neglected, especially if we consider large scale real-world problems.

3 The Trailblazer Search with an Abstract Map

TBS maintains a map including all the details every time the problem solver takes a single step(fig.1). This is useless since a single move results in updating the map for the entire area of the trail, even though the move will only affect the structure of the trail in a small region surrounding the location where the move took place. Following the idea that the map should be locally updated, we divide the map and conduct routing in the partitions. In order to keep track of connections between partitions, we create a global map over the partitions. Therefore, the problem solver hierarchically maintains the information about the problem space. We call the search algorithm that uses the hierarchical map, the *Trailblazer Search with an Abstract map* (*TBSA*). This section describes the maintenance method and the use of this two-layered hierarchical map.

3.1 A Hierarchical Construction of a Map

TBSA regards the problem space as a set of exclusive partitions called the *subproblem spaces*. The partitions will be made by a simple function distinguishing nodes of one partition from those of the other partitions. In a subproblem space, some nodes directly connect to nodes that belong to another distinct subproblem space. We call them *boundary nodes*. A *boundary* lies between connected boundary nodes of different subproblem spaces.

TBSA maintains the map in two layers(fig.2): (1)the *local maps* hold the detailed information about the corresponding subproblem space, and (2)the *abstract map* holds the information about how each subproblem space interconnects with one another. Each local map records every search step the problem solver takes in corresponding partitions of the problem space. The abstract map only records boundary nodes and edges connecting those

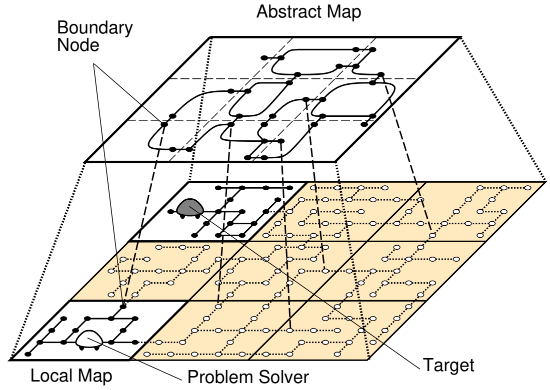

Figure 2: The detailed local maps and the abstract global map

nodes. In the abstract map, the cost of an edge is the cost of the path that connects the boundary nodes, and that is calculated from a local map.

Due to the hierarchical map, TBSA systematically manages the information that has been acquired during the search process. As a result, the problem solver makes decisions in a shorter time for each step.

3.2 Hierarchical Map Maintenance and Reconstruction of Paths

In the following, we describe, (1) how TBSA constructs and maintains a hierarchical map, and (2) how TBSA calculates the distance between any two nodes according to the hierarchical map.

If the problem solver takes a step within a certain subproblem space, map maintenance proceeds only in a corresponding local map. If the problem solver strides across the boundary between two distinct subproblem spaces, map maintenance proceeds in the abstract map. In both procedures, we can use the same routing algorithms on each map. Let p_n represent a node that the problem solver reaches after taking n steps. For n being a non-negative integer, assume that the problem solver moves from a node p_{n-1} to a node p_n on its nth step. $S(x)$ identifies a subproblem space to which each node x belongs. Map maintenance proceeds differently, according to whether both p_{n-1} and p_n belong to the same subproblem space or not.

- *When the step is taken within the same subproblem space* ($S(p_{n-1}) = S(p_n)$),
  1. Add the edge $(p_{n-1}, p_n, c(p_{n-1}, p_n))$ to the local map of subproblem space $S(p_{n-1})$, where $c(p_{n-1}, p_n)$ is the actual cost to traverse the edge.
  2. Update the local map of $S(p_{n-1})$.
- *When the step is taken between two distinct subproblem spaces* ($S(p_{n-1}) \neq S(p_n)$),
  1. Add the edge $(p_{n-1}, p_n, c(p_{n-1}, p_n))$ to the abstract map.
  2. For any explored boundary node q in subproblem space $S(p_{n-1})$, add the edge

