Auton Agent Multi-Agent Syst (2010) 21:69–107
DOI 10.1007/s10458-009-9102-0

Multi-agent real-time pursuit

Cagatay Undeger · Faruk Polat

Published online: 2 August 2009
Springer Science+Business Media, LLC 2009

Abstract In this paper, we address the problem of multi-agent pursuit in dynamic and partially observable environments, modeled as grid worlds; and present an algorithm called Multi-Agent Real-Time Pursuit (MAPS) for multiple predators to capture a moving prey cooperatively. MAPS introduces two new coordination strategies namely Blocking Escape Directions and Using Alternative Proposals, which help the predators waylay the possible escape directions of the prey in coordination. We compared our coordination strategies with the uncoordinated one against a prey controlled by Prey A*, and observed an impressive reduction in the number of moves to catch the prey.

Keywords Real-time pursuit · Multi-agent search · Real-time search · Path planning

1 Introduction

Pursuing a moving target is one of the most challenging problems in areas such as robotics and computer games. The first challenge here is to successfully plan a dynamic path towards a changing goal by a single agent. 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. There are only a few number of algorithms capable of pursuing a moving target in environments with obstacles, two of which are moving target search (MTS) [1] and moving target evaluation search (MTES) [2,3]. The second challenge in that area raises when the number of predators involved in the search is increased. In this case, the single agent path search problem becomes a search against a moving prey with multiple coordinated agents called multi-agent pursuit,

C. Undeger · F. Polat (✉)
Department of Computer Engineering, Middle East Technical University, 06531 Ankara, Turkey
e-mail: polat@ceng.metu.edu.tr

C. Undeger
e-mail: cagatay@undeger.com

on which there is not much successful study done so far, especially against moving preys in environments with obstacles.

In this paper, we focus on developing a successful and practically usable multi-agent pursuit algorithm that can be applied to real-world problems in domains such as modeling and simulation, game programming and robotics; and propose Multi-Agent Real-Time Pursuit (MAPS), which is capable of pursuing a moving prey with multiple coordinated predators in partially observable grid worlds with obstacles, and employs two coordination strategies namely *blocking escape directions* (BES) and *using alternative proposals* (UAL). The first strategy is executed before the path search for determining the *blocking location*, which is an estimated point that the agent may possibly waylay the prey at. Then the *blocking location* is fed as an input to the path planner, which is MTES in our case, but can also be any other planner. And the latter strategy, which can only work with MTES, is performed after the path search for selecting the best estimated direction from the alternative moving directions that are proposed by the path planner.

We compared our coordinated pursuit algorithm with uncoordinated one against a moving prey guided by Prey-A*, and observed that the number of moves to catch the prey is significantly reduced by multiple agents in coordination.

The organization of the paper is as follows: The related work on path planning, multi-agent pursuit and preys are given in Sect. 2. In Sect. 3, MTES is described in summary. Section 4 introduces our new algorithm, MAPS, and Sect. 5 presents the performance analysis of MAPS. Finally, Sect. 6 is the conclusion.

2 Related work

2.1 Off-line search algorithms

In the context of navigation, 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 find the whole solution in advance before starting execution, and can be uninformed, which do not require the domain knowledge such as Dijkstra's algorithm [4], or can be informed, which uses domain knowledge such as A* [5,6], genetic algorithms [7,8], random trees [9–11], probabilistic roadmaps [12,13].

2.2 Incremental search algorithms

Off-line path planning algorithms are hard to use for large dynamic environments because of their time requirements. One solution is to make off-line algorithms to be incremental [14], which is a continual planning technique that make use of information from previous searches to find solutions to the problems potentially faster than are possible by solving the problems from scratch. D* [15,16], Focused D* [17], D* Lite [18–20] and MT-Adaptive A* [21] are some of the well-known optimal incremental heuristic search algorithms applied to path planning domain. These algorithms are efficient in most cases, but sometimes a small change in the environment may cause to re-plan almost a complete path from scratch, which requires polynomial time and does not meet real-time constraints. That's why these algorithms are usually considered as efficient off-line path planning algorithms.

2.3 Real-time search algorithms

2.3.1 Static target algorithms

Due to the efficiency problems of off-line techniques, a number of on-line approaches that determine only one step ahead towards the goal in real-time are proposed. Tangent-Bug [22] is one of the former heuristic search algorithms. It is based on the Bug algorithm [23], and uses vision information to reach the target. It constructs a local tangent graph (LTG), a limited visibility graph, in each step only considering the obstacles in the visible set. Then, the agent either moves to the locally optimal direction on the current LTG or moves along the obstacle borders depending on the conditions.

Learning Real-Time A* (LRTA*), introduced by Korf [24], is another former generic heuristic search algorithm, which is applicable to real-time path search for fixed goals. LRTA* builds and updates a table containing admissible heuristic estimates of the distance from each state in the problem space to the fixed goal state, which are learned in exploration time. In the early runs, the algorithm does not guarantee optimality, but when the heuristic table is converged, the solutions generated become optimal. Although LRTA* is convergent and optimal, the algorithm is able to find poor solutions in the first run. To solve the problem, Korf also proposed a variation of LRTA*, called real-time A* (RTA*) [24], which gives better performance in the first run, but is lack of learning optimal table values. If you have only one chance to reach the goal, RTA* is surely a better choice. Since we have employed RTA* in our blocking location validation phase, which will be described later on, we briefly present RTA* in Algorithm 1.

Algorithm 1 An iteration of RTA* Algorithm [24]

```
1: Let $x$ be the current state of the problem solver
2: 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 $x'$
3: Move to a neighbor with the minimum $f(x')$ value. Ties are broken randomly
4: Update the value of $h(x)$ to the second best $f(x')$ value
```

Koenig proposed a new version of LRTA* that uses look-ahead depth more effectively to examine the local search space [25]. In each planning episode, the algorithm performs an A* search from the current state towards the goal state until either the goal state is about to be expanded or the number of states expanded reaches the look-ahead depth. Next, the heuristic values of expanded states are updated using Dijkstra's algorithm, and the agent follows the path minimizing the heuristic values. Most recently, Koenig and Likhachev proposed a second version of this LRTA* variation called real-time adaptive A* (RTAA*) [26]. For the sake of efficiency and simplicity, RTAA* replaces Dijkstra's algorithm with another one, which updates the heuristic values of expanded states more efficiently, but with less informed values. Some other versions of LRTA* could be found in [27–33].

In the literature, there are also some probabilistic on-line search algorithms based on genetic algorithms [34], random trees [35] and probabilistic roadmaps [36] since a significant portion of the previous search data generated by these algorithms could be still valid after an environmental change, and can be used in real-time for deciding on the next move.

Recently, two real-time search algorithms, real-time edge follow (RTEF) [37,38] and real-time target evaluation search (RTTES) [39,40] have been proposed for partially observable environments. Although these algorithms were developed for static targets, they were having the potential to handle moving targets with little modification. This potential inspired the moving target version of this algorithm called moving target evaluation search (MTES) [2,3].

2.3.2 Moving target algorithms

Since LRTA*, RTA* and their variations are all limited to work on fixed goals, Ishida and Korf proposed another algorithm called moving target search (MTS) [1]. Their algorithm is built on LRTA* and capable of pursuing a moving target. The algorithm maintains a table of heuristic values, representing 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 (i.e., y changes), the learning process has to start all over again that causes a performance bottleneck in heuristic depressions. Therefore, two MTS extensions called *Commitment to Goal* (MTS-c) and *Deliberation* (MTS-d) are proposed to improve the solution quality [1]. In order to use the learned table values more effectively, MTS-c ignores some of the target's moves while in a heuristic depression, and MTS-d performs an off-line search (deliberation) to update the heuristic values if the agent enters a heuristic depression.

Moving target evaluation search (MTES) proposed in [2,3] is able to detect the closed directions around the agent, and determine the estimated best direction that avoids the nearby obstacles leading to a static or moving target from a shorter path. It was reported that a significant improvement was observed over MTS-c and MTS-d.

2.4 Multi-agent real-time search algorithms

The algorithms described so far are only applicable to the problem of reaching a static or moving prey with single or multiple predators without coordination. Moving predators in coordination to pursuit a moving prey is a challenging problem, and most of the studies done so far only focus on multi-agent coordination in environments that are free of obstacles (non-hazy). The pursuit problem is originally proposed by Benda et al. [41], which was involving four coordinated predators pursuing a prey moving randomly. The environment was a non-hazy grid world, and the agents were allowed to move only in horizontal and vertical directions. According to the experiments, the authors concluded that an organization with one controlling predator and three communicating predators performs the best for solving the problem. Note that the coordination is centralized in this case.

2.4.1 Machine learning algorithms

In the literature, there are two common ways for studying pursuit problem, which are either hand-crafted coordination strategies or machine learning algorithms that let the predators learn themselves how to cooperate in order to catch the prey. For instance, in [42], a new reinforcement learning method, Two Level Reinforcement Learning with Communication (2LRL), is used to provide cooperative action selection in a multi-agent predator-prey environment. In 2LRL, the decision mechanism of the agents is divided into two hierarchical levels, in which the agents learn to select their target in the first level and to select the action directed to their target in the second level.

In [43], another reinforcement learning algorithm is employed to make four predators learn to pursue a moving prey in non-hazy environments. The authors assumed that the sensor and communication ranges of predators are limited, and hence a predator does not know the location of other predators and the prey all the time. Therefore they used Q-learning with partially observable Markov decision process and two kinds of predictions. The first prediction is the location of the other predators and the prey, and the second one is the possible moving direction of the prey in the next step. In their model, a state is defined with the velocity of the predator, the existence of any predators in communication range and/or observing the prey, and the relative coordinates/angles of the prey and the center of gravity of other predators. As a result, the authors observed that the predators can learn cooperative behavior and different roles, and the way in which the predators organize themselves depends on the initial locations of the predators, the style of the target movement, and the speed differences of the predators and the prey.

In [44], a recent variation of reinforcement learning algorithm known as TD-FALCON (A Temporal Difference Fusion Architecture for Learning, COgnition, and Navigation) is used for developing a cooperative strategy to surround a prey in all directions by four predators. TD-FALCON is an extension of predictive adaptive resonance theory (ART) networks for learning multi-model pattern mappings across multiple input channels. TD-FALCON makes use of a 3-channel architecture representing the current state, the set of available actions and the values of the feedbacks (rewards) received from the environment. The FALCON network is used to predict the value of performing each available actions in the current state. Then the values are processed to select an action, the action is executed, and the received reward (if any) is used to update FALCON network. The authors compared non-cooperative and cooperative (TD-FALCON) predator teams in a 16×16 sized non-hazy grid world, and observed about 15% success rate increase with the help of cooperation.

Another instance of learning algorithms for pursuit is introduced by Haynes and Sen in [45,46]. They employed strong typed genetic programming (STGP) to evolve pursuit algorithms represented as Lisp S-expressions for predators and preys moving in a 30×30 sized non-hazy toroidal grid world, which has left-right and bottom-top edges bend and connected to each other forming an infinite sized environment. They reported that good building blocks or subprograms are being identified during the evolution, and the performance of the best evolved program is comparable to a manually derived greedy strategy proposed by Korf [47].

Different from the work of Haynes and Sen, the generic algorithm is used by Yong and Miikkulainen [48] to evolve (and coevolve) neural network controllers, rather than program controllers. Co-evolution in this domain refers to maintaining and evolving individuals for taking different roles in a pursuit task. In the study, enforced subpopulations (ESP), a powerful and fast problem solver, is used to evolve three different strategies, which are a single centralized neural network, multiple distributed communicating neural networks, and multiple distributed non-communicating (Co-evolved) neural networks. Three predators were trained in a series of incrementally more challenging tasks obtained by starting with a static prey first, increasing the speed of prey in later iterations, and ending with the same speed as the predators. A 100×100 sized non-hazy toroidal grid world is used, in which the agents can move in horizontal and vertical directions. As a result of experiments, the authors reported that evolving several distinct autonomous, cooperating neural networks to control a team of predators is more efficient and robust than evolving a single centralized controller. This claim was contradictory to that reported by Benda et al. And very interestingly, they also observed that non-communicating distributed neural networks perform better than the communicating ones because of niching in coevolution, which obtains a set of simpler subtasks, and optimizes each team member separately and in parallel for one specialized subtask.

2.4.2 Hand-crafted algorithms

In [49], a hand-crafted coordination strategy that uses a game theoretic approach is suggested to solve the pursuit problem in non-hazy grid worlds, where the predators are coordinated implicitly by incorporating the global goal of a group of predators into their local interests using a payoff function. In fact, that study is not presenting a classical predator-prey problem since the predators does not always behave for the sake of the global objective. In that model, a predator should take into account the coalitions he may participate in along with their incomes, and decide the best coalition for him. For every move reducing/increasing the manhattan distance to the prey, the predator is given a positive/negative payment. Additionally, the amount of global utility is shared among the predators. Therefore, the predators should also consider the global objective, which is to block maximum number of prey's escape directions that are north, south, east and west. In this study, a predator is said to be blocking an escape direction d only if he moves towards the direction opposite to direction d (e.g., moving west towards the prey if the escape direction is to east), and d_p is smaller than d_a, where d_p is the distance of the predator from the prey along a line perpendicular to direction d, and d_a is the distance of predator from the prey along direction d. Our multi-agent pursuit algorithm is similar to this work in the sense that it is based on the strategy of blocking escape directions, but our definition of escape directions is significantly different.

Another hand-crafted coordination strategy is proposed by Kitamura et al. in [50]. Their coordination algorithm is build on a multi-agent version of RTA* called multi-agent real-time A* (MARTA*) [51], which can work in hazy environments, but is only for static goals. Kitamura et al. introduced two organizational strategies to MARTA* namely *repulsion* and *attraction*, where the repulsion strengths the discovering effect by scattering agents in a wider search space, and in contrast, the attraction strengths the learning effect by making agents update estimated costs in a smaller search space more actively. They performed their experiments in 120×120 sized maze grids with random obstacles with a ratio of 40%, and also in 15-puzzles. As a result, the repulsion showed a good performance with mazes in which deep heuristic depressions are spotted, and the attraction showed a good performance with 15-puzzles in which shallow depressions are distributed all over.

In [52], a multi-agent pursuit algorithm is proposed for fully known grid worlds with randomly placed obstacles. The authors proposed an application domain called multiple agent moving target (MAMT), where the agents are permitted to see or communicate with other agents only if they are in line of sight, and can move in horizontal or vertical directions simultaneously. Different from the previous algorithms described, the predators cannot see the prey all the time and need to explore a hazy environment. But, one missing point in their approach is that the predators only use coordination to search different parts of the environment when the prey is hiding, but cannot chase the prey in coordination when they see the prey.

Another similar problem studied in the literature is called the evasion, the goal of which is to detect all the unknown targets within a region in cooperation, and to guarantee that the region is completely searched and all the possible targets are detected in a finite time. In [53,54], evasion problem is examined, and cooperative search strategies (swarm flying patterns) are proposed for unmanned air vehicles (UAVs) against mobile and evasive targets. In these studies, UAVs are arranged into a flight configuration that optimizes UAVs integrated sensing capability while allowing quick reconfiguration of the topology in the event that some of the UAVs become unavailable (e.g., killed).

The last hand-crafted coordination strategy we will examine is proposed by Kota et al. [55]. Their coordination approach is based on deflecting the predators from the centroid of the group meanwhile attracting themselves towards the prey. Although the environment they used is free of obstacle, they introduced an artificial haze to their environment by making the predators lose track of the prey location from time to time. In such cases, the predators use the last observed location of the prey for deciding the next move. Their algorithm calculates the direction of the next move using the weighted sum of the attraction vector towards the prey and the repulsion vector away from the centroid. As a results, the authors stated that their algorithm shows moderate performance, and hence they are studying for better strategies.

2.5 Prey algorithms

When we look at the prey algorithms, we usually see hybrid techniques mixing a number of reactive strategies. For instance, the strategy, moving randomly in any possible direction not blocked by a predator, is commonly used in pursuit problems [49,43,1,52]. In the study of Ishida and Korf [1], the avoid strategy of the prey is developed as to move towards a position as far from the predators as possible using MTS algorithm. In [43,55], the prey is let to escape from the predators along a straight line or a circle. In [43], additionally a third method is also used as moving in the opposite direction of the predator if the prey sees only one predator, and otherwise moving in the direction that bisects the largest angle of its field of view in which there are no predators. In [52], a weighted combination of four sub-strategies: moving towards a direction that maximizes the distance from the predator's location, moving towards a direction that maximizes the mobility by preferring a move that leads to more move choices, moving towards a position that is not in line of sight of predators and moving randomly, are used. There are also some studies focussing on evolving behavioral strategies [45,46].

Since these reactive algorithms are not good enough to challenge successful predator algorithms, an off-line strategy called Prey A*, which is slow but more powerful, is developed [2,3]. In this paper, we employed Prey A* in our experiments to challenge our pursuit algorithms. Prey-A* generates two grids, $costs_{predator}$ and $costs_{prey}$, whose sizes are the same as the size of the environment, and have one to one mapping to the cells of the grid world. Each cell of the $costs_{predator}$ contains the length of the optimal path from the nearest predator to the cell, and similarly, each cell of the $costs_{prey}$ stores the length of the optimal path from the prey to the cell. The objective is to find a cell such that the number of moves from the nearest predator to the cell (the cost in $costs_{predator}$) is maximized, and the prey will not be caught by the predators during the travel to the cell through the optimal path. This is checked by ensuring that each cell on the optimal path satisfies $costs_{predator}[cell] - \alpha.costs_{prey}[cell] > 0$, where α is computed by the formula $speed_{predator}/speed_{prey}$. In order to find the best cell, the algorithm examines each non-obstacle cell within a limited search window centered at the prey location, and moves one step towards the selected one.

3 MTES algorithm

In this section, we summarize *real-time moving target evaluation search (MTES)* [2,3], on which one of our coordination algorithms, using alternative proposals (UAL), is built. MTES is a real-time single-agent path search (predator) algorithm for moving targets that works on partially observable planar grid worlds, where any grid cell can either be free or obstacle. Predator has no control over the movements of the prey and is able to move north, south, east or west direction in each step, and uses Euclidian distance metric for heuristic estimations.

It is assumed that the predator knows the location of the prey all the time, but perceives the obstacles around him within a limited region. The unknown part of the grid world is assumed to be free of obstacle by the agent, until it is explored. The agent maintains a tentative map, which holds the known part of the grid world, and updates it as he explores the environment. Therefore, when an obstacle is mentioned, that refers to the known part of that obstacle.

MTES repeats the steps in Algorithm 2 until reaching the target or detecting that the target is inaccessible, and MTES makes use of a heuristic, real-time target evaluation (RTTE-h) (see Algorithm 3), 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).

Algorithm 2 An iteration of MTES Algorithm [2,3]

```
Require: $s$: current cell the agent is on
1: Let $d$ be proposed direction to move on cell $s$, which is determined by RTTE-h (Algorithm 3)
2: if $d$ exists then
3:    Let $n$ be the set of neighbor cells of $s$ with minimum visit count
4:    Let $c$ be the cell in $n$ with maximum utility. Ties are broken randomly
5:    Move to cell $c$
6:    Increment the visit count of cell $s$ by one
7:    Insert cell $s$ into the history
8: else
9:    if History is not empty then
10:       Clear all the History
11:       Jump to 1
12:   else
13:       Destination is unreachable, stop the search with failure
14:   end if
15: end if
```

Algorithm 3 RTTE-h heuristic [2,3]

```
Require: $s$: current cell the agent is on
Ensure: Final proposed direction to move (if any) on cell $s$
Ensure: Utilities of neighbor cells of $s$
1: Mark all the moving directions of cell $s$ as open (non-closed)
2: Propagate four diagonal rays from cell $s$
3: for each ray $r$ hitting an obstacle do
4:    Let $o$ be the obstacle hit by ray $r$
5:    Extract border $b$ of obstacle $o$
6:    Detect closed directions of cell $s$ using border $b$
7:    Analyze border $b$ to extract geometric features of obstacle $o$
8:    Evaluate result and determine best direction to avoid obstacle $o$ (Algorithm 4)
9: end for
10: Merge individual results of entire rays to determine a proposal (Algorithm 5)
```

In the first step, MTES calls RTTE-h heuristic function, which returns a moving direction (proposed direction) and the utilities of neighbor cells (see Algorithm 5) according to that proposed direction. Next, MTES selects one of the neighbor cells on non-closed directions

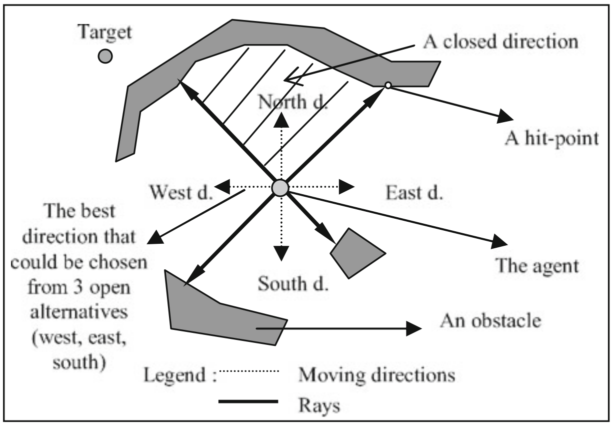

Target
A closed direction
North d.
A hit-point
West d.
East d.
The best direction that could be chosen from 3 open alternatives (west, east, south)
South d.
The agent
An obstacle
Legend : Moving directions
Rays

Fig. 1 Sending rays to split moving directions [38]

(described later on in this section), 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 by one 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.

To better handle moving targets, the following procedure is applied in MTES. 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).

Real-Time Target Evaluation heuristic (RTTE-h) method given in Algorithm 3 propagates four diagonal virtual rays (propagated in the mind of the agent) away from the agent location (line 2 in Algorithm 3) 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 and identify closed directions (see Fig. 1), the boundary of the obstacle is extracted by starting from the hit-point and tracing the edges towards the left side until making a complete tour around the obstacle (line 5) and analyzed (line 6). Next, the obstacle border is re-traced from both left and right sides to determine geometric features of the obstacle (line 7). These features are evaluated and a moving direction to avoid the obstacle is identified (line 8). After all the obstacles are evaluated, the results are merged (line 10) in order to propose a final moving direction and to compute the utilities of neighbor cells of the agent location accordingly. Details of these steps are not given in this paper except analyzing an obstacle border (line 7), evaluating individual obstacle features (line 8), and merging entire results (line 10) since only these parts are related with the modifications we have made.

3.1 Analyzing an obstacle border

After extracting an obstacle border, the border information is represented as a polygonal area, which is stored as a list of vertices. Later on that information is analyzed in order to determine the geometric features of the obstacle (see [40] for details). Border analysis (line 7 in Algorithm 3) 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 in Definitions 1 to 7 (see Fig. 2 for illustrations):

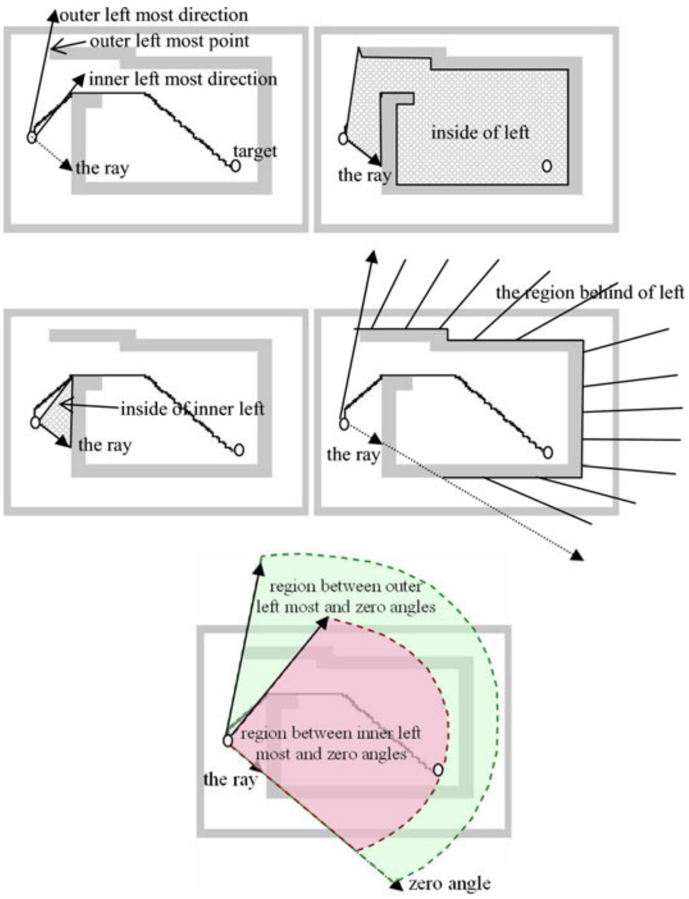

outer left most direction
outer left most point
inner left most direction
target
the ray
inside of left
the ray
inside of inner left
the ray
the region behind of left
the ray
region between outer left most and zero angles
region between inner left most and zero angles
the ray
zero angle

Fig. 2 Geometric features of an obstacle: Outer leftmost and inner leftmost 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*) [40]

Definition 1 (*Outer leftmost 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 leftmost direction*.

Definition 2 (*Inner leftmost 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 the *outer leftmost 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 leftmost 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 leftmost direction* in ccw order and the target is not inside of left.

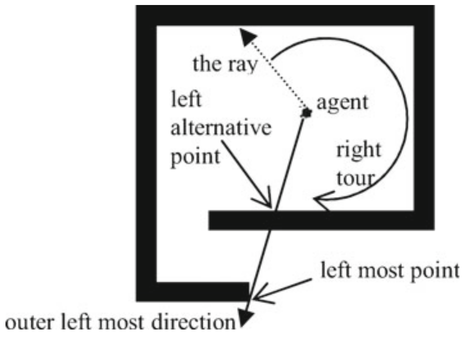
Definition 6 (*Outer-left-zero angle blocking*) *True* if target is in the region obtained by sweeping the angle from the ray direction to the *outer leftmost direction* in ccw order.

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 leftmost 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 given in Definition 8 is extracted (see Fig. 3 for illustration):

Definition 8 (*Left alternative point*) The last vertex in the *outer leftmost direction* encountered during the right tour until the *outer rightmost direction* is determined.

Fig. 3 Left alternative point [40]



3.2 Evaluating individual obstacle features

In individual obstacle evaluation step (line 8 in Algorithm 3), 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. The method is given in Algorithm 4 (see [40] for details and proof of correctness), which requires the path length estimations given in Definitions 9 to 11 (see [2,3] for details) in addition to the acquired geometric features of the obstacle.

