Real-Time Moving Target Search

Cagatay Undeger[1] and Faruk Polat[2]

[1] Savunma Teknolojileri Muhendislik ve Tic.A.S., 06510 Ankara, Turkey
[2] Middle East Technical University, 06531 Ankara, Turkey
cundeger@stm.com.tr,polat@ceng.metu.edu.tr

Abstract. In this paper, we propose a real-time moving target search algorithm for dynamic and partially observable environments, modeled as grid world. The proposed algorithm, Real-time Moving Target Evaluation Search (MTES), is able to detect the closed directions around the agent, and determine the best direction that avoids the nearby obstacles, leading to a moving target which is assumed to be escaping almost optimally. We compared our proposal with Moving Target Search (MTS) and observed a significant improvement in the solution paths. Furthermore, we also tested our algorithm against A* in order to report quality of our solutions.

Key words: Path Planning, Real-Time Search, Moving Target Search

1 INTRODUCTION

Pursuing a moving target is one of the most challenging problems in areas such as robotics, computer games, etc. Off-line and incremental path planning algorithms are not able to handle moving targets in real-time, and most of the on-line search algorithms are specifically designed for partially observable environments with static targets. The most well known algorithm for moving targets is Moving Target Search (MTS) [2], which maintains a heuristic table that contains estimated costs of paths between every pair of coordinates.

In this paper, we propose a new moving target search algorithm, Real-Time Moving Target Evaluation Search (MTES), which is build on Real-Time Target Evaluation Search [9] developed for partially observable environments with static targets. MTES is able to estimate the distance to the target considering the intervening obstacles and discard some non-promising alternative moving directions in real-time in order to guide the agent to a moving target. The method sends virtual (non-physical) 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, a single moving direction is determined. To show the performance of our algorithm, we compared MTES with MTS-c, MTS-d and A*. For the experiments, we randomly generated grids of different types, and developed a successful prey

algorithm (Prey-A*) in order to challenge the algorithms used in the experiments. The results showed that MTES produces near optimal solutions, and outperforms MTS-c and MTS-d significantly.

The related work is given in Section 2. In section 3, MTES is described in details. In Section 4, the performance analysis of MTES is presented. Finally Section 5 is the conclusion.

2 RELATED WORK

Off-line path planning algorithms [6] are hard to use for large dynamic environments and for moving targets because of their time requirements. One way to make these algorithms more efficient is to change them from off-line to incremental [7, 4] in order to avoid re-planning from scratch. Although these algorithms work fine with partially observable environments, they are sometimes not efficient enough and usually not capable of handling moving targets. There are also a number of on-line approaches [5, 3, 8, 9, 2]. As a matter of fact, only few of these on-line algorithms can be adapted against a moving target.

Moving Target Search (MTS) [2] is a well known real-time algorithm for pursuing a moving target, which is built on Learning Real-Time A*. The algorithm maintains a table of heuristic values, presenting the function $h(x, y)$ for all pairs of locations x and y in the environment, where x is the location of the agent and y is the location of the target. The original MTS is a poor algorithm in practice because when the target moves, the learning process has to start all over again. Therefore, two MTS extensions namely *Commitment to Goal* (MTS-c) and *Deliberation* (MTS-d) [2] are proposed to improve the solution quality. In order to use the learned table values more effectively, MTS-c ignores some of the target's moves, and MTS-d performs an off-line search while in a heuristic depression.

When we look at the prey algorithms, we usually cannot see very successful studies. Mostly the focus is on the predators, and the prey algorithms commonly use hybrid techniques mixing the reactive strategies [1, 2]. Since these strategies are not challenging enough for our algorithms, we developed an off-line strategy, which is slow but more powerful with respect to its escape capability.

3 MOVING TARGET SEARCH

3.1 Problem Description

In our study, the environment is a grid world, where any grid cell can either be free or obstacle. There is a single agent (predator) that aims to reach a static or moving target (prey). Both are randomly located far from each other in non-obstacle grid cells. The predator is expected to reach the prey from a short path avoiding obstacles in real-time. We assume that the predator knows the prey's location all the time, but has limited perception, and is only able to sense the obstacles around him within a square region centered at the agent location. The size of the square is $(2v+1)$x$(2v+1)$, where v is the **vision range**. We used the

term *infinite vision* to emphasize that the agent has unlimited sensing capability and knows the entire grid world before the search starts. The unknown parts of the grid world is assumed to be free of obstacle by the agent, until it is explored. Therefore, when we say an obstacle, we refer to the known part of that obstacle. The agent can only perform four actions in each step; moving to a free neighbor cell in north, south, east or west direction. The prey has unlimited perception and knows all the grid world and the location of the predator all the time. The search continues until the predator reaches the prey.