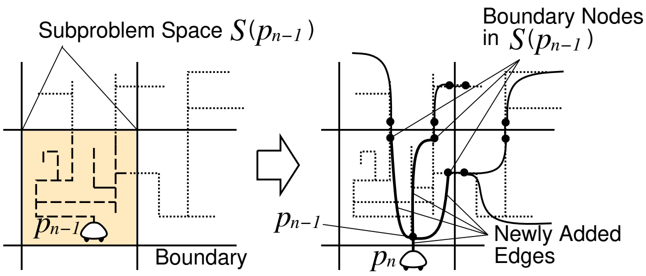

Figure 3: Updating of the abstract map, when a step is taken across the boundary

$(q, p_{n-1}, C_{S(p_{n-1})}(q, p_{n-1}))$ to the abstract map. Here $C_{S(p_{n-1})}(q, p_{n-1})$ is the actual cost of the path between node q and node p_{n-1}, derived from the local map of $S(p_{n-1})$.

3. Update the abstract map.

(fig.3)

The same map maintenance is done when the target takes a step.

We now show how to calculate the map cost $C(x, y)$ between two nodes x and y, where x is the location of the problem solver and y of the target. The function $C_{S(x)}$ denotes the cost referred from the local map of subproblem space $S(x)$ to which the node x belongs. The function C_{Abst} denotes the cost referred from the abstract map. The distance between two nodes located in the same subproblem space can be directly referred from the corresponding local map. On the other hand, the distance between two nodes located in distinct subproblem spaces must be calculated by reconstructing the path from the local maps and the abstract map.

$$C(x,y) = \begin{cases} C_{S(x)}(x,y), \\ \quad \textit{when } S(x) = S(y) \\ \min_{v,w}\{C_{S(x)}(x,v) + C_{Abst}(v,w) + C_{S(y)}(w,y)\}, \\ \quad \textit{when } S(x) \neq S(y) \end{cases}$$

Here, v and w indicate any explored boundary node located inside subproblem spaces $S(x)$ and $S(y)$, respectively.

TBSA differs from TBS only in the way of constructing the map and of calculating the distance between two nodes on the map. TBSA follows the policy of TBS that separates the process of capturing the target into the search phase and the chase phase. Thus TBSA inherits the simplicity and completeness of TBS.

In the following of this paper, we will analyze the search problem on a two-dimensional grid space. Although grid-like space represents only some features of many problem types, the search problem on such a space is not extremely special. This is because, if we regard junctions as nodes and connections between junctions as edges, we can naturally convert the grid-like space to a general graph-like one.

In a grid-like problem space, we consider partitions of fixed size, made by horizontally and vertically dividing the grid at fixed intervals. This method for partitioning allows TBSA to create hierarchies with small overhead. For general graph-like problem spaces, however, the partitioning method becomes a major issue. Later, we discuss a method that creates hierarchies in such problem spaces.

4 Formal Analysis of TBSA

TBS and TBSA both update the map every time the problem solver or the target moves. We analyze the worst-case time complexity of map maintenance for each step of the problem solver or the target. We consider the rectangular grid-like problem space as discussed above. Let n be the total number of the nodes in the problem space. If we use Dijkstra's algorithm[Aho *et al.*, 1974] for routing, the stepwise worst case time complexity of map maintenance is $O(n^2)$ for TBS. If we equally partition the problem space into d subproblem spaces, each local map contains n/d nodes at most. The abstract map contains $4\sqrt{n/d} \times d = 4\sqrt{nd}$ nodes at most, since each subproblem contains $4\sqrt{n/d}$ boundary nodes. Hence, for each step of the problem solver or the target, the stepwise worst-case time complexity of map maintenance is $O((n/d)^2)$ for the local map, and $O(16nd) = O(nd)$ for the abstract map. The partitioning number d ranges between 1 and n. If we coarsely partition the problem space, i.e., take a small d, the problem solver or the target tends to move within a subproblem space. Hence, the term of $O((n/d)^2)$ becomes dominant in the stepwise cost of map maintenance. When $d = 1$, the term of $O((n/d)^2)$ becomes $O(n^2)$. In this case, a single local map works as a detailed entire map. On the other hand, if we finely partition the problem space into small pieces, i.e., take a large d, the problem solver or the target will frequently stride across boundaries. Hence, the term of $O(nd)$ becomes dominant in the stepwise cost of map maintenance. When $d = n$, the term of $O(nd)$ becomes $O(n^2)$. In this case, the abstract map works as a detailed entire map. Now if we let d be $d \propto \sqrt[3]{n}$, both $O((n/d)^2)$ and $O(nd)$ become $O(n^{\frac{4}{3}})$, which are less than the value $O(n^2)$ for TBS.

