SINGLE AND MULTI AGENT REAL-TIME PATH SEARCH IN DYNAMIC AND PARTIALLY OBSERVABLE ENVIRONMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞATAY ÜNDEĞER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JANUARY 2007

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Faruk Polat
Supervisor

Examining Committee Members

Prof. Dr. H. Altay Güvenir	(Bilkent Univ.)	_____
Prof. Dr. Faruk Polat	(METU,CENG)	_____
Assoc. Prof. Dr. İ. Hakkı Toroslu	(METU,CENG)	_____
Assoc. Prof. Dr. Veysi İşler	(METU,CENG)	_____
Assoc. Prof. Dr. Halit Oğuztüzün	(METU,CENG)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Çağatay Ündeğer

Signature :

ABSTRACT

SINGLE AND MULTI AGENT REAL-TIME PATH SEARCH IN DYNAMIC AND PARTIALLY OBSERVABLE ENVIRONMENTS

Ündeğer, Çağatay

Ph.D., Department of Computer Engineering

Supervisor : Prof. Dr. Faruk Polat

January 2007, 121 pages

In this thesis, we address the problem of real-time path search in partially observable grid worlds, and propose two single agent and one multi-agent search algorithm.

The first algorithm, Real-Time Edge Follow (RTEF), is capable of detecting the closed directions around the agent by analyzing the nearby obstacles, thus avoiding dead-ends in order to reach a static target more effectively. We compared RTEF with a well-known algorithm, Real-Time A* (RTA*) proposed by Korf, and observed significant improvement.

The second algorithm, Real-Time Moving Target Evaluation Search (MTES), is also able to detect the closed directions similar to RTEF, but in addition, determines the estimated best direction that leads to a static or moving target from a shorter path. Employing this new algorithm, we obtain an impressive improvement over RTEF with respect to path length, but at the cost of extra computation. We compared our algorithms with Moving Target Search (MTS) developed by Ishida and the off-line path planning algorithm A*, and observed that MTES performs significantly better than MTS, and offers solutions very close to optimal ones produced by A*.

Finally, we present 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 (BES) and Using Alternative Proposals (UAL), which help the predators waylay the possible escape directions of the prey in

coordination. We compared our coordination strategies with the uncoordinated one, and observed an impressive reduction in the number of moves to catch the prey.

Keywords: Path Planning, Real-Time Search, Multi-Agent Pursuit

ÖZ

DEĞİŞKEN VE KISMİ GÖZLEMLENEBİLİR ORTAMLARDA TEK VE ÇOKLU ETMEN GERÇEK ZAMANLI YOL ARAMA

Ündeğer, Çağatay

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Faruk Polat

Ocak 2006, 121 sayfa

Bu tezde, kısmen gözlemlenebilir ızgara dünyalardaki gerçek zamanlı yol arama problemi için iki farklı tek etmenli ve bir çoklu etmenli arama algoritması önerilmektedir.

İlk algoritma olan Real-Time Edge Follow (RTEF), sabit bir hedefe daha etkin şekilde ulaşabilmek için yakında bulunan engelleri analiz ederek etmenin çevresindeki kapalı yönleri tespit edebilme, dolayısıyla çıkmaz sokaklardan sakınabilme kabiliyetine sahiptir. RTEF'in performansını gösterebilmek için Korf tarafından önerilen Real-Time A* (RTA*) dikkate alınmış olup, yapılan deneysel çalışma sonucunda dikkate değer bir iyileşme gözlemlenmiştir.

İkinci algoritma olan Real-Time Moving Target Evaluation Search (MTES) de RTEF gibi kapalı yönleri tespit edebilmekte, ancak ek olarak, sabit veya hareketli bir hedefe en kısa yoldan giden rotayı daha doğru olarak tahmin edebilmektedir. Ek hesaplama maliyeti olsa da, bu yeni algoritma ile RTEF'e karşı yol uzunluğu açısından etkileyici bir iyileşme elde edilmiştir. Önerilen algoritmalar, Ishida tarafından geliştirilen Moving Target Search (MTS) ve çevrim dışı yol planlama algoritması A* ile test edildi ve MTES'in MTS'den çok daha iyi performans gösterdiği ve A* tarafından sağlanan en iyi çözümlere çok yakın sonuçlar sunduğu gözlemlendi.