3.2 MTES Algorithm

MTES makes use of a heuristic, Real-Time Target Evaluation (RTTE-h), which analyzes obstacles and proposes a moving direction that avoids these obstacles and leads to the target through shorter paths. To do this, RTTE-h geometrically analyzes the obstacles nearby, tries to estimate the lengths of paths around the obstacles to reach the target, and proposes a moving direction. RTTE-h 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). MTES repeats the steps in Algorithm 1 until reaching the target or detecting that the target is inaccessible.

In the first step, MTES calls RTTE-h heuristic function, which returns a moving direction and the utilities of neighbor cells according to that proposed direction. Next, MTES selects one of the neighbor cells on open directions with the minimum *visit count*, which stores the number of visits to the cell. If there exists more than one cell having the minimum *visit count*, the one with the maximum utility is selected. If utilities are also the same, then one of them is selected randomly. After the move is performed, the *visit count* of the previous cell is incremented and the cell is inserted into the *history*. The set of previously visited cells forms the *history* of the agent. History cells are treated as obstacles. Therefore, if the agent discovers a new obstacle during the exploration and realizes that the target became inaccessible due to history cells, the agent clears the history to be able to backtrack.

Algorithm 1 An Iteration of MTES Algorithm

```
1: Call RTTE-h to compute the proposed direction and the utilities of neighbor cells.
2: if a direction is proposed by RTTE-h 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
```

In moving target search problem, the target may sometimes pass through the cells the agent previously walked through. In such a case, there is a risk that the history blocks the agent to reach the target since history cells are assumed to be obstacles and may close some gateways required to return back. If this situation occurs at some point, the agent will surely be able to detect this at the end, and clear the history, opening all the closed gateways. Therefore, the algorithm is capable of searching moving targets without any additions. As a matter of fact, the only drawback of the history is not the possibility that it can block the way to the target entirely, but it can sometimes prevent the agent to reach the target through shorter paths by just closing some of the shortcuts. To reduce the performance problems of this side effect, the following procedure is applied. Assuming that (x_1, y_1) and (x_2, y_2) are the previous and newly observed locations of the target, respectively, and R is the set of cells the target could have visited in going from (x_1, y_1) to (x_2, y_2), the algorithm clears the history along with visit counts when any cell in set R appears in history or has non-zero visit count. In the algorithm, R can be determined in several ways depending on the required accuracy. The smallest set has to contain at least the newly observed location of the target, (x_2, y_2). One can choose to ignore some of the set members and only use (x_2, y_2) to keep the algorithm simple, or one may compute a more accurate set, which has the cells fall into the ellipse whose foci are (x_1, y_1) and (x_2, y_2), and the sum of the radii from the foci to a point on the ellipse is constant m, where m is the maximum number of moves the target could have made in going from (x_1, y_1) to (x_2, y_2).

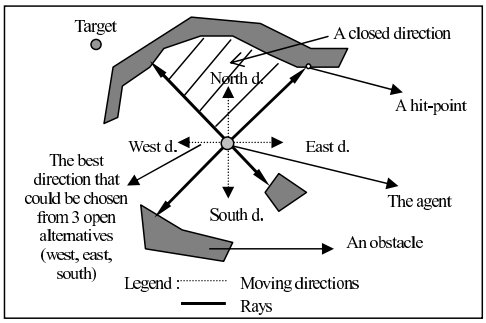

Fig. 1. Sending rays to split moving directions

The RTTE-h heuristic method in Algorithm 2 propagates four diagonal rays away from the agent location (line 2 in Algorithm 2) to split north, south, east and west moving directions as shown in Fig 1. The rays move outwards from the agent until they hit an obstacle or maximum ray distance is achieved. 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, the target is unreachable from the current location. To detect closed

regions, the boundary of the obstacle is extracted (line 4) and analyzed (line 5). Next, the obstacle border is re-traced from both left and right sides to determine geometric features of the obstacle (line 6). These features are evaluated and a moving direction to avoid the obstacle is identified (line 7). After all the obstacles are evaluated, results are merged to propose a final moving direction (line 9).

Algorithm 2 RTTE-h Heuristic

1: Mark all the moving directions as open.
2: Propagate four diagonal rays.
3: **for** each ray hitting an obstacle **do**
4: Extract the border of the obstacle by tracing the edges from left side until making a complete tour around the obstacle.
5: Detect closed directions.
6: Extract geometric features of the obstacle.
7: Evaluate the results and determine a direction to avoid the obstacle.
8: **end for**
9: Merge individual results, propose a direction to move, and compute utilities of neighbor cells.

The obstacle border extraction and closed direction detection phases of the algorithm use the same methods presented in [8, 9]. The geometric features of an obstacle are determined in two phases: left analysis and right analysis (line 6 in Algorithm 2). In left/right analysis, the known border of the obstacle is traced edge by edge towards the left/right sides starting from the hit point, making a complete tour around the obstacle border. During this process, several geometric features of the obstacle are extracted, which are described in Definitions 1 to 8 and illustrated in Fig. 2.

Definition 1 (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.*

Definition 2 (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).*

Definition 3 (Inside of left). *True if the target is inside the polygon whose vertices starts at agent's location, jumps to outer left most point, follows the border of the obstacle to the right and ends at the hit point of the ray.*

Definition 4 (Inside of inner left). *True if the target is inside the polygon that starts at agent's location, jumps to the inner left most point, follows the border of the obstacle to the right and ends at the hit point of the ray.*

Definition 5 (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.*

Definition 6 (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.*

Definition 7 (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:

Definition 8 (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. 2).*

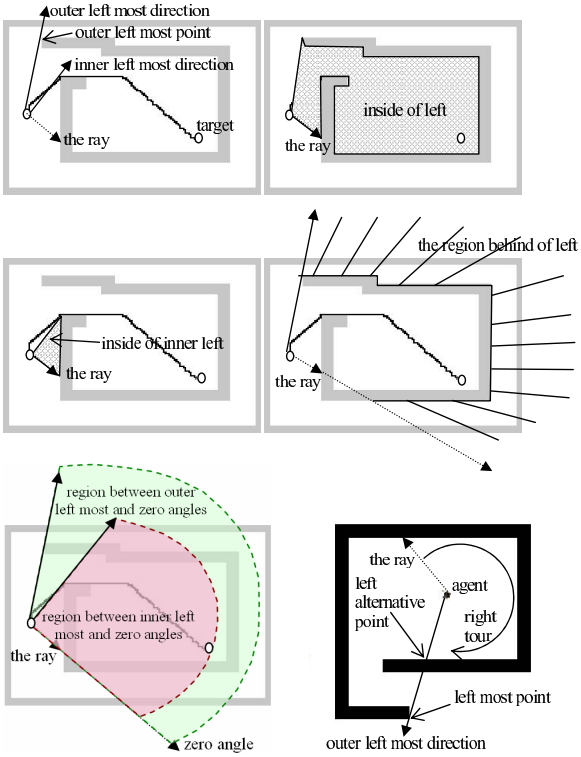

Fig. 2. Geometric features of obstacles

In obstacle evaluation step (line 7 in Algorithm 2), if an obstacle blocks the line of sight from the agent to the target, we determine a moving direction that avoids the obstacle meanwhile reaching the target through a shorter path. The method is given in Algorithm 3, which requires the path length estimations given in Definitions 9 to 11.

Definition 9 (d_{left}). *The approximated length of the path which starts from the agent location, jumps to the outer left most point, and then follows the path determined by Algorithm 4 (see Fig. 3).*

Definition 10 ($d_{left.alter}$). *The approximated 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, and finally follows the path determined by Algorithm 4 (see Fig. 3).*

Definition 11 ($d_{left.inner}$). *The approximated length of the path passing through the agent location, the inner left most point, and the target (see Fig. 3).*

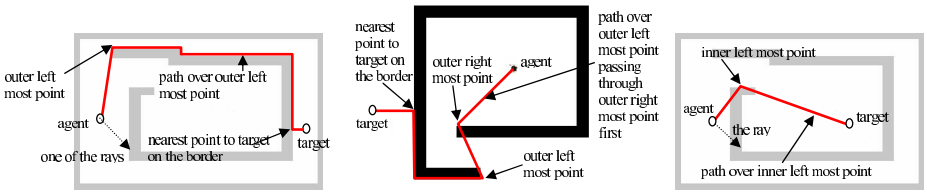

Fig. 3. Examples of d_{left} (left), $d_{left.alter}$ (middle) and $d_{left.inner}$ (right)

Algorithm 4 is internally used in computations of d_{left} and $d_{left.alter}$, and the sub-function *isoutwardsfacing* is called for detecting if a border segment, whose both ends touch the line passing through the *outer left most point* and the target point, is outwards facing (see Fig. 4). 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 the merging phase (line 9 in Algorithm 2), the evaluation results for all obstacles are used to determine a single moving direction. This proposed direction will be passed to MTES algorithm (Algorithm 1) for final decision. The details of the merging algorithm can be found in [9].

The complexity of MTES is $O(w.h)$ per step, where w is the width and h is the height of the grid world. Since increasing the grid size decreases the efficiency, a **search depth (d)** can be introduced in order to limit the worst case complexity. A search depth is a rectangular area of size $(2d+1)$x$(2d+1)$ centered at agent location, which makes the algorithm treat the cells beyond the rectangle as non-obstacle. With this limitation, the complexity becomes $O(d^2)$.

4 PERFORMANCE ANALYSIS

In this section, we present our experimental results on MTS-c, MTS-d, MTES and A*. As being an off-line algorithm, we executed A* in each step from scratch. For the test runs, we used 9 randomly generated sample grids of size 150x150. Six of them were the *maze* grids, and three of them were the *U-type* grids (see Fig. 5). For each grid, we produced 15 different randomly generated predator-prey location pairs, and made all the algorithms use the same pairs for fairness. Our tests are performed with 10, 20, 40 and *infinite* vision ranges and search depths. To test the algorithms, we developed a deliberative off-line prey algorithm (Prey-A*), which is powerful but not very efficient. To prevent the side effects caused

Algorithm 3 Evaluation Phase

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

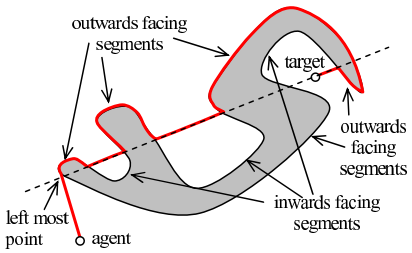

Fig. 4. Exemplified path length estimation, and outwards/inwards facing segments

Algorithm 4 Path length estimation

Require: t : target point
Require: s : outer left most point
Require: n : the nearest point to the target
Require: $+$: next border point (left of)
Require: $-$: previous border point (right of)
Require: $insert(p)$: inserts a point to the estimated path
Require: $clasify(p_1, p_2, p_3)$: if the edge formed by the points p_1, p_2, p_3 does a left turn then returns true, else returns false
Require: $isoutwardsfacing(side, p_1, p_2)$: see Algorithm 5

```
1: let prev = s
2: let prevleft = true
3: insert(s)
4: for each border point v between s+ and n do
5:    if v = n then
6:       if isoutwardsfacing(prevleft, prev, t) then
7:          insert(all border points between prev+ and v)
8:       end if
9:       insert(t)
10:      return length of estimated path
11:   end if
12:   let vleft = not clasify(s, t, v)
13:   if prevleft ≠ vleft then
14:      let z = intersection point of lines (s, t) and (v−, v)
15:      if not isoutwardsfacing(prevleft, prev, z) and z is between prev and t then
16:         insert(t)
17:         return length of estimated path
18:      end if
19:      if isoutwardsfacing(prevleft, prev, z) then
20:         insert(all border points between prev+ and v)
21:      else
22:         insert(v)
23:      end if
24:      let prev = v
25:      let prevleft = vleft
26:   end if
27: end for
```

Algorithm 5 The function $isoutwardsfacing(side, p_1, p_2)$

Require: t : target point
Require: s : outer left most point
Require: $len(n_1, n_2)$: returns distance between points n_1 and n_2
Require: $positive(m)$: if $len(m, t) \leq len(s, t)$ or $len(m, t) \leq len(s, m)$ then returns true, else returns false
Require: $slen(m)$: if $positive(m)$ then returns $+len(s, m)$ else returns $-len(s, m)$

```
1: if (side and slen(p1) < slen(p2)) or (not side and slen(p1) > slen(p2)) then
2:    return true
3: else
4:    return false
5: end if
```

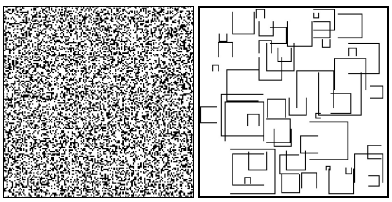

Fig. 5. A maze grid (left), a U-type grid (right)

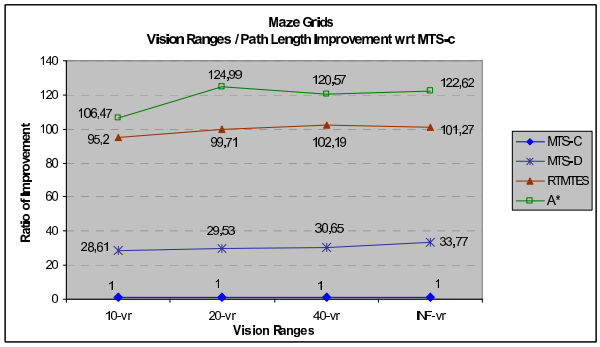

Fig. 6. Average of path length results of maze grids for increasing vision ranges

by the efficiency difference, the predator and the prey are executed alternately in performance tests. We also assumed that the prey is slower than the predator, and skips 1 move after each 7 moves.

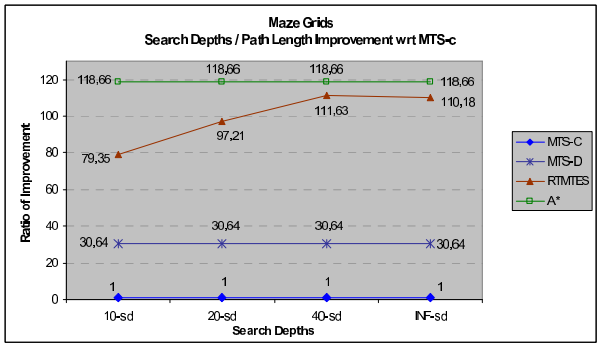

Fig. 7. Average of path length results of maze grids for increasing search depths

With respect to vision ranges and search depths, the averages of path lengths on maze grids are given in Figures 6 and 7, and the averages of path lengths on U-type grids are given in Figures 8 and 9, respectively. In the charts, the horizontal axis is either the vision range or the search depth, and the vertical axis contains the path length of MTS-c divided by that of the compared algorithm. The results showed that MTES performs significantly better than MTS-c and MTS-d even with small search depths, and usually offers near optimal solutions that are almost as good as the ones produced by A*. Especially in U-type grids, MTES mostly outperforms A*. When we examined this interesting result in details, we observed that they behave very differently in sparse parts of the grid. MTES prefers performing diagonally shaped manoeuvres for approaching targets located in diagonal directions, whereas A* prefers performing L-shaped

manoeuvres in such cases. Since the agents are only permitted to move in horizontal and vertical directions, these two manoeuvre patterns have equal path distances to a static target, but for a moving target, the strategy difference significantly affects the behavior of the prey in U-type grids and sometimes makes A* worse than MTES.

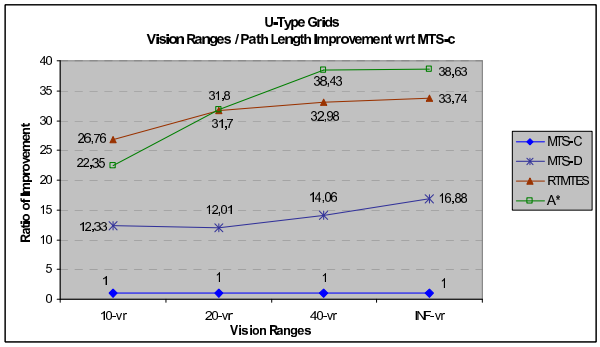

Fig. 8. Average of path length results of U-type grids for increasing vision ranges

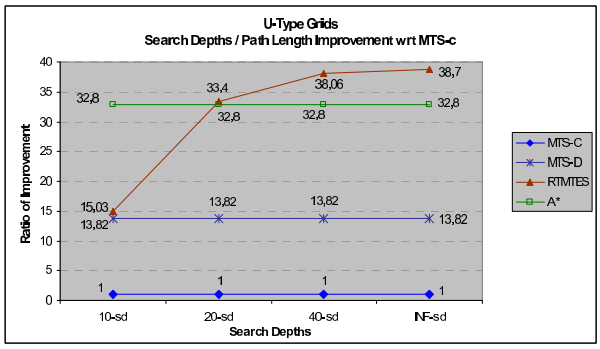

Fig. 9. Average of path length results of U-type grids for increasing search depths

We also examined the step execution times of the algorithms running on an AMD Athlon 2500+ computer. In Table 1, the worst case and the average number of moves executed per second are shown. The rows are for the compared algorithms and the columns are for the search depths. We see that MTS-c and MTS-d have low and almost constant step execution times whereas the efficiency of MTES is tied to the search depth and obstacle ratio, and hence the appropriate depth should be chosen according to the required efficiency. A* is the worst as expected.

Table 1. Worst case and average number of moves/second for increasing search depths

Maze grids

Depth	10-sd	20-sd	40-sd	INF-sd
MTS-c	1063/2676	1063/2676	1063/2676	1063/2676
MTS-d	937/2412	937/2412	937/2412	937/2412
MTES	531/1101	212/692	82/413	23/283
A*	20/189	20/189	20/168	20/189

U-type grids

Depth	10-sd	20-sd	40-sd	INF-sd
MTS-c	1063/2855	1063/2855	1063/2855	1063/2855
MTS-d	1062/2498	1062/2498	1062/2498	1062/2498
MTES	793/1257	400/747	133/348	57/233
A*	8/104	8/104	8/104	8/104

5 CONCLUSION

In this paper, we have examined the problem of pursuing a moving target in grid worlds, introduced a moving target search algorithm, MTES, and presented the comparison results of MTS-c, MTS-d, MTES and A* against a moving target controlled by Prey-A*. With respect to path lengths, the experiments showed that MTES performs significantly ahead of MTS-c and MTS-d, and competes with A*. In the test runs, we have also observed that the two MTS versions are significantly different from each other. Although, MTS-d performs acceptably good, MTS-c almost never offers good solutions. In terms of step execution times, we observed that MTS-c and MTS-d are the most efficient algorithms, and almost spend constant time in each move. But their solution path lengths are usually unacceptably long. MTES follows MTS, and its efficiency is inversely proportional to the increase in obstacle density. Finally, A* is always the worst.

References

1. M. Goldenberg, A. Kovarsky, X. Wu, and J. Schaeffer. Multiple agents moving target search. *Int'l Joint Conf. on Artificial Intelligence, IJCAI*, pages 1536–1538, 2003.
2. T. Ishida and R. Korf. Moving target search: A real-time search for changing goals. *IEEE Trans Pattern Analysis and Machine Intelligence*, 17(6):97–109, 1995.
3. S. Koenig and M. Likhachev. Real-time adaptive a*. *5th Int'l Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 281–288, 2006.
4. S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search*. *6th Int'l Joint Conf. on Autonomous Agents and Multiagent Systems*, 2007.
5. R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.
6. S. Russell and P. Norving. *Artificial Intelligence: a modern approach.* Prentice Hall, Inc., 1995.
7. A. Stentz. The focussed D* algorithm for real-time replanning. *In Proceedings of the Int'l Joint Conference on Artificial Intelligence*, 1995.
8. C. Undeger and F. Polat. Real-time edge follow: A real-time path search approach. *IEEE Transaction on Systems, Man and Cybernetics, Part C*, 37(5):860–872, 2007.
9. C. Undeger and F. Polat. Rttes: Real-time search in dynamic environments. *Applied Intelligence*, 27:113–129, 2007.