Appl Intell (2007) 27:113–129
DOI 10.1007/s10489-006-0023-1

RTTES: Real-time search in dynamic environments

Cagatay Undeger · Faruk Polat

Published online: 13 November 2006

Abstract In this paper we propose a real-time search algorithm called Real-Time Target Evaluation Search (RTTES) for the problem of searching a route in grid worlds from a starting point to a static or dynamic target point in real-time. The algorithm makes use of a new effective heuristic method which utilizes environmental information to successfully find solution paths to the target in dynamic and partially observable environments. The method requires analysis of nearby obstacles to determine closed directions and estimate the goal relevance of open directions in order to identify the most beneficial move. We compared RTTES with other competing real-time search algorithms and observed a significant improvement on solution quality.

Keywords Real-time search · Path planning

1 Introduction

Path planning can be described as finding a path from an initial point to a target point if there exists one. Path planning algorithms are either off-line or on-line. Off-line algorithms like A* [1, 2] find the whole solution in advance before starting execution, and suffer from execution time in dynamic or partially observable environments due to frequent re-planning requirements. In on-line case, an agent repeatedly plans its next move in limited time and executes it. There are several real-time algorithms such as Real-Time A* (RTA*), Learning Real-Time A* (LRTA*) [3, 4], Moving Target Search [5], Bi-directional Real-Time Search [6], Real-Time Horizontal A* [7]. They are not designed to be optimal, and usually find poor solutions with respect to path length. Furthermore, there exist some hybrid solutions such as incremental heuristic search algorithms; D* [8,9], Focused D*[10], D*Lite [11–13], which are optimal and more efficient than off-line path planning algorithms. However, they are still slow for some real-time applications, and are not applicable to moving targets. A comparison of D*Lite and LRTA* can be found in [14].

Recently we have developed a real-time search algorithm called Real-Time Edge Follow (RTEF) [15, 16] that uses a powerful heuristic function, RTEF-Alternative Reduction Method (RTEF-ARM), to discard some non-promising alternative moving directions in real-time to guide the agent to a static or dynamic target. Although RTEF is able to determine the closed (non-promising) directions successfully, it is weak in selecting the right move from the remaining alternatives as it uses the poor Euclidian distance heuristic. Therefore, we focused on a new method for better selection and improved the performance of RTEF [17].

In this paper, we propose a real-time search algorithm (Real-Time Target Evaluation Search—RTTES) capable of estimating the distance to the target more accurately considering the intervening obstacles. The method sends rays away from the agent in four directions, and determines the obstacles that the rays hit. For each such obstacle, we extract its border and determine the best direction that avoids the obstacle if the target is blocked by the obstacle. Hence, we have a number of directions each avoiding an obstacle hit by a ray. Then by using these directions and a resolution mechanism that will be described later, a single moving direction is determined.

C. Undeger · F. Polat (✉)
Middle East Technical University, Ankara, Turkey
e-mail: polat@ceng.metu.edu.tr

C. Undeger
e-mail: cundeger@ceng.metu.edu.tr

We randomly generated a number of grids of different types (random, maze and U-type) and compared RTTES with RTA* and RTEF algorithms on these sample grids in terms of path length and execution time. We observed a significant improvement in the path length over RTA* and RTEF in all types of grids, and in the execution time over RTA* in most of the grid types. Furthermore, we also observed that the solution paths of RTTES nearly converged to optimal paths on the average.

The organization of the paper is as follows: The related work on path planning is given in Section 2. In Section 3, RTTES is described in detail, the complexity analysis of RTTES and its proof of correctness is given. Section 4 presents the performance analysis of the algorithm and finally, Section 5 is the conclusion.

2 Related work

Off-line path planning algorithms such as Dijkstra's algorithm [18] and A* [1, 2] are hard to use for large dynamic environments because of their time requirements. One solution is to make off-line algorithms to be incremental [19] to avoid re-planning from scratch. D* [8, 9], focused D* [10], and D* Lite [11–13] are some of the well-known optimal incremental heuristic search algorithms. They are efficient in most cases, but sometimes a small change in the environment may cause to re-plan almost a complete path from scratch. There are also some probabilistic off-line algorithms that use genetic algorithms [20–22], random trees [23–25] and probabilistic road-maps [26, 27]. Genetic algorithms encode candidate solution paths as chromosomes and make use of evolution meta-heuristics to find acceptable solutions. Random tree based algorithms search the target in obstacle-free space in randomly generated trees. Probabilistic road-map algorithms generate connected graphs (road-maps) in obstacle-free space randomly, and try to connect initial and target points to the road-map to search paths.

Due to the efficiency problems of off-line techniques, a number of on-line approaches such as Learning Real-Time A* (LRTA*), Real-Time A* (RTA*) [4], LRTA*(k) [28], weighted LRTA*, upper-bounded LRTA* [29], Real-Time Horizontal A* (RTHA*) [7], Bug [30], Tangent-Bug [31], Execution Extended Rapidly Exploring Random Trees [32], Probabilistic Road-maps with Kinodynamic Motion Planner [33], Navigation Among Movable Obstacles (NAMO) [34] and Real-Time Adaptive A* [35] are proposed. LRTA* generates and updates a table containing admissible heuristic estimates of the distance from any state to the fixed goal state to reach the target. LRTA* is shown to be convergent and optimal, but the algorithm is able to find poor solution in the first run. Being a variation of LRTA*, RTA* gives better performance in the first run, but is lack of learning optimal table values. RTA* repeats the steps given in Algorithm 1 until reaching the goal [4].

Algorithm 1. An Iteration of RTA* Algorithm

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'.
2: Move to a neighbor with the minimum $f(x')$ value. Ties are broken randomly.
3: Update the value of $h(x)$ to the second best $f(x')$ value.

In its original form, RTA* considers immediate successors to determine the move and update the current estimate which is poor to estimate the real cost. It can easily be extended to have any arbitrary look-ahead depth. Although this improvement is shown to reduce the number of moves to reach the goal significantly, it requires exponential time and is not practical for large look-ahead depths [16].

Recently, Shimbo and Ishida introduced two LRTA* variations known as weighted LRTA* and upper-bounded LRTA* [29] for controlling 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).

Since LRTA*, RTA* and their variations are all limited to fixed goals, Ishida and Korf proposed another algorithm called Moving Target Search (MTS) for moving targets [5]. Their algorithm maintains a table that consists of $h(x, y)$ estimating the distance between x and y, where x and y are the positions of the problem solver and the target, respectively. MTS is a poor algorithm in practice because when the target moves (i.e., y changes), the learning process has to start all over again that causes a performance bottleneck.

Tangent-Bug [31], a similar approach to our proposed algorithm, is based on the Bug algorithm [30], and uses vision information to reach the target. It constructs a local tangent graph (LTG), a limited visibility graph, in each step considering the obstacles in the visible set. The sensed obstacles are modeled as thin walls and assumed to be the only obstacles in the environment. The agent moves to the locally optimal direction on the current LTG until reaching the target or detecting a local minimum (when hit an obstacle boundary). If a local minimum is detected, the agent switches to the boundary following mode, and move along the boundary until the distance to the target starts decreasing. After leaving the boundary, the agent switches to the first mode again. Although this approach seems to be similar to ours in the sense that it moves to locally optimal directions to go around the nearby obstacles and follows the obstacle borders, it only considers the obstacles in active visible set, and follows the boundaries while walking. But our approach that will be described later on can also consider obstacles known but not currently visible, and border following process is just performed in the mind of the agent, not physically executed.

In [15, 16], a new on-line path search algorithm, Real-Time Edge Follow (RTEF), is proposed for grid-type environments. RTEF uses a new heuristic, Real-Time Edge Follow Alternative Reduction Method (RTEF-ARM), which effectively makes use of global environmental information. With this heuristic, the agent can detect closed directions (the directions that cannot reach the target) using the perceptual data and the tentative map he/she discovered, and determine his/her next move from the open directions. In the following section we give a compact description of RTEF as our method proposed in this paper is build upon RTEF.

2.1 Real-time edge follow

RTEF aims to search a path from an initial location to a static or dynamic target in real-time. The basic idea behind the algorithm is to eliminate the closed directions that cannot reach the target point. RTEF executes the steps shown in Algorithm 2 until reaching the target or determining that the target is unreachable. RTEF internally uses the heuristic method, RTEF-ARM, to find out open and closed directions and hence to eliminate non-beneficial movement alternatives. To avoid infinite loops and re-visiting the same locations redundantly, RTEF either uses *visit count* or *history*, or both.

The algorithm maintains the number of visits, *visit count*, to the grid cells. The agent moves to one of the neighbor cells in open directions with minimum visit count. If there exists more than one cell having minimum visit count, the one with the minimum Euclidian distance to the target is selected. If Euclidian distances are also the same, then one of them is selected randomly. The set of previously visited cells forms the *history* of the agent. History cells are treated as obstacles. If the agent discovers a new obstacle and realizes that the target became inaccessible due to history cells, the agent clears the history to be able to backtrack.

Algorithm 2. An Iteration of RTEF Algorithm

1: Call RTEF-ARM to determine the set of open directions
2: **if** Number of open directions > 0 **then**
3: Select the best direction from open directions with the smallest visit count using Euclidian distance.
4: Move to the selected direction.
5: Increment the visit count of previous cell by one.
6: Insert the previous cell into the history.
7: **else**
8: **if** The history is not empty **then**
9: Clear all the history
10: Jump to 1
11: **else**
12: Destination is unreachable, stop search with failure.
13: **endif**
14: **endif**

Every obstacle either fully or partially known has a boundary, which is actually a sequence of connected edges shaping the obstacle [16]. RTEF-ARM sends four rays away from the agent in diagonal directions. The region between two adjacent rays forms a possible moving direction for the agent. Hence, the agent has four moving directions (north, south, east and west). RTEF-ARM extracts the border of each obstacle hit by any ray and then analyzes the so-called regions to determine open and closed moving directions, as summarized in Algorithm 3.