Son olarak, birden fazla avcı ile hareketli bir hedefi kordineli şekilde kovalama ka-

biliyetine sahip bir çoklu etmen yol arama algoritması olan Multi-Agent Real-Time Pursuit (MAPS) geliştirilmiştir. MAPS, avın muhtemel kaçış yönlerini avcıların koordineli olarak kesebilmesine yardımcı olan Kaçış Yönlerini Kapama (BES) ve Alternatif Önerileri Kullanma (UAL) olarak isimlendirilen iki yeni koordinasyon stratejisi kullanmaktadır. Önerilen bu stratejiler, koordinasyonsuz olanla mukayese edildi ve avı yakalama adım sayılarında çarpıcı bir iyileşme gözlemlendi.

Anahtar Kelimeler: Yol Planlama, Gerçek Zamanlı Arama, Çoklu Etmen Kovalama

ACKNOWLEDGMENTS

I would like to thank Prof. Dr. Faruk Polat for his excellent advice, help and mentoring during the development of this work.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Path planning can be described as finding a sequence of moves from an initial state (starting point) to a goal state (target point), or as finding out that no such sequence exists. Path-planning algorithms can be either off-line or on-line. Off-line algorithms find the whole solution in advance before starting execution, and suffer from execution time in dynamic or partially observable environments due to frequent re-planning requirements. On the other hand, on-line algorithms require planning and execution phases to be coupled in a way that the agent repeatedly plans its next move in limited time and executes it. These algorithms are not designed to be optimal, and usually find poor solutions with respect to path length. Furthermore, there exist some hybrid solutions such as incremental heuristic search, which are optimal and more efficient than off-line path planning algorithms. However, they are still slow for some real-time applications.

In the path planning domain, it is very common to assume that the goal state is static. When this assumption is relaxed for covering changing goals, the problem becomes very complicated, which can only be handled by a few number of algorithms. Off-line and incremental path planning algorithms are not able to deal with this problem since they require re-planning towards the changing goal from scratch in each step, which takes a considerable time. Unfortunately, most of the on-line search algorithms are not also able to solve the problem since they are usually designed for partially observable environments, but not for changing goals, and store search information collected during the exploration, which is difficult to be updated when the goal changes.

In this thesis, we first propose a real-time path planning algorithm, Real-Time Edge Follow (RTEF) [50, 47], developed for searching a static target in partially

observable planer grid worlds (e.g., mazes). RTEF uses a powerful heuristic function called RTEF-Alternative Reduction Method (RTEF-ARM), which sends rays away from the agent in four diagonal directions, and analyzes the obstacles that the rays hit in order to discard some non-promising (closed) alternative moving directions in real-time to successfully guide the agent to the target avoiding dead-ends. To show the effectiveness of the algorithm, we randomly generated a number of grids of different types (random, maze and U-type) and compared RTEF in these grids with a well-known on-line search algorithm introduced by Korf [28] called Real-Time A* (RTA*), and its extended version, RTA* with n-look-ahead depth. As a result of the experiments, we obtained impressive improvements in the path lengths over both RTA* versions.

Although RTEF is able to determine the closed directions successfully, it is weak in selecting the right move from the remaining alternatives as it uses the poor Euclidian distance heuristic. Therefore, we focused on a new method to be able to make better use of environmental information available to the agent in order to select the best moving direction avoiding nearby obstacles to capture a static or moving target (prey) as soon as possible. With this intuition, we introduce our second real-time search algorithm, Real-Time Moving Target Evaluation Search (MTES) [49, 48], capable of estimating the distance to the target more accurately considering the intervening obstacles. Similar to RTEF, the method sends rays away from the agent in four directions, and determines the obstacles that the rays hit. For each such obstacle, we extract its border and determine the best direction that avoids the obstacle if the target is blocked by the obstacle. Hence, we have a number of directions each avoiding an obstacle hit by a ray. Then by using these directions and a resolution mechanism that will be described later, a single moving direction is determined. Since the methodology, which makes our second algorithm handle the moving targets more successfully, is a generic technique, we also employed it in our first algorithm, and extended RTEF to handle moving targets better. In order to test our search algorithms, we also developed a prey algorithm, Prey-A*. Since our objective is not to test the prey algorithms, Prey-A* is developed to be an off-line algorithm, which is inefficient in terms of execution time, but powerful with respect to its escape capability. The agent (predator) and the prey algorithms are executed alternately in the experiments in order to prevent side effects that could be caused by the efficiency difference. We compared RTEF

and MTES with an on-line search algorithm, Moving Target Search (MTS) [15], and an off-line path planning algorithm, A* [38]. The experiments showed that MTES significantly over-performs RTEF and MTS, and competes with A*.