Definition 9 (d_{left}) The approximated length of the path which starts from the agent location, jumps to the *outer leftmost point*, and then follows a part of the border of the obstacle from left side, and finally jumps to the target (Figs. 4, 5).

Definition 10 ($d_{left.alter}$) The approximated length of the path which starts from the agent location, jumps to the *outer rightmost point*, and then follows a part of the border of the obstacle from left side, and finally jumps to the target (see Fig. 6).

Definition 11 ($d_{left.inner}$) The approximated length of the path passing through the agent location, the *inner leftmost point*, and the target (see Fig. 7).

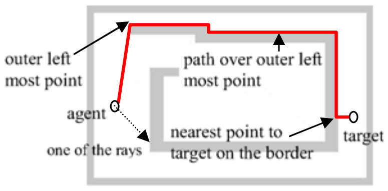

outer left most point
path over outer left most point
agent
one of the rays
nearest point to target on the border
target

Fig. 4 Exemplified d_{left} estimation [40]

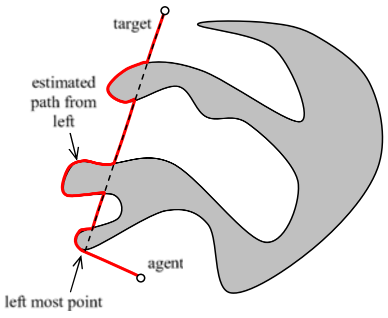


Fig. 5 A more complex d_{left} estimation [2,3]

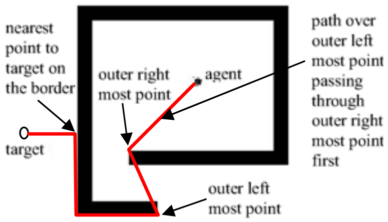

nearest point to target on the border
target
outer right most point
agent
path over outer left most point passing through outer right most point first
outer left most point

Fig. 6 Exemplified $d_{left.alter}$ estimation [40]

Algorithm 4 Individual obstacle evaluation step [40]

```
Require: $s$: current cell the agent is on
Require: $o$: evaluated obstacle
Require: $g$: geometric features of obstacle $o$ (Definitions 1 to 8)
Require: $p$: path length estimations of obstacle $o$ (Definitions 9 to 11)
Ensure: Proposed direction to move on cell $s$ avoiding obstacle $o$
Ensure: Estimated distance to target on cell $s$ avoiding obstacle $o$
1: if (behind of left and not inside of right) or (behind of right and not inside of left) then
2:   Case 1:
3:   if outer leftmost angle + outer rightmost angle $\geq 360$ then
4:     Case 1.1:
5:     if distance from agent to outer leftmost point $<$ distance from agent to left alternative point then
6:       Case 1.1.1: Let estimated distance be $min(d_{left}, d_{right.alter})$, and propose outer leftmost direction as moving direction
7:     else
8:       Case 1.1.2: Let estimated distance be $min(d_{left.alter}, d_{right})$, and propose outer rightmost direction as moving direction
9:     end if
10:  else
11:    Case 1.2:
12:    if $d_{left} < d_{right}$ then
13:      Case 1.2.1: Let estimated distance be $d_{left}$, and propose outer leftmost direction as moving direction
14:    else
15:      Case 1.2.2: Let estimated distance as $d_{right}$, and propose outer rightmost direction as moving direction
16:    end if
17:  end if
18:  Mark obstacle as blocking the target
19: else if behind of left then
20:  Case 2:
21:  if Target direction angle $\neq 0$ and outer-right-zero angle blocking then
22:    Case 2.1: Let estimated distance be $d_{left}$, and propose outer leftmost direction as moving direction
23:  else
24:    Case 2.2: Let estimated distance be $d_{right.inner}$, and propose inner rightmost direction as moving direction
25:  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: Let estimated distance be $d_{right}$, and propose outer rightmost direction as moving direction
31:  else
32:    Case 3.2: Let estimated distance be $d_{left.inner}$, and propose inner leftmost 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: Let estimated distance be $d_{left.inner}$, and propose inner leftmost 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: Let estimated distance be $d_{right.inner}$, and propose inner rightmost direction as moving direction
42:    Mark obstacle as blocking the target
43:  end if
44: end if
```

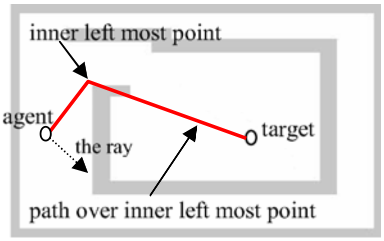


Fig. 7 Exemplified $d_{left.inner}$ estimation [40]

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 definitions). So, we have additional estimated target distances d_{right}, $d_{right.alter}$ and $d_{right.inner}$. The details of path estimation can be found in [2,3].

3.3 Merging entire results

In the result merging step (line 10 in Algorithm 3), 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 MTES algorithm (see Algorithm 2) for final decision. The merging algorithm is given in Algorithm 5 (see [40] for details).

The most critical step of the merging phase is to compute the moving direction to get around *the most constraining obstacle*, which is the obstacle that is marked as blocking the target on direct flying direction and which has the largest estimated path length (see Definitions 9 to 11). The reason why we determine the moving direction (line 6 in Algorithm 5) 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. The algorithm works even if the intervening obstacles are ignored but the following technique is employed 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 leftmost direction* outside ccw angle between ray s and pd_r, and has *outer rightmost direction* inside ccw angle between r and s, then the obstacle o_s blocks pd_r and proposed direction should be swept to *outer leftmost 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. 8. 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.

A complete sample illustrating the entire process of MTES can be seen in Fig. 9. In the sample, there exist three obstacles, two of which are blocking the prey (obstacle A and B). Assuming that all the obstacles are known by the agent, MTES first propagates 4 diagonal

Algorithm 5 Merging phase [40]

```
Require: s: current cell the agent is on
Require: t: current cell the target is on
Require: r: individual results of entire rays, which are determined in Algorithm 3
Ensure: Final proposed direction to move (if any) on cell s
Ensure: Utilities of neighbor cells of s
 1: if All the directions to neighbor cells of s are closed then
 2:     Return no moving direction and halt with failure
 3: end if
 4: Determine the most constraining obstacle m using results in r
 5: if m exists then
 6:     Let final proposed direction d be a direction that gets around obstacle m
 7: else
 8:     Let final proposed direction d be the direct flying direction from cell s to cell t
 9: end if
    {Compute utility of each neighbor cell of s}
10: for Each neighbor cell n of s do
11:     if Direction to cell n is closed then
12:         Let utility of cell n be zero
13:     else
14:         Let dif be the smallest angle between direction d and the direction of cell n
15:         Let utility of cell n be $(181 - dif)/181$
16:     end if
17: end for
```

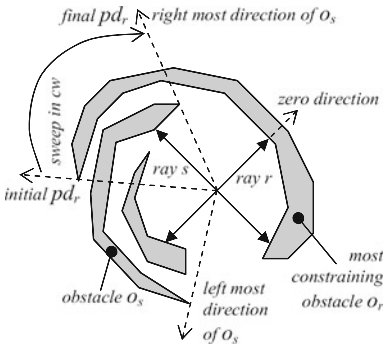

final pd_r
right most direction of o_s
sweep in cw
zero direction
ray s
ray r
initial pd_r
most constraining obstacle o_r
obstacle o_s
left most direction of o_s

Fig. 8 An example of avoiding the intervening obstacles [40]

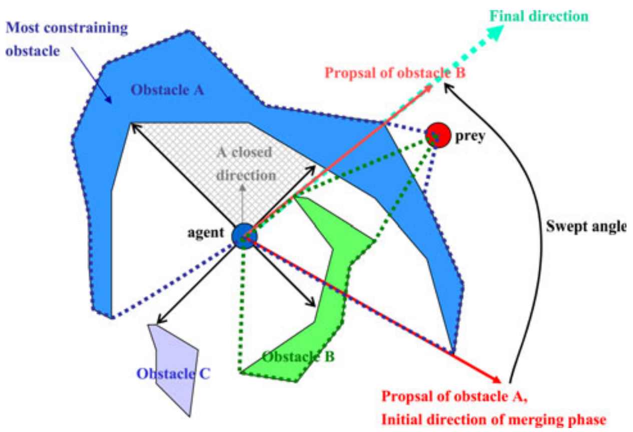


Fig. 9 Illustration of entire process of RTTE-h heuristic [2,3]

rays that hit obstacle A, B and C. Then border extraction, close direction detection, obstacle border analysis and individual obstacle feature evaluation phases are executed for each ray hitting an obstacle. In border extraction phases, borders of obstacles are determined by performing complete tours around the obstacles starting from the hit points of rays. During border extraction, the rays that share the same obstacles are also determined. Next, in closed direction detection phases, north moving direction is closed since north-west and north-east rays share the same obstacle (obstacle A), and the inner region bordered by these two rays does not contain the prey. Afterwards, obstacle border analysis phases are initiated, and geometric features of obstacles such as outer leftmost direction, left alternative point, are determined. In individual obstacle feature evaluation phases, estimated best and second best paths, their lengths and initial moving directions are determined for each obstacle. While evaluating an obstacle, all the other obstacles are ignored, and just the obstacle itself is taken into account. For obstacle A and B, there are two alternative paths around left and right sides of each obstacle, totally making four paths. There is no path determined for obstacle C since it does not block the prey. When all the phases for the entire rays are completed, result merging phase is performed. In that phase, obstacle A is chosen as the most constraining obstacle since minimum of its estimated path lengths is longer than minimum of estimated path lengths of obstacle B. Most constraining obstacle proposes south-east moving direction, but that direction is blocked by obstacle B. Therefore, the proposed direction is swept in counter clockwise angle until obstacle B does not intersect the direction any more. That is the final proposed direction, but it should be mapped into one of the four possible moving directions taking into account unclosed directions and their utilities. Finally, the east direction will probably be chosen to move (visit counts may change the result) since final proposed direction is closest to that direction.

3.4 Analysis of the algorithm

The worst case complexity of MTES is $O(w \cdot h)$ per step, where w is the width and h is the height of the grid. Here $w \cdot h$ represents the length of longest obstacle border that is possible in a grid world. The worst case environments are rarely possible in practice, and enlarging the area will not drop the performance very sharply most of the time since the average length of obstacle borders does not strictly depend on the grid size. For instance, in an urban area, the sizes of buildings are similar independent of how large the city is. But, note that such worse cases are usually possible in complex mazes.

Since increasing the grid size decreases the efficiency, a **search depth (d)** can be introduced in order to limit the worst case complexity of MTES. 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, the complexity of MTES becomes $O(d^2)$ (see [38] for details).

For the proof of correctness and completeness of the MTES algorithm please see [38,2,3].

4 MAPS algorithm

In this section, we present our pursuit algorithm called multi-agent real-time pursuit (MAPS). First, we start with the assumptions of our problem domain.

In our environment, there are multiple agents (predators) that aim to reach a moving target (prey) in coordination. The predator group and the prey are randomly located far from each other in non-obstacle grid cells. The predators are expected to reach the prey in coordination

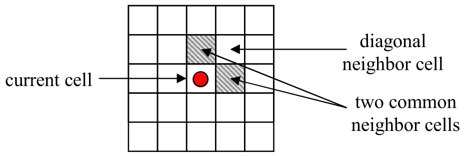


Fig. 10 Common neighbors of diagonal cell and current cell

as soon as possible avoiding the obstacles in real-time. The prey is escaping from the predators using Prey-A* [2,3], and its location is assumed to be known by the predators all the time. This assumption can be relaxed if a hidden prey search is integrated into the predator behaviors.

The predators perceive the obstacles around them within a square region centered at the agent location. The size of the square is $(2v + 1) \cdot (2v + 1)$, where v is the **vision range**. The unknown parts of the grid world is assumed to be free of obstacle by the agent, until it is explored. The term **infinite vision** is used to emphasize the setting where the predators has unlimited sensing capability and knows the entire grid world before the search starts. The prey has unlimited perception and knows all the grid world and the location of the predators all the time.

The predators and the prey can only perform nine actions in each step, which are staying still or moving to a non-obstacle neighbor cell in horizontal (east, west), vertical (north, south) or diagonal (north-east, north-west, south-east, south-west) directions. The effects of actions are all deterministic. In order to decide on moving to a diagonal neighbor cell in north-east, north-west, south-east or south-west direction, both the predators and the prey should make sure that the *common neighbors* (see Definition 12) of the diagonal neighbor cell and the current cell is also free (see Fig. 10 for illustration). We assume that a cell cannot be shared by more than one predator. Therefore the predators also should not move to a cell that is blocked by another predator. The size of the grid cells is assumed to be 1×1 unit, and the coordinates of the agents are stored in continuous space (i.e., in floating numbers). All the moves take 1 unit step for fairness. Therefore, some diagonal moves may not cause the cell coordinate of the agent to be changed if the step size is not long enough to reach the next cell.

Definition 12 (*Common neighbors*) Common neighbors of two cells (source cells) are defined as the cells that are neighboring both source cells at the same time.

The predators and the prey move sequentially in each iteration, and the first step is always taken by the prey. Next, the predators make their moves one by one considering the new location of the prey and the other predators. The prey is assumed to be caught when the prey and any of the predators are in the same cell.