5 Experimental Results of TBSA

We empirically evaluated the performance of TBSA, and for comparison, TBS. The problem space is a 50×50 rectangular grid, organized as a torus. We randomly placed obstacles on junctions, that prevent the problem solver or the target from moving to the occupying junctions. The obstacle occupation of the whole problem space was set in the range from 0% to 40% at intervals of 5%. According to preliminary experiments that tested the best division, we partitioned the problem space into 25 subproblem spaces by dividing each side of the grid equally into 5 parts. We set the speed of the target to 4/5 that of the problem solver by skipping the turn of the target once every five turns. We performed experiments for each TBSA and TBS, and for different strategies of targets. The strategies are, (1)*Avoid*: the target moves to the furthest adjacent node from the problem solver, (2) *Meet*: the target moves cooperatively to meet the

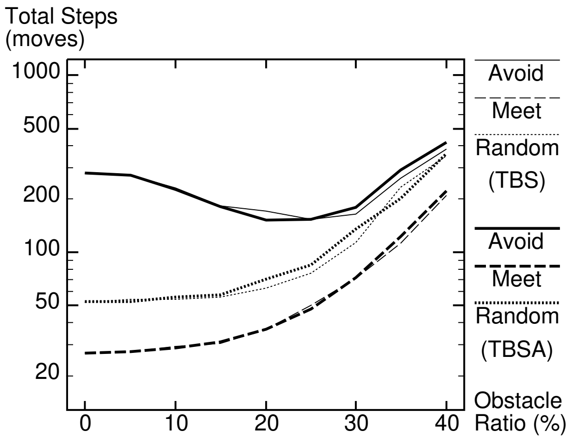

Figure 4: Total number of steps

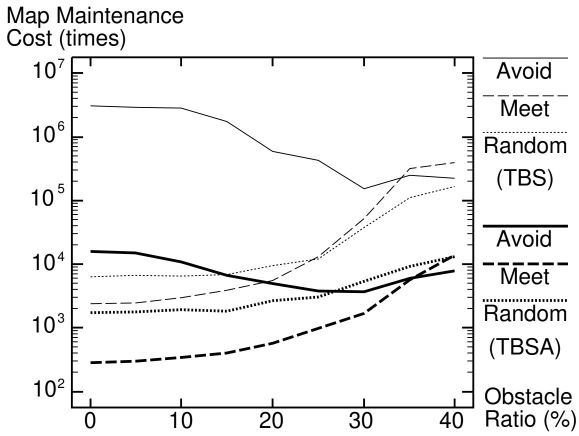

Figure 5: Cost of map maintenance

problem solver, i.e., searches for the problem solver, and (3)*Random*: the target moves randomly. For each experiment, we randomly created 100 sample grids, and averaged the results over all the samples. For each experiment, we counted the total number of search steps and the cost of map maintenance. The cost of map maintenance is regarded as the number of references to records on the routing table for each time the map is updated. Further, to evaluate the algorithm's reactiveness, we measured the cost of map maintenance for each 10 steps of the problem solver. Figures 4 and 5 show the total number of steps and the total map maintenance cost. Figure 6 shows the result of the cost of map maintenance for each 10 steps up to 150 steps, where the target's strategy is *Avoid* and the obstacle ratios are 0% and 20%.

As shown in figure 4, TBSA shows the same tendency as TBS for the total number of search steps. This is because TBSA differs from TBS only in how the map is maintained; the policy of movement is the same for both. When the obstacle ratio increases from 0% to 25%, it becomes easier for a systematic search like TBS and TBSA to capture the *Avoiding* targets. This shows that problem solvers can get out of dead ends faster than the targets[Chimura and Tokoro, 1994].

On the other hand, as shown in figure 5, TBSA significantly reduces the cost of map maintenance, compared with TBS. When the obstacle ratio is 40%, TBSA's cost of map maintenance reduces to 0.035 times that of TBS for the target's strategy of *Avoid* or *Meet*, and to 0.080 times for *Random*. The cost of map maintenance is especially high when the capturing process of the target includes a long chase phase. This is because, in the chase phase, the problem solver must make a combined map of both the problem solver and the target. The chase phase tends to become shorter when the obstacle ratio increases, since *Avoiding* targets tend to get trapped in dead ends. This explains why the *Avoid* curve dips as the obstacle ratio increases. The chase phase is usually longer for a target with the *Avoid* strategy than for one with other strategies. Hence, the biggest improvement of TBSA against TBS appears when the target's strategy is *Avoid*. For an *Avoiding* target and for an obstacle ratio of 0%, TBSA's cost of map maintenance is 0.005 times that of TBS. Within a practical time, TBSA can solve much larger problems than TBS, such as the moving target problem on a 200 × 200 grid.

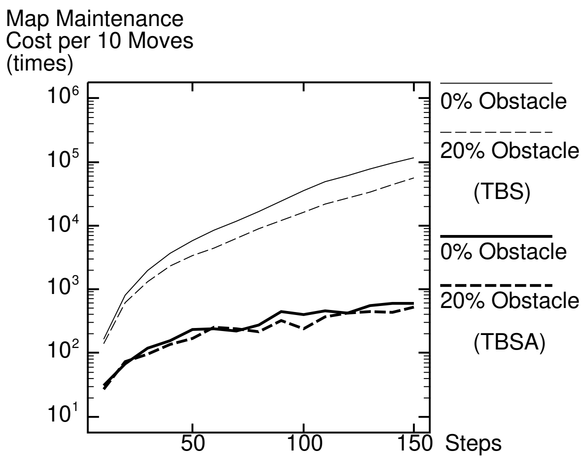

Figure 6: Cost of map maintenance for every 10 steps

Figure 6 shows that the cost of map maintenance of TBSA for each 10 steps increases slowly, while that of TBS increases rapidly, as the search proceeds. This result indicates significant improvement of the problem solver's reactiveness. We can confirm the improvement on a computer display; TBSA responds instantly, while TBS spends quite a long time to take a single step. Nevertheless, the reaction time of TBSA is not constant. Since the number of nodes in a subproblem space is limited to a constant, we predict that the maintenance cost of a local map is constant as well. On the other hand, since the abstract map controls local maps, and since the number of local maps is regarded as unlimited, as for the abstract map, we cannot set any plausible boundary to the cost of maintenance. However, the increase in the number of nodes included in the abstract map is so slow that maintenance cost of the abstract map has little impact on the total cost of map maintenance.