By increasing the number of predators involved in the environment, the single agent path search problem can be extended to a search against a static or moving prey with multiple coordinated agents. This problem introduces a recent research area called multi-agent pursuit, on which there is not much successful study done so far, especially against moving preys in environments with obstacles. Finally, we propose our last algorithm called 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. MAPS is built on MTES 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. The *blocking location* is fed as input to the path planner. And the latter strategy is performed after the path search for selecting the best estimated direction from the alternative moving directions, which 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.

Concerning the organization of the thesis, in Chapter 2, we briefly review the existing and related work on path planning, multi-agent pursuit and prey algorithms. In Chapter 3, we describe our first real-time search algorithm, Real-Time Edge Follow (RTEF) in details, and we introduce our second real-time search algorithm, Real-Time Moving Target Evaluation Search (MTES) in Chapter 4. In Chapter 5, we present our last algorithm, Multi-Agent Real-Time Pursuit (MAPS). And finally, we conclude our study and discuss the possible future research directions in Chapter 6.

CHAPTER 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 either informed or uninformed. Uninformed planning algorithms do not use any domain specific information in order to guide the search. Dijkstra's algorithm [44] is a well-known uninformed algorithm, which is complete and optimal. It works on general purpose weighted graphs, but commonly applied to path planning problems. For instance, Knuffner [31, 32] embedded Dijkstra's algorithm into his simulation system, where the aim was to find a collision free path between an initial and target point on a virtual 3D terrain. The terrain was divided into cells, which have vertical, horizontal and diagonal costs of walking through, and a weighted graph is constructed from the data in order to use Dijkstra's algorithm. Contrary to uninformed algorithms, informed ones use domain specific knowledge (heuristics) to increase efficiency. A* [38, 9] is one of the best-known efficient path planning algorithms, which is guided by an admissible heuristic function. In path planning domain, it is common to use euclidian or manhattan distance for the heuristic function.

The pre-mentioned algorithms are all deterministic, and will always find the same solution given the same input. But there are also some probabilistic off-line algorithms such as genetic algorithms, random trees and probabilistic roadmaps. Genetic algorithms [36, 43] encode candidate solution paths as chromosomes and make use of evolution meta-heuristics to find acceptable solutions. Each chromosome of the population is usually designed to be representing a single route, and may contain feasible and infeasible solutions together. Crossover causes the parts of these solutions

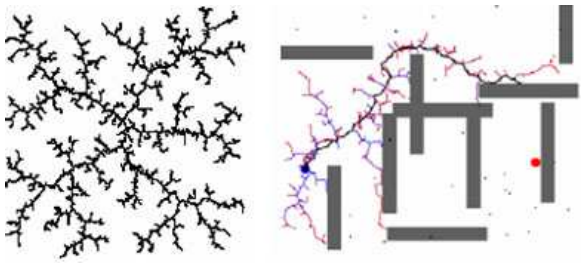

Figure 2.1: Random tree samples [2]

to be exchanged, and mutation causes the parts to be redefined and made smoother. Random tree based algorithms [3, 33, 34] search the target in obstacle-free space in randomly generated trees, which are illustrated in Figure 2.1. In the original method, a single source (agent location) is used to grow the random tree in order to reach the target point. Later on, the random tree approach is enhanced to use two source random trees starting from the agent and the target points in parallel, and try to catch an intersection among the points of distinct trees to find a path in between. Probabilistic roadmap algorithms [18, 39] use a three phased approach. They generate a connected graph (roadmap) randomly in obstacle-free space, then try to connect initial and target points to the roadmap, and finally search a path on the roadmap between initial and target points.