MAPS offers two different coordination strategies: *blocking escape directions* (BES) and *using alternative proposals* (UAL). One may selectively enable one or both of these strategies in order to increase the pursuit performance. The first strategy, *blocking escape directions*, is executed before the path search, and moves the target coordinate of path planner from the prey's current location to the prey's possible future escape location in order to better waylay the prey. Therefore, the path planner will determine a path for reaching the possible escape location instead of the current location of the prey. This strategy is generic and can be integrated into any moving target search algorithm. The second strategy, *using alternative proposals*, is performed after the path search for selecting from the alternative moving directions proposed by the path search algorithm. Since the alternative moving directions and their estimated path lengths can be determined by our path search heuristic, advance real-time target evaluation (ARtte), the second strategy is not generic and only applicable to our path planner. MAPS given in Algorithm 6 is executed by all the predators independently in each step until one of the predators catch the prey. The algorithm first determines a location, called the *blocking location*, for the predator to move in order to waylay the prey considering all the other predators and the possible escape directions of the prey (line 1 in Algorithm 6). Then the *blocking location* (see Definition 13) is used as the target coordinate to be reached in the path search algorithm (lines 2 and 4).

Definition 13 (*Blocking location*) The intersection point, which the predator and the prey will possibly meet at the same time if they both insist on continuously moving to that point at full speed, and there is no obstacles on the way. The *blocking location* is computed based on the assumption that the prey will move to a fixed direction called the *escape direction* (see Definition 14), which passes through that point.

Definition 14 (*Escape direction*) A direction that the prey may move to in order to escape from the predators. *Escape directions* are chosen heuristically, and there is no guarantee that the prey will move that way.

Algorithm 6 An iteration of MAPS Algorithm

```
Require: $p$: this predator
Require: $s$: current cell predator $p$ is on
1: Let $(tx, ty)$ be blocking location for predator $p$ determined by BES (Algorithm 7)
2: Let $st$ be set of closed directions for cell $s$ determined by ARtte $(tx, ty)$ (Algorithm 9)
3: Let $pr$ be best and second best direction proposals determined by ARtte $(tx, ty)$ (Algorithm 9)
4: if $pr$ does not exist then
5:   if $History$ is not empty then
6:     Clear all the $history$.
7:     Jump to 3.
8:   else
9:     The prey is unreachable, stop search with failure.
10:  end if
11: end if
12: Let $dir$ be final proposed moving direction determined by UAL $(pr)$ (Algorithm 12)
13: Let $ut[s]$ be set of utilities for neighbor cells of $s$ determined using $st$ and $dir$ (Algorithm 13)
14: Let $n$ be set of neighbor cells with smallest visit count and non-zero utility in $ut[s]$
15: Let $m$ be the neighbor cell to move with the highest utility in $n$. Ties are broken randomly
16: if $m$ exists then
17:   Perform one step move to cell $m$
18:   if Cell $s$ is changed after move then
19:     Increment the visit count of previous cell $s$ by one
20:     Insert previous cell $s$ into the history
21:   end if
22: else
23:   Clear all the $history$ in order to search for an alternative way
24:   The way is temporarily blocked by another predator, stop search and wait for one iteration.
25: end if
```

For performing path search, a modified version of RTTE-h heuristic (see Algorithm 3), called Advance Real-Time Target Evaluation (ARtte), is used. ARtte analyzes the current state, and produces a set of proposals for the next move. There are three possible outcomes. ARtte may propose two alternative moving directions (the *best* and the *second best directions*), or propose one moving direction (the *best direction*), or may not propose anything at all if the target is unreachable. If any proposal is made by ARtte, the proposal is evaluated in order to select a final moving direction (line 12). According to the final moving direction, the utilities of the eight neighbor cells are computed next (line 13). Then, a move is selected considering the utilities of the neighbor cells (lines 14 and 15). Finally, the move (if there exists one) is performed (line 17) and the agent information is updated for the next step (lines 18–20).

In order to avoid infinite loops and re-visiting the same locations redundantly, MAPS uses *visit counts* and *history* together. 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 becomes inaccessible due to history cells, the agent clears the history to be able to backtrack. The algorithm maintains the number of visits, **visit count**, to the grid cells, and the agent moves to one of the neighbor cells with non-zero utility and minimum *visit count*. If there exists more than one cell satisfying the condition, the one with the maximum utility is selected. If they are also the same, then one of them is selected randomly. In situations where there is no cell having non-zero utility (the way may be temporarily blocked by another predator), the agent stays still for one step, and clears the *history* to be able to search alternative ways in the next step (lines 23 and 24).

If the agent selects a cell to move from the eight neighbors, he performs a one-step move towards the direction of that cell (line 17). If a neighbor cell in horizontal or vertical direction is selected, we move to that cell horizontally or vertically, and the step will directly cause the cell of the agent to be changed, but if a neighbor cell in diagonal direction is selected, we compute the direction from the agent location to the corner of the diagonal cell, and move to that direction, which may not sometimes cause the cell of the agent to be changed. Therefore, in addition to the cell coordinate of the predator, we also maintain its real coordinate in continuous space. In the following sections, some of the phases mentioned above are described in more details.

4.1 Determining the blocking location

In order to waylay the prey in its escape direction, a *blocking location* for the predator to move is determined using Algorithm 7. This phase is optional and only executed unless *escape direction blocking* is disabled (checked by line 3 in Algorithm 7). If it is disabled or there is only one predator, the current location of the prey is returned as the *blocking location* (line 4). To compute the *blocking location* of the predator, the algorithm first needs to determine n possible *escape directions* (lines 9), where n is the number of predators. The first *escape direction* is always chosen as the vector directed from the location of the prey to the location of the nearest predator to the prey. Other *escape directions* are distributed uniformly based on the first *escape direction*, and computed as vectors, which are originated from the prey location, and angle differences of which from its two neighbor vectors are all equal (see Fig. 11 for illustration). Thus, we allocate 360 degrees to escape directions such that we get equal angle differences. After selecting the *escape directions*, the algorithm assigns *escape directions* to predators optimally (see Fig. 12) such that the total distance from predators to *blocking locations* waylaying their assigned *escape directions* is minimized (line 10). In this assignment procedure, the first *escape direction* should always be matched to the nearest predator to the prey and the nearest predator should always aim to reach the prey location as its *blocking location*. When the assignment is completed, the initial *blocking location* of the predator is determined based on the *escape direction* assigned to him (lines 11–12). And as the final step, the computed *blocking location* is validated and corrected if required (lines 13–16). The validation is done by checking if there exists a path from the prey to the *blocking* *location* or not. If no path is determined, then the nearest location to the *blocking location* is selected (line 16). To find this location, RTA* algorithm (see Algorithm 1) is executed (line 15) starting from the prey location until the *blocking location* is reached or the number of moves executed by RTA* exceeds a threshold computed proportional to the Manhattan distance from the prey to the *blocking location* (lines 13 and 14).

Algorithm 7 Determining the *blocking location*

```
Require: $p$: this predator
Require: $s$: current cell predator $p$ is on
Require: $t$: current cell the prey is on
Require: $n$: number of predators
Ensure: Blocking location
 1: Let $(hx, hy)$ be the coordinate of cell $p$
 2: Let $(tx, ty)$ be the coordinate of cell $t$
 3: if n=1 or BES is disabled then
 4:   Let blocking location $(bx, by)$ be $(tx, ty)$
 5: else
 6:   if predator $p$ is the nearest predator to $(tx, ty)$ then
 7:     Let blocking location $(bx, by)$ be $(tx, ty)$
 8:   else
 9:     Determine $e$ as the set of $n$ escape directions
10:     Determine assignment $m$ that maps $n$ predators to $e$ optimally
11:     Let $esc$ be the escape direction assigned to predator $e$ in $m$
12:     Determine initial blocking location $(ix, iy)$ using $esc$ (Algorithm 8)
13:     Let $\beta$ be the path search limit multiplier (e.g., 2 times)
14:     Let $d$ be $\beta.(abs(hx - ix) + abs(hy - iy))$
15:     Run RTA* (Algorithm 1) from $(hx, hy)$ to $(ix, iy)$ until $(ix, iy)$ is reached or # steps $> d$
16:     Let blocking location $(bx, by)$ be nearest point to $(ix, iy)$ determined during RTA* search
17:   end if
18: end if
```

In order to compute *blocking location* given the *escape direction* (see Algorithm 8), we assume that the speeds of the prey and the predator are known, but we have no constraint on their speeds. The prey may be slower or faster than the predators. In case the prey is faster, there is still hope for the predators to catch the prey since the environment is complicated, and at the end the prey may have to wait somewhere or change direction for avoiding obstacles and dead-ends. The *blocking location* is chosen as the coordinate on the way of the prey's

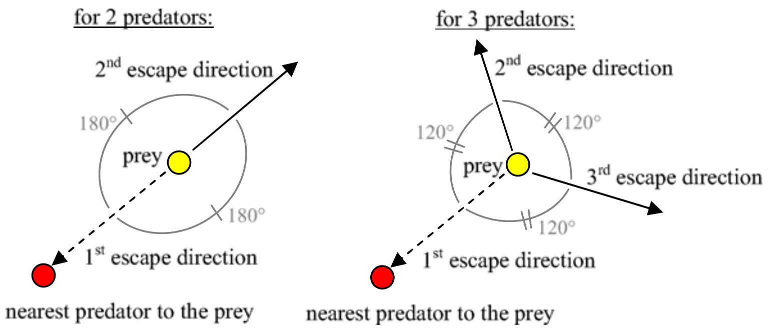


Fig. 11 Determining the escape directions

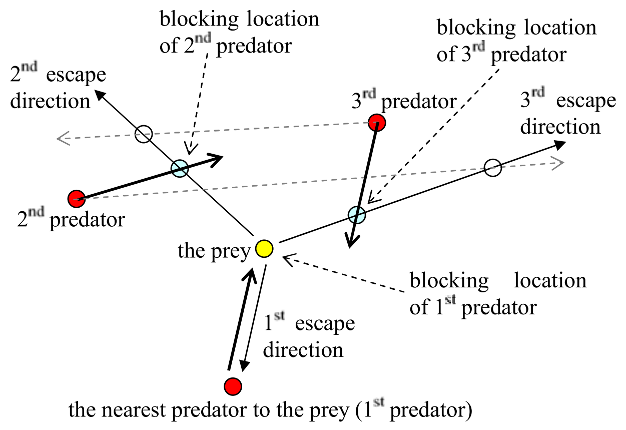


Fig. 12 Assigning escape directions to predators minimizing the total walking cost to the blocking locations

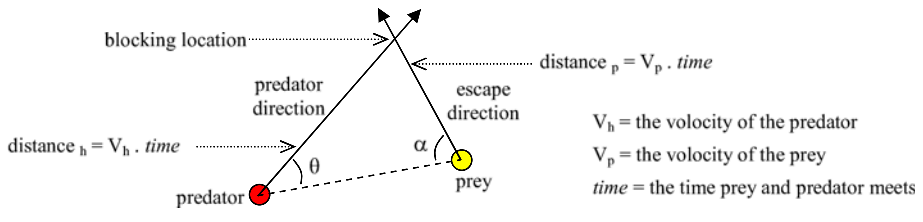
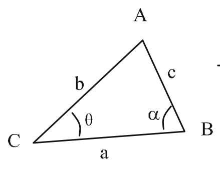


Fig. 13 Determining the blocking location



$$\frac{a}{\mathrm{Sin}(A)} = \frac{b}{\mathrm{Sin}(B)} = \frac{c}{\mathrm{Sin}(C)} \qquad \frac{\text{distance}_h}{\mathrm{Sin}(\alpha)} = \frac{\text{distance}_p}{\mathrm{Sin}(\theta)} \qquad \frac{V_h}{\mathrm{Sin}(\alpha)} = \frac{V_p}{\mathrm{Sin}(\theta)}$$

Fig. 14 Sinus theorem

escape direction, which the predator can reach the prey at if the prey decides to follow the *escape direction* in the next step and keeps his decision until reaching that point. This is illustrated in Fig. 13.

To compute θ given in Fig. 13, we base our formula on sinus theorem shown in Fig. 14, and according to the sinus theorem we get the following formula:

$$\theta = \arcsin(\sin(\alpha) \cdot V_p / V_h)$$

Note that θ is feasible only if the input to arcsin is in the range $[-1, +1]$, therefore we need to examine the input and act accordingly using Algorithm 8.

If α is close to 0 or 180 degrees (checked by line 5 in Algorithm 8), we skip the computation and select the *blocking location* as the location of the prey (line 23) since this will make the *predator direction* (see Fig. 13) aim to the prey location and be almost parallel to the *escape direction* causing errors in floating point computations. Otherwise, we additionally

Algorithm 8 Computing the *blocking location*

```
Require: $p$: this predator
Require: $v_h$: velocity of predator $p$
Require: $v_p$: estimated velocity of the prey
Require: $es$: escape direction originated from the prey that is assigned to predator $p$
Ensure: Blocking location
1: Let $\epsilon$ be a small number (e.g., 0.5)
2: Let $\varepsilon$ be a very small number (e.g., 0.05)
3: Let $\alpha$ be the smallest angle between $es$ and the direction from prey to predator $p$
4: Let $d_{max}$ be maximum permitted distance between blocking location and the prey (e.g., 100)
5: if $\epsilon < \alpha < 180 - \epsilon$ then
6:   if $\sin(\alpha) \cdot (1 + \varepsilon) \cdot v_p/v_h \leq 1$ then
7:     Let $\theta$ be $\arcsin(\sin(\alpha) \cdot (1 + \varepsilon) \cdot v_p/v_h)$
8:     if $\theta < 180 - \alpha - \epsilon$ then
9:       Let $pdir$ be predator direction originated from predator $p$, which is determined using $\theta$ (see Fig. 13)
10:      Let $bl$ be the intersection point of lines passing through $es$ and $pdir$
11:      if Distance between $bl$ and prey $> d_{max}$ then
12:        Let blocking location $(bx, by)$ be the point with distance $d_{max}$ from the prey in the direction
           of $es$
13:      else
14:        Let blocking location $(bx, by)$ be $bl$
15:      end if
16:    else
17:      Let blocking location $(bx, by)$ be a far point with distance $d_{max}$ from the prey in the direction of $es$
         {It is not possible to catch the prey}
18:    end if
19:  else
20:    Let blocking location $(bx, by)$ be a far point with distance $d_{max}$ from the prey in the direction of $es$
       {It is not possible to catch the prey}
21:  end if
22: else
23:   Let blocking location $(bx, by)$ be the location of the prey since the direction to the prey is almost parallel
      to the direction of $es$
24: end if
```

check if $\sin(\alpha) \cdot (1 + \varepsilon) \cdot v_p/v_h$ is less than or equal to 1 (line 6) since arc sinus of numbers greater than 1 is undefined. In the formula, ε is a very small number (e.g., 0.05), which makes the formula over estimate the prey speed in order to let the predator reach the *blocking location* a little bit earlier than the prey. If this conditional check is also satisfied, then we can compute θ using the formula $\arcsin(\sin(\alpha) \cdot (1 + \varepsilon) \cdot v_p/v_h)$ (line 7). And last, we have to do a test to see if θ is less than $180 - \alpha - \epsilon$ (line 8) since sinus theorem is unreliable (in our problem, it is not guaranteed that a feasible triangle is always formed) and may give incorrect results if α is greater than 90 degrees. Here, ϵ is a small number (e.g., 0.5) preventing the floating point errors. Satisfying this final test means it is possible for predator to catch the prey, and the algorithm returns the *blocking location* as the intersection point of two lines passing through the *escape direction* and the *predator direction* (line 10) (see Fig. 13). But we limit the distance of the *blocking location* to the prey with distance d_{max} (lines 11 and 12) in order to reduce the computational cost of the *blocking location* validation (see lines 13–15 in Algorithm 7). In all the other conditions, the predator is not able to catch the prey, and the best way to act is to move parallel to the escape direction. But, since we cannot give a direction as an output, we need to select the *blocking location* as a far point with distance d_{max} from the prey in the *escape direction* (lines 17 and 20).

4.2 Searching for a path

For determining the next move to reach the *blocking location* of the predator, we use a modified version of RTTE-h (see Algorithm 3) called Advanced Real-Time Target Evaluation (ARtte) given in Algorithm 9. In ARtte, all the phases are the same as RTTE-h except *evaluating individual obstacle features* (line 8 in Algorithm 9), *merging entire results* (line 10), and the *outputs*.

Algorithm 9 The algorithm *$ARtte$*

```
Require: p: this predator
Require: s: current cell predator p is on
Require: tx, ty: blocking location of predator p
Ensure: Closed directions of cell s.
Ensure: Best and second best (if any) moving directions on cell s
Ensure: Estimated distances from cell s to prey through best and second best moving directions
 1: Mark all the moving directions of cell s as open (non-closed)
 2: Propagate four diagonal rays from cell s
 3: for each ray r hitting an obstacle do
 4:    Let o be the obstacle hit by ray r
 5:    Extract border b of obstacle o
 6:    Detect closed directions of cell s using border b
 7:    Analyze border b to extract geometric features of obstacle o
 8:    Evaluate result and determine best and second best (if any) directions to avoid obstacle o (combined
       evaluation method using Algorithms 4 and 10)
 9: end for
10: Merge individual results of entire rays to determine proposals (Algorithm 11)
```

We have modified the method in *evaluating individual obstacle features* phase (see Algorithm 4) to determine both the best and the second best proposals for individual obstacles (see Fig. 15 for an illustration). First, we have assumed that all the proposals in Algorithm 4 were pointing to the best direction and its estimated path length. Next we have modified *Case 1.2* as shown in Algorithm 10 in order to propose two alternatives, the best and the second best directions and their estimated path lengths.

We have also modified the method in *merging entire results* phase (see Algorithm 5) to determine the best and the second best proposals considering all the obstacles. The new method given in Algorithm 11 is similar to the previous one. But, instead of only determining the best moving direction, both the best and the second best (if exists) moving directions are determined, and the utility computations are removed since they are performed in Algorithm 13 from now on. The details of the moving direction determination for getting around the *most constraining obstacle* is the same as the previous algorithm.

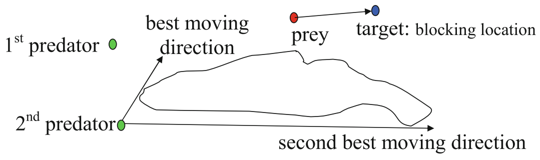


Fig. 15 The best and the second best proposed moving directions

Algorithm 10 Modified case 1.2 of individual obstacle evaluation phase

```
1: Case 1.2:
2: if $d_{left} < d_{right}$ then
3:     Case 1.2.1:
4:     Propose outer left most direction as the best direction and let $d_{left}$ be its estimated path length
5:     Propose outer right most direction as the second best direction and let $d_{right}$ be its estimated path length
6: else
7:     Case 1.2.2:
8:     Propose outer right most direction as the best direction and let $d_{right}$ be its estimated path length
9:     Propose outer left most direction as the second best direction and let $d_{left}$ be its estimated path length
10: end if
```

Algorithm 11 Modified merging phase

```
Require: $p$: this predator
Require: $s$: current cell predator $p$ is on
Require: $t$: cell of blocking location for predator $p$
Require: $r$: individual results of entire rays, which are determined in Algorithm 9
Ensure: Best and second best (if any) moving directions on cell $s$
Ensure: Estimated distances to cell $t$ through best and second best moving directions
1: if All the directions to neighbor cells of $s$ are closed then
2:     Return no moving direction and halt with failure
3: end if
4: Determine the most constraining obstacle $m$ using results in $r$
5: if $m$ exists then
6:     Determine best and second best (if any) moving directions that get around obstacle $m$
7: else
8:     Determine best moving direction as direct flying direction from cell $s$ to cell $t$
9: end if
```

4.3 Selecting the final moving direction

Following the selection of proposals, the best and the second best (if exists) moving directions, the pursuit algorithm calls UAL to decide on a *final proposed moving direction* (see Algorithm 12). If there is only one predator or there exists only one direction proposed or *using alternative proposals* is disabled (checked by line 1 in Algorithm 12), then we just return the best direction as the final direction. Otherwise we require analyzing the proposals furthermore keeping in mind the locations of all the other predators and the prey.

The motivation behind UAL algorithm comes from a difficulty encountered in the pursuit problem. Imagine that we have a large piece of obstacle in the environment, and the predators and the prey are all moving round that obstacle in the same circular direction continuously (see Fig. 16). In such a case, the prey can not be captured by the predators forever if the prey's speed is equal to or greater than the speed of the predators. This problem could only be solved if the circling direction of one of the predators could be switched to the opposite direction in order to stop the prey do that circular movement. Therefore, by selecting the second best moving direction, the UAL algorithm explicitly tries to switch the circling direction and to obtain the desired behavior. As a result of our experiments, we observed that we mostly managed our goal, and the desired behavior occurred if the alternative route was not significantly longer than the best one. However, we also noticed that our first strategy, BES, was also able to obtain the same goal implicitly in some cases.

To compute the *final proposed moving direction*, the algorithm first determines n possible *escape directions* (line 8), where n is the number of predators. The method used to select these directions is the same as the one used in determining the *blocking location*. The first

Algorithm 12 Selecting the final moving direction

```
Require: $p$: this predator
Require: $n$: number of predators
Require: $pr$: set containing best and second best (if exists) proposals for the next move
Ensure: *Final moving direction*
1: if n=1 or UAL is disabled or there is no second best proposal in $pr$ then
2:   Let *final moving direction* be the best direction proposed in $pr$
3: else
4:   Let $nr$ be the nearest predator to the prey
5:   if this predator is $nr$ then
6:     Let *final moving direction* be the best direction proposed in $pr$
7:   else
8:     Determine $e$ as the set of $n$ *escape directions*.
9:     Determine assignment $m$ that maps $n$ predators to $e$ optimally.
10:    Let $edir$ be the *escape direction* assigned to predator $p$ in $m$.
11:    Let $dif$ be the smallest angle between $edir$ and prey-predator vector.
12:    Let *attraction factor* $a_f$ be $0.5 + dif/360$
13:    Let $cw$ be the clockwise angle from prey-predator vector to $edir$
14:    Let $ccw$ be the counter clockwise angle from prey-predator vector to $edir$
15:    if $cw < cww$ then
16:      Let *attraction direction* $a_d$ be 90 degree left of predator-prey vector
17:    else
18:      Let *attraction direction* $a_d$ be 90 degree right of predator-prey vector
19:    end if
20:    Let $angle1$ be the smallest angle between $a_d$ and the best direction proposed in $pr$
21:    Let $angle2$ be the smallest angle between $a_d$ and the second best direction proposed in $pr$
22:    Let $d1$ be the estimated path length of the best direction in $pr$
23:    Let $d2$ be the estimated path length of the second best direction in $pr$
24:    Let $angle\_factor$ be $(angle1/180 + 1)/(angle2/180 + 1)$
25:    Let $utility\_factor$ be $a_f.angle\_factor$
26:    Let $utility\_alternative$ as $utility\_factor.(d1/d2)$
27:    if $utility\_alternative \geq 1$ then
28:      Let *final moving direction* be the second best direction proposed in $pr$
29:    else
30:      Let *final moving direction* be the best direction proposed in $pr$
31:    end if
32:  end if
33: end if
```

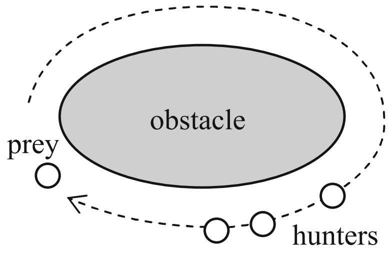


Fig. 16 Moving round an obstacle continuously

escape direction is chosen as the vector directed from the prey to the nearest predator to the prey, and the others are determined based on the first one. The *final proposed moving direction* of the nearest predator to the prey is always selected as the best direction (lines 5 and 6) since we would like to have someone following the prey from a short path, but the others may decide going from longer routes for the sake of better coordination. After selecting the *escape directions*, the algorithm assigns *escape directions* to predators optimally (see Fig. 17 for illustration) such that the total angle difference between the directions from the prey to

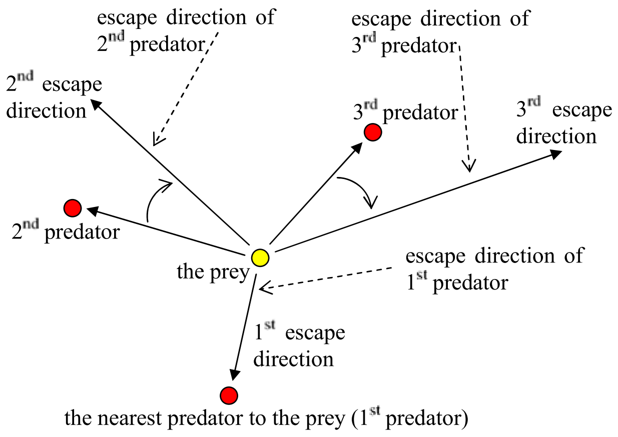

escape direction of 2nd predator
escape direction of 3rd predator
2nd escape direction
3rd predator
3rd escape direction
2nd predator
the prey
escape direction of 1st predator
1st escape direction
the nearest predator to the prey (1st predator)

Fig. 17 Assigning escape directions to predators minimizing the total angle difference

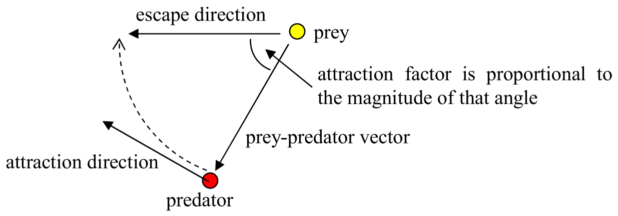


Fig. 18 Computing attraction direction

the predators and their assigned *escape directions* is minimized and the first *escape direction* is assigned to the nearest predator to the prey (line 9). Then we compute the *attraction factor* and the *attraction direction* as given in lines through 10 to 18. *Attraction factor* is a number between 0.5 and 1.0, and proportional to the angle difference between the direction from the prey to the predator and the *escape direction*. *Attraction direction* is the direction the predator should move in order to get closer to the *escape direction* (see Fig. 18 for illustration). Finally we compute the utility of the second best alternative (lines 20–26), and if the utility is greater than or equal to 1, we select the second best direction, otherwise we select the best direction as the *final proposed moving direction* (lines 27–30). The utility formula increases when *attraction direction* is closer to the second best direction or the estimated path length of second best route is closer to the best one; and is determined such that the second best alternative route may be selected only when it is at most two times longer than the best one.

4.4 Computing the utilities of neighbor cells

After determining the final proposed moving direction, we compute the utilities of eight neighbor cells (see Algorithm 13) considering the proposed direction. The utility is a rating value indicating the availability of the neighbor cell (unavailable cells have zero utility), and the angle difference between the proposed direction and the direction of the neighbor cell (smaller difference is better).

First of all, we set the utilities of all the neighbor cells, which are obstacle or temporarily blocked by another predator, to *zero* (line 19 in Algorithm 13). Otherwise we branch

Algorithm 13 Computing the utilities of neighbor cells

```
Require: $p$: this predator
Require: $s$: current cell predator $p$ is on
Require: $st$ : set containing closed directions of cell $s$, which may be north, south, east or west
Require: $dir$ : final proposed direction
Ensure: Utilities of the neighbor cells of $s$
1: for Each eight neighbor cell $c$ of $s$ do
2:   if Cell $c$ is not an obstacle and not occupied by another predator then
3:     Let $dif$ be the smallest angle between $dir$ and the direction of cell $c$
4:     if Cell $c$ is a diagonal neighbor cell then
5:       Let $m$ be the two common neighbors of cell $c$ and cell $s$
6:       if Cells in $m$ are not obstacle and at least one of the directions to cells in $m$ is not marked as closed
         in $st$ then
7:         Let utility of cell $c$ be $(181 - dif)/181$
8:       else
9:         Let utility of cell $c$ be $zero$
10:      end if
11:    else
12:      if Direction to cell $c$ is not marked as closed in $st$ then
13:        Let utility of cell $c$ be $(181 - dif)/181$
14:      else
15:        Let utility of cell $c$ be $zero$
16:      end if
17:    end if
18:  else
19:    Let utility of cell $c$ be $zero$
20:  end if
21: end for
```

according to whether the neighbor cell is a diagonal (north-east, north-west, south-east or south-west) one, or not. If the cell is a diagonal one, then we determine the two *common neighbors* (see Definition 12) of the diagonal cell and the cell the predator is on (line 5). If common neighbors are both not obstacles, and at least one of the directions to these cells are not closed by the ARtte algorithm (line 6), then we set the utility of the diagonal cell to $(181 - dif)/181$ (line 7), where dif is the smallest angle between the final proposed moving direction and the direction of the diagonal neighbor cell (line 3). Otherwise, we set the utility of the diagonal cell to $zero$ (line 9). If the neighbor cell is a horizontal or vertical one, then we set the utility to $(181 - dif)/181$ (line 13) if the direction to that cell is not closed (line 12), else we set the utility to $zero$ (line 15).

A complete sample illustrating the entire process of MAPS is given in Fig. 19. In the sample, we assume that all the computations are performed from the view point of the first predator. The escape directions are computed first and next the blocking locations are determined (BES). Since the blocking location of the first predator falls into an obstacle, it is corrected before used by the path search algorithm. Later on, the best and the second best moving directions are determined (ARtte). Finally, these proposals are examined by the algorithm (UAL) in order to decide on a final moving direction.

4.5 Analysis of the algorithm

In each move of a predator, MAPS performs three time consuming phases, which are path search with ARtte, validation of the *blocking location* with RTA*, and the optimal assignments of *escape directions* to predators.

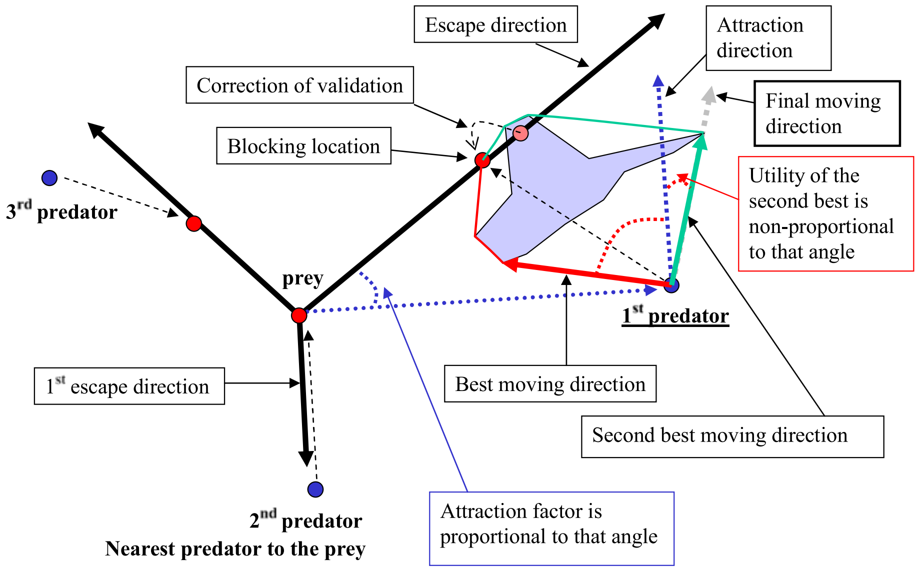


Fig. 19 A complete sample illustrating the entire process of MAPS

In ARtte, there is only one modification towards determining the second best direction, and the rest of the algorithm is the same as Rtte-h. We have just exported an additional information, which potentially existed in Rtte-h, but was not used. That modification does not change the complexity, which is $O(w \cdot h)$ per step, where w is the width and h is the height of the grid. Proof of completeness is also the same since we haven't done any modification on close direction detection step, which the proof of completeness is built on. And the last important feature of ARtte derived from Rtte-h is the search depth. Since increasing the grid size decreases the efficiency, we usually use a search depth (d) in order to limit the worst case complexity with $O(d^2)$.

The validation of the *blocking location* takes time proportional to d_{max}, where d_{max} is the maximum permitted distance between the *blocking location* and the prey. Thus the complexity of this phase is $O(d_{max})$.

And finally, the complexity of the optimal assignment is $O((n-1)!)$, where n is the number of predators. Therefore the total complexity becomes $O(d^2 + d_{max} + (n-1)!)$, but n and d_{max} will most probably be taken not very high in practice, so we may assume the complexity be $O(d^2)$ since path search will be the most time consuming part. Additionally, if one needs to have many predators, then the optimal assignment requirement can be relaxed by using a greedy algorithm.

The final question that is not answered yet is the proof of completeness of the pursuit algorithm as a whole, which actually depends on answering the question "is it possible to guarantee catching of the prey in a finite time?". The answer is yes for a static prey, which comes from the completeness of the path planning algorithm against a static target. But the answer is not straightforward against a moving target since there are some difficulties and preconditions.

Firstly, in our problem, there is an hazy environment full of obstacles, thus predators cannot move freely and they have to find paths toward the moving prey meanwhile avoiding obstacles. That prevents having predetermined search patterns, which also makes difficult to find an analytical proof.

Secondly, in order to guarantee a catch, some preconditions must hold. If the speed of the prey is faster than the predators, the prey can run away forever if there is no dead-ends the prey can get in. If the speed of the prey is slower, it is still not always possible to guarantee a catch because that depends on the path planners used for both prey and the predators. The pursuit problem necessarily requires real-time path planning algorithms, which can only provide sub-optimal solutions. Therefore, if the performance of the real-time path planner of the prey is better than that of the predators on the average, the prey will make less mistakes than the predators do. Therefore it may be possible for the prey to escape forever if its speed is not significantly slower than the predators or there is not enough number of predators in the field.

Finally, it is also not straightforward to determine the number of predators required for a successful pursuit since that strictly depends on the topography of the environment. If there are many gateways providing corridors for the prey to escape, more predators will be required to block these gateways. However, if the environment is full of free spaces, a few predators will be just enough.

5 Experimental results

In this section, we present the experimental results of our coordinated pursuit algorithm against moving preys guided by Prey-A*. In each iteration, the prey and the predators are executed alternately in order to prevent the side effects caused by the difference in efficiency of the algorithms. For coordination, we used four strategies: without coordination (None), with *blocking escape directions* (BES), with *using alternative proposals* (UAL) and with both BES and UAL (BES + UAL).

For the test runs, we used 9 randomly generated sample grids of size 150 × 150. Six of them were the *maze* grids (see Fig. 20), and three of them were the *U-type* grids (see Fig. 21). The *maze* grids were produced with the constraint that every two non-obstacle cells are always connected through a path. For each obstacle ratio (25%, 30% and 35%), two test mazes were randomly generated. The obstacle ratio is chosen not to be more than 35% in order to make enough room for prey to escape. The *U-type* grids were created by randomly putting *U*-shaped obstacles of random sizes (5 to 30 cells) on an empty grid limiting the number of *U*-type obstacles with 70, 90 or 120. For each grid, three different strategies (*one*

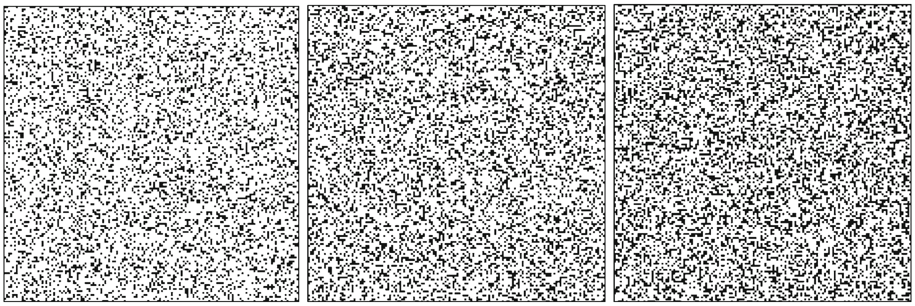


Fig. 20 Maze grids with 25% (*left*), 30% (*middle*) and 35% (*right*) obstacles

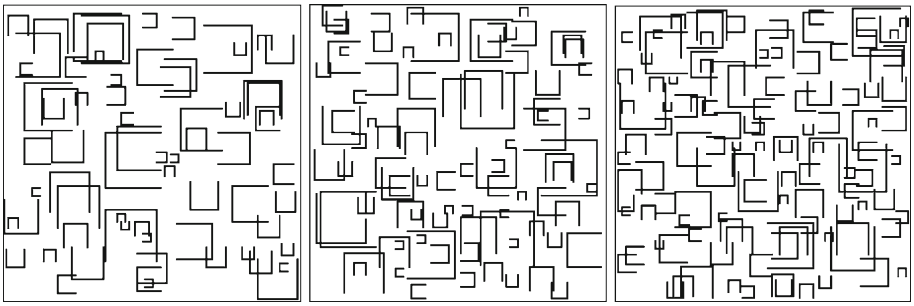


Fig. 21 U-type grids with 70 (*left*), 90 (*middle*) and 120 (*right*) u-type obstacles

corner, *one side* and *all sides*) were used to select the initial predator locations, and for each strategy, 15 different predator-prey location sets were generated and kept the same for all different test configurations for fairness. To get random locations, the grid world was divided into 5 columns and 5 rows, which formed 25 regions. With the *one corner* strategy, the predators were randomly located together in one of the four corner regions of the grid world, and with the *one side* strategy, the predators were randomly located together in one of the four side regions. Finally, using the *all sides* strategy, the predators were randomly located in any of the side regions of the grid world. The prey was always randomly located in the center region.

In the experiments, we assumed that the prey knows the entire grid world and the location of the predators all the time, and the predators always know the location of the prey, but perceive the grid world up to predefined *vision range*. Our tests were performed with 10, 20 and *infinite* vision ranges and 40 search depth. Additionally, we assumed that the prey is slightly slower than the predators, and skips 1 move after each 24 moves.

In addition to the experiments that will be examined in the following sections, we have also demonstrated the behavior of uncoordinated and coordinated predators (using BES) in an empty environment in Fig. 22. We only used the coordination algorithm BES since UAL will not change the result in an empty grid (there is no second best proposal in such a case).

5.1 Analysis of path lengths

In this section, we examine the effect of coordination on the number of moves to catch a moving prey. In Fig. 23, we present the experimental results with respect to team size, vision range and initial locations of the predators, and in Fig. 24, we present the results in terms of grid types. In the charts, the horizontal axis is the team size, the vision range or the initial locations of the predators, and the vertical axis is the number of moves to reach the target.

When we examine the results, we see that increasing the number of predators involved in the coordinated search significantly reduces the number of moves to catch the prey, and the solutions with coordination are clearly better than the ones without coordination. The coordination strategies, BES and BES + UAL, are very competitive to each other, and usually perform much better than UAL. With more than 2 predators, BES is slightly ahead of BES + UAL, but with 2 predators, BES usually becomes worse than BES + UAL. Examining the reasons, we observe that when two predators exist, the two escape directions are selected in exactly the opposite directions (180 degrees between them), therefore the first blocking location is selected as the location of the prey, and the second one is selected usually as a far point in front of the preys moving direction. When the predators using BES are following the

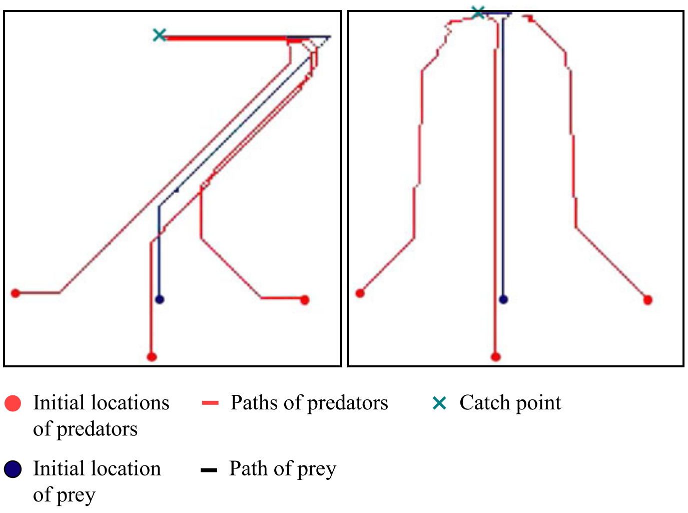


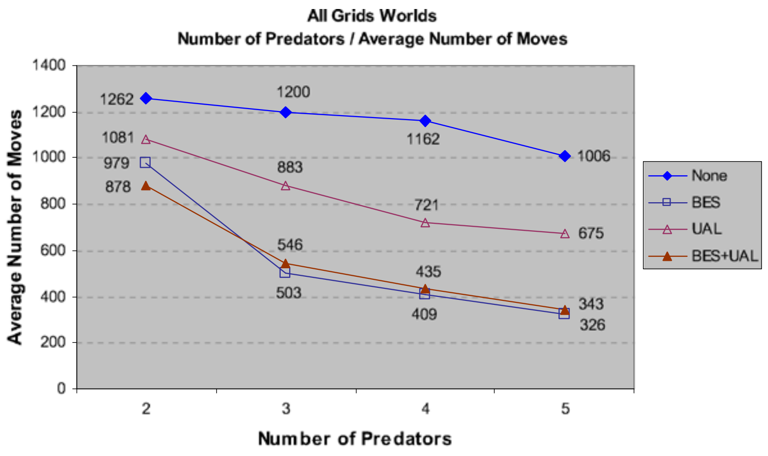
Fig. 22 The behavior of uncoordinated predators (*left*) and the behavior of coordinated predators using BES (*right*)

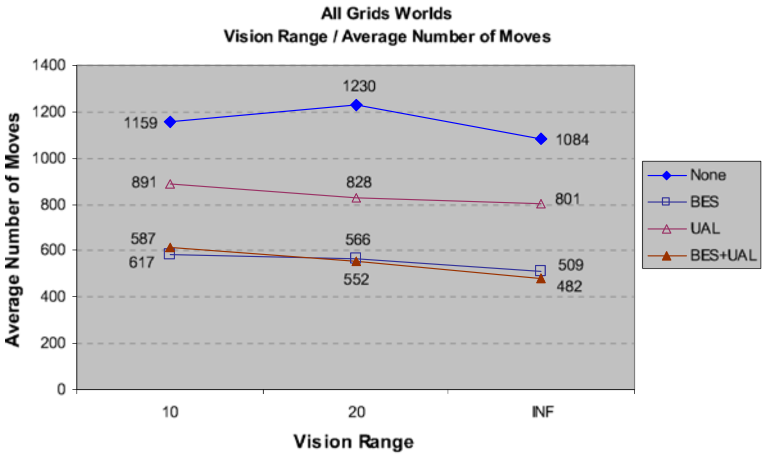
prey behind, blocking the second escape direction (laying in front of the prey) is a hard task for the second predator, thus it decides to follow the prey in a direction parallel to the escape direction, which is also almost parallel to the moving direction of the prey. This behavior sometimes makes the second predator follow the similar path as the first one, which can be better avoided if integrated with UAL.

In Figs. 25, 26, 27 and 28, we exemplify routes of four different strategies in a maze grid with 30% obstacles followed by 2, 3, 4 and 5 predators, respectively. In the example, the initial locations of the predators were selected from the bottom-right corner. The general aim of the prey is moving to the top-left corner first, waiting there until the predators get nearer, and finally performing a quick manoeuvre in order to escape from the predators and move to the bottom-right corner. From the figures, we see that without coordination, the predators usually move together on a line, and in coordination, the predators spread to the environment in order to surround the prey better and to close the potential gateways that the prey may escape through. We also observe that BES + UAL strategy spread the predators the most.

According to the experimental results, vision range does not seem to affect the results much. For instance, the average number of moves to reach the prey is 814 with 10 vision range, and 719 with infinite vision range. This is not very surprising since our grid worlds are not very complicated (obstacle percentage is less than or equal to 35), and increasing the vision range does not gain much. But interestingly, the initial locations of the predators are not also affecting the results. This possibly means that positional distribution of predators around the prey and coordination behaviors are mostly insignificant when predators are far away from the prey, which is the case for initial setting, but becomes significant when they get closer, where initial distribution is mostly corrupted so far.

We also examined the average and the standard deviations of the number of moves to catch the prey for various grid types and coordination strategies. The results given in Table 1 show that the standard deviations of the strategy with no coordination are the highest and tend to decrease slightly with the increase in the number of predators involved in the search. The strategies, BES and BES + UAL, are again the best and have standard deviations close to each





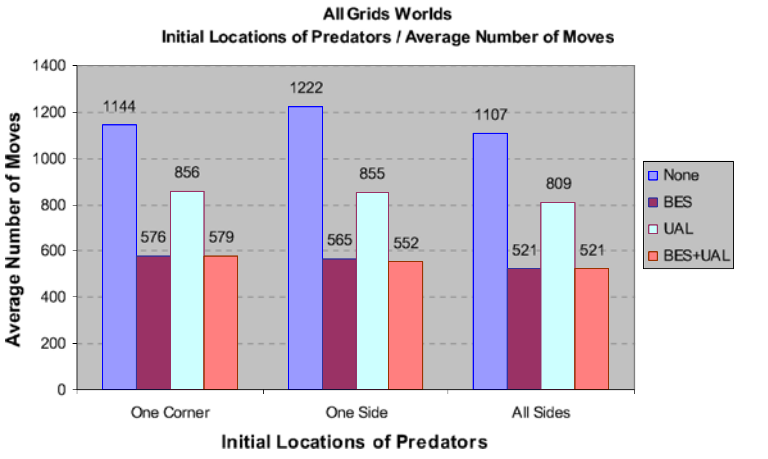
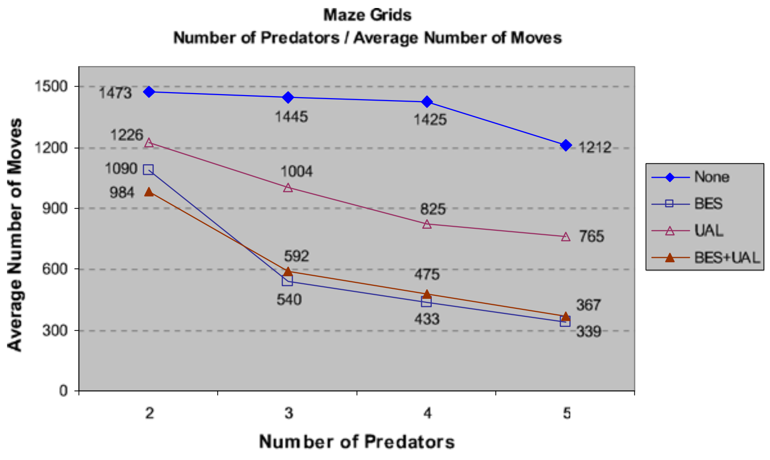


Fig. 23 Average number of moves to reach a moving prey for different number of predators (*top*), vision ranges (*middle*) and initial locations of predators (*bottom*)



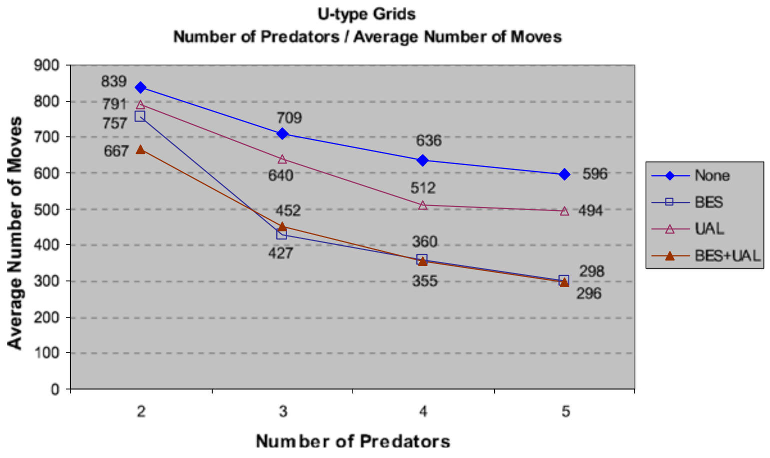


Fig. 24 Average number of moves to reach a moving prey in maze (*top*) and U-type (*bottom*) grids

other. With 2 predators, BES + UAL has the lowest standard deviation, and with more than 2 predators, BES becomes the best. UAL follows BES and BES + UAL in the third place, and has significantly higher standard deviations, but better than having no coordination for sure.

With respect to grid types, we observe that the maze grids with 35% obstacles are the most difficult ones for the predators, and the U-types grids are the easiest. One interesting result was that mazes with 25% obstacles were more difficult for predators than mazes with 30% obstacles. This shows that the obstacle ratio is not strictly the determining factor for the difficulty of the maze when pursuing a moving prey. Although there is more obstacles, a prey may not sometimes be able to escape easily if there are many dead-ends.

5.2 Analysis of step execution times

We have also examined the step execution times of MAPS with different coordination strategies and predator team sizes running on a laptop computer with 1.66 GHz Solo processor. In Table 2, the average number of moves executed per second per predator in maze and U-type

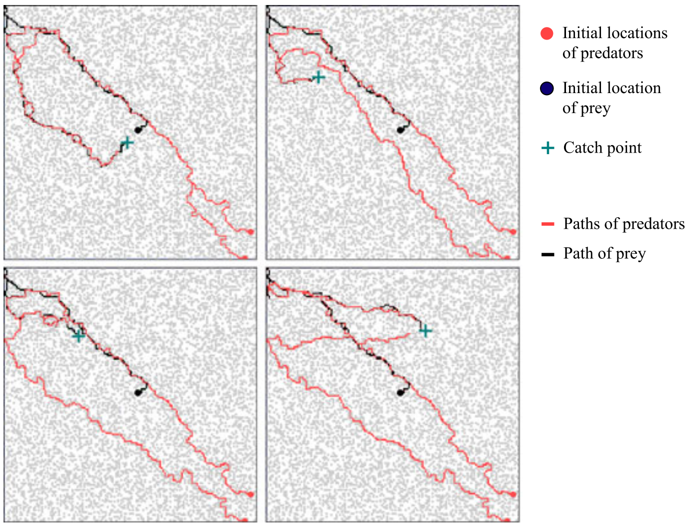


Fig. 25 2 predators against a moving prey: No coordination (*top-left*), coordination with BES (*top-right*), UAL (*bottom-left*) and BES + UAL (*bottom-right*)

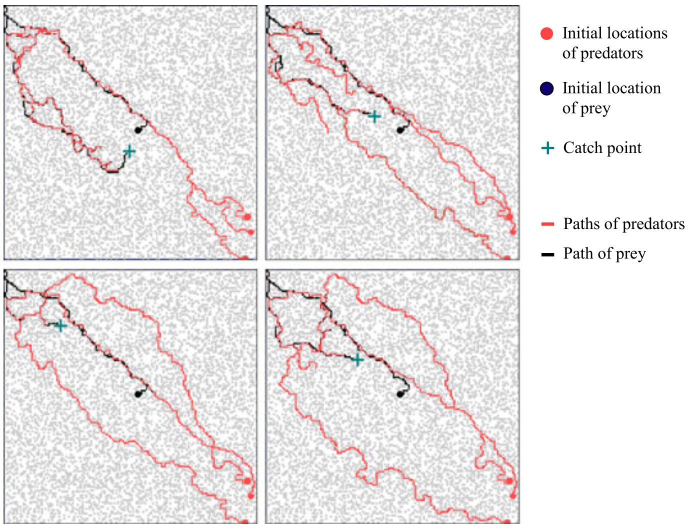


Fig. 26 3 predators against a moving prey: No coordination (*top-left*), coordination with BES (*top-right*), UAL (*bottom-left*) and BES + UAL (*bottom-right*)

grids are shown. The rows are for the compared coordination strategies and the columns are for the predator team sizes from 2 to 5.

The results showed that increasing the number of predators does not reduce the efficiency much, and the most efficient algorithms are MAPS with no coordination and MAPS with UAL, which perform almost the same speed. MAPS with BES and MAPS with BES + UAL perform slightly slower since computation and validation of blocking locations take time. We

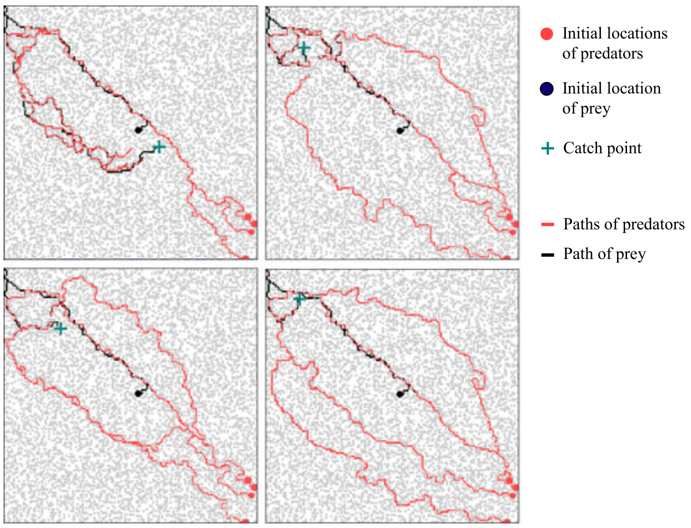


Fig. 27 4 predators against a moving prey: No coordination (*top-left*), coordination with BES (*top-right*), UAL (*bottom-left*) and BES + UAL (*bottom-right*)

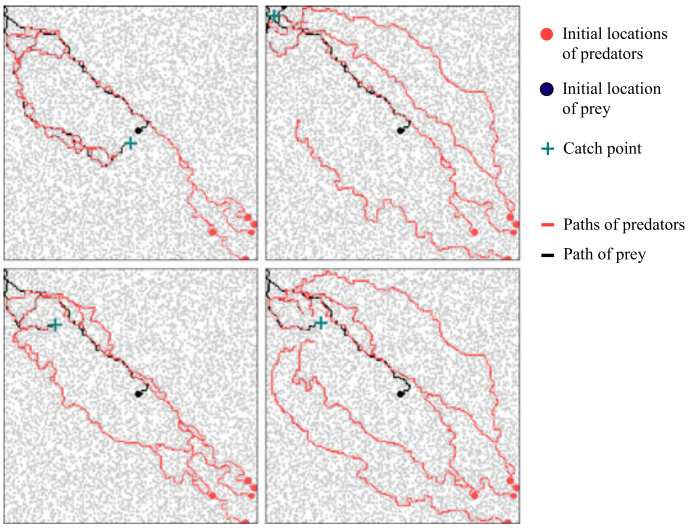


Fig. 28 5 predators against a moving prey: No coordination (*top-left*), coordination with BES (*top-right*), UAL (*bottom-left*) and BES + UAL (*bottom-right*)

also see that the step execution times are the lowest in maze grids with 25% obstacles, and the highest in maze grids with 35% obstacles since the worse case complexity of MAPS depends on both the search depth, which is 40 in our experiments, and the sizes of the obstacles in the environment, which are the largest in maze grids with 35% obstacles.

Table 1 The average number of moves and their standard deviations to reach a moving prey using different coordination strategies with 2, 3, 4 and 5 predators

Number of predators	None		BES		UAL		BES + UAL	
	Average	SD	Average	SD	Average	SD	Average	SD
All grids								
2	1262	1460	979	867	1081	866	878	626
3	1200	1111	503	340	883	807	546	369
4	1162	1253	409	265	721	695	435	267
5	1006	1136	326	169	675	711	343	172
Maze grids with 25% obstacles								
2	1151	947	921	886	1095	821	913	646
3	1020	805	544	461	819	773	585	455
4	1128	1219	426	321	761	931	511	354
5	909	874	320	191	778	920	375	237
Maze grids with 30% obstacles								
2	871	635	811	622	966	738	914	669
3	812	712	522	400	797	687	606	484
4	727	596	430	330	655	619	466	309
5	733	674	326	179	608	663	340	159
Maze grids with 35% obstacles								
2	2398	4322	1538	1716	1619	1706	1124	977
3	2504	2874	556	363	1396	1596	585	386
4	2421	3244	443	266	1060	1129	446	254
5	1994	3035	371	167	909	1153	386	188
U-type grids								
2	839	442	757	450	791	419	667	350
3	709	405	427	202	640	382	452	222
4	636	385	360	184	512	297	355	187
5	596	350	298	146	494	309	296	125

6 Conclusion and future work

In this paper, we have focused on pursut domain and presented a multi-agent real-time pursuit algorithm, MAPS, which employs two coordination strategies called *blocking escape directions* (BES) and *using alternative proposals* (UAL). We compared four coordination configurations: *no coordination*, *coordination with BES*, *coordination with UAL* and *coordination with BES + UAL*, and observed that coordination significantly reduces the number of moves to catch a moving target. We also observed that coordination with BES and BES + UAL performs the best.

As a future work, we think there is still much to do on multi-agent pursuit domain from the view point of both predators and preys, especially in environments with obstacles. In the paper, we have proposed a pursuit algorithm, which estimates the escape directions of the prey analytically with less considering the environment, but it would be very valuable if the topography of the environment is taken into account for determining where the prey may move to. We have also assumed that the location of the prey is always known by the

Table 2 The average number of moves per second per predator for different coordination strategies and predator team sizes

Algorithm	2 Predators	3 Predators	4 Predators	5 Predators
Maze grids with 25% obstacles				
None	765	772	775	774
BES	558	537	531	523
UAL	763	738	735	685
BES + UAL	557	519	519	517
Maze grids with 30% obstacles				
None	491	495	450	441
BES	380	347	329	328
UAL	453	444	435	413
BES + UAL	355	326	322	317
Maze grids with 35% obstacles				
None	216	210	206	207
BES	191	180	169	169
UAL	211	210	199	198
BES + UAL	187	177	166	166
U-type grids				
None	547	534	507	492
BES	408	388	376	375
UAL	550	522	504	447
BES + UAL	418	389	377	376

predators. This assumption can be relaxed, and the coordination algorithms can be extended to be able to estimate the location of the prey and search the environment in situations where the prey is not seen. And lastly, we have employed a deliberative prey algorithm in order to place a powerful rival against the predators. Although this algorithm is strong enough most of the time, we think there is still much research to be done on the prey algorithms in terms of both escape capability and execution time efficiency.

References

1. Ishida, T., & Korf, R. (1995). Moving target search: A real-time search for changing goals. *IEEE Trans Pattern Analysis and Machine Intelligence, 17*(6), 97–109.
2. Undeger, C. (2007). Single and multi agent real-time path search in dynamic and partially observable environments. Ph.D. thesis in Computer Engineering Department of Middle East Technical university.
3. Undeger C., & Polat F. (2008). Real-time moving target evaluation search. *IEEE Transaction on Systems, Man and Cybernetics, Part C, 39*(3), 366–372.
4. Tanenbaum, A. (1996). *Computer networks*. New Jersey: Prentice-Hall.
5. Russell, S., & Norving, P. (1995). *Artificial intelligence: A modern approach*. New Jersey: Prentice Hall.
6. Gutmann, J., Fukuchi, M., & Fujita, M. (2005). Real-time path planning for humanoid robot navigation. In *Internationl joint conference on artificial intelligence IJCAI-05* (pp. 1232–1237). Denver, CO: Professional Book Center.
7. Michalewicz, Z. (1986). *Genetic algorithms + data structure = evolution programs*. New York: Springer.

8. 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.
9. Cheng, P., & LaValle, S. M. (2002). Resolution complete rapidly-exploring random trees. In *Proceedings of IEEE international conference on robotics and automation* (pp. 267–272).
10. LaValle, S., & Kuffner, J. (1999). Randomized kinodynamic planning. In *Proceedings of the IEEE international conference on robotics and automation (ICRA'99)*.
11. LaValle, S. M., & Kuffner, J. J. (2001). Rapidly-exploring random trees: Progress and prospects. *Algorithmic and computational robotics: New directions* (pp. 293–308). Wellesley, MA: A K Peters.
12. Kavraki, L., & Latombe, J. (1998). Probabilistic roadmaps for robot path planning, ser. In *Practical motion planning in robotics: current and future directions*. Massachusetts: Addison-Wesley.
13. 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*.
14. Koenig, S., Likhachev, M., Liu, Y., & Furcy, D. (2004). Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine, 25*(2), 99–112.
15. Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE international conference on robotics and automation*.
16. Mudgal, A., Tovey, C., Greenberg, S., & Koenig, S. (2005). Bounds on the travel cost of a mars rover prototype search heuristic. *SIAM Journal on Discrete Mathematics, 19*(2), 431–447.
17. Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the international joint conference on artificial intelligence*.
18. Koenig, S., & Likhachev, M. (2002a). D* lite. In *Proceedings of the national conference on artificial intelligence* (pp. 476–483).
19. Koenig, S., & Likhachev, M. (2002b). Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the international conference on robotics and automation*.
20. Koenig, S., & Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *Transactions on Robotics, 21*(3), 354–363.
21. Koenig, S., Likhachev, M., & Sun, X. (2007). Speeding up moving-target search*. In *6th international joint conference on autonomous agents and multiagent systems*.
22. Kamon, I., Rivlin, E., & Rimon, E. (1996). A new range-sensor based globally convergent navigation algorithm for mobile robots. In *Proceedings of the IEEE international conference on robotics and automation* (Vol. 1, pp. 429–435).
23. Lumelsky, V. J., & Skewis, T. (1987). Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algoritmica, 2*, 403–430.
24. Korf, R. (1990). Real-time heuristic search. *Artificial Intelligence, 42*(2–3), 189–211.
25. Koenig, S. (2004). A comparison of fast search methods for real-time situated agents. *AAMAS 2004* (pp. 864–871).
26. Koenig, S., & Likhachev, M. (2006). Real-time adaptive a*. In *5th international joint conference on autonomous agents and multiagent systems* (pp. 281–288).
27. Hernandez, C., & Meseguer, P. (2005). Lrta*(k). In *International joint conference on artificial intelligence IJCAI-05* (pp. 1238–1243).
28. Undeger, C. (2001). Real-time mission planning for virtual human agents. M.Sc. thesis in Computer Engineering Department of Middle East Technical University.
29. Shimbo, M., & Ishida, T. (2003). Controlling the learning process of real-time heuristic search. *Artificial Intelligence, 146*(1), 1–41.
30. Thorpe, P. (1994). A hybrid learning real-time search algorithm. Master's thesis, Computer Science Department, University of California at Los Angeles.
31. Edelkamp, S., & Eckerle, J. (1997). New strategies in real-time heuristic search. In *Proceedings of the AAAI-97 workshop on on-line search* (pp. 30–35).
32. Furcy, D., & Koenig, S. (2000). Speeding up the convergence of real-time search. In *Proceedings of AAAI* (pp. 891–897).
33. Furcy, D., & Koenig, S. (2001). Combining two fast-learning real-time search algorithms yields even faster learning. In *Proceedings of the 6th European conference on planning*.
34. Konar, A. (2000). *Artificial intelligence and soft computing: Behavioral and cognitive modeling of human brain*. Florida: CRC Press LLC.
35. Bruce, J., & Veloso, M. (2002). Real-time randomized path planning for robot navigation. In *Proceedings of international conference on intelligent robots and systems* (pp. 2383–2388).
36. Hsu, D., Kindel, R., Latombe, J., & Rock, S. (2002). Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research, 21*(3), 233–255.

37. 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 2001*.
38. Undeger, C., & Polat, F. (2007). Real-time edge follow: A real-time path search approach. *IEEE Transaction on Systems, Man and Cybernetics, Part C, 37*(5), 860–872.
39. Undeger, C., & Polat, F. (2006). Real-time target evaluation search. In *5th Internaltional joint conference on autonomous agents and multiagent systems, AAMAS-06* (pp. 332–334).
40. Undeger, C., & Polat, F. (2007). Rttes: Real-time search in dynamic environments. *Applied Intelligence, 27*, 113–129.
41. Benda, M., Jagannathan, V., & Dodhiawalla, R. (1986). On optimal cooperation of knowledge sources. Technical Report No.BCS-G2010-28, Boeing Advanced Technology Center.
42. Erus, G., & Polat, F. (2007). A layered approach to learning coordination knowledge in multiagent environments. *Applied Intelligence, 27*(3), 249–267.
43. Ishiwaka, Y., Sato, T., & Kakazu, Y. (2003). An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning. *Elsevier Journal on Robotics and Autonomous Systems, 43*(4), 245–256.
44. Xiao, D., & Tan, A. (2005). Cooperative cognitive agents and reinforcement learning in pursuit game. In *Proceedings of 3rd international conference on computational intelligence, robotics and autonomous systems (CIRAS'05)*.
45. Haynes, T., & Sen, S. (1996). Evolving behavioral strategies in predators and prey. *Springer Book on Adaptation and Learning in Multiagent Systems, 1042*, 113–126.
46. Haynes, T., & Sen, S. (1997). The evolution of multiagent coordination strategies. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.4981.
47. Korf R. (1992). A simple solution to pursuit games. In *Working papers of the 11th international workshop on distributed artificial intelligence* (pp. 183–194).
48. Yong, C., & Miikkulainen, R. (2001). Cooperative coevolution of multi-agent systems. Technical report: AI01-287, University of Texas at Austin.
49. Levy, R., & Rosenschein, J. (1992). A game theoretic approach to the pursuit problem. In *11th international workshop on distributed artificial intelligence*.
50. Kitamura, Y., Teranishi, K., & Tatsumi, S. (1996). Organizational strategies for multiagent real-time search. In *Proceedings of international conference on multi-agent systems (ICMAS-96)* (pp. 150–156).
51. Knight, K. (1993). Are many reactive agents better than a few deliberative ones? In *Proceedings of the 10th international joint conference on artificial intelligence* (pp. 432–437).
52. Goldenberg, M., Kovarsky, A., Wu, X., & Schaeffer J. (2003). Multiple agents moving target search. In *International joint conference on artificial intelligence, IJCAI* (pp. 1536–1538).
53. Vincent, P., & Rubin, I. (2004). A framework and analysis for cooperative search using uav swarms. In *Proceedings of the 2004 ACM symposium on applied computing*.
54. Altshuler, Y., Yanovskya, V., Wagner, I. A., & Bruckstein, A. M. (2008). Efficient cooperative search of smart targets using uav swarms. *Robotica, 26*, 551–557.
55. Kota, R., Braynov, S., & Llinas, J. (2003). Multi-agent moving target search in a hazy environment. *IEEE international conference on integration of knowledge intensive multi-agent systems* (pp. 275–278).