6 Related Research

In the field of AI planning, there has been plenty of research dealing with *hierarchical problem solving*. According to [Prieditis and Janakiraman, 1993], it is a three step problem-solving paradigm:(1)*abstraction*, (2)*problem solving*, and (3)*reconstitution*. After organizing an abstract problem space, a plan is first inferred in that problem space. The intermediate states of the abstract plan are then used as intermediate goals to guide the search for a more detailed plan. Knoblock showed that this method can reduce the search complexity from exponential to linear when searching for a solution with the same length[Knoblock, 1991]. In TBSA, the phase of organizing the hierarchical map corresponds to the step of *abstraction*, and the phase of reconstructing paths from the hierarchical map corresponds to the steps of *problem solving* and *reconstitution*. The major issue of TBSA and hierarchical problem solving in general, is the need of a cost effective abstraction scheme. The *ALPINE* system[Knoblock, 1990] automatically forms abstraction hierarchies by analyzing the structure of the problem space before *problem solving*. Even though this automation isn't directly applicable to the dynamic nature of map construction, the idea of learning hierarchies is useful for TBSA. For example, instead of using a predefined function for partitioning the problem space, TBSA might initially explore a small region of the search space and predict an appropriate partition for the whole problem space. This will be one way of making TBSA applicable to more general search problems other than problems on a grid.

Hierarchical problem solving is regarded as an important technique for realizing real-time problem solving[Strosnider and Paul, 1994]. We introduced the paradigm into the map maintenance of TBS in order to improve the problem solver's reactiveness.

7 Conclusions and Future Work

We dealt with the problem of searching moving targets. The information about the region explored by the search algorithm is predicted to be especially useful for searching moving targets. However, the concern for this method is that the map becomes larger as the algorithm progresses. TBSA is a method that uses a hierarchical map of where the algorithm has explored. TBSA handles the problem space in partitions, called local maps. It maintains an abstract map on top of the local maps, in order to systematically manage the information of the problem space.

We formally and empirically analyzed TBSA in a problem where a problem solver searches for a target on a grid-like problem space with randomly placed obstacles. First, we formally showed that TBSA reduces the cost of map maintenance for each step of the problem solver from $O(n^2)$ to $O(n^{\frac{4}{3}})$, where n is the number of nodes in the problem space. Then, with results from simulations, we showed that TBSA actually reduced the cost of map maintenance. TBSA inherits the simplicity and completeness of TBS; however, there is significant improvement in the algorithm's reactiveness. We found that, within a plausible time, TBSA could solve problems of sizes up to a 200×200 square grid. TBS could not solve a problem of this size within a plausible time.

For larger problems, it might be beneficial to construct a hierarchical map that has more than two layers. It will be useful if the search algorithm can control the number of layers adaptively. We plan to research this strategy further.

References

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Chimura and Tokoro, 1994] Fumihiko Chimura and Mario Tokoro. The Trailblazer Search: A New Method for Searching and Capturing Moving Targets. In *Proceedings of AAAI-94*, pages 1347–1352, 1994.

[Chimura, 1994] Fumihiko Chimura. *The Trailblazer Search: A Study on Searching Moving Targets*. PhD thesis, Department of Computer Science, Faculty of Science and Technology, Keio University, 1994.

[Ishida and Korf, 1991] Toru Ishida and Richard E. Korf. Moving Target Search. In *Proceedings of IJCAI-91*, pages 204–210, 1991.

[Ishida, 1992] Toru Ishida. Moving Target Search with Intelligence. In *Proceedings of AAAI-92*, pages 525–532, 1992.

[Knoblock, 1990] Craig A. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *Proceedings of AAAI-90*, pages 923–928, 1990.

[Knoblock, 1991] Craig A. Knoblock. Search Reduction in Hierarchical Problem Solving. In *Proceedings of AAAI-91*, pages 686–691, 1991.

[Korf, 1990] Richard E. Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42(2–3):189–211, 1990.

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

[Prieditis and Janakiraman, 1993] Armand Prieditis and Bhasker Janakiraman. Generating Effective Admissible Heuristics by Abstraction and Reconstitution. In *Proceedings of AAAI-93*, pages 743–748, 1993.

[Strosnider and Paul, 1994] Jay K. Strosnider and C. J. Paul. A Structured View of Real-Time Problem Solving. *AI Magazine*, 15(2):45–66, 1994.