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 [26], 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* [41, 37], focused D* [42], and D* Lite [22, 23, 24] are some of the well-known optimal incremental heuristic search algorithms applied to path planning domain. In these algorithms, an initial optimal path is generated off-line, and then the agent is allowed to follow the path. During the navigation, the agent observes the environment, and if any environmental change is detected, the agent partially re-plans the existing solution. 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 exponential time and

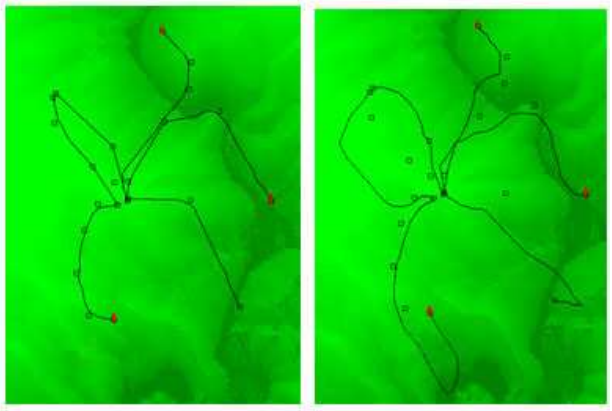

Figure 2.2: A path update sample: Initial paths of three agent groups (left), refined paths after some time considering several threats (right) [46]

does not meet real-time constraints. That's why these algorithms are considered as efficient off-line path planning algorithms.

There are also some incremental algorithms that can continuously refine given paths. Undeger's energy based algorithm [46] is such an instance. In this study, an off-line path is generated in the first step. Later on, the algorithm is let to continuously refine the path considering the environmental changes. A path update method based on energy minimization of path points commonly applied in image processing, called the *snakes*, is used. It is supposed that there are a number of energy sources such as threats that the agent should avoid. The basic idea is to escape from the energy sources and in parallel to keep the path as smooth as possible considering the obstacles. Figure 2.2 is an example run of the proposed algorithm. Although this approach is an efficient way of updating a path, the result is not satisfactory all the time, because this process can only modify a path locally, but not radically.

2.3 Single Agent 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 are proposed. Tangent-Bug [17] is one of the former heuristic search algorithms, which has some similarities with our proposed algorithm that will be described later on. It is based on the Bug algorithm [51], and uses vision information to reach the target. It constructs a local tangent graph (LTG), a limited visibility graph, in each step considering the obstacles in the visible set. The sensed obstacles are modeled as thin

walls and assumed to be the only obstacles in the environment. The agent moves to the locally optimal direction on the current LTG until reaching the target or detecting a local minimum (when hit an obstacle border). If a local minimum is detected, the agent switches to the border following mode, and move along the border until the distance to the target starts decreasing. After leaving the border, the agent switches to the first mode again. Although this approach seems to be similar to ours in the sense that it moves to locally optimal directions to go around the nearby obstacles and follows the obstacle borders, it only considers the obstacles in active visible set, and follows the boundaries while walking. But our approach can also consider obstacles known but not currently visible, and border following process is just performed in the mind of the agent, not physically executed.

Learning Real-Time A* (LRTA*), introduced by Korf [28], 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. The initial table values are set to *zero* and the agent is made to learn exact goal distances 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*) [28], 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. RTA* repeats the steps given in Algorithm 1 until reaching the goal.

Algorithm 1 An Iteration of RTA* Algorithm

1: Let x be the current state of the problem solver. Calculate $f(x') = h(x') + k(x, x')$ for each neighbor x' of the current state, where $h(x')$ is the current heuristic estimate of the distance from x' to a goal state, and $k(x, x')$ is the cost of the move from x to x'.
2: Move to a neighbor with the minimum $f(x')$ value. Ties are broken randomly.
3: Update the value of $h(x)$ to the second best $f(x')$ value.

Since original RTA* only considers immediate successors to determine the next

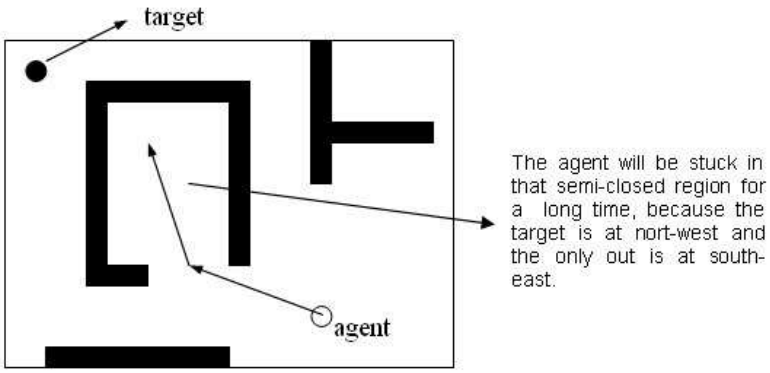

Figure 2.3: A heuristic depression sample: The agent, directed by RTA*, will be stuck in the semi-closed region shown in the figure for a long time searching the same region hopