The Trailblazer Search: A New Method for Searching and Capturing Moving Targets

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
{chimura, mario}@mt.cs.keio.ac.jp

Abstract

This paper proposes a new search algorithm for targets that move. Ishida and Korf presented an algorithm, called the *moving target search*, that captures a target while deciding each search step in constant time (Ishida & Korf 1991). However, this algorithm requires many search steps to solve problems, if it uses a heuristic function that initially returns inaccurate values. The *trailblazer search* stores path information of the region it has searched and exploits this information when making decisions. The algorithm maintains a *map* of the searched region, and chases the target once it falls on a path found on the map. We empirically show that the algorithm's map function can significantly reduce the number of search steps, compared with the moving target search. We also discuss the efficiency of the trailblazer search, taking the maintenance cost of the map into consideration.

Introduction

Heuristic search for moving targets models the case where the location of the goal changes dynamically. This assumption is realistic because there are many applications in which the goal moves before the search reaches the original location. Search for moving targets is a real-time task that interleaves decision and execution of search steps (Korf 1990). The search is illustrated intuitively when you try to meet someone in a crowd. You have to be careful in performing the search: there is a possibility that you and the person will cover the same ground many times before you actually meet. We study the search in an abstract search space as multiple paths finding to every possible location of the goal.

Ishida and Korf proposed the *moving target search* (MTS) algorithm (Ishida & Korf 1991). MTS tries to learn the exact distance to the target, while exploring the search space. It starts with an initial heuristic estimate of the distance. After the exact distance to the target is known, the search is just the process of catching the target up. The distinctive characteristic of MTS is that it makes decisions in constant time. The major concern in MTS is that it requires many search steps to solve a problem, if the initial heuristic estimate of the distance to the target greatly differs from the exact value.

This paper investigates an underlying intelligence that reduces the number of search steps, in the search for moving targets. We propose the *trailblazer search*: a method that stores path information of the region it has searched, and exploits this information when making decisions.[1] Information of the searched region is organized into a *map* that contains paths and associated costs to every site in the region from the current location of the algorithm. The trailblazer search uses a heuristic function, in an algorithm that conducts a systematic search and avoids exploring the same region twice. Once the target crosses a path on the map, the algorithm follows the path to catch the target up. Under the assumption of moving faster than the target, the trailblazer search captures the target.

The trailblazer search makes decisions using a map that gets larger as the algorithm steps further. We will discuss the efficiency of the trailblazer search, taking the maintenance cost of the map into consideration.

Search for Moving Targets

We study the search for moving targets in an abstract search space: a connected and undirected graph with a unit cost on each edge. A problem solver searches for a target representing the goal. At any point during search, both the problem solver and the target are assigned nodes in the graph that represent their states, or simply denote their locations. They can move to any node adjacent to their current locations. We assume that they move on alternate turns. The problem solver and the target are assigned initial nodes at the beginning of the search; the search ends when their locations coincide. We say a node is *explored* if it has been the location of the problem solver.

We make three assumptions to assure a solution to the search. The first is that the problem solver always

*Also with Sony Computer Science Laboratory Inc., 3-14-13 Higashigotanda, Shinagawa–ku, Tokyo 141, Japan.

[1] We use the term "trailblazer" in the sense of a guide or pathfinder since it uses a map.

knows the location of the target. The second is that the problem solver has a heuristic function that returns an estimated distance between any two nodes in the search space. We further assume that the heuristic function is initialized to return non-overestimated values, that is the function is *admissible* (Pearl 1984). The last assumption is that the problem solver moves faster than the target. In the particular search space defined above, we realize this by eventually skipping the turn of the target.

Search for moving targets is a real-time task that interleaves decision and execution of search steps. This is the result of practical resource limitations that do not allow a method to first find a sequence of steps that leads to the goal, and then execute the sequence. Korf presented the *learning real-time A** (LRTA*) algorithm that reaches the goal while making decisions in constant time (Korf 1990). Basically, LRTA* starts with an initial heuristic estimate of the distance from a node in the search space to the goal and, while exploring the search space, tries to find the exact distance. In the learning step, the heuristic distance $h(x)$ between node x and the goal is updated to the value of $min_i\{c(x, x_i) + h(x_i)\}$, where x_i is a node adjacent to x, $c(x, x_i)$ is the actual distance from x to x_i, and $h(x_i)$ is the heuristic distance between x_i and the goal. This update is done to ensure that $h(x)$ is not smaller than the estimated length of the path from x to the goal that goes through a node adjacent to x.

Ishida and Korf extended the learning method of LRTA* to tackle the search problem for moving targets (Ishida & Korf 1991). Their algorithm, the *moving target search* (MTS), learns the exact distance between any pair of nodes in the search space. This capacity is added because both the problem solver and the target can move to any location in the search space. Once the problem solver knows the complete set of exact distance values, the search task is reduced to moving to the adjacent nodes that are closer to the target. MTS is guaranteed to reach the target if the search problem follows the above assumptions. The worst case time complexity of MTS is $O(N^3)$, and the worst case space complexity is $O(N^2)$, where N is the number of nodes in the search space.

The major concern of LRTA* and MTS is that they require a significant number of search steps to reach the goal, if their heuristic functions initially return inaccurate values. In MTS, excessive search steps is typically shown using a target on a plane with randomly placed obstacles. When trapped in dead ends, the problem solver moves back and forth because the obstacles prevent direct movement to the target placed just beyond the obstacles. Ishida presented an extension to MTS that conducts a *lookahead* search in order to get out of dead ends fast (Ishida 1992). His algorithm, the *Intelligent Moving Target Search* (IMTS), considers the tradeoff between the increased computation cost to explore nodes with the lookahead search, and the reduced execution cost to reach the target with fewer steps. The learning method in LRTA* and that in MTS is a sort of *reinforcement learning*; learning from positive and negative rewards of executing search steps. Whitehead, and also Koenig and Simmons studied the issue of excessive search steps in the context of reinforcement learning (Whitehead 1991; Koenig & Simmons 1993).

The Trailblazer Search

We propose a new search algorithm for targets that move. We aim at reducing the number of search steps, yet guaranteeing the accomplishment of the task. The basic idea is to store path information of the region where the algorithm has searched, and exploit this information for the task. The information of the searched region is especially useful when the target is moving because, as the region expands, there is a good chance that the target will cross a path that the problem solver has already used for search.

The problem solver records every search step it takes by remembering an undirected edge connecting the departure and arrival nodes of the step. This record is organized into a graph called the **trail**. The **map** is a table calculated from the trail and it contains the information of minimum cost paths from the current location of the problem solver to any node it has explored. The map is a partial map of the whole search space, and is relative to the search steps that the problem solver has taken so far. Thus we consider it as a relative search tree rooted at the current location of the problem solver. In the following, we describe how we maintain the map.

Let p_n be the node the problem solver reaches after n steps, where n is a non-negative integer. We assume that p_n takes values from a set of integers that identify the actual nodes in the search space. Let V_{p_n} and E_{p_n} be the sets of accumulated nodes and edges at location p_n. Weighted and undirected edges are denoted by triplets, $(p_n, p_{n+1}, c(p_n, p_{n+1}))$, where $c(p_n, p_{n+1})$ is the actual cost to traverse the edge.

Definition 1 (The Trail) *Let p_0 be the initial node of the problem solver. The trail of the problem solver at location p_0 is a weighted and undirected graph $G_{p_0} = (V_{p_0}, E_{p_0})$, where the set of nodes V_{p_0} is $\{p_0\}$ and the set of edges E_{p_0} is $\{\}$.*

Assume that the problem solver moves from p_{n-1} to p_n on its nth step with actual cost $c(p_{n-1}, p_n)$, where n is a positive integer. The trail of the problem solver at location p_n is a weighted and undirected graph $G_{p_n} = (V_{p_n}, E_{p_n})$, where the set of nodes V_{p_n} is $V_{p_{n-1}} \cup \{p_n\}$ and the set of edges E_{p_n} is $E_{p_{n-1}} \cup \{(p_{n-1}, p_n, c(p_{n-1}, p_n))\}$.

We assume that the problem solver is able to record the trail of the target and to maintain the map of the target. This means that the location of the target is regarded as a sort of explored node for the problem

solver. Using the above procedure, we define the trail of the target at location q_m to be a graph $G'_{q_m} = (V'_{q_m}, E'_{q_m})$, where q_m is the node the target reaches after m steps, where m is a non-negative integer. The problem solver is able to refer to a graph $G = (V_{p_n} \cup V'_{q_m}, E_{p_n} \cup E'_{q_m})$ at location p_n as the total trail of the search.

We use Dijkstra's *shortest path algorithm*, a *routing algorithm*, to calculate the map. Dijkstra's algorithm has a time complexity of $O(N^2)$ when the number of nodes in the search space is fixed to N (Aho, Hopcroft, & Ullman 1974). Let $C(x, y)$ be the cost of the minimum cost path from the current location x of the problem solver to some node y in the trail. The path goes only through nodes in the trail, and the cost of the path is the sum of the costs of the edges that constitute it. The routing algorithm calculates a *routing table* that holds the value for $C(x, y)$, and the node x' succeeding x on the path to y.[2] The value of $C(x', y)$ is simply $C(x, y) - c(x, x')$, where $c(x, x')$ is the cost of the edge between x and x'. The value of $C(x, y)$ is ∞ if there is no path between x and y. In other words, we know that there is a path from x to y if $C(x, y)$ has a finite value. The map is nothing else than the routing table calculated from the trail by the routing algorithm.

We use **trailblazer search** to refer generally to algorithms that maintain a map to perform a search. A trailblazer search has two distinct phases: (1) a *search phase* in which the map has no path to the target, and the search space is heuristically searched, and (2) a *chase phase* in which the map has a path to the target that is deterministically followed to catch the target up. The phases are characterized by the evaluation functions they use to determine which adjacent node to move to next. In the search phase, decisions are made using the heuristic estimates of the distances from the adjacent nodes to the goal. In the chase phase, decisions are made using the costs on the map that indicate the best known paths from the adjacent nodes to the goal. We describe a particular instance of the trailblazer search that uses hill-climbing in the search phase.

The Trailblazer Search with Hill-Climbing

Let x and y be the locations of the problem solver and the target. Let $h(x, y)$ be the heuristic estimate of the distance between x and y, and $C(x, y)$ be the cost of the minimum cost path between x and y. To eliminate unnecessary re-exploration of the same node, we assume that the problem solver holds the set of explored nodes. We also assume that the problem solver records, for each explored node, the *parent* node from which it has arrived to the node for the first time.

[2]We applied some minor modification to Dijkstra's algorithm to avoid repeated calculation for known paths.

Procedures of the problem solver when it is its own turn to move.

Update the routing table according to the location x of the problem solver. For each node x' adjacent to x, read $C(x', y)$ to find if there is a path from x' to y. If there is no path on the map, enter the search phase. Otherwise, enter the chase phase.

- *In the search phase*:
  For each non-explored node x' adjacent to x, calculate $h(x', y)$, move to the node x' with minimum $h(x', y)$. If all of the adjacent nodes have been explored, move to the node x' that is the parent node of x. Assign the value of x' to x as the new location.
- *In the chase phase*:
  Move to the adjacent node x' with minimum $C(x', y)$, and assign the value of x' to x as the new location.
- *In both the search phase and the chase phase*:
  Record the move of the problem solver in the trail.

Procedures of the problem solver when it is the target's turn to move.

- *In both the search phase and the chase phase*:
  Record the move of the target in the trail.

If a tie occurs such that two node evaluations return the same cost, the tie is broken randomly.

The algorithm is *complete* in the sense that it never fails to capture the target if the search problem follows the assumptions in the previous section. As a short proof, consider first the search phase. Since the search space is finite, the algorithm must either find the target or the trail of the problem solver and the target must overlap before all the search space is explored. If the trails overlap, the algorithm enters the chase phase because there is now a path to the target on the map, whatever steps the target may take. The cost of the path never increases when the problem solver and the target alternate turns. Hence, under the assumption that the problem solver moves faster than the target, it will eventually reach the target.

We will now analyze the complexity of the algorithm. Let N be the number of nodes in the search space, and M the number of search steps. The time complexity of the trailblazer search is the number of search steps plus the time complexity of map maintenance. The time complexity of map maintenance on the Mth step of the problem solver is $O((2*M)^2) = O(M^2)$. Although we omit a formal proof, this can be seen intuitively because the total number of steps of the problem solver and the target, and thus the number of nodes in the total trail, is at most $2*M$ when the problem solver moves M steps. The algorithm re-calculates the map every time the problem solver moves. Thus the time complexity of map maintenance is $O(\sum_{i=0}^{M} i^2)$ when the problem solver moves M steps. Since this value is bounded by $O(M^3)$, the time complexity of the trail-

blazer search is $O(M + M^3)$ when the problem solver moves M steps.

In the search phase, the steps of the problem solver are along a search tree that grows to exhaust the search space. Since this tree has at most $N-1$ edges and it takes $2*(N-1)$ steps to traverse the whole tree, in the search phase, the worst case of the number M of search steps is $2*(N-1)$. In the chase phase, the steps of the problem solver are along a path found on the map. Since the initial cost of this path is at most $N-1$, in the chase phase, the worst case of M is $(N-1)/\alpha$, where α is the difference in speed between the problem solver and the target; it takes $(N-1)/\alpha$ steps to catch the target up. In each phase of the search, M is bounded by $O(N)$, if we assume α is a constant.

Consequently, the algorithm's worst case time complexity is $O(N + N^3)$ that is bounded by $O(N^3)$. The worst case space complexity is $O(N^2)$ because this is the size of the complete routing table. Note that these values are the same as those for MTS. However, the factor that determines the worst case time complexity of the trailblazer search differs from that of MTS. In MTS, the determining factor is the number of search steps. In the trailblazer search, it is the time complexity of map maintenance, since the worst case of the number of search steps reduces from $O(N^3)$ to $O(N)$, compared with MTS. This indicates that the trailblazer search considers the tradeoff between the computation cost to maintain the map, and the execution cost of the search steps.

Performance of the Trailblazer Search

We empirically evaluate the performance of the trailblazer search. We also discuss its efficiency, while considering the maintenance cost of the map. The search space of our problem is a rectangular grid with randomly placed obstacles. The problem solver and the target move along the grid from junctions to adjacent junctions, but not to those occupied by obstacles. Logically, increasing the number of obstacles changes the regularity of the search space and causes the heuristic function to return an increasing number of errors. The grid is a square of 50 junctions on each side, organized as a torus.

We implemented the trailblazer search with hill-climbing (denoted by TBS), and, for comparison, the moving target search algorithm by Ishida and Korf (denoted by MTS). The problem solver uses the Manhattan distance as its heuristic function. We performed four experiments, using for each, a different movement strategy of the target. (1) *Avoid*: the target moves to the furthest adjacent junction from the problem solver, estimated by the Manhattan distance, (2) *Stationary*: the target does not move, (3) *Random*: the target moves randomly, (4) *Meet*: the target moves to meet the problem solver, i.e., searches for the problem solver. With the *Avoid* strategy, the target executes the same learning method as MTS in the hope to flee from the problem solver cleverly. With the *Meet* strategy, the target uses the same search method as the problem solver (TBS matches TBS, MTS matches MTS). 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. The problem solver and the target are initially separated diagonally on the square grid at a distance of 50 junctions in the Manhattan distance.

We set the obstacle ratio between 0% and 40% at intervals of 5%, and randomly created 100 sample grids for each obstacle ratio. Both TBS and MTS solved the same samples, and we averaged the total number of search steps of the problem solver over the 100 samples. For TBS, we also measured the actual complexity of map maintenance while solving the samples. The actual complexity is calculated by counting the number of references to entries on the routing table for each time the map is updated. We do this because references to entries are the actual operations that decide the complexity of map maintenance. We will briefly mention some results of the actual complexity of map maintenance.

Figure 1 shows the number of search steps when we change the obstacle ratio. We obtained similar results for problem sizes ranging from 20 to 50 junctions on each side of the grid. TBS and MTS show almost the same performance when there are few obstacles. However, as the obstacle ratio increases and the heuristic function begins to return inaccurate values, TBS significantly reduces the number of search steps, compared with MTS. When the obstacle ratio is 40% and when the target strategy is *Avoid*, *Stationary*, *Random*, *Meet*, the proportion of the number of search steps of MTS to that of TBS is $3475/405 \simeq 8$, $1586/444 \simeq 3$, $6313/376 \simeq 16$, $20981/180 \simeq 100$, respectively. The reason for the gain is that TBS can use the map to detour walls of obstacles, and get out of dead ends fast.

By the nature of their movements, targets with the *Random* strategy do not move far from their initial locations. When the obstacle ratio is high, targets with the *Avoid* strategy get trapped in dead ends and do not move far either. Thus, for a systematic search like TBS and a high obstacle ratio, targets with the *Random* and the *Avoid* strategies are as easy (takes as much search steps) to capture as targets with the *Stationary* strategy. Actually, with a high obstacle ratio, there is a slight tendency that moving targets will be easier (takes less search steps) to capture. The fact that TBS collects more information of the search space from the movements of the target, explains the result. We predict that these results will hold, even if we change the speed ratio between the problem solver and the target because change in speed does not affect the target's strategy.

As seen in figure 1, when there are many obstacles, MTS shows a counterintuitive behavior in that targets with the *Meet* strategy are far more difficult to capture than those using the *Avoid* strategy, even though tar-

