Real-Time Edge Follow: A Real-Time Path Search Approach

Cagatay Undeger and Faruk Polat

***Abstract*—Real-time path search is the problem of searching a path from a starting point to a goal point in real-time. In dynamic and partially observable environments, agents need to observe the environment to track changes, explore to learn unknowns, and search suitable routes to reach the goal rapidly. These tasks frequently require real-time search. In this paper, we address the problem of real-time path search for grid-type environments; we propose an effective heuristic method, namely a real-time edge follow alternative reduction method (RTEF-ARM), which makes use of perceptual information in a real-time search. We developed several heuristics powered by the proposed method. Finally, we generated various grids (random-, maze-, and U-type), and compared our proposal with real-time A*, and its extended version real-time A* with n-look-ahead depth; we obtained very significant improvements in the solution quality.**

***Index Terms*—Path planning, real-time heuristic search.**

I. Introduction

PATH planning can be described as either finding a path (a sequence of moves) from an initial state (starting point) to a goal state (target point), or finding out that no such sequence exists. Path planning algorithms are divided into two broad categories: off-line and on-line. Off-line path planning algorithms find the whole solution in advance, before starting execution. In other words, an agent plans a complete path and then follows it until the goal state is reached or the plan becomes infeasible. If the plan is infeasible, the agent needs to replan and continue execution. Dijkstra's algorithm [7], [8], [23] and A* [18] are typical off-line search algorithms, which are complete and optimal. There are also off-line techniques based on genetic algorithms [13], [17], [22] and random trees [1], [2], [15], [16]. In dynamic environments, the need for replanning emerges frequently and off-line algorithms are not suitable. On-line search algorithms require the planning and execution phases to be coupled, such that the agent repeatedly plans and executes the next move. Real-time path planning algorithms such as Real-Time A* (RTA*), Learning Real-Time A* (LRTA*) [6], [14], Moving Target Search (MTS) [4], and Bidirectional Real-Time Search [5] are typical on-line algorithms that are not designed to be optimal. Furthermore, there are some hybrid solutions such as incremental heuristic search algorithms D* [20], Focused D* [21], and D*Lite [9], [10], which are optimal and more efficient than off-line path planning algorithms. A comparison of D*Lite and LRTA* can be found in [11].

Manuscript received April 12, 2005; revised October 21, 2005. This paper was recommended by Associate Editor E. Trucco.

The authors are with the Department of Computer Engineering, Middle East Technical University, 06531 Ankara, Turkey (e-mail: cundeger@ceng.metu.edu.tr; polat@ceng.metu.edu.tr).

Digital Object Identifier 10.1109/TSMCC.2007.900663

In dynamic or partially observable environments, off-line path planning algorithms suffer from execution/response time, whereas on-line algorithms yield low quality solutions in terms of path length. Incremental heuristic search algorithms try to merge advantages of both approaches to obtain better execution/response time without sacrificing optimality. However, they are still slow for some real-time applications. Furthermore, almost none of these algorithms can be applied to moving target search problems. Only few are capable of handling moving targets, but their performance is, in general, not acceptable. Our motivation is to develop an on-line path search algorithm that offers lower total execution time and shorter paths, and is applicable to moving targets. To achieve this objective, we have focused on the most critical problem of on-line path search in complex girds, namely revisiting the same locations (states) too many times, which causes unacceptably long paths.

We propose an effective heuristic method called the real-time edge follow alternative reduction method (*RTEF-ARM*). We have also developed several algorithms that we call *RTEF algorithms*; they are based on RTEF-ARM for real-time path planning in grid environments. Unlike many real-time search algorithms, RTEF algorithms are able to make use of global environmental information. The basic idea is to analyze the perceptual information and known parts of the environment to eliminate the closed directions that do not lead the target point; this helps to determine which way to move. Environmental information is effectively used, resulting in much shorter paths and lower execution times. We have compared the RTEF algorithms with a well-known algorithm, namely RTA* and its extended version (RTA* with n-look-ahead depth) introduced by Richard Korf [14], and obtained significant improvements over both.

The rest of this paper is organized as follows. Related work on path planning is given in Section II. In Section III, RTEF and RTEF-ARM are described in detail. The complexity analysis of RTEF-ARM is given in Section IV, and its proof of correctness is presented in Section V. The performance analysis of our method is presented in Section VI. Section VII is the conclusion.

II. Related Work

Off-line path planning algorithms are hard to use for large dynamic environments because of their time requirements. One solution is the incremental heuristic search algorithms [12], which are continual planning techniques that make use of information from previous search results to find solutions to the problems, potentially faster than those possible by solving the problems from scratch. D* [20], focused D* [21], and D* Lite [9], [10] are some of the well-known optimal incremental heuristic search

algorithms applied to path planning. In D*, an agent plans an optimum path off-line, and then follows this path until a change in the environment occurs that triggers replanning. Sometimes, a small change in the environment may cause replanning of almost a complete path which surely takes a considerable time. Therefore, these algorithms can be considered as efficient off-line path planning algorithms because they do not plan the next step regularly, but partially replan the whole path on every environmental change.

Because of the efficiency problems of off-line planning techniques, a number of approaches that can work on-line without having any precomputed solutions are proposed. LRTA* [14] introduced by Korf is one of the real-time heuristic search algorithms for fixed goals. It builds and updates a table containing admissible heuristic estimates of goal distances of all the states. The initial table values are set to zero and the agent is made to explore the search space and learn exact goal distances through a finite number of trials. It was proven that the algorithm is complete and the table values converge to optimal values after a fixed number of trials, which can be quite large. Although LRTA* is capable of learning in real-time, the quality of solutions for the first trial is generally poor. To control the amount of effort required to achieve a short-term goal (to safely arrive at a location in the current trial) and a long-term goal (to find better solutions through repeated trials), Shimbo and Ishida introduced two techniques known as weighted LRTA* and upper-bounded LRTA* [19]. Korf also proposed another heuristic search algorithm called RTA*, which gives better performance in the first run but does not guarantee optimal solutions [14]. It repeats the following steps until reaching a goal state:

Step 1) Let x be the current state of the problem solver. Calculate $f(x') = h(x') + k(x, x')$ for each neighbor x' of the current state, where $h(x')$ is the current heuristic estimate of the distance from x' to a goal state, and $k(x, x')$ is the cost of the move from x to x'.

Step 2) Move to a neighbor with the minimum $f(x')$ value. Ties are broken randomly.

Step 3) Update the value of $h(x)$ to the second best $f(x')$ value.

For real-time path search in regular grid environments, LRTA* and RTA* are usually guided with Euclidian or Manhattan distance heuristics, and they are effective. However, if the grid is too large and there are many semi-closed regions with large open areas inside, the agent may get stuck in these regions for a long time due to the heuristic depression. A heuristic depression [4] is a set of states, which do not contain the goal state and have heuristic values less than or equal to those of a set of immediately and completely surrounding states. It is actually a local maximum, whose heuristic values have to be filled up before the agent can escape. A heuristic depression is illustrated on an example in Fig. 1.

The original RTA* uses *1-depth look-ahead* heuristic function, which is too poor to estimate the real cost of neighboring states. The algorithm can be improved by using heuristic function with n-look-ahead depth [14]. The structure of the algorithm is the same as RTA* except for the computation of $h(x')$. Instead of computing the $h(x')$ of a neighbor cell x' only from the cell itself, $(n-1)$ level neighbors of x' are used; thus, the search space is expanded up to a predefined look-ahead depth (n). When the look-ahead depth is set to 1, the effect is the same as the original RTA*. Although tests show that this improvement reduces the number of moves to reach the goal significantly, it requires exponential time in the look-ahead depth. Therefore, large look-ahead depths are not preferred in practice.

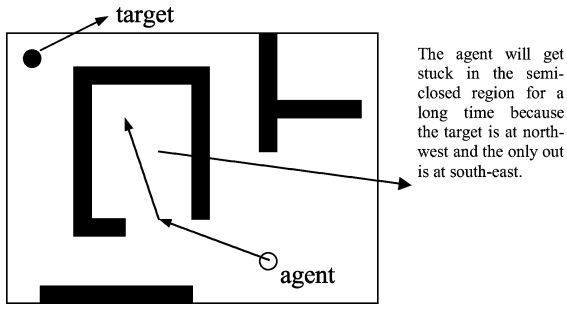

Fig. 1. Agent, directed by LRTA* and RTA*, will get stuck in the semiclosed region shown in the figure for a long time searching the same region hopelessly until the heuristic depression is filled.

LRTA*, RTA*, and their variations are all limited to fixed goal states. Ishida and Korf proposed an algorithm for moving targets called MTS [4]. MTS maintains a table of size N^2 that consists of heuristic values $h(x, y)$ for all x, y where $h(x, y)$ is the estimated distance between the problem solver location x and the target position y. MTS suffers from poor performance in case the target moves require the learning process to be started from scratch. This is even worse when the agent gets into a heuristic depression. To solve this problem, they introduced two methods called commitment to goals and deliberation. An interested reader can find more about these improvements in [4].

III. RTEF

In this section, we will describe a new real-time path search algorithm called RTEF for grid environments. We defined the environment as a rectangular planar grid. Each cell of the grid may be empty or may contain an agent, a target, or an obstacle. Although there is no restriction to have dynamic targets, we assumed that the target is static and its location is always known by the agent.

RTEF aims to find a path from an initial location to a fixed or moving target in real-time. The basic idea is to eliminate closed directions (the directions that cannot reach the target point) in order to determine which way to go (open directions). For instance, if the agent is able to realize that moving north and east will not lead to the target, he/she will prefer moving south or west. RTEF uses RTEF-ARM to find out open and closed directions and, hence, to eliminate nonbeneficial movement alternatives using perceptual information and uncovered tentative map. Initially, the agent is at a starting point and the goal is to reach a static or a dynamic target. The agent can move north, south, east, or west. Before each move, the RTEF-ARM algorithm is executed to detect the closed directions from the current cell. Then RTEF-ARM sets the heuristic values of the closed alternatives to infinity. The cost of moving to the next cell plus

Algorithm 1 Original RTEF Algorithm

Require: x is the current state of the agent
1: **if** x is the goal state **then** halt
2: $OpenDirections \leftarrow RTEF\_ARM(x)$ {RTEF_ARM(x) is the edge following alternative reduction function that determines the set of open directions that possibly reach the target location}
3: **if** $|OpenDirections| > 0$ **then**
4: Select the best direction in $OpenDirections$ using any heuristic function
5: Move to the selected direction
6: Insert x into the history
7: **else**
8: **if** $|OpenDirections| = 0$ and the history is not empty **then**
9: Clear all the history
10: Jump to 2
11: **else**
12: Destination is unreachable, stop search with failure.
13: **end if**
14: **end if**

the distance to the target is used as the heuristic function to select the move from the open alternatives. After performing the move, the previous cell is marked as an obstacle in the original RTEF [24]. This ensures that every cell is visited at most once in fully known environments and loops are prevented.

Definition 1 (History): The set of previously marked/visited cells is called the *History* of the agent. Cells in the history are assumed to be obstacles. Sometimes, the history might need to be cleared when it blocks the agent.

This definition assumes that the target is static. For moving targets, a simple check should be added to the algorithm to clear the history when the target touches any cell included in the history. The RTEF-ARM is based on the idea that every obstacle has a boundary, which is actually formed from a set of connected edges shaping the obstacle. RTEF-ARM splits the moving directions (north, south, east, and west) by a set of rays sent away from the agent, and analyzes each region to understand whether it is closed or open. Algorithm 1 sketches the RTEF algorithm which uses history to prevent loops and clears the whole history when necessary. In this paper, we separate RTEF-ARM from RTEF and generalize it. In addition, several new variations of the RTEF algorithm are proposed and tested.

Before going into details of the algorithm, it is better to clarify the concept of an obstacle. An obstacle in the mind of an agent is a set of neighboring grid cells that shape an object with a boundary. Note that each such cell can be a real-world obstacle cell or a history cell. A single cell might also form an obstacle. Due to the limited sensor capability, the agent may only perceive part of the real obstacle; in such a case, the known part will be considered as an obstacle. Furthermore, sometimes two or more real-world obstacles can be merged to give rise to a single obstacle because of the history cells that neighbor the real objects. These cases are illustrated in Fig. 2.

A. Edge-Following Alternative Reduction Method

RTEF-ARM finds out open directions that possibly reach the target location. RTEF-ARM executes the following steps: *initialization, ray-sending, edge-following*, and *edge-analyzing* as given in Algorithm 2. The details of these steps are given in the following sections.

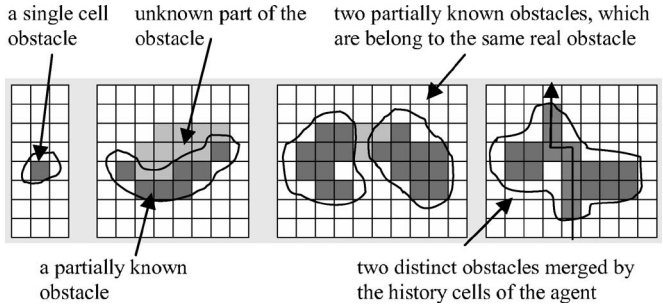

Fig. 2. Obstacle samples.

Algorithm 2 RTEF-ARM Algorithm

1: *Initialization Phase*: Mark all moving directions as open.
2: *Ray-Sending Phase:* Propagate all rays (4 diagonal rays are used in our implementation).
3: *Edge-following and edge-analyzing*:
4: **for** each ray hitting an obstacle **do**
5: Edge-Following phase: Follow the edges of the obstacle starting from the hit-point of the ray and moving to left, and find out an island and an hit-point-island if exists.
6: Edge-Analyzing phase: Analyze the edges using the island, hit-point-island and the target, and find out closed directions.
7: If number of open directions is zero, stop with failure (target is unreachable).
8: **end for**

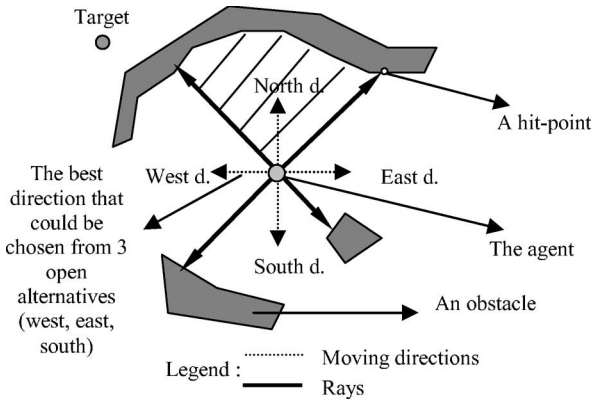

Fig. 3. Sending grays to split north, south, east, and west directions.

1) Ray-Sending: In this phase of RTEF-ARM, four diagonal rays splitting north, south, east, and west are propagated away from the agent as shown in Fig. 3. The known environment is split by the rays into four regions. The rays go away from the agent until hitting an obstacle or maximum ray distance is achieved. Types of ray-hitting are exemplified in Fig. 4. The reason for choosing only four diagonal rays is due to the nature of the grid world and movement alternatives. However, the idea can easily be generalized to n rays in different environment representations such as hexagonal grid worlds, polygonal worlds, etc.

2) Edge-Following: Four rays split the area around the agent into four regions. A region is said to be closed if the target is inaccessible from any cell in that region. If all the regions are closed, then the target is unreachable from the current location. To detect closed regions, the boundaries of obstacles that the rays hit are analyzed. If the edges on the boundary of an obstacle are

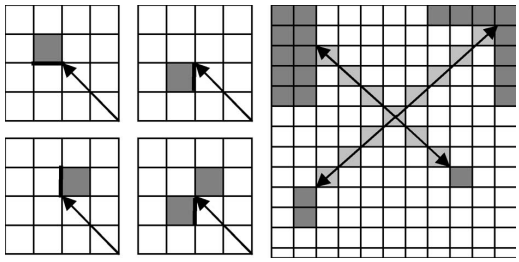

Fig. 4. Ray propagation and hitting.

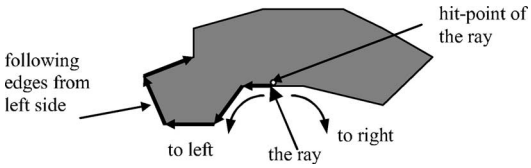

Fig. 5. Identifying the obstacle boundary.

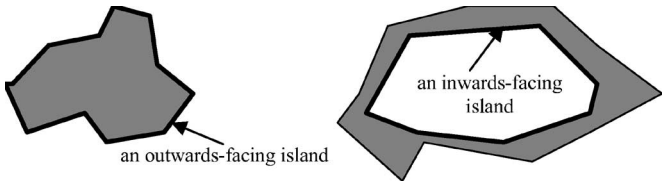

Fig. 6. Island types: outward-facing (*left*), inward-facing (*right*).

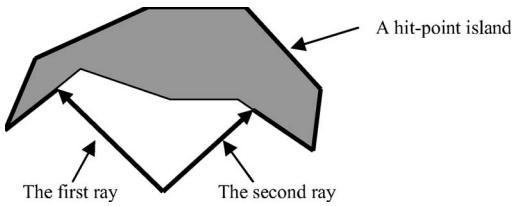

Fig. 7. Two rays hitting the same obstacle at two different points form a hit-point island.

followed in a fixed direction (to left or right) starting from a hit-point, we always return to the same point as illustrated in Fig. 5.

Definition 2 (Island): By following the edges to the left and returning to the same starting point, a polygonal area is formed as the boundary of the obstacle. We call this polygonal area an *island* (stored as a list of vertices forming the boundary of the obstacle). As shown in Fig. 6, there are two kinds of islands: *outward-facing* and *inward-facing islands*. The target is unreachable from current location if it is inside an outward-facing island or outside an inward-facing island.

Definition 3 (Hit-point island): More than two rays can hit the same obstacle. As illustrated in Fig. 7, an additional virtual polygonal area called *hit-point island* is formed when we reach the hit-point of another ray on the same obstacle while following the edges.

A hit-point island borders one or more agent moving directions. If the target point is not inside the hit-point island, all the directions that are bordered by the hit-point island are closed; otherwise (the target is outside the hit-point island) all the directions not bordered by the hit-point island are closed; this is illustrated in Fig. 8.

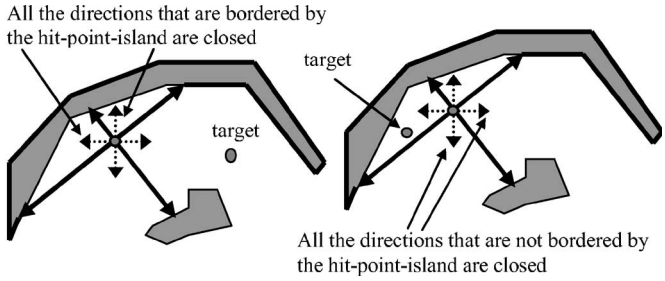

Fig. 8. Analyzing hit-point islands and eliminating moving directions.

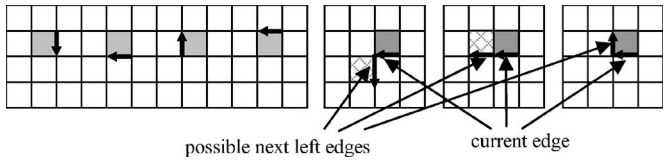

Fig. 9. Finding next left edge: The figure on the *left* shows four possible current edges. The figures on the right illustrate all possible next states for the current edge.

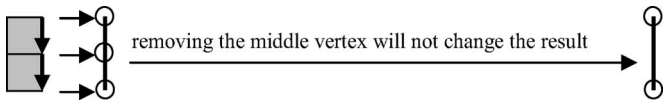

Fig. 10. After following right edges of two neighbor cells shown in figure, three vertex points are generated, but the middle one is unnecessary since removal of it will not change the resulting shape.

The current implementation of RTEF-ARM uses left direction in edge following. Alternative cases in edge following in the grid world are illustrated with some typical examples in Fig. 9.

Islands and hit-point islands are stored as vertex lists and passed to the edge-analyzing phase described in the next section. A vertex is identified *unnecessary* if its removal does not change the shape of the island or hit-point-island polygons (Fig. 10 contains a simple example). Note that we eliminate unnecessary vertices during the edge-following phase to decrease the workload of the edge-analyzing phase.

3) Edge-Analyzing: The edge-following and edge-analyzing phases are highly coupled. The edge-analyzing phase is performed after each edge-following phase in order to find out closed directions in the light of newly discovered edge information. The edge-analyzing phase for each ray is performed as given in Algorithm 3. Note that function `isInside(x, y, p)` returns *true* if coordinates (x, y) are inside polygon p and function `isClockwise(p)` returns *true* if the vertices of polygon p are ordered in clockwise direction (i.e., if polygon p is outward-facing with respect to the agent).

4) Integration of RTEF-ARM to Real-Time Path Search: RTEF-ARM is a general method for evaluating moving directions; it can be integrated into various grid-type algorithms. In our study, we applied RTEF-ARM to real-time path search and developed RTEF algorithms. In general, RTEF uses RTEF-ARM to find out open directions that possibly reach the target location. Later on, one of the open directions is selected

Algorithm 3 Edge-Analyzing

Require: (x, y) : coordinates of the target
Require: i : the list of vertices forming the island border,
Require: h : the list of vertices forming the hit-point-island border,
if $isClockwise(i) = isInside(x, y, i)$ **then**
  Close the entire directions (the target is unreachable){**Case 1**}
else if $|h| > 0$ **then**
  if $isInside(x, y, h)$ **then**
    if $isClockwise(h)$ **then**
      Close the directions between 1st and 2nd hit-points on i in counter clockwise direction {**Case 2**}
    else
      Close the directions between 1st and 2nd hit-points on i in clockwise direction {**Case 3**}
    end if
  else
    if $isClockwise(h)$ **then**
      Close the directions between 1st and 2nd hit-points on i in clockwise direction {**Case 4**}
    else
      Close the directions between 1st and 2nd hit-points on i in counter clockwise direction {**Case 5**}
    end if
  end if
end if

for the next move. After performing the move, agent information is updated in order to prevent infinite loops. We used several techniques for selecting from open directions and updating agent information. Table I describes several variations of RTEF algorithms.

TABLE I
VARIATIONS OF RTEF ALGORITHMS

RTEF-History is the original RTEF algorithm introduced in [24]. It selects one of the open directions for the next move using the Euclidian distance heuristic function. After moving to the next cell, the previous cell is added to the history. The history cells are considered as obstacles in RTEF-ARM, thus the visited cells are never visited again unless the history is cleared. If the grid is fully known, the history is never cleared during the search and the target is reached (if a solution exists) without revisiting any previously visited cells. But if the grid is not fully unknown, some of the newly discovered obstacles may block the way the agent is going, so all or a part of the history is needed to be cleared in order to be able to return back. The RTEF-History is the RTEF version, which clears all the history when the agent is totally blocked by a newly discovered obstacle.
RTEF-History-BC, where BC is an abbreviation for "Border Clear", is the same as RTEF-History except the history clearing method. When the agent is totally blocked by a newly discovered obstacle, RTEF-History-BC only clears the history cells encountered during the last edge-following phase. The reason behind this idea can be summarized as follows: Although there is no way found to reach the target, the target may still be reachable if the history is not empty yet, because the history cells may be blocking the way. If this is the case, failing of the last search must partially or totally be because of these history cells. Thus the algorithm goes over all the edges followed in the last edge-following phase, and clears all the history cells on these edges. This edge-following and history-clearing loop must continue until the last history edge blocking the agent is cleared or no history cell could be found to clear.
RTEF-Visited Count is a variation that does not use any history. To prevent infinite loops, number of visits (visited count) is stored for each cell. Thus the algorithm selects the neighbor cell having the minimum visited count value. If there are more than one minimum, the algorithm uses the Euclidian distance heuristic function. After moving to the next cell, the visited count of the previous cell is increased by one.
RTEF-Visited Count-History is a mixture of RTEF-Visited Count and RTEF-History. The algorithm uses both visited count and history together and clears all the history when the agent is totally blocked by a newly discovered obstacle.
RTEF-Visited Count-History-BC is a mixture of RTEF-Visited Count and RTEF-History-BC. The algorithm uses both visited count and history together and only clears the history cells found in the last edge-following phase.
RTEF-RTA* is the RTEF version, which performs RTA* integrated with RTEF-ARM. The algorithm uses the heuristic estimation update method of RTA* to guide the search and prevent infinite loops, and uses RTEF-ARM to reduce the possible moving directions. The closed directions found are considered as obstacles from the view point of the cell where the RTEF-ARM is executed, thus a cell might be evaluated different from different neighbors of that cell. This condition forces us to make a little change in RTA* heuristic estimation update. RTA* sets the second best heuristic estimation of the neighbors to the current cell before moving to another one. If there is just one alternative to select, the second best estimation will be infinite. If this is the case and there are some directions closed by RTEF-ARM, we should not mark the current cell as infinite, but mark with the best one, because there is a two way communication with these closed cells. Although you may not move to these neighbors from the current cell, you may go these cells from some other cells and may need to move to the current cell in order to escape from the area.
RTEF-RTA*-Penalty is similar to the RTEF-RTA* except a penalty extension. As mentioned in the previous section, the closed directions found are considered as obstacles from the view point of the cell where the RTEF-ARM is executed. So a cell might be evaluated as an obstacle from the view point of some of the neighbor cells but that may not be the case for some others. To prevent these non-promising cells from being visited from some other neighbors, the algorithm gives a penalty to the heuristic estimation of these cells according to a pre-defined rule. The main motivation of the rule comes from the fact that the heuristic estimation of a cell inside a closed region can not be better than the minimum heuristic estimation computed among all the other neighbors, because the algorithm prefers the most-promising cell having the minimum heuristic estimation rather than the cell inside the closed region. Thus the heuristic estimation of the neighbor cells inside closed regions can be set to the minimum heuristic estimation if they are smaller. We call this technique as RTEF-RTA*-penalty-0. If we add a penalty value (n) to the minimum heuristic estimation, it is called RTEF-RTA*-penalty-n.

IV. COMPLEXITY OF RTEF-ARM

The most time-consuming phases of RTEF-ARM are the edge-following and the edge-analyzing phases. In the edge-following phase, the edges of at most four obstacles are followed in just one pass. Each edge point is checked with the hit points of three other rays in order to detect any other hit to the same obstacle. This decreases the efficiency by a constant factor. The point is then inserted into a list in constant time. Thus, the worst case complexity of this phase is proportional to the number of edges of the largest obstacle in the environment. In the edge-analyzing phase, the edges on the islands and hit-point islands are analyzed in several passes. Since the number of edges of islands and hit-point islands can be at most one plus the number of edges of the largest obstacle in the environment, the worst case complexity of this phase is also proportional to the number of edges of the largest obstacle in the environment. Thus, we can formulate the RTEF-ARM complexity in a formal way if we can describe the number of edges of the largest obstacle in the environment. Having the assumption that the environment is a rectangle grid world, the largest obstacle could have at most $(w-1) \times (h-1)$ edges (exemplified in Fig. 11), where w and h are the width and height of the rectangle, respectively. Therefore, the worst case complexity of RTEF for each move becomes $O(w \times h)$.

The worst case setting is rarely possible in practice. We set up a test platform on an AMD 2500+ computer and conducted a number of experiments on three types of grids: random-, maze-, and U-type to measure the average number of moves per unit

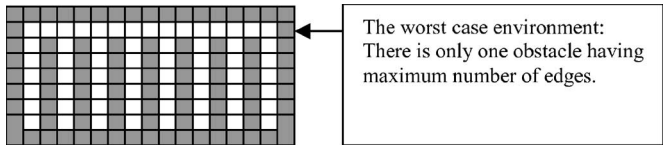

Fig. 11. Worst case environment for a 17 × 9 sized grid: There is only one obstacle, and it has the maximum number of edges that is possible.

TABLE II
AVERAGE NUMBER OF MOVES PER SECOND FOR DIFFERENT SIZED GRIDS

	100x100	200x200	400x400	800x800
Random Grids	695.24	379.67	191.42	78.87
U-Type Grids	249.56	126.56	73.06	32.37
Maze Grids	93.04	17.24	4.97	1.52

TABLE III
PERFORMANCE DECREASE RATIO WITH RESPECT TO THE PREVIOUS SIZE FOR DIFFERENT SIZED GRIDS

	100x100	200x200	400x400	800x800	Average ratio
Random grids	-	1.83	1.98	2.43	2.08
U-Type grids	-	1.97	1.73	2.26	1.98
Maze grids	-	5.40	3.47	3.27	4.04

time and the performance decrease ratios for different sizes of the grids. The results are given in Tables II and III.

In practice, the performance of the RTEF is quite high in random- and U-type grids. Increasing the size does not drop the performance very sharply because the average obstacle size is not strictly dependent on the grid size. But, the performance in mazes is low, since they are similar to the worst case grids; and increasing the grid size by two drops the moves per second by four on the average. The advantage of the algorithm in mazes is that the agent reaches the target without visiting any previously visited cells because the maze is known in advance, resulting in almost optimal path lengths.

A. Depth-Limited RTEF-ARM: Setting Constant Time Complexity

As can be seen from Table II, increasing the grid size decreases the efficiency. To solve this problem, a search depth limit is introduced and the cells beyond the depth limit are treated as free cells. The time complexity becomes $O(d^2)$, because the size of the search rectangle will be 2 d in both dimensions, where d is the search depth limit.

RTEF-Visited-Count, RTEF-Visited-Count-History, RTEF-RTA*, and RTEF-RTA*-Penalty all work fine with this extension. However, RTEF-History and RTEF-History-BC that use only *History* to prevent loops can go into an infinite loop. Without a depth limit, the history is only cleared when an unknown obstacle is detected. But, when we include a depth limit, the history may need to be cleared not just because of an unknown obstacle but also a known obstacle that does not fit into the search rectangle. Additionally, RTEF-Visited-Count-History-BC with depth limit may not be able to clear all the history cells required for backtracking. The search depth may prevent extracting the entire border of a blocking obstacle, thus the border-clear technique may not be able to detect that it is stuck in the area due to that obstacle. On the other hand, RTEF-Visited-Count-History with depth limit works fine because the agent inserts every visited cell into the history. As a result, the agent will be able to detect that he is blocked at a particular time and, hence, clear all the history to open all the blocking cells.

V. PROOF OF CORRECTNESS OF RTEF-ARM

It will be enough to prove that RTEF-ARM only closes the directions not leading to the target. RTEF-ARM has four phases: *initialization, ray-sending, edge-following*, and *edge-analyzing*. In the *initialization phase*, all the directions are set to open. In the *ray-sending phase*, four rays are propagated in four diagonal directions until hitting an obstacle or a maximum ray distance is reached. If a ray does not hit an obstacle, it does not cause a direction to be closed. Note that whether the ray hits something or not, the agent can move along one of the two sides of the ray until reaching the end point. In the *edge-following phase*, the algorithm is able to determine *islands*, and *hit-point islands*. Following the edges of an obstacle (partially or fully known) starting from the hit-point of a ray in only one direction, we always reach the same point we started at, or hit-point of another ray.

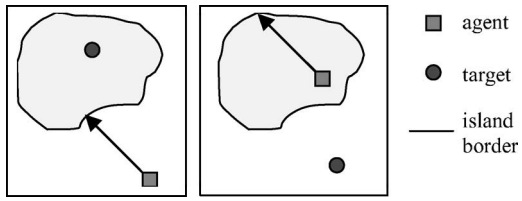

Fig. 12. Case1—unreachable targets. Outward-facing island with a target inside (*left*), inward-facing island with a target outside (*right*).

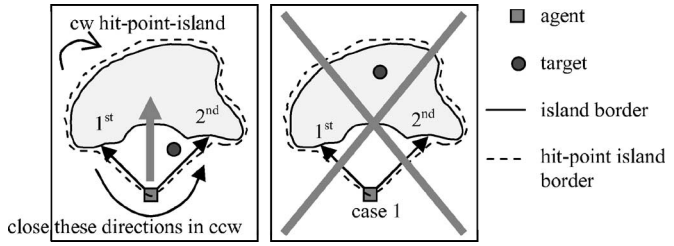

Fig. 13. Illustration of Case 2 (left), an illustration seems to be Case 2 but already covered by Case 1 (right).

In the *edge-analyzing phase*, the algorithm could find all possible closed directions and never closes a direction that must be open. Five cases (Case 1–5) to mark a direction as closed are shown (as per Algorithm 3). For the remaining cases, no conclusion can be made, which we name as Case 0.

Case 1: If the ray hits the outer border of an obstacle, the agent must be outside the obstacle and the island must be outward-facing. If the agent is outside the obstacle and the target is inside the obstacle, then it is clear that the target cannot be reached from any direction. If the ray hits the inner border of an obstacle, the agent must be inside the obstacle. If the agent is inside the obstacle and the target is outside the obstacle, then the target cannot be reached from any direction. These two cases are illustrated in Fig. 12.

Case 2: Case 2 is possible if Case 1 is not satisfied and there exists a hit-point island and the target is inside the hit-point island which is clockwise-oriented (cw). We figure out that the agent must go into the region bordered by the hit-point island and close directions outside the hit-point island as illustrated in Fig. 13. Note that the example on the right side of Fig. 13 is not an instance of Case 2 as it is already covered by Case 1, and it is clear that the target cannot be reached from any of the directions that are not inside the hit-point island.

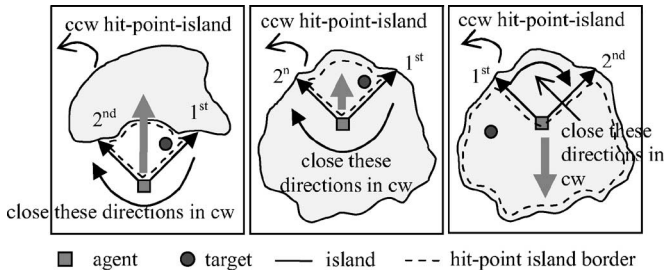

Fig. 14. Illustrations of Case 3.

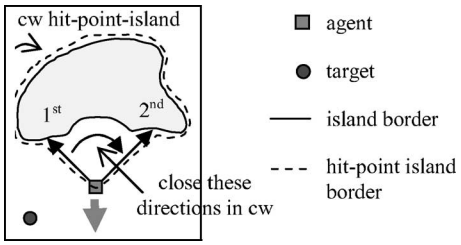

Fig. 15. Illustration of Case 4.

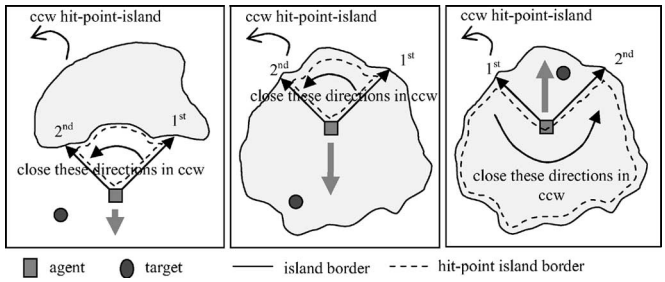

Fig. 16. Illustration of Case 5.

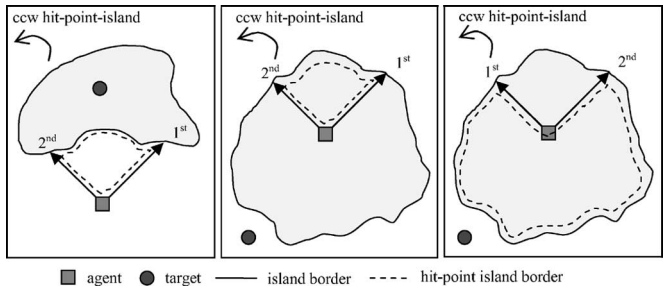

Fig. 17. Illustrations seem to be Case 5 but covered by Case 1.

TABLE IV
CASES WHERE NO HIT-POINT ISLAND IS FOUND

Agent and Island	Target	Mapping
agent is outside the island, which means having an outwards-facing island	outside island	Case 0
	inside island	Case 1
agent is inside island, which means having an inwards-facing island	outside island	Case 1
	inside island	Case 0

TABLE V
CASES WHERE THERE IS A HIT-POINT ISLAND

Agent & Island	Hit-Point Island	Target	Target	Mapping
agent is outside the island, which also means having an outwards-facing island	clockwise oriented	outside hit-point-island	outside island	Case 4
			inside island	impossible
		inside hit-point-island	outside island	Case 2
			inside island	Case 1
	counter clockwise oriented	outside hit-point-island	outside island	Case 5
			inside island	Case 1
		inside hit-point-island	outside island	Case 3
			inside island	impossible
agent is inside the island, which also means having an inwards-facing island	clockwise oriented	outside hit-point-island	outside island	impossible
			inside island	impossible
		inside hit-point-island	outside island	impossible
			inside island	impossible
	counter clockwise oriented	outside hit-point-island	outside island	Case 1
			inside island	Case 5
		inside hit-point-island	outside island	impossible
			inside island	Case 3

Case 3: Case 3 occurs when the target is inside the hit-point island which is counterclockwise-oriented (ccw). The agent must go into the region bordered by the hit-point island (see examples in Fig. 14). In this case, we must close the directions between the first and second hit points (encountered during the edge-following) in clockwise direction. From the figure, it is clear that the target cannot be reached from any of the directions that are not inside the hit-point island.

Case 4: The algorithm considers this case only if Case 1 is not satisfied and there is a hit-point island found. Case 4 is possible if the target is outside the hit-point island which is cw. We realize that the agent must leave the region bordered by the hit-point island; this is exemplified in Fig. 15. In this case, we must close the directions between the first and second hit points in clockwise direction. It is clear that the target cannot be reached from any of the directions that are inside the hit-point island.

Case 5: Case 5 occurs when the target is outside the hit-point island, which is ccw. The agent needs to leave the region bordered by the hit-point island as illustrated in Fig. 16. In this case, we must close the directions between the fist and second hit points in counterclockwise direction. From the figure, it is clear that the target cannot be reached from any of the directions that are inside the hit-point island. There are three other cases that look like Case 5, but, in fact, covered by Case 1. These cases are illustrated in Fig. 17.

We have demonstrated that all directions that are closed by the algorithm are feasible. Furthermore, we must also show that the algorithm could find all the possible closed directions. We enumerated all possible situations and mapped them to the cases given in Tables IV and V. Table IV contains four situations where no hit-point island is found. As a result, we can either conclude that the target is unreachable (Case 1) or say nothing (Case 0). Table V contains the situations where a hit-point island is found. Note that some situations in the table have no real-world interpretations (named "impossible" in the table) and thus they are not considered by the algorithm. Thus, the proof is complete.

VI. PERFORMANCE ANALYSIS

In this section, we report the performance results of the RTEF algorithms listed in Table VI. We used RTA* as the basis and evaluated the performance increase of various RTEF algorithms on three different types of grids: *random-*, *maze-*, and *U-type* described as follows. We randomly generated 16 grids of size

TABLE VI
RTEF ALGORITHMS AND THEIR ABBREVIATIONS

RTEF-History	RT-H
RTEF-History-BC	RT-HBC
RTEF-Visited Count	RT-VC
RTEF-Visited Count-History	RT-VCH
RTEF-Visited Count-History-BC	RT-VCHBC
RTEF-RTA*	RT-RTA*
RTEF-RTA*-Penalty-0	RT-RTA*p0
RTEF-RTA*-Penalty-1	RT-RTA*p1
RTEF-RTA*-Penalty-2	RT-RTA*p2
RTEF-RTA*-Penalty-3	RT-RTA*p3
RTEF-RTA*-Penalty-4	RT-RTA*p4

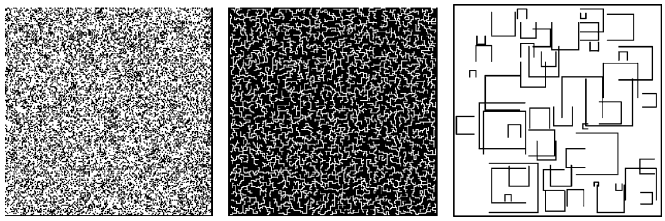

Fig. 18. Sample random-, maze-, and U-type grids.

200×200 and tested the algorithms on a Centrino 1.5-GHz laptop.

Random grids are generated randomly based on a specified obstacle ratio (the percentage of the obstacle cells). We used three random grids generated with obstacle ratios 0.3, 0.35, and 0.4. *Maze grids* are the ones where every two nonobstacle cells are always connected through a path (usually one path). Two parameters, *obstacle ratio* and *corridor size* (the minimum corridor width in the maze), are used to produce mazes. The corridor size effect is obtained by scaling small mazes (e.g., a 200×200 maze can be obtained by scaling a 50×50 maze). Nine different mazes are generated using obstacle ratios 0.3, 0.5, and 0.7 and corridor sizes 1, 2, and 4 and used in the experiments. *U-type grids* are created by randomly putting U-shaped obstacles of random sizes on an empty grid. Taking into consideration the number of U-type obstacles, minimum and maximum width and height of U-shaped obstacles, we used four different U-type grids where the number of U-type obstacles is 30, 50, 70, 90 and minimum/maximum U sizes are 5/50. Three of the input grids used in our experiments, one for each type, are given in Fig. 18.

Initially, agent and target locations are randomly generated for each grid type such that the distance between them is at least half of the maze size. These locations are kept the same in all the experiments with different configurations.

We assume that an agent perceives the environment up to a limit, which is called visual depth (v). Being at its center, the agent can only sense cells within the rectangular area of size $2v \times 2v$. We used the statement *full vision* to emphasize that the agent has infinite visual depth and knows the entire grid world before the search starts.

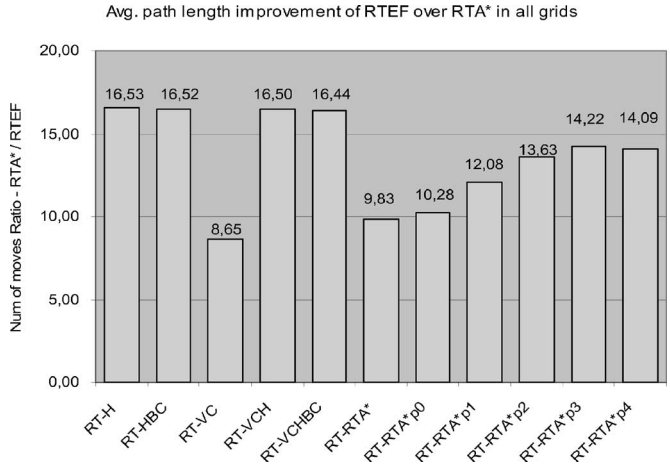

Fig. 19. Average path length decrease of RTEF over RTA* in all the grids.

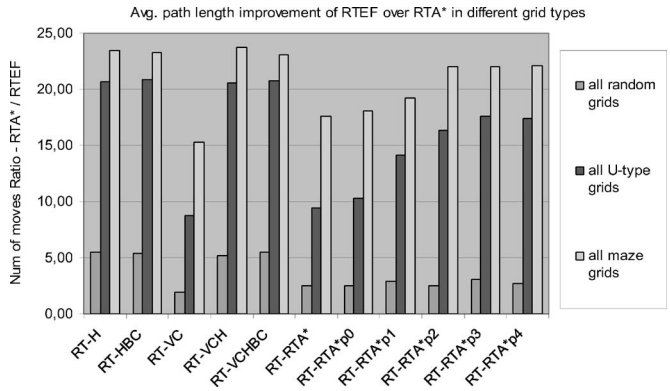

Fig. 20. Average path length decrease of RTEF over RTA* in maze-, random-, and U-type grids.

A. First Experiment: Comparison of RTA* and RTEF Algorithms

We tested 12 different algorithms (11 RTEFs + 1 RTA*) on 16 grids with four different visual depths (10, 20, 40, and full). Thus, 768 test configurations were generated, and ten runs were performed for each configuration, making 7680 runs in total. The results demonstrate that RTEF-ARM finds much shorter paths (a significant improvement in the solution quality) in almost all the tested configurations. In terms of total execution time, RTEF-ARM seems to be better than the original RTA* in most of the tested configurations, although execution time per movement is quite high.

1) Path Length Analysis: The ratio of the number of moves of RTA* to that of any RTEF algorithm is used as a metric to compare the lengths of the solution paths. Larger the ratio, better is the solution quality of the RTEF algorithm. Figs. 19–Fig. 22 show the average solution quality of the RTEF algorithms compared to RTA* considering all grids, grids of different types (random-, maze-, and U-type), and different visual depths (10, 20, 40, and full) and grids with different corridor sizes (1, 2, and 4), respectively.

All the RTEF algorithms perform better than RTA* in all the grids. This is expected since RTEF-ARM is an improvement

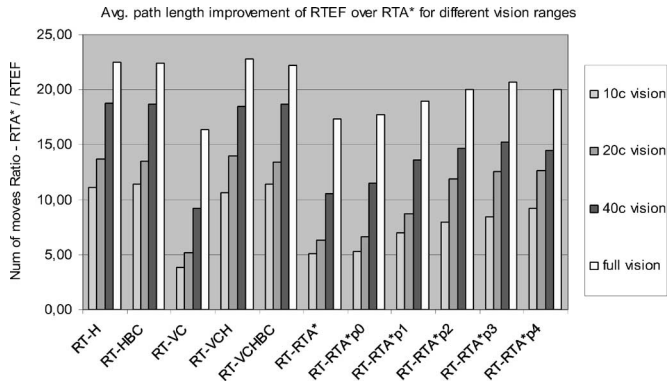

Fig. 21. Average path length decrease of RTEF over RTA* with visual depths 10, 20, 40, and full. Note that c stands for cell.

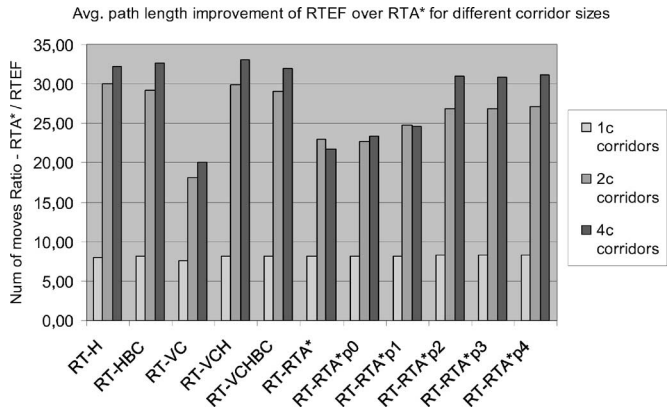

Fig. 22. Average path length decrease of RTEF over RTA* with corridor sizes 1, 2, and 4 cells.

over RTA* without any drawbacks. Furthermore, the most beneficial improvement is obtained in mazes, next in U-type, and then in random grids. This is due to the fact that the difference between the suboptimal solutions found by RTA* and the optimal one is maximal in mazes. Increasing the *visual depth* yields better solutions (shorter paths) for RTEF-ARM over RTA* because RTEF-ARM is able to make use of environmental information. RTEF-ARM performs better than RTA* when the *corridor size* increases since wider corridors increase the average branching factor and the area of heuristic depression to be filled up. When the branching factor is high, RTA* has lots of alternatives to pursue while the RTEF-ARM is able to classify the alternatives intelligently. The RTEF algorithms integrated with history are the best because history cells are merged with real obstacles yielding the larger obstacles that cause the agent to be more explorative. For RTEF-RTA*-Penalty, penalty value 3 seems to be the best. RTEF-Visited-Count, RTEF-RTA*, and RTEF-RTA*-Penalty (penalty<2) perform the worst. The best improvement increase (81.63) was encountered in mazes with obstacle ratio 0.5 and corridor size 2, due to the difficulty level of maze and the high branching factor. The worst improvement (1.08) was encountered in maze grids with obstacle ratio 0.3 and corridor size 1. The second worst was 1.26 in random grids with obstacle ratio 0.3.

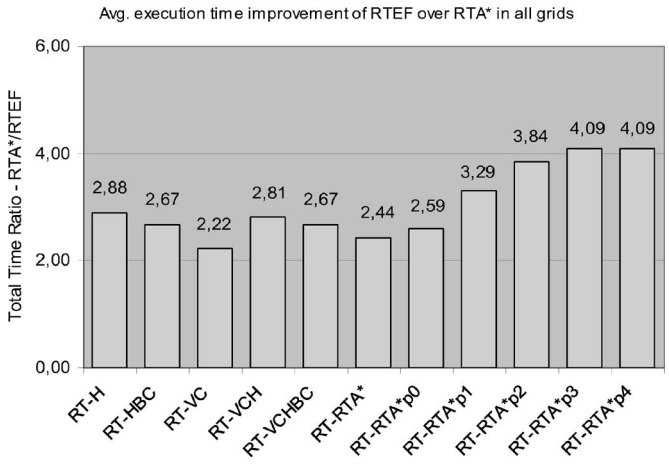

Fig. 23. Average total execution time decrease of RTEF over RTA* in all the grids.

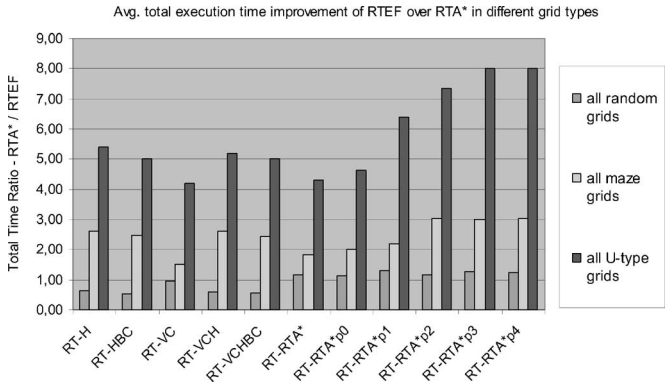

Fig. 24. Average total execution time decrease of RTEF over RTA* in random-, maze-, and U-type grids. Note that the ones below 1 are not an improvement for RTEF.

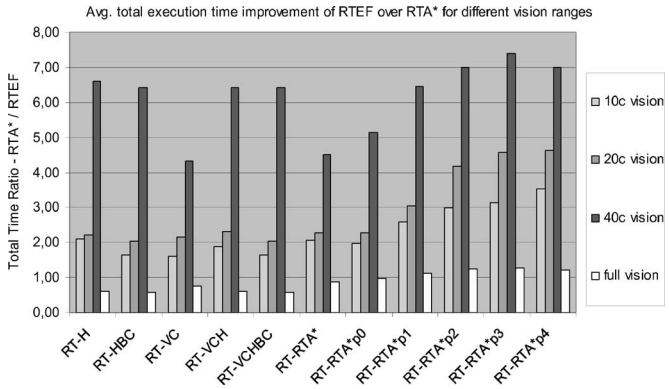

Fig. 25. Average total execution time decrease of RTEF over RTA* with 10, 20, 40 cell and full vision ranges. Note that the ones below 1 are not an improvement for RTEF.

2) Execution Time Analysis: In this section, we will compare the total execution times of RTEF and RTA* for the experiments reported in the previous section. Similarly, the ratio of execution time of RTA* to that of any RTEF algorithm is reported. Figs. 23–26 show the speed-up by considering all cases, grids of different types (random-, maze-, and U-type), different visual

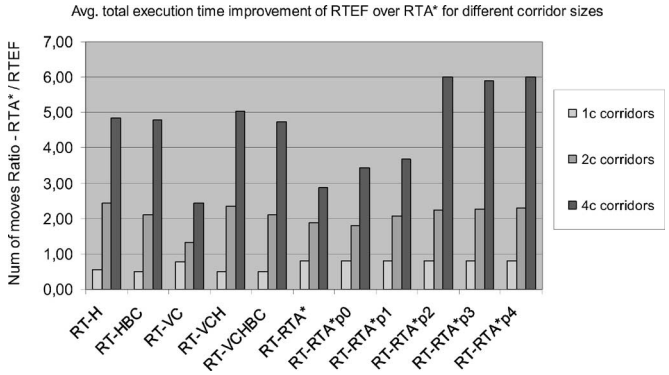

Fig. 26. Average total execution time decrease of RTEF over RTA* with 1, 2, and 4 cell corridor sizes. Note that the ones below 1 are not an improvement for RTEF.

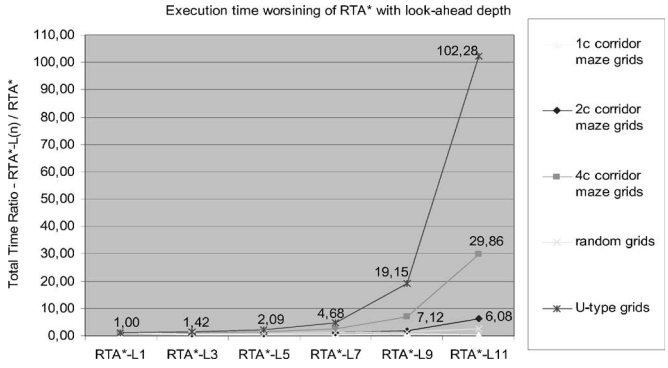

Fig. 27. Total execution time increase of RTA* with look-ahead depth 3, 5, 7, 9, and 11 over RTA*. Note that lower values are better.

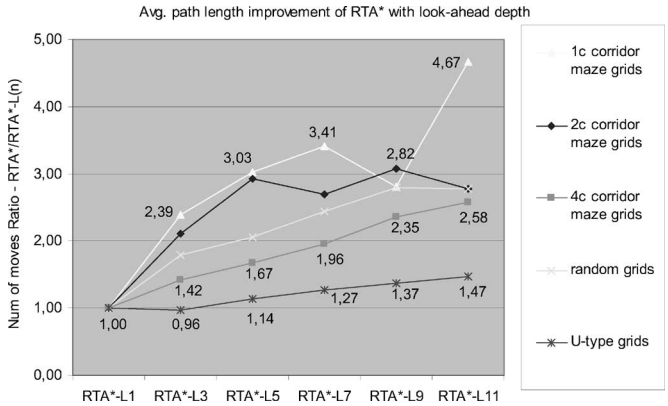

Fig. 28. Average path length decrease of RTA* with look-ahead depth 3, 5, 7, 9, and 11 over RTA*.

depths (10, 20, 40, and full) and grids with different corridor sizes (1, 2, and 4), respectively.

The RTEF algorithms perform better than RTA* in U-type grids. This is due to the decrease in both path length and the cost of edge-following (U-type grids have less obstacle edges). Increasing the visual depth is beneficial up to a point; and having the complete grid information (full vision) does not bring more efficiency because knowing the entire obstacle borders (which may not be useful all the time) makes the edge-following phase costly. Such a case can be seen in the experiments reported in Fig. 25. When the corridor size gets larger, RTEF performs better since the average branching factor increases and the edge-following phase becomes less costly. On the average, all the RTEF algorithms perform better than RTA* in maze- and U-type grids, which are the most difficult ones. Since random grids are the easiest grids for RTA*, RTEF algorithms generally perform worse than RTA*. The RTEF-RTA*-Penalty-3 is efficient and returns shorter solution paths. Penalties greater than 3 did not bring any performance improvement, but even a reduction in some cases. Although history computations are costly, RTEF algorithms with history take less execution time due to their ability to return the shortest solution paths. The RTEF-Visited-Count, RTEF-RTA*, and RTEF-RTA*-Penalty-0 are the most inefficient algorithms.

We have also performed a number of experiments with different types of grids and visual depths which are not included here. We observed that the best average speed-up (15.26) was encountered in U-type grids with visual depth 40 and the worst speedup (0.11) was with random grids with full vision (random grids were easy for RTA* and knowing the entire obstacle borders due to full vision increases the edge-following cost).

B. Second Experiment: The Effect of Look-Ahead Depth

In the second experiment, five different look-ahead depths (3, 5, 7, 9, and 11), and four different visual ranges (10, 20, 40, and full), are used with RTA* in 16 grids. Thus, 320 test configurations were generated, and ten runs were performed for each configuration, making 3200 runs in total.

The results show that the path length improvement of RTA* with reasonable look-ahead depths is insignificant compared to the RTEF algorithms. The time per move and total execution time of RTA* with large look-ahead depths are too high because the time complexity is exponential in the size of the look-ahead depth. Fig. 27 shows the execution time ratios of RTA* with various look-ahead depths (1–11) to RTA* without look-ahead depth. Sudden increase in the execution time highly depends on the grid type, which affects the average branching factor and the area of heuristic depression needed to be filled up. The sharpness of the increment is too high in grids with wide corridors, and low in grids with narrow corridors.

1) Path Length Analysis: The average path length decrease of RTA* with look-ahead depth of 3, 5, 7, 9, and 11 with respect to RTA* is shown in Fig. 28, and the comparison of RTA* with various look-ahead depths, RTEF-Visited-Count-History, and RTEF-RTA*-Penalty-3 can be found in Fig. 29.

As can be seen easily from Fig. 28, increasing the look-ahead depth does not always improve the solution, although we expect shorter paths. This is easily observed from the results of maze grids with 1-cell- and 2-cell corridors. Since the results seemed to be strange at first, we examined the test runs in detail, and found out the problem, which was also mentioned in [3]. The reason was to choose the wrong alternative at a very critical decision point because of stopping the search at an immature depth guiding a local optimal. This is exemplified using one of our problematic runs shown in Fig. 30. In the example, although RTA* with look-ahead depth 7 could easily reach the target,

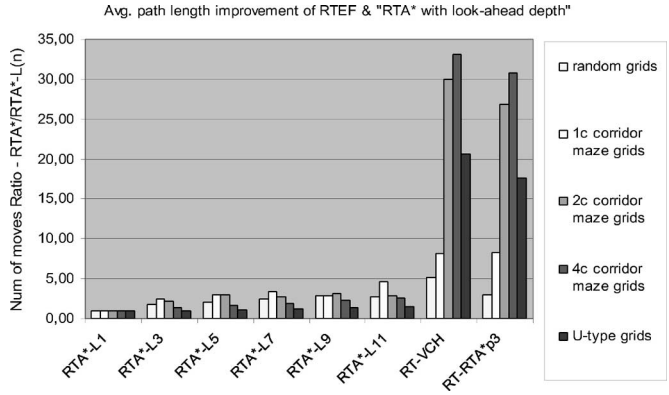

Fig. 29. Average path length decrease of RTEF-Visited-Count-History, RTEF-RTA*-Penalty-3, and RTA* with look-ahead depth 3, 5, 7, 9, and 11 over RTA*.

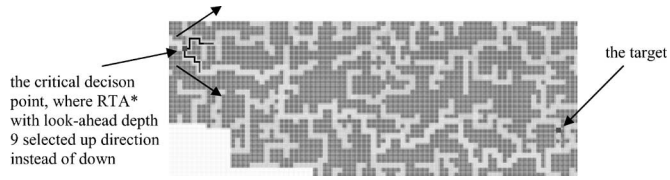

Fig. 30. Critical decision point of RTA* with look-ahead depth 9.

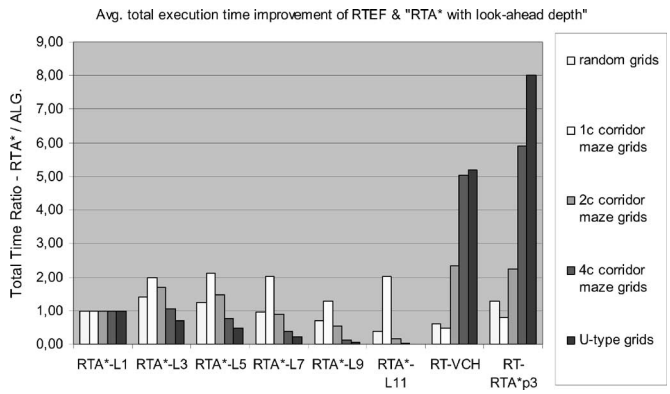

Fig. 31. Average total execution time decrease of RTEF-Visited-Count-History, RTEF-RTA*-Penalty-3, and RTA* with look-ahead depth 3, 5, 7, 9, and 11 over RTA*. Note that ratios below 1 mean becoming worse.

RTA* with look-ahead depth 9 selected a wrong direction at a very early stage, and had to travel almost the entire maze. Thus instead of 576 moves, it took 11 704 moves to reach the target.

2) Execution-Time Analysis: The improvements in the execution time of RTA* with look-ahead depth, RTEF-Visited-Count-History, and RTEF-RTA*-Penalty-3 over RTA* can be seen in Fig. 31. The results show that using look-ahead depth is not sufficient for RTA* to overtake the total time efficiency of RTEF in maze grids having wide corridors. However, RTA* with look-ahead depth performed better than RTEF with small look-ahead depths in random grids and maze grids with 1-cell corridors.

C. Third Experiment: Effect of Search Depth

In the third experiment, four different search depths (10, 20, 40, and 80), and four different vision ranges (10, 20, 40, and full),

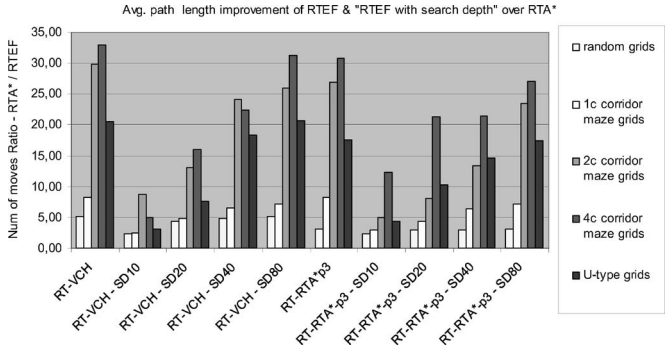

Fig. 32. Average path length decrease of RTEF-Visited-Count-History and RTEF-RTA*-Penalty-3 with 10, 20, 40, and 80 cell depth over RTA* without depth limit.

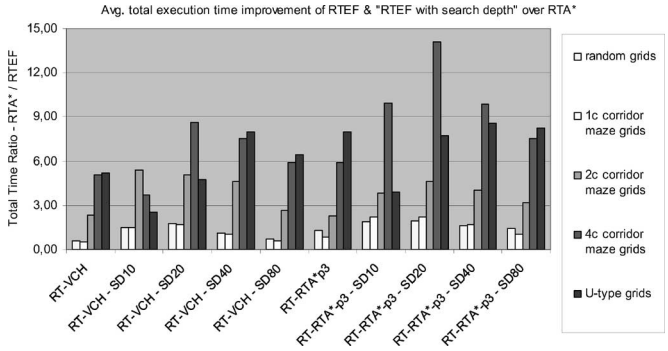

Fig. 33. Average execution time decrease of RTEF-Visited-Count-History and RTEF-RTA*-Penalty-3 with 10, 20, 40, and 80 cell depth over RTA* without depth limit.

were used with two RTEF algorithms: RTEF-Visited-Count-History and RTEF-RTA*-Penalty-3, in 16 grids. In all, 512 test configurations were generated, and ten runs were performed for each configuration, making 5120 runs in total. The average solution length and execution time ratios of RTEF algorithms with various search depths are shown in Figs. 32 and 33.

The results show that when a search depth is specified, the path lengths get longer, but the time spent per move decreases significantly. Thus, if the path lengths do not increase sharply (which do not if a very small depth is not used), the total execution time usually decreases. As a conclusion, we can say that if the time per move and total execution time are critical, it is better to use a search depth. But, if path lengths form the most significant item, it is better to use a large search depth to have a bounded complexity, or not to use any.

D. Comparison With Optimal Solution Paths

Finally, we have conducted a number of experiments to compare the path lengths of 8 of 12 algorithms (7 RTEFs + 1 RTA*) with optimal path lengths on 16 grids. We implemented the off-line path planning algorithm A* [18] to compute the optimal path lengths. Note that 160 different agent-target locations are used, and we assume that the mazes are fully known (full vision case). Finally, we computed the ratio of path length of each of these eight algorithms to that of optimal solutions to clearly see the proximity of solutions to the optimal ones (Fig. 34).

		RT-H	RT-HBC	RT-VCH	RT-VCHBC	RT-RTA*	RT-RTA*p3	RT-RTA*p4	RTA*
Random 0.30	Avg	1,272	1,281	1,283	1,269	1,214	1,191	1,200	1,588
	Std	0,083	0,086	0,074	0,076	0,088	0,095	0,100	0,254
Random 0.35	Avg	1,643	1,628	1,628	1,638	1,589	1,499	1,575	3,676
	Std	0,579	0,586	0,586	0,582	0,554	0,389	0,483	1,922
Random 0.40	Avg	1,384	1,384	1,383	1,383	6,185	4,024	4,231	26,332
	Std	0,259	0,260	0,258	0,258	5,121	3,353	2,361	22,261
Maze 1c x 0.3	Avg	2,924	3,860	2,948	2,974	1,525	1,461	1,455	2,194
	Std	3,266	4,005	3,263	3,236	0,258	0,276	0,248	0,684
Maze 1c x 0.5	Avg	1,005	1,005	1,005	1,005	1,005	1,005	1,005	21,418
	Std	0,004	0,004	0,004	0,004	0,004	0,004	0,004	9,491
Maze 1c x 0.7	Avg	1,000	1,000	1,000	1,000	1,000	1,000	1,000	20,640
	Std	0,000	0,000	0,000	0,000	0,000	0,000	0,000	18,747
Maze 2c x 0.3	Avg	1,568	1,507	1,550	1,567	1,600	1,200	1,212	2,121
	Std	0,318	0,305	0,327	0,317	0,466	0,125	0,131	0,571
Maze 2c x 0.5	Avg	1,001	1,001	1,001	1,001	1,000	1,000	1,000	124,921
	Std	0,002	0,002	0,002	0,002	0,000	0,000	0,000	47,574
Maze 2c x 0.7	Avg	1,000	1,000	1,000	1,000	1,000	1,000	1,000	11,961
	Std	0,000	0,000	0,000	0,000	0,000	0,000	0,000	8,534
Maze 4c x 0.3	Avg	1,560	1,555	1,588	1,561	7,626	3,894	1,942	18,126
	Std	0,485	0,474	0,457	0,484	5,023	2,752	1,010	12,570
Maze 4c x 0.5	Avg	1,017	1,017	1,017	1,017	1,000	1,000	1,000	91,888
	Std	0,044	0,044	0,044	0,044	0,000	0,000	0,000	87,633
Maze 4c x 0.7	Avg	1,005	1,005	1,005	1,005	1,000	1,000	1,000	46,929
	Std	0,014	0,014	0,014	0,014	0,000	0,000	0,000	55,477
U-type 30	Avg	1,600	1,540	1,621	1,600	3,890	1,708	1,751	19,286
	Std	0,618	0,618	0,649	0,618	2,816	0,884	0,931	35,029
U-type 50	Avg	1,893	2,008	1,800	1,925	3,356	1,493	1,744	59,066
	Std	0,894	1,153	0,915	0,899	2,829	0,462	1,213	76,739
U-type 70	Avg	1,247	1,292	1,285	1,452	13,900	1,701	1,735	19,888
	Std	0,303	0,346	0,340	0,570	20,480	0,970	1,256	18,865
U-type 90	Avg	3,828	3,964	2,907	3,593	23,072	4,212	4,663	58,325
	Std	2,053	2,523	0,991	1,403	17,358	2,795	2,720	51,438
		RT-H	RT-HBC	RT-VCH	RT-VCHBC	RT-RTA*	RT-RTA*p3	RT-RTA*p4	RTA*
Global Average		1,559	1,628	1,501	1,562	4,373	1,774	1,719	33,022
Global Stdev		1,239	1,499	1,068	1,162	8,927	1,700	1,487	50,417

Fig. 34. Average ratio of RTEF algorithms and RTA* solution path lengths over optimal path lengths, and their standard deviations.

The solutions found by RTEF algorithms are very close to optimal ones on the average. On the other hand, RTA* generates solutions that are very far away from optimal solutions on the average. The best performance is obtained by RTEF-VCH. The solutions are only 1.501 times longer than the optimal ones on the average, and the standard deviation is 1.068. The worst performance is obtained by RTA*. The solutions are 33.022 times longer than the optimal ones on the average, and the standard deviation is 50.417, which is unacceptably high. When we closely look at the results from the viewpoint of different types of grids, we see that RTEF algorithms and RTA* show opposite behaviors most of the time. When the obstacle ratio increases and grids become complicated, RTEF algorithms converge to optimum (e.g., the results are exactly optimal in maze grids with obstacle ratio 0.7); on the contrary, RTA* is far away from the optimal solutions (e.g., the results are 124 times longer than the optimal ones in maze grids with two-cell corridors and obstacle ratio 0.5). As a result, we can conclude that RTEF algorithms are able to find out solutions very close to optimal solutions.

VII. Conclusion

We have shown that RTEF is able to make use of environmental information acquired during the search successfully, and brings significant performance improvement over RTA* with respect to path lengths in all types of grids. Especially, the improvement is the highest in grids with wide corridors (e.g., U-type grids and maze grids with corridor size greater than 1) because their high branching factor and large heuristic depression area to be filled up make the grids difficult for RTA*. In addition, the path length improvement of RTA* with reasonable look-ahead depths is still insignificant compared to RTEF algorithms.

With respect to execution time, we have observed that the total execution time to reach the goal is usually better than RTA* in maze and U-type grids although the time per move of

RTEF is very high compared to RTA*. This improvement is due to much shorter paths found by RTEF. But, the total execution time of RTEF is worse than RTA* in random grids because random grids are not very challenging for both RTEF and RTA*. RTA* with look-ahead depth offers very little execution time improvement with very small look-ahead depths (e.g., 2–5); but, the total execution time becomes unreasonably high with larger look-ahead depths. Finally, we have introduced a search depth to RTEF guaranteeing a constant complexity. The test results clearly demonstrate that search depth significantly decreases time per move, but this advantage is balanced with longer paths, and, thus, the total execution time generally becomes more or less the same on the average. Although the focus of this paper was not moving targets, some of the RTEF versions can very easily be adapted to moving targets. Currently, we are studying moving targets and multiagent versions of RTEF.

REFERENCES

[1] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *Proc. IEEE Int. Conf. Intell. Robots Syst.* Piscataway, NJ: IEEE, 2002, pp. 2383–2388.

[2] P. Cheng and S. M. LaValle, "Resolution complete rapidly-exploring random trees," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2002, pp. 267–272.

[3] B. Hamidzadeh and S. Shekhar, "DYNORAII: A real-time path planning algorithm," *Int. J. Artif. Intell. Tools*, vol. 2, no. 1, pp. 93–115, Mar. 1993.

[4] T. Ishida and R. E. Korf, "Moving target search: A real-time search for changing goals," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 17, no. 6, pp. 97–109, Jun. 1995.

[5] T. Ishida, "Real-time bidirectional search: Coordinated problem solving in uncertain situations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 18, no. 6, pp. 617–628, Jun. 1996.

[6] K. Knight, "Are many reactive agents better than a few deliberative ones?," in *Proc. 13th Int. Joint Conf. Artif. Intell.*, 1993, pp. 432–437.

[7] J. J. Kuffner, Jr. and J. C. Latombe, "Goal-directed navigation for animated characters using realtime path planning and control," in *Proc. CAPTECH 1998: Workshop Model. Motion Capture Techn. Virtual Environ.*, Geneva, Switzerland, Nov.1998, pp. 26–28.

[8] J. J. Kuffner, Jr. and J. C. Latombe, "Fast synthetic vision, memory, and learning models for virtual humans," in *Proc. Comput. Anim.*, Piscataway, NJ, May 1999, pp. 118–127.

[9] S. Koenig and M. Likhachev, "D* Lite," in *Proc. Natl. Conf. Artif. Intell.*, 2002, pp. 476–483.

[10] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *Proc. Int. Conf. Robot. Autom.*, 2002, pp. 968–975.

[11] S. Koenig, "A comparison of fast search methods for real-time situated agents," in *Proc. Auton. Agents Multi-Agent Syst.*, 2004, pp. 864–871.

[12] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy, "Incremental heuristic search in artificial intelligence," *Artif. Intell. Mag.*, vol. 25, no. 2, pp. 99–112, 2004.

[13] A. Konar, *Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of Human Brain*. Boca Raton, FL: CRC, 2000.

[14] R. E. Korf, "Real-time heuristic search," *Artif. Intell.*, vol. 42, no. 2–3, pp. 189–211, 1990.

[15] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," presented at the IEEE Int. Conf. Robot. Autom., May 1999, Detroit, MI.

[16] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, B. R. Donald, K. M. Lynch, and D. Rus, Eds. Wellesley, MA: A. K. Peters, 2001, pp. 293–308.

[17] Z. Michalewicz, *Genetic Algorithms + Data Structure = Evolution Programs*. New York: Springer-Verlag, 1986.

[18] S. Russell and P. Norving, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[19] M. Shimbo and T. Ishida, "Controlling the learning process of real-time heuristic search," *Artif. Intell.*, vol. 146, no. 1, pp. 1–41, May 2003.

[20] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, vol. 4, May 1994, pp. 3310–3317.

[21] A. Stentz, "The focussed D* algorithm for real-time replanning," in *Proc. Int. Joint Conf. Artif. Intell.*, Aug. 1995, pp. 1652–1659.

[22] K. Sugihara and J. K. Smith, "Genetic algorithms for adaptive planning of path and trajectory of a mobile robot in 2D terrains," Dept. Inf. Comput. Sci. Univ. Hawaii, Tech. Rep. ICS-TR-97-04, May 1997.

[23] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1996, pp. 349–351.

[24] C. Undeger, F. Polat, and Z. Ipekkan, "Real-time edge follow: A new paradigm to real-time path search," presented at the GAME-ON 2001, London, U.K.

Cagatay Undeger received the B.Sc. degree from Kocaeli University, Kocaeli, Turkey, in 1998 and the M.S. degree from the Ankara, Turkey, Middle East Technical University, in 2001, where he is currently working toward the Ph.D. degree.

His current research interests include the field of real-time search.

Faruk Polat received the B.Sc. degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1987 and the M.S. and Ph.D. degrees in computer engineering from Bilkent University, Ankara, in 1989 and 1993, respectively.

He is currently a Professor in the Department of Computer Engineering, Middle East Technical University. During 1992–1993, he was engaged in research as a Visiting NATO Science Scholar in the Department of Computer Science, University of Minnesota, MN. His current research interests include artificial intelligence, multiagent systems, and object-oriented data models.