Algorithm 3. RTEF-ARM Algorithm

1: Mark all moving directions as open.
2: Propagate four diagonal rays.
3: **for** each ray hitting an obstacle **do**
4: Trace the edges of the obstacle starting from the hit-point of the ray and moving to left, extract the border of the obstacle, and find out an island and an hit-point-island if exists.
5: Analyze the edges using the island, hit-point-island and the target, and detect closed directions.
6: If number of open directions is zero, stop with failure (target is unreachable).
7: **end for**

In RTEF-ARM, four diagonal rays splitting north, south, east and west directions are propagated away from the agent as shown in Fig. 1. The rays go away from the agent until hitting an obstacle or maximum ray distance is achieved.

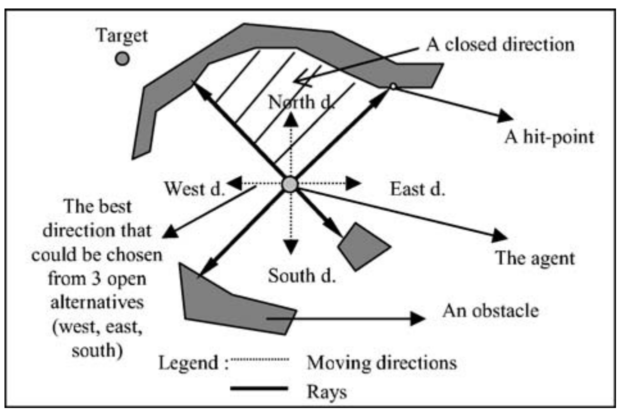

Fig. 1 Sending rays to split north, south, east and west directions

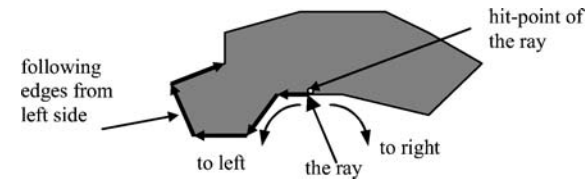

Fig. 2 Identifying the obstacle boundary

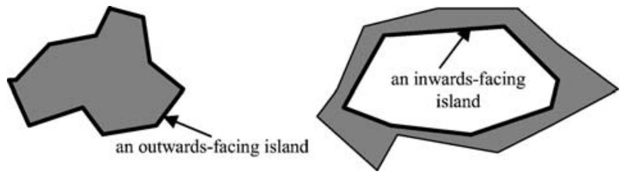

Fig. 3 Island types: outwards facing (left), inwards facing (right)

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 traced by going towards left side starting from a hit-point, we always return to the same point as illustrated in Fig. 2.

By following the edges of an obstacle hit by the ray 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. 3, there are two kinds of islands: *outwards-facing* and *inwards-facing islands*. The target is unreachable from agent location if it is inside an outwards-facing island or outside an inwards-facing island.

It is possible that more than one ray can hit the same obstacle. As illustrated in Fig. 4, an augmented 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 inside the hit-point-island) all the directions not bordered by the hit-point-island are closed; this is illustrated in Fig. 5.

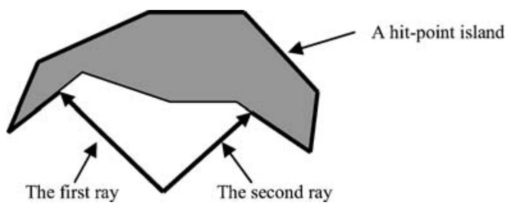

Fig. 4 Two rays hitting the same obstacle at two different points form a hit-point island

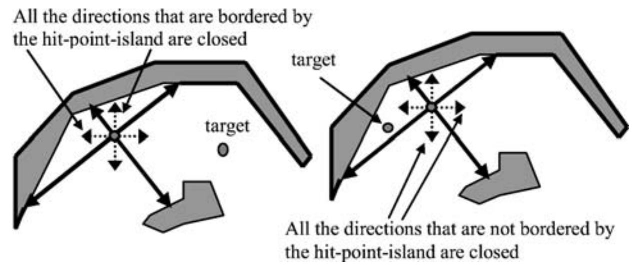

Fig. 5 Analyzing hit-point islands and eliminating moving directions

Islands and hit-point-islands are stored as vertex lists and passed to the closed direction determination step shown in Algorithm 4. Note that function `isInside(x,y,p)` returns *true* if coordinates (x, y) is inside polygon p and function `isClockwise (p)` returns *true* if the vertices of polygon p is ordered in clockwise direction (i.e., if polygon p is outwards facing with respect to the agent)

Algorithm 4. Determining Closed Directions

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)
else if $|h| > 0$ **then**
  if *isInside*(x, y, h) **then**
    if *isClockwise*(h) **then**
      Close the directions between 1^{st} and 2^{nd} hit-points on i in counter clockwise direction
    else
      Close the directions between 1^{st} and 2^{nd} hit-points on i in clockwise direction
    end if
  else
    if *isClockwise*(h) **then**
      Close the directions between 1^{st} and 2^{nd} hit-points on i in clockwise direction
    else
      Close the directions between 1^{st} and 2^{nd} hit-points on i in counter clockwise direction
    end if
  end if
end if

3 Real-time target evaluation search

Agents that use less informed heuristics such as Euclidian distance cannot precisely evaluate the cost differences of

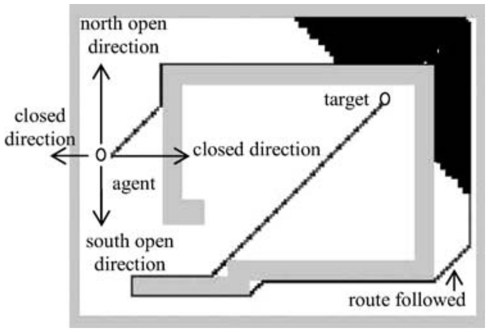

Fig. 6 RTEF can detect which directions are open, but cannot evaluate the cost differences successfully just using Euclidian distance heuristic

neighbor states and hence usually make wrong decisions in selecting their next moves towards the target. Although RTEF attempts to solve this problem to some extend by detecting closed directions correctly, it is also poor in estimating real cost because it uses Euclidian distance heuristic to select the moving direction from open ones. Figure 6 shows the route an agent follows guided by RTEF. Initially the agent has two open (north and south) directions. Due to the Euclidian distance heuristic, the agent prefers the north direction leading to a very long route to the target. If the agent had selected to move south, the route would have been much shorter. The problem of determining the right moving direction from open alternatives is the motivation behind RTTES algorithm.

RTTES makes use of a heuristic (RTTE) which analyzes obstacles and proposes a moving direction that avoids these obstacles and leads to the target through shorter paths. To do this, RTTE geometrically analyzes the obstacles nearby, tries to estimate the lengths of paths around obstacles to reach the target and proposes a moving direction. RTTE works in continuous space to identify the moving direction which is then mapped to one of the actual moving directions (north, south, east and west). The effectiveness of RTTES is illustrated on the previous example in Fig. 7. Here, RTTE identifies three

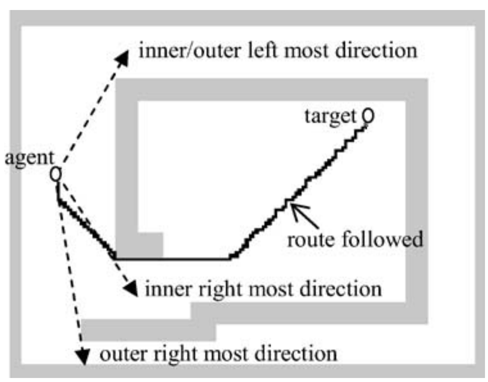

Fig. 7 RTTES chooses inner right most direction because it seems to be the shortest of all (not optimally computed)

possible moving directions, and evaluated that the middle one (which we name as the *inner right most direction*) is approximately the shortest.

We assume that the environment is a rectangular planar grid which is partially known by the agent. The agent located at a particular cell is required to reach a target cell avoiding obstacles in real-time. The agent is able to move north, south, east or west in each step. The agent has limited perception and maintains a tentative map containing obstacles in its memory as he/she explores the environment. Therefore, when we say an obstacle, we refer to the known part of that obstacle. RTTES repeats the steps in Algorithm 5 until reaching the target or detecting that the target is inaccessible. In the first step, RTTES calls RTTE heuristic function given in Algorithm 6 which returns a moving direction and the utilities of neighbor cells according to that proposed direction. Next, among the four possible cells RTTES selects the non-obstacle cell with highest utility and lowest visit count, if there exists one. If not, RTTES attempts to clear history similar to RTEF algorithm.

Algorithm 5. RTTES Algorithm

```
 1: Call RTTE to compute the proposed direction and the
    utilities of neighbor cells.
 2: if a direction is proposed by RTTE then
 3:     Select the neighbor cell with the highest utility
        from the set of non-obstacle neighbors with the
        smallest visit count.
 4:     Move to the selected direction.
 5:     Increment the visit count of previous cell by one.
 6:     Insert the previous cell into the history.
 7: else
 8:     if History is not empty then
 9:         Clear all the History.
10:         Jump to 1
11:     else
12:         Destination is unreachable, stop the search with
            failure.
13:     end if
14: end if
```

Although, Algorithm 5 can handle moving targets, we can still have some improvements for better performance. A moving target may sometimes come to points the agent previously walked through, and the history or visit count may prevent agent to aim the target through shortcuts. To solve this side effect, the agent should clear all the history and visit counts when he/she observes that the target moves into a cell which appears in the history or into a cell with none-zero visit count.

Algorithm 6. RTTE Algorithm

1: Mark all the moving directions as open.
2: Propagate four diagonal rays.
3: **for** each ray hitting an obstacle **do**
4: Trace and extract the border of the obstacle.
5: Analyze the border by re-tracing it from left and right sides.
6: Detect closed directions.
7: Evaluate results and determine a direction to avoid obstacle.
8: **end for**
9: Merge individual results, propose a direction to move, and compute utilities of neighbor cells.

RTTE propagates four diagonal rays away from the agent location, and analyzes the obstacles these rays hit to find out the best direction to move. If a ray hits an obstacle before exceeding the maximum ray distance, the obstacle border is extracted by tracing cells on the border starting from the hit-point. Concurrently, we also find the point on the border which is closest to the target. This point will be used in calculating the estimated path lengths. Next, the border is re-traced from both left and right sides to determine the additional geometric features that will be described in the next section. Then the closed directions are determined. The obstacle features are evaluated and a moving direction to avoid the obstacle is identified. After all the obstacles are evaluated, the results are merged in order to propose a final moving direction.

In RTTE, ray sending, border extraction and closed direction detection steps are the same as RTEF-ARM. Additionally, RTTE performs three more steps shown in lines 5, 7 and 9 for extracting additional geometric features and estimating the moving direction that minimizes the path to the target. Details of these steps are given in the following sub-sections.

3.1 Analyzing an obstacle border

When a ray hits an obstacle, its border is extracted and analyzed. Border analysis is done by tracing the border of an obstacle from left and right. In left analysis, the known border of the obstacle is traced edge by edge towards the left starting from the hit point, making a complete tour around the obstacle border. During the process, several geometric features of the obstacle are extracted. These features are described below (See Fig. 8 for illustrations):

Definition 1. Obstacle features

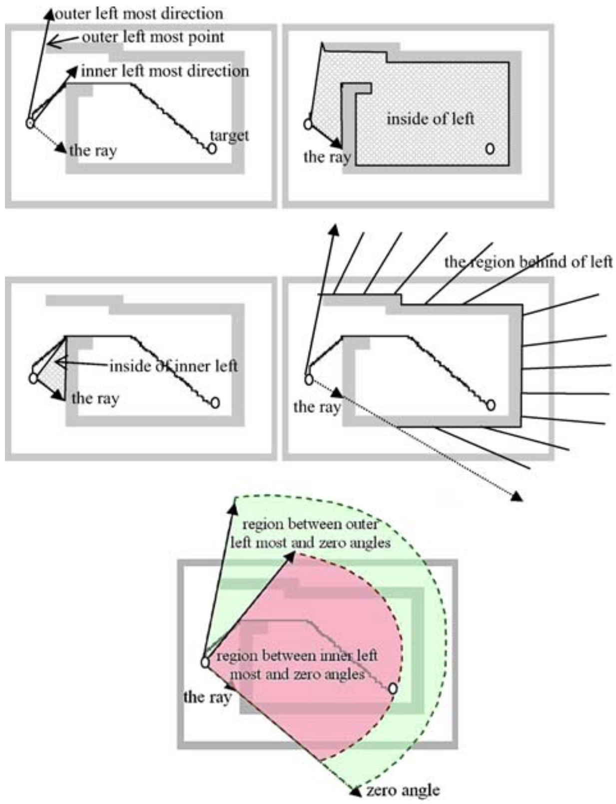

Fig. 8 Outer left most and inner left most directions (left-top), Inside of left (right-top), Inside of inner left (left-middle), Behind of left (right-middle), Outer-left-zero angle blocking and Inner-left-zero angle blocking (bottom)

- *Outer left most direction:* Relative to the ray direction, the largest cumulative angle is found during the left tour on the border vertices. In each step of the trace, we move from one edge vertex to another on the border. The angle between *the two lines* (TWLNS) starting from the agent location and passing through these two following vertices is added to the cumulative angle computed so far. Note that the added amount can be positive or negative depending on whether we move in counter-clockwise (*ccw*) or clockwise (*cw*) order, respectively. This trace (including the trace for the other geometric features) continues until the sum of the largest cumulative angle and the absolute value of smallest cumulative angle is greater than or equal to 360. The largest cumulative angle before the last step of trace is used as the *outer left most direction*.
- *Inner left most direction:* The direction with the largest cumulative angle encountered during the left tour until reaching the first edge vertex where the angle increment is negative and the target lies between TWLNS. If such a situation is not encountered, the direction is assumed to be $0 + \varepsilon$, where ε is a very small number (e.g., 0.01).
- *Inside of left: True* if the target is inside the polygon whose vertices starts at agent's location, jumps to the *outer left most point*, follows the border of the obstacle to the right and ends at the hit point of the ray.
- *Inside of inner left: True* if the target is inside the polygon that starts at agent's location, jumps to the *inner left most*

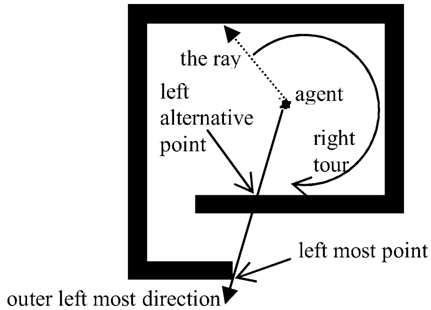

Fig. 9 Left alternative point

point, follows the border of the obstacle to the right and ends at the hit point of the ray.

- *Behind of left: True* if the target is in the region obtained by sweeping the angle from the ray direction to the *outer left most direction* in ccw order and the target is not inside of left.
- *Outer-left-zero angle blocking: True* if target is in the region obtained by sweeping the angle from the ray direction to the *outer left most direction* in ccw order.
- *Inner-left-zero angle blocking: True* if target is in the region obtained by sweeping the angle from the ray direction to the *inner left most direction* in ccw order.

In right analysis, the border of the obstacle is traced towards the right side and the same geometric properties listed above but now symmetric ones are identified. In the right analysis, additionally the following feature is extracted:

- *Left alternative point:* The last vertex in the *outer left most direction* encountered during the right tour until the *outer right most direction* is determined (See Fig. 9).

3.2 Evaluating individual results

In this phase, if an obstacle blocks the line of sight from the agent to the target, we determine a direction to move avoiding the obstacle to reach the target through a shorter path. In addition, the length of the path through the moving direction to the target is estimated. In the evaluation phase, the following features are used in addition to all the acquired geometric obstacle features given in Definition 1.

Definition 2. Estimated Target Distances

- d_{left}: The approximated distance, which is computed by finding the length of the path which starts from the agent location, jumps to the *outer left most point*, and then follows the border of the obstacle from left side until reaching the nearest point to the target on the border, and finally jumps to the target (See Fig. 10).
- $d_{left.alter}$: The approximated distance, which is computed by finding the length of the path which starts from the agent location, jumps to the *outer right most point*, and then to the *outer left most point*, follows the border of the obstacle from left side until reaching the nearest point to the target on the border, and finally jumps to the target (See Fig. 11).
- $d_{left.inner}$: The approximated distance, which is computed by finding the length of the path which starts from the agent location, continues with the *inner left most point*, and finally jumps to the target (See Fig. 12).

Algorithm 7. Evaluation Phase

if (*behind of left* and not *inside of right*) or (*behind of right* and not *inside of left*) **then**
  {**Case 1**}
  if *outer left most angle* + *outer right most angle* $\geq$ 360 **then**
    {**Case 1.1**}
    if distance from agent to *outer left most point* is smaller than distance from agent to *left alternative point* **then**
      {**Case 1.1.1**} Assign estimated distance as $min(d_{left}, d_{right.alter})$ and propose *outer left most direction* as moving direction
    else
      {**Case 1.1.2**} Assign estimated distance as $min(d_{left.alter}, d_{right})$ and propose *outer right most direction* as moving direction
    end if
  else
    {**Case 1.2**}
    if $d_{left} < d_{right}$ **then**
      {**Case 1.2.1**} Assign estimated distance as d_{left} and propose *outer left most direction* as moving direction
    else
      {**Case 1.2.2**} Assign estimated distance as d_{right} and propose *outer right most direction* as moving direction
    end if
  end if
  Mark obstacle as blocking the target
else if *behind of left* **then**
  {**Case 2**}
  if Target direction angle $\neq 0$ and *outer-right-zero angle blocking* **then**
    {**Case 2.1**} Assign estimated distance as d_{left} and propose *outer left most direction* as moving direction
  else
    {**Case 2.2**} Assign estimated distance as $d_{right.inner}$ and propose *inner right most direction* as moving direction
  end if

26: Mark obstacle as blocking the target
27: **else if** *behind of right* **then**
28: {**Case 3**}
29: **if** Target direction angle $\neq 0$ and *outer-left-zero angle blocking* **then**
30: {**Case 3.1**} Assign estimated distance as d_{right} and propose *outer right most direction* as moving direction
31: **else**
32: {**Case 3.2**} Assign estimated distance as $d_{left.inner}$ and propose *inner left most direction* as moving direction
33: **end if**
34: Mark obstacle as blocking the target
35: **else**
36: {**Case 4**}
37: **if** (*inside of left* and not *inside of right*) and (*inner-left-zero angle blocking* and not *inside of inner left*) **then**
38: {**Case 4.1**} Assign estimated distance as $d_{left.inner}$ and propose *inner left most direction* as moving direction
39: Mark obstacle as blocking the target
40: **else if** (*inside of right* and not *inside of left*) and (*inner-right-zero angle blocking* and not *inside of inner right*) **then**
41: {**Case 4.2**} Assign estimated distance as $d_{right.inner}$ and propose *inner right most direction* as moving direction
42: Mark obstacle as blocking the target
43: **end if**
44: **end if**

The estimated target distances over right side of the obstacle are similar to those over left side of the obstacle, and computed symmetrically (the terms *left* and *right* are interchanged in Definition 2). So, we have additional estimated target distances d_{right}, $d_{right.alter}$ and $d_{right.inner}$.

The evaluation procedure given in Algorithm 7 is executed for each obstacle (line 7 in Algorithm 6). The algorithm may propose a single moving direction that avoids a single obstacle, or may not propose any direction at all. In the algorithm,

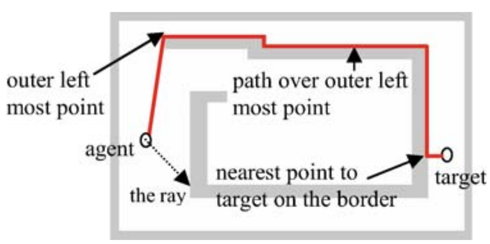

Fig. 10 Exemplified distance estimation from agent to target over outer left most point

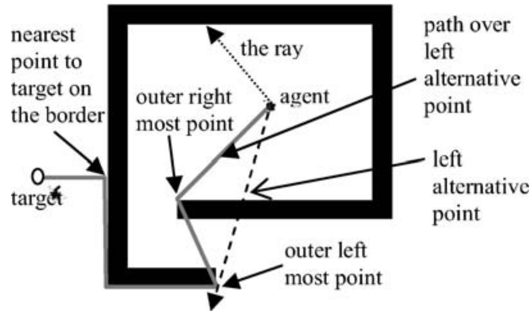

Fig. 11 Exemplified distance estimation from agent to target over left alternative point

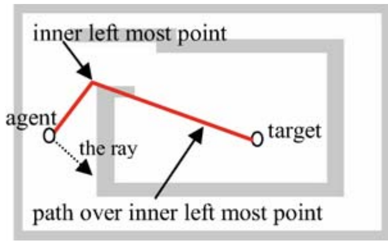

Fig. 12 Exemplified distance estimation from agent to target over inner left most point

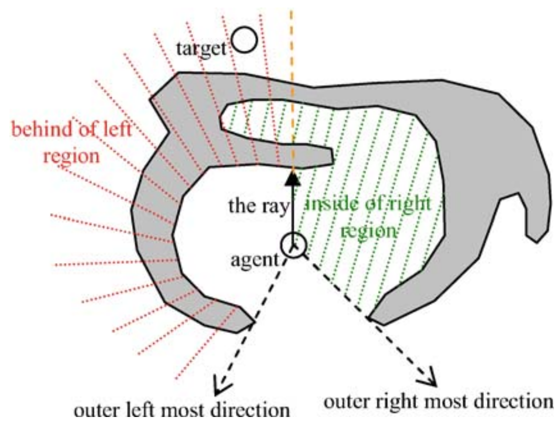

Fig. 13 *Case 1*: Target is behind of left region and not inside of right region (means not inside of the overlap area of behind of left and inside of right regions)

lets consider four top-level if-conditions in lines 1, 20, 29 and 38, which correspond to *Cases 1, 2, 3* and *4* respectively.

The algorithm enters *Case 1* only if the target is certainly behind the obstacle and the target can be reached by either going around the obstacle through the *outer left most point* or the *outer right most point*. The if-condition preceding *Case 1* consists of two disjuncted sub-conditions. The first one, *"behind of left and not inside of right"*, is satisfied when the target is behind the left side of the obstacle and we are sure that we cannot go to the target from the inner right region since the target is not *inside of right*. Hence, we need to go around the obstacle to reach the target. This case is exemplified in Fig. 13. The second sub-condition is symmetric to the first one.

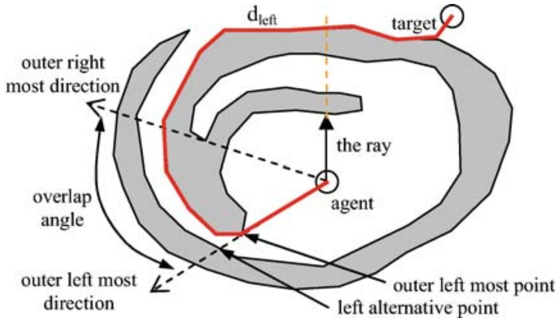

Fig. 14 *Case 1.1.1*: Since outer left most angle + outer right most angle ≥ 360 and outer left most point is nearer to the agent than left alternative point, the outer left most direction will be proposed

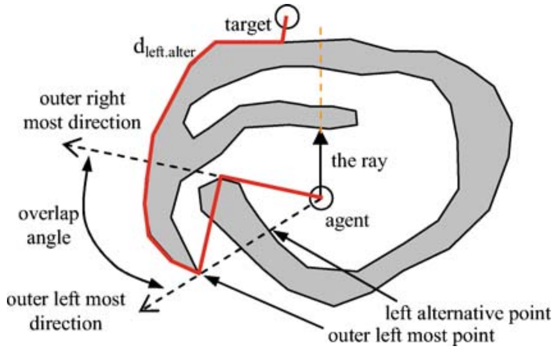

Fig. 15 *Case 1.1.2*: Since outer left most angle + outer right most angle ≥ 360 and left alternative point is nearer to the agent than outer left most point, the outer right most direction will be proposed

Case 1 has two second-level if-conditions in lines 2 and 10, which cover *Case 1.1* and *Case 1.2* respectively. The if-condition preceding *Case 1.1* checks if the sum of the *outer left most angle* and the *outer right most angle* is greater or equal to 360 degree. The condition is satisfied if the swept angles from left and right to opposite orientations meet each other and some angle overlap occurs. This means that the agent is surrounded by the obstacle in all directions, the target is outside, and the agent needs to go out from the nearest exit. In this case, if the nearest exit is determined as the corner of the *outer left most edge*, *Case 1.1.1* otherwise *Case 1.1.2* is executed. The cases are illustrated in Figs. 14 and 15. If the if-condition preceding *Case 1.1* is not satisfied, *Case 1.2* is executed, which means there is no angle overlap and the agent is only surrounded by the obstacle in some directions but not all. In this case, the edge minimizing the route distance to the target is determined, and either *Case 1.2.1* or *Case 1.2.2* is executed depending on the value of d_{left} and d_{right}. This case is exemplified in Fig. 16.

The algorithm enters *Case 2* only if the target is certainly blocked by the obstacle, and the target can be reached by going through either the corner of the *outer left most edge* or the *inner right most edge*. In such a case, there are two possible regions the target can be located in. The first one handled in *Case 2.1* is between the *outer left most edge* and

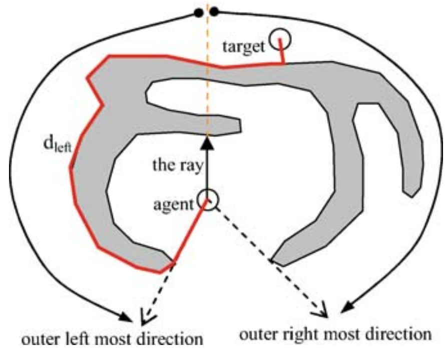

Fig. 16 *Case 1.2.1*: Since outer left most angle + outer right most angle < 360 and the estimated path length to the target through the outer left most point is shorter than the right one, the outer left most direction will be proposed

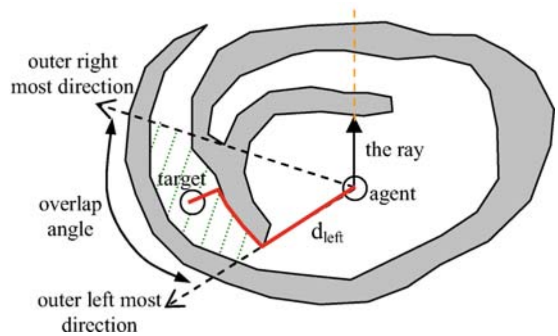

Fig. 17 *Case 2.1*: Since the target is at the direction that falls into the overlap angle, the outer left most direction will be proposed

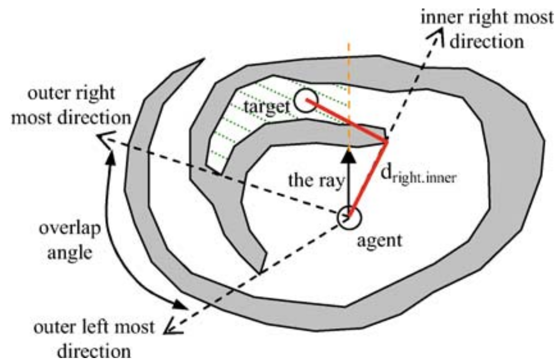

Fig. 18 *Case 2.2*: Since the target is not at the direction that falls into the overlap angle, the inner right most direction will be proposed

the *outer right most edge* (see Fig. 17), and the target inside that region can be reached by going through the corner of the *outer left most edge*. The second one handled in *Case 2.2* lies in the inner part of the obstacle (see Fig. 18), and the target inside this region can be reached by going through the corner of the *inner right most edge*. We will not go into the details of *Case 3* since it is symmetric to *Case 2*.

The algorithm enters *Case 4* if none of the previous top-level if-conditions are satisfied. *Case 4* has two second-level if-conditions in lines 39 and 43, which cover *Case 4.1* and *Case 4.2* respectively. The if-condition preceding *Case 4.1*

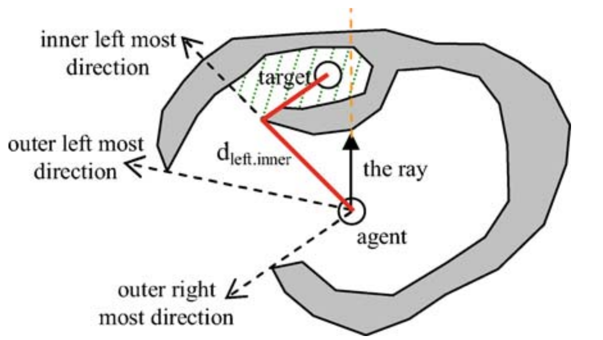

Fig. 19 *Case 4.1*: Since target is inside of left region, but not right, and inner-left-zero angle blocking and not inside of inner left, the inner left most direction will be proposed

consists of two conjuncted sub-conditions. The first one, *"inside of left and not inside of right"*, is satisfied when the target is inside of the left but not right region, thus we are sure that we need to enter the left region but we don't know yet if the flying direction is feasible. The second sub-condition, *"inner-left-zero angle blocking and not inside of inner left"*, is satisfied if the target is behind the *inner left most edge*, thus the flying direction is not feasible. If both sub-conditions hold, we known that the agent needs to enter the left region through the corner of the *inner left most edge*. This case is illustrated in Fig. 19. We will not examine *Case 4.2* since it is also symmetric to *Case 4.1*.

If none of the above conditions are satisfied, the algorithm does not propose any moving direction meaning the flying direction may still be a feasible choice to move.

3.3 Merging entire results

In the merging phase, the evaluation results (moving direction and estimated distance pairs) for all obstacles are used to determine a final moving direction to reach the target. The proposed direction will be passed to RTTES algorithm (Algorithm 5) for final decision. The merging algorithm is given in Algorithm 8.

Algorithm 8. Merging Phase

```
if all the directions to neighbor cells are closed then
    propose no moving direction and halt with failure
end if
Select the obstacle (most constraining obstacle) that
is marked as blocking the target and maximize the
distance to the target, if there exists one
if most constraining obstacle exists then
    identify a moving direction that gets around the
    most constraining obstacle avoiding the
    remaining obstacles
else
    select the moving direction as the direct flying
    direction to the target
end if
{Compute utility of each neighbor cell}
for each neighbor cell do
    if direction of the neighbor cell is closed then
        set utility to zero
    else
        set utility to (181 − dif)/181, where dif is
        smallest angle between the proposed moving
        direction and the direction of the neighbor cell
    end if
end for
```

The most critical step of the algorithm is to compute the moving direction to get around *the most constraining obstacle*. The reason why we determine the moving direction based on the most constraining obstacle is the fact that it might be blocking the target the most. We aim to get around the most constraining obstacle and to do this we have to reach its border. In case there are some other obstacles on the way to the most constraining obstacle, we need to avoid them and determine the moving direction accordingly. Our algorithm works even if we ignore the intervening obstacles but we employ the following technique in order to improve solution quality with respect to path length.

Let the final direction to be proposed by the algorithm considering ray r be pd_r. Initially pd_r is set to the direction dictated by the most constraining obstacle o_r hit by ray r. Assume that pd_r is computed in the left tour. Note that the pd_r was determined during the counter clockwise (ccw) tour started from the hit point of ray r. If pd_r is blocked by some obstacles, pd_r can be changed by sweeping pd_r in clockwise direction until pd_r is not blocked by any obstacle or pd_r becomes the direction of ray r. By definition, we know that r is guaranteed to reach the border of obstacle o_r before hitting any other obstacle. In order to determine intervening obstacles, we check obstacles (not equal to o_r) hit by the other rays fall into ccw angle between r and pd_r. If an obstacle o_s hit by ray s has *outer left most direction* outside ccw angle between ray s and pd_r, and has *outer right most direction* inside ccw angle between r and s, then the obstacle o_s blocks pd_r and proposed direction should be swept to *outer left most direction* of obstacle o_s. Using this information we compute the direction nearest to pd_r between r and pd_r and not blocked by the intervening obstacles. The method is exemplified in Fig. 20. The similar mechanism is also used to compute the proposed direction for pd_r detected in the right tour, but this time, left/right and ccw/cw are interchanged.

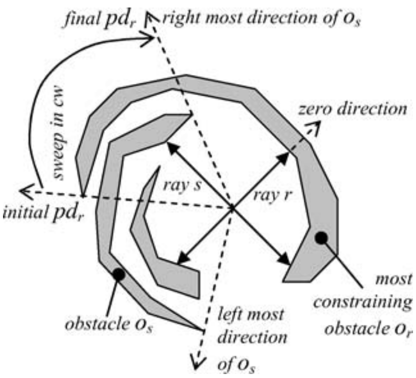

Fig. 20 An example of avoiding the intervening obstacles

3.4 Analysis of the algorithm

In this section we give the computational complexity of our algorithm and its proof of correctness.

In each move, RTTE performs steps similar to RTEF. In RTTE, the number of passes over obstacle borders is greater than that of RTEF and in each pass more time is consumed. As a result, at each step RTTES is slower than RTEF. Although the RTEF seems to be more efficient than RTTE, its worst case complexity is the same as that of RTEF, which is $O(w.h)$ per step, where w is the width and h is the height of the grid [16].

Since increasing the grid size decreases the efficiency, a search depth (d) can be introduced similar to RTEF in order to limit the worst case complexity of RTTE. A search depth is a rectangular area of size $(2d + 1)\cdot(2d + 1)$ centered at agent location, which makes the algorithm treat the cells beyond the rectangle as non-obstacle. With this limitation, complexity of RTTE becomes $O(d^2)$ per step [16].

A single iteration of RTTES given in Algorithm 5 is similar to RTEF shown in Algorithm 2, which is proved to be correct [16]. It uses visit count and history to prevent infinite loops, which is the same as RTEF. The difference is in the selection of moving directions. RTEF selects an open direction minimizing the Euclidian distance to the target; on the other hand RTTES selects an open direction maximizing the utility computed by RTTE heuristic which measures the actual distance to the target more precisely than Euclidian distance. The algorithm is complete in the sense that if the target is accessible, the agent will surely find his/her way to the target without entering any infinite loop.

4 Performance analysis

In this section, we present the results of the comparisons of RTTES, RTEF and RTA* on randomly generated sample grids of size 200 × 200. We used RTTES and RTEF with two variations: the first one uses *visit count* (*VC*), the second one uses both *visit count* and *history* (*VCH*) to prevent infinite loops. Thus, we present performance of algorithms RTA*, RTEF-VC, RTEF-VCH, RTTES-VC and RTTES-VCH.

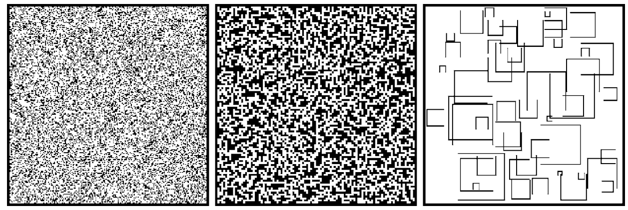

Fig. 21 A random grid (left), a maze grid (middle), a *U*-type grid (right)

We used grids of three different types: *random*, *maze* and *U-type* (see Fig. 21). Three *random* grids were generated randomly based on different obstacle ratios (30%, 35% and 40%). Nine *maze* grids were produced with the constraint that every two non-obstacle cells are always connected through a path, which is usually one. Two parameters, obstacle ratio (30%, 50% and 70%) and corridor size (1, 2 and 4 cell corridors) were used to produce mazes. Four *U-type* grids were created by randomly putting *U*-shaped obstacles of random sizes on an empty grid. We took into consideration the number of *U*-type obstacles (30, 50, 70 and 90), minimum and maximum sizes of *U*-shaped obstacles (between 5 and 50 cell sizes) to create *U*-type grids. For each grid, we produced 10 different randomly generated agent-target location pairs taken on opposite sides of the grid, and made all the algorithms use the same pairs for fairness.

In our experiments, we assumed that the agent perceives the environment up to a limit, which is called *vision range* (*v*). Being at its center, the agent can only sense cells within the rectangular area of size $(2v + 1)\cdot(2v + 1)$. We used the statement *infinite vision* to emphasize that the agent has unlimited sensing capability and knows the entire grid world before the search starts. Our tests are performed with the vision ranges 10, 20, 40 and *infinite* cells and with search depths 10, 20, 40, 80 and *infinite* cells to limit the worst case complexity of RTEF and RTTES.

4.1 Analysis of path lengths

In order to compare the solution path lengths of algorithms RTA*, RTEF and RTTES, we used 16 grid worlds with 4 different vision ranges (10, 20, 40, infinite) and with 5 different search depths (10, 20, 40, 80, infinite), totally making 320 test configurations. For each configuration, we performed 10 runs per algorithm totally making 16000 runs. As a result, we observed that RTTES-VCH performs significantly better than the other algorithms. The average of path length results of maze, random and *U*-type grids are given in Fig. 22.

Later on, we split the test runs into seven categories: maze grids with 30%, 50% and 70% obstacles, random grids with 30%, 35% and 40% obstacles, and *U*-type grids, and evaluated the results with respect to *vision range* and *search depth*.

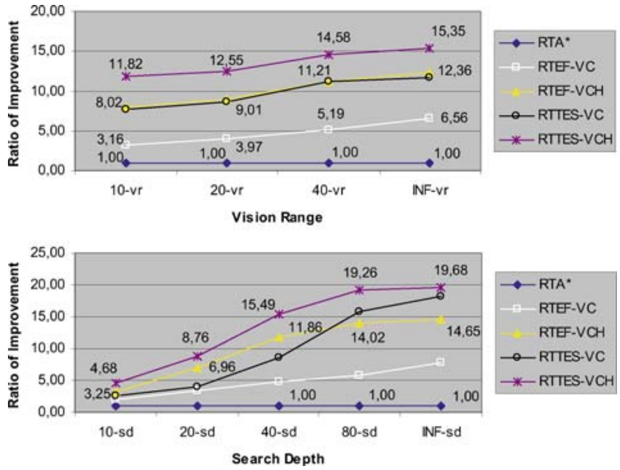

Fig. 22 Average of path length results of maze, random and *U*-type grids for increasing visual ranges (top) and search depths (bottom)

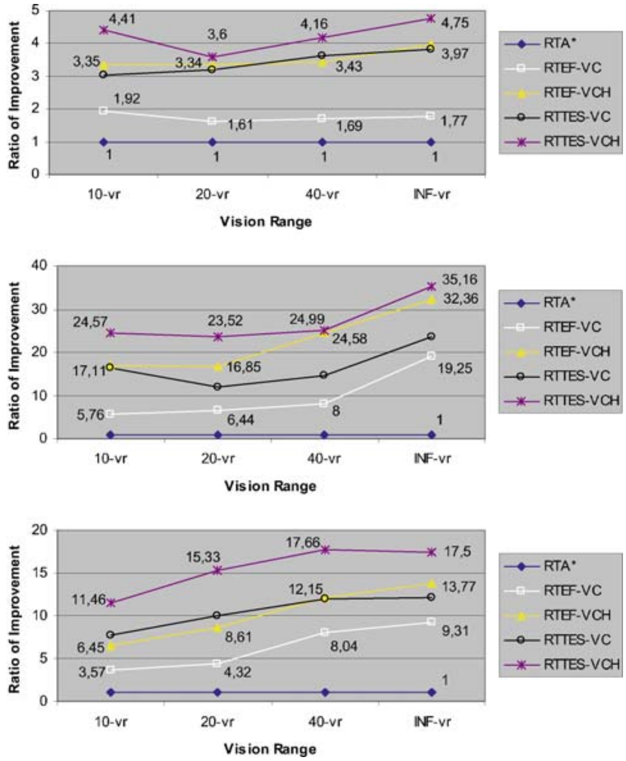

Fig. 23 Path length results of maze grids with 30% (top), 50% (middle) and 70% (bottom) obstacle ratios for increasing vision ranges

In some of the charts shown in the following sections, increasing the vision range or search depth does not always improve the solution. This problem usually appears because of the characteristic of the grid, the misleading changes in the shape of the grid known by the agent or stopping the search at an immature depth guiding a local optimal. When the agent choices a wrong alternative on a critical decision point, the rest of the search is significantly affected (for more information, see [36] and [16]).

4.1.1 *Effect of vision range*

The charts in Figs. 23, 24, and 25 present the effect of vision range on path lengths in maze, random and *U*-type grids, respectively. The horizontal axis is the vision range (10, 20, 40 and infinite) and the vertical axis contains the ratio of improvement in the path length with respect to RTA* (the path length of RTA* divided by that of the compared algorithm). Note that the ratio is always 1 for RTA*.

According to the results, we observe that increasing vision range improves the performance, which is not too steep. Especially, in random grids it is almost ineffective since obstacles are not very large. When we order the algorithms according to their performance, we see that RTTES-VCH is the best, RTA* is the worst and RTEF-VC is the second worst all the time. For RTTES-VC and RTEF-VCH, there is no obvious ordering since results change depending on the grid type. Although RTTES-VC is powered by a precise distance estimation heuristic, it does not use the history and hence its performance decreases in grids with very high obstacle ratios. As a result RTTES-VC performs worse than RTEF-VCH in maze grids with 50% obstacles and random grids with 40% obstacles. But in other grid types, RTTES-VC is either better than RTEF-VCH or head to head.

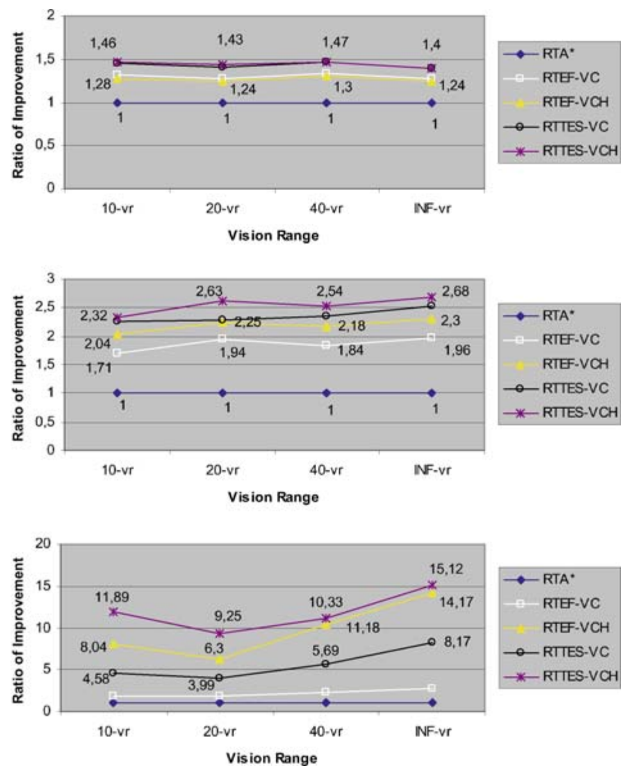

Fig. 24 Path length results of random grids with 30% (top), 35% (middle) and 40% (bottom) obstacle ratios for increasing vision ranges

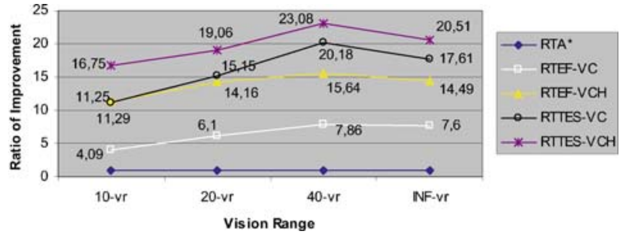

Fig. 25 Path length results of *U*-Type grids for increasing vision ranges

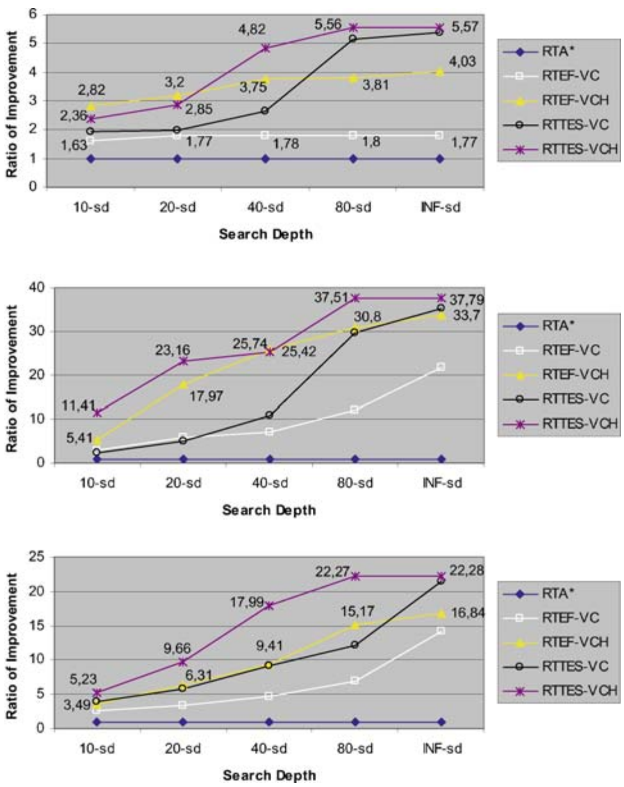

Fig. 26 Path length results of maze grids with 30% (top), 50% (middle) and 70% (bottom) obstacle ratios

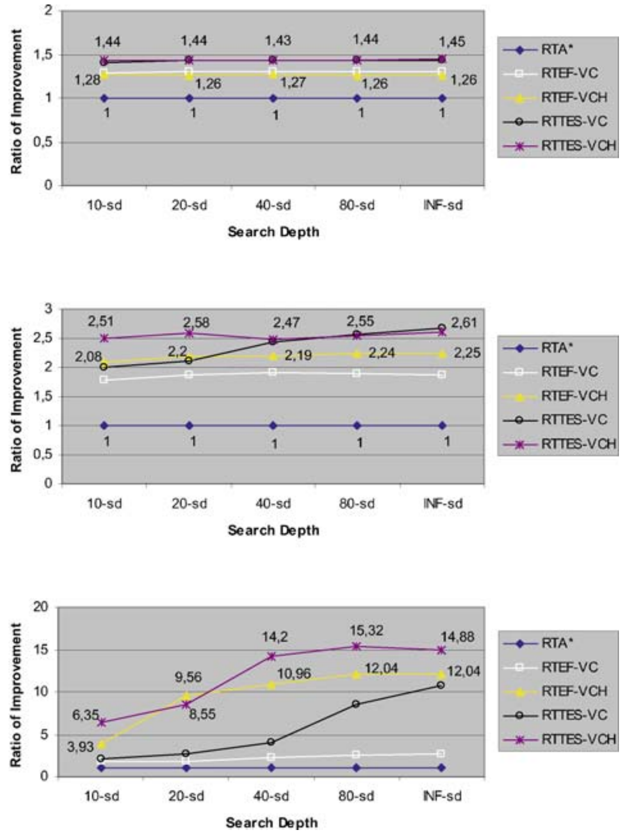

Fig. 27 Path length results of random grids with 30% (top), 35% (middle) and 40% (bottom) obstacle ratios

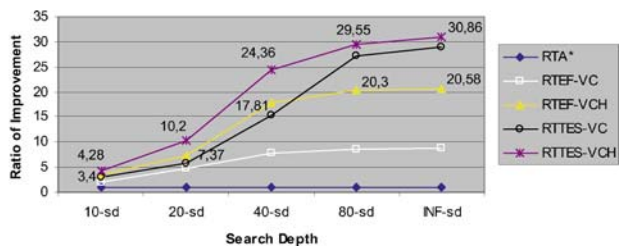

Fig. 28 Path length results of *U*-Type grids

4.1.2 *Effect of search depth*

The Figs. 26, 27, and 28 show the effect of search depth on path lengths in maze, random and *U*-type grids, respectively. The horizontal axis is the search depths (10, 20, 40, 80 and infinite) and the vertical axis contains the ratio of improvement in the path length with respect to RTA* again.

The results show that RTEF and RTTES are more sensitive to the search depth change than the vision range change, especially the RTTES since limited search depth crops the obstacle borders, which may mislead the distance estimation procedure of RTTE heuristic. According to the outcomes, the scene seems to be almost the same as the previous one. RTTES-VCH is the best again, RTA* and RTEF-VC are the worst, RTTES-VC and RTEF-VCH are competing each other.

There is one important conclusion that can be drawn from the charts. The path length of RTTES-VCH with 80 and infinite search depths are almost the same. However, with 80 search depth, the number of moves executed per second is about 2.18 times higher on maze grids, 2.02 times higher on random grids and 1.58 times higher on *U*-type grids, which is given next.

4.2 Analysis of execution times

To compare the execution times of the algorithms, we used the same test configurations and runs mentioned before, and evaluated the results of 16000 runs for different search depths and grid types. We examined the results in two categories:

Table 1 Average number of moves per second in maze grids

Search depth	10-c	20-c	40-c	80-c	INF-c
maze grids with 30% obstacles					
RTA*	3048	3048	3048	3048	3048
RTEF-VC	2358	1945	1576	1296	1071
RTEF-VCH	1767	1140	609	280	158
RTTES-VC	1602	1135	886	745	567
RTTES-VCH	1009	550	285	137	79
maze grids with 50% obstacles					
RTA*	3113	3113	3113	3113	3113
RTEF-VC	1991	1091	298	58	12
RTEF-VCH	1485	671	159	41	19
RTTES-VC	1085	450	132	27	6
RTTES-VCH	702	272	73	22	9
maze grids with 70% obstacles					
RTA*	3134	3134	3134	3134	3134
RTEF-VC	2091	1277	403	85	17
RTEF-VCH	1631	834	238	59	27
RTTES-VC	1160	542	171	41	9
RTTES-VCH	781	331	102	33	14

Table 2 Average number of moves per second in random grids

Search depth	10-c	20-c	40-c	80-c	INF-c
random grids with 30% obstacles					
RTA*	2857	2857	2857	2857	2857
RTEF-VC	2503	2380	2458	2281	1987
RTEF-VCH	1744	1250	854	514	355
RTTES-VC	1707	1585	1526	1434	1378
RTTES-VCH	950	605	379	229	158
random grids with 35% obstacles					
RTA*	3005	3005	3005	3005	3005
RTEF-VC	2119	1648	1316	981	617
RTEF-VCH	1401	780	343	138	51
RTTES-VC	1319	874	670	517	285
RTTES-VCH	709	341	135	61	29
random grids with 40% obstacles					
RTA*	3093	3093	3093	3093	3093
RTEF-VC	2026	1329	497	160	83
RTEF-VCH	1297	572	133	46	20
RTTES-VC	1108	616	194	71	25
RTTES-VCH	572	223	66	25	10

Table 3 Average number of moves per second in *U*-type grids

Search depth	10-c	20-c	40-c	80-c	INF-c
RTA*	3075	3075	3075	3075	3075
RTEF-VC	2611	1961	1067	601	507
RTEF-VCH	2068	1374	648	262	177
RTTES-VC	1854	1058	453	219	167
RTTES-VCH	1194	639	290	131	83

step and total execution times. The results are presented in the following sub-sections.

4.2.1 Number of moves per second

In this section, we present the step execution performance of RTA*, RTEF and RTTES, which were run on a Centrino 1.5 GHz laptop. In Tables 1–3, the average number of moves executed per second in maze, random and *U*-type grids can be seen. The rows of the tables are representing the compared algorithms and the columns are for the search depths.

RTA* is the most efficient algorithm, which performs about 3046 moves/sec and has almost constant step execution time. The second place is for RTEF-VC, and it performs about 613 moves/sec with infinite search depth and 2242 moves/sec with 10 search depth. The third place is shared by RTEF-VCH and RTTES-VC. Using infinite search depth, RTTES-VC shows better performance with 348 moves/sec versus 115 moves/sec; and using 10 search depth, RTEF-VC performs better with 1627 moves/sec versus 1405 moves/sec. And finally, RTTES-VCH gets the last place since its computational cost is high. RTTES-VCH executes 54 moves/sec using infinite search depth and 845 moves/sec using 10 search depth.

The execution time performances of RTEF and RTTES with limited search depth relative to unlimited search depth are given in Fig. 29. The horizontal axis is the search depths (10, 20, 40 and 80), and the vertical axis is the execution time performance (the number of moves per second with limited search depths divided by that with infinite search depths). When we compare the effect of search depth on path lengths and step execution times, we observe that increasing search depth increases the step execution performance much more than path lengths.

4.2.2 Total execution times

The total execution time is depended on the total number of moves performed to reach the target and the time spent per move. The results are shown in Fig. 30. The horizontal axis is the search depths (10, 20, 40, 80 and infinite), and the vertical axis is the ratio of improvement in the total execution time with respect to RTA* (the total execution time of RTA* divided by that of the compared algorithm).

The results show that although the step execution of RTEF and RTTES is highly inefficient compared to RTA*, the total time spent per run is less in maze and *U*-type grids since the path lengths are significantly shorter. In random grids, the performance of RTEF and RTTES drops much, especially

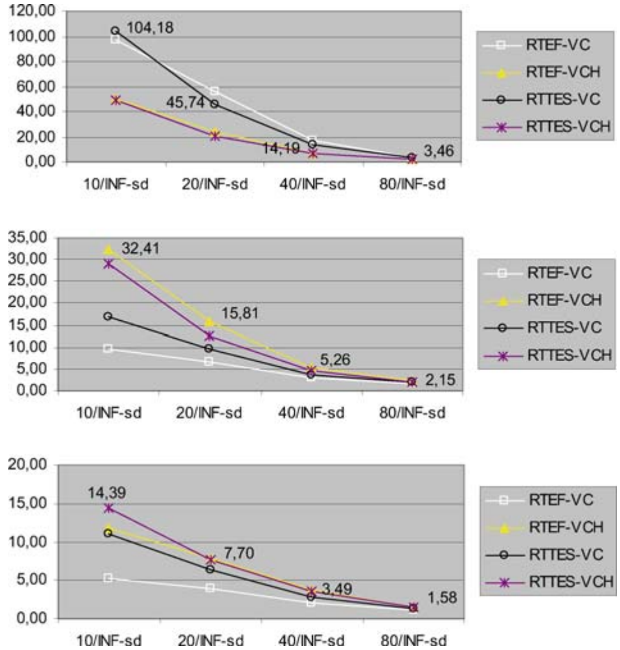

Fig. 29 The step execution performance increase of limited search depths over infinite search depths on maze (top), random (middle) and U-Type (bottom) grids

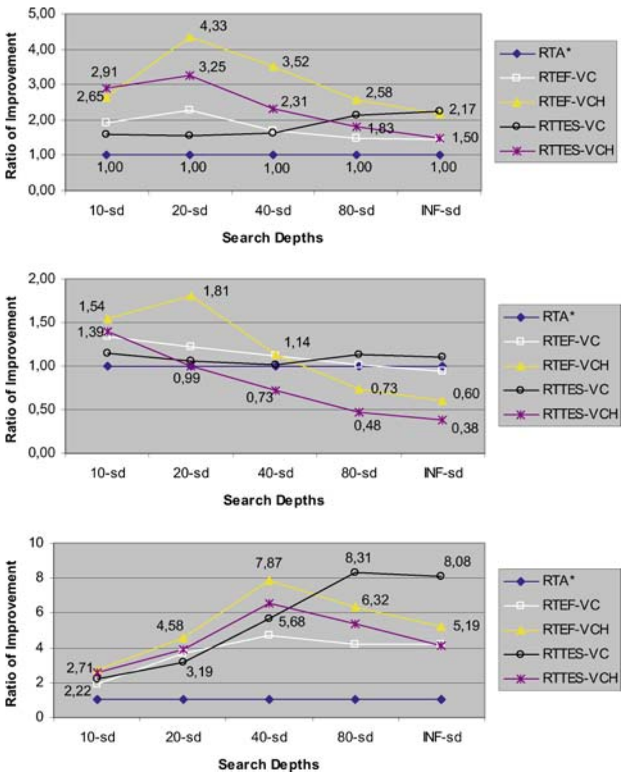

Fig. 30 Total execution time results of maze (top), random (middle) and U-type (bottom) grids

Table 4 Average ratios of algorithms' path lengths over optimal path lengths and their standard deviations

	RTA*	RTEF-VC	RTEF-VCH	RTTES-VC	RTTES-VCH
			maze grids		
Avg	30.746	1.967	1.347	1.047	1.044
Std	45.312	2.744	1.219	0.091	0.078
			random grids		
Avg	10.532	3.577	1.432	1.216	1.197
Std	16.877	3.837	0.388	0.138	0.118
			u-type grids		
Avg	39.141	10.467	1.903	1.128	1.148
Std	52.222	14.776	0.961	0.178	0.186
			all grids		
Avg	26.806	5.337	1.561	1.130	1.129
Std	38.137	7.119	0.856	0.136	0.127

with high search depths because random grids are usually easy for RTA*, thus path length improvements of RTEF and RTTES are not very significant. The step execution of RTEF is more efficient than RTTES, and path length reduction with RTTES is not very large compared to step execution time increase, therefore total execution times of RTEF usually seem to be better than RTTES on the average.

4.3 Comparison with optimal solution paths

In the last experiment, we compared the path lengths of RTEF-VC, RTEF-VCH, RTTES-VC, RTTES-VCH and RTA* with the optimal ones. We used the off-line path planning algorithm A*, and generated the optimal paths assuming that the grids are fully known (infinite vision). We present the proximity of solutions to the optimal ones in Table 4 computed as the ratio of the algorithms' path lengths divided by the optimal path lengths.

The results show that the path lengths of RTTES variations are only 1.13 times longer than the optimal ones on the average, whereas those of RTEF-VC, RTEF-VCH and RTA* are 5.33, 1.56 and 26.80 times longer, respectively. Also, the standard deviations in path lengths obtained by RTTES algorithms are significantly less than the others. Concerning the types of grids, we see that the best improvement was obtained in U-type grids, which was expected due to the weakness of RTEF in these grids [16].

5 Conclusion

In this paper, we have focused on real-time search for grid-type problems, and presented an effective heuristic method (RTTE) and a real-time search algorithm (RTTES) based on

Table 5 A breif comparison of RTA*, RTEF and RTTES

RTA*	RTA* relies on a user specified heuristic function to decide next move in each step. Since it is not able to incorporate environmental information into its decision, it always finds longer paths compared to other algorithms. However, time per move cost is very low (i.e., $O(1)$). By using large look-ahead depths, the path lengths can be made significantly shorter, but it requires exponential time in the length of look-ahead depth.
RTEF	In addition to the user specified heuristic function, RTEF explores the environment and is able to detect closed directions correctly. Path lengths are significantly reduced compared to RTA*, but the time per move increases because of additional computational cost, namely $O(w.h)$ where w and h are width and height of environment, respectively. However, the complexity of algorithm can be reduced and bounded with the help of search depth.
RTTES	In addition to its capability of identifying closed directions, RTTES is able to analyze the extracted border information in details to assess which direction to move is the best. The path lengths are significantly reduced compared to RTEF and they are very close to optimal ones in all types of grids. Although the complexity of the algorithm is the same as RTEF, the time spent per move is almost doubled since more time consuming computations are performed. However, in terms of total time spent, RTEF and RTTES are head to head.

RTTE. We have compared RTTES with RTA* and RTEF with the help of more than 16000 test runs. A brief comparison of RTA*, RTEF and RTTES can be found in Table 5.

With respect to path lengths, experimental results showed that RTTES-VCH is able to make use of environmental information very successfully to improve the solutions, and performs the best in all types of grids. The second and third places are shared by RTTES-VC and RTEF-VCH, which are usually going head to head. And finally, the forth and fifth places are owned by RTEF-VC and RTA* respectively. Concerning the total execution times, we have observed that although the step execution time of RTA* is low, RTTES and RTEF perform much better than RTA* in maze and U-type grids. In random grids, the performance of RTEF and RTTES drops much, especially with high search depths since effective usage of environmental information gains less.

We have also seen that RTTES is able to find almost optimal path to the target in fully known grids. The results are only 1.13 times longer than the optimal on the average, and have standard deviation less than 0.13. This ratio is a significant success for a real-time algorithm, which leaves a very little space for later improvements in all types of grids with respect to path length. But we think that there are still some improvements that could be done. In path length computations of d_{left}, $d_{left.alter}$, d_{right} and $d_{right.alter}$, the algorithm follows the border of the obstacle from left/right side until reaching the nearest point to the target on the border. This process may sometimes lead to over estimation in situations where the obstacles are concave. Therefore, we are planning to improve this heuristic as a future work.

References

1. Russell S, Norving P (1995) Artificial intelligence: A modern approach. Prentice Hall, Inc.
2. Gutmann J, Fukuchi M, Fujita M (2005) Real-time path planning for humanoid robot navigation. International joint conferance on artificial intelligence IJCAI-05, pp 1232–1237
3. Knight K (1993) Are many reactive agents better than a few deliberative ones? In: Proceedings of the 10th int'l joint conf. on artificial intelligence, pp 432–437
4. Korf R (1990) Real-time heuristic search. Artif Intell 42(2–3):189–211
5. Ishida T, Korf R (1995) Moving target search: a real-time search for changing goals. IEEE Trans Patt Anal Mach Intell 17(6):97–109
6. Ishida T (1996) Real-time bidirectional search: coordinated problem solving in uncertain situations. IEEE Trans Patt Anal Mach Intell 18(6)
7. Undeger C (2001) Real-time mission planning for virtual human agents. M.Sc. Thesis in Computer Engineering Department of Middle East Technical University, 2001
8. Stentz A (1994) Optimal and efficient path planning for partially-known environments. In: Proceedings of the IEEE international conference on robotics and automation
9. Mudgal A, Tovey C, Greenberg S, Koenig S (2005) Bounds on the travel cost of a mars rover prototype search heuristic. SIAM J Discrete Math 19(2):431–447
10. Stentz A (1995) The focussed D* algorithm for real-time replanning. In: Proceedings of the int'l joint conference on artificial intelligence
11. Koenig S, Likhachev M (2002) D* lite. In: Proceedings of the national conference on artificial intelligence, pp 476–483
12. Koenig S, Likhachev M (2002) Improved fast replanning for robot navigation in unknown terrain. In: Proceedings of the international conference on robotics and automation
13. Koenig S, Likhachev M (2005) Fast replanning for navigation in unknown terrain. IEEE Trans Robo 21(3):354–363
14. Koenig S (2004) A comparison of fast search methods for real-time situated agents. AAMAS 2004 pp 864–871
15. Undeger C, Polat F, Ipekkan Z (2001) Real-time edge follow: A new paradigm to real-time path search. In: The proceedings of GAME-ON
16. Undeger C, Polat F (2007) Real-time edge follow: a real-time path search approach. IEEE Transaction on Systems, Man and Cybernetics, Part C
17. Undeger C, Polat F (2006) Real-time target evaluation search. In: 5th int'l joint conf on autonomous agents and multiagent systems, pp 332–334
18. Tanenbaum A (1996) Computer networks. Prentice-Hall, Inc.
19. Koenig S, Likhachev M, Liu Y, Furcy D (2004) Incremental heuristic search in artificial intelligence. Artif Intell Magazine
20. Konar A (2000) Artificial intelligence and soft computing: behavioral and cognitive modeling of human brain. CRC Press LLC
21. Michalewicz Z (1986), Genetic algorithms + data structure = evolution programs. Springer-Verlag, New York
22. Sugihara K, Smith J (1997) Genetic algorithms for adaptive planning of path and trajectory of a mobile robot in 2d terrains. Technical Report, number ICS-TR-97-04, University of Hawaii, Department of Information and Computer Sciences

23. Cheng P, LaValle SM (2002) Resolution complete rapidly-exploring random trees. In: Proceedings of IEEE int'l conf on robotics and automation, pp 267–272
24. LaValle S, Kuffner J (1999) Randomized kinodynamic planning. In: Proceedings of the IEEE international conference on robotics and automation (ICRA'99)
25. LaValle SM, Kuffner JJ (2001) Rapidly-exploring random trees: progress and prospects, ser. Algorithmic and Computational Robotics: New Directions. A K Peters, Wellesley, MA, pp 293–308
26. Kavraki L, Latombe J (1998) Probabilistic roadmaps for robot path planning, ser. In Practical Motion Planning in Robotics: Current and Future Directions. Addison-Wesley
27. Sanchez G, Ramos F, Frausto J (1999), Locally-optimal path planning by using probabilistic roadmaps and simulated annealing. In: Proceedings IASTED robotics and applications international conference
28. Hernndez C, Meseguer P (2005) Lrta*(k). Int'l joint conf on artificial intelligence IJCAI-05, pp 1238–1243
29. Shimbo M, Ishida T (2003) Controlling the learning process of real-time heuristic search. Artif Intell 146(1):1–41
30. LVJ, Skewis T (1987) Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. Algoritmica 2:403–430
31. Kamon I, Rivlin E, Rimon E (1996) A new range-sensor based globally convergent navigation algorithm for mobile robots. In: Proc of the IEEE int'l conf on robotics and automation vol. 1, pp 429–435
32. Bruce J, Veloso M (2002) Real-time randomized path planning for robot navigation. In: Proceedings of int'l conf on intelligent robots and systems, pp 2383–2388
33. Hsu D, Kindel R, Latombe J, Rock S (2002) Randomized kinodynamic motion planning with moving obstacles. Int J Robotics Res 21(3):233–255
34. Stilman M, Kuffner J (2005) Navigation among movable obstacles: Real-time reasoning in complex environments. Int J Humanoid Robo 2(4):1–24
35. Koenig S, Likhachev M (2006) Real-time adaptive a*. In: 5th int'l joint conference on autonomous agents and multiagent systems, pp 281–288
36. Hamidzadeh B, Shekhar S (2005) Dynoraii: A real-time path planning algorithm. Int J Artif Intell Tools 2(1):93–115