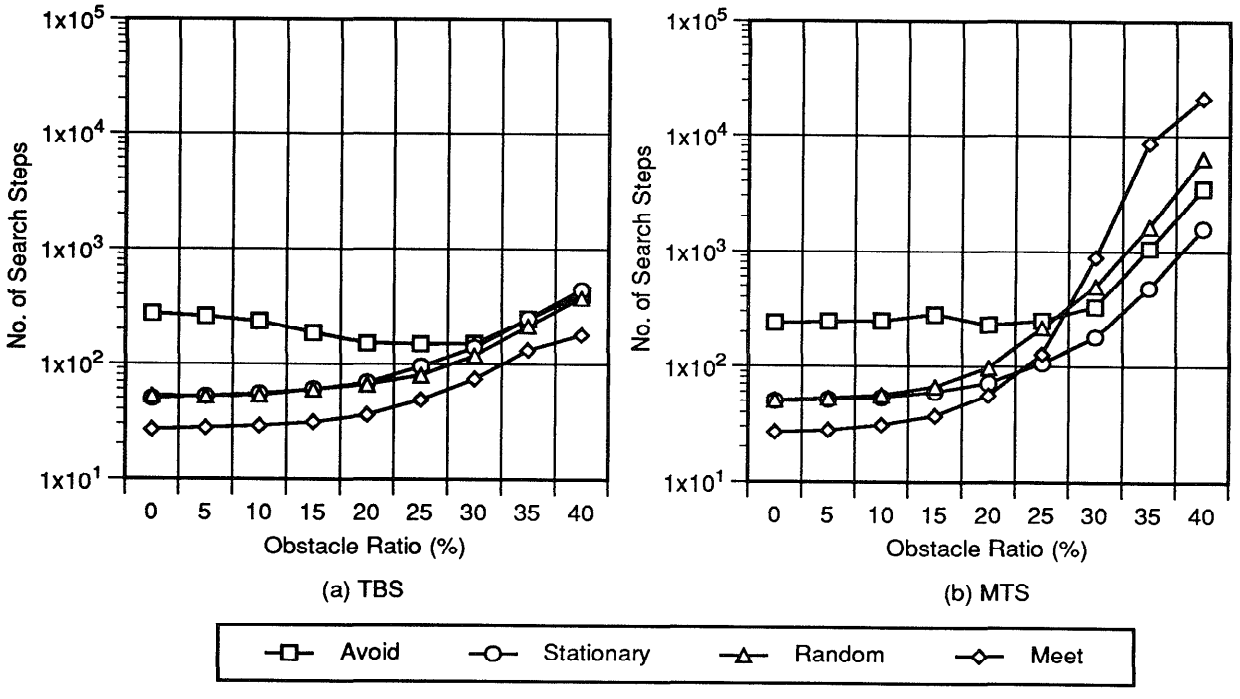

Figure 1: Results of the number of Search Steps

gets with the *Meet* strategy cooperate with the problem solver. Ishida and Korf pointed out this result in their paper (Ishida & Korf 1991). This doesn't happen in TBS; in TBS, targets with the *Meet* strategy are the easiest to capture, regardless of the obstacle ratio. Increasing the target's speed increases the efficiency of TBS, reducing the number of search steps. As we decrease the target's speed, the number of search steps approaches that TBS needs to capture targets with the *Stationary* strategy.

An interesting characteristic of TBS is that it does not take the optimal path even if there are no obstacles (and hence the Manhattan distance gives the correct distance to the target). When the obstacle ratio is 0% and the target executes the *Avoid* strategy, TBS takes 275 steps to capture the target while MTS takes the optimal path of 239 steps. This happens because the current implementation of TBS is completely *faithful* to the map, in the sense that the problem solver always follows the map whenever a path is found, even if the distance to the target in the Manhattan distance is smaller than the cost of the path.

Evaluating the Real Efficiency of the Trailblazer Search

For TBS, the drastic reduction of search steps is due to the benefit of the map. Here we estimate the total cost of search, and discuss the real efficiency of TBS, while considering the maintenance cost of the map. We emphasize that, from the viewpoint of the worst case time complexity, TBS and MTS have identical efficiency. Compared with IMTS (the extended form of MTS), TBS exchanges the cost to explore information with a lookahead search, for the cost to exploit information of the searched region. The total cost of search is the sum of the cost to make decisions on search steps and the cost to execute them. We simply assume that the decision cost is proportional to its time complexity, and that the execution cost is proportional to the number of search steps. We further assume that the search steps are decided in a computer and are executed in the real world. That is, we weight the decision cost by $1/\beta$, where β is a large number. Formally, this makes the total cost for TBS, $T_{TBS} = M_{TBS} + M_{TBS}^3/\beta$, and for MTS, $T_{MTS} = M_{MTS} + M_{MTS}/\beta$, where M_{TBS} and M_{MTS} are the numbers of search steps that each algorithm requires to solve a problem. The fact that TBS maintains a map to make decisions, and that MTS makes a constant time decision, justifies the difference between T_{TBS} and T_{MTS}.

For example, we set β to 10^6. This is a plausible figure because, in the real world, search steps usually take seconds of time to execute, while, in current computers, the unit operation of map maintenance takes on the order of microseconds. When the obstacle ratio is over 30% and the target is moving, the empirical results of the actual complexity of map maintenance ranges from $4.8*10^3$ (30% obstacles, *Meet* strategy tar-

get) to $6.7 * 10^4$ (40% obstacles, *Random* strategy target). These values have small impact on T_{TBS}, when we weight them by 10^{-6}. Thus, in our empirical search space, when the obstacle ratio is high and the target is moving, T_{TBS} is smaller than T_{MTS} because, as we observe in figure 1, M_{TBS} is smaller than M_{MTS} by a factor of 10 in average. This indicates the applicability of TBS to real domain problems that have the same size (50*50 = 2500 nodes) as our empirical search space. For large scale problems, however, the maintenance cost of the map increases the time TBS uses to make decisions, and this becomes an issue for TBS if we look at its ability to react to the real world.

Currently, TBS maintains a full size map in the sense that it includes information concerned with all the explored nodes. The aim is to utilize plenty of alternative paths to efficiently and intelligently reach the target. The idea of limiting the size of the map, while preserving efficiency, is extremely important for two reasons: to improve the ability of TBS to react to the real world, and to reduce the average space complexity.

Conclusion

We dealt with search for moving targets, and proposed an efficient method to capture moving targets. The *trailblazer search* maintains a map of the searched region and uses this for the search. We compared the properties of the trailblazer search with those of the *moving target search* (Ishida & Korf 1991). Formally, the two algorithms have the same worst case time complexity of $O(N^3)$ where N is the number of nodes in the search space. However, the determining factors of the values are different. For the moving target search, the number of search steps determines its worst case time complexity. For the trailblazer search, this value is determined by the maintenance of the map, since the number of search steps reduces from $O(N^3)$ to $O(N)$ when compared with the moving target search.

To examine this difference, we tested the performance of the algorithms on a grid-like search space with randomly placed obstacles. We showed that when the algorithms use inaccurate heuristic functions, the trailblazer search significantly reduces the number of search steps, when compared with the moving target search. In a square grid with 50 junctions on each side and an obstacle occupation ratio of 40%, where the problem solver and the target searched for each other, we obtained a 100–fold reduction. We also estimated the impact of map maintenance on the total cost of the trailblazer search. We made an assumption that the map is maintained in a computer and the search steps are executed in the real world, and hence weighted the cost to maintain the map by a plausible factor. In our empirical search space, the impact of map maintenance is small when compared with the execution cost of search steps. This indicates the applicability of the trailblazer search to real domain problems. For large scale problems, a parallel implementation of the map maintenance method decreases the cost to maintain the map (Chandy & Misra 1982).

As a consequence, we showed that, for the purpose of efficient search, we can tradeoff the execution cost of search steps for the computation cost to maintain a map. When memory is limited, it is useful to consider a mixed strategy that uses the heuristic function while it returns accurate values, and maintains the map only if necessary. We plan to research this strategy further.

Acknowledgements

We had a fruitful discussion with Toru Ishida of Kyoto University. Members of Sony Computer Science Laboratory helped to improve initial concepts and colleagues at Keio University supported us invaluably. Finally, F. Chimura would like to express his deepest gratitude to his family for their support.

References

Aho, A. V.; Hopcroft, J. E.; and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Reading, Mass.: Addison-Wesley.

Chandy, K. M., and Misra, J. 1982. Distributed Computation on Graphs: Shortest Path Algorithms. *Communications of the ACM* 25(11):833–837.

Ishida, T., and Korf, R. E. 1991. Moving Target Search. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 204–210.

Ishida, T. 1992. Moving Target Search with Intelligence. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 525–532.

Koenig, S., and Simmons, R. G. 1993. Complexity Analysis of Real-Time Reinforcement Learning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 99–105.

Korf, R. E. 1990. Real-Time Heuristic Search. *Artificial Intelligence* 42:189–211.

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

Whitehead, S. D. 1991. A Complexity Analysis of Cooperative Mechanisms in Reinforcement Learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 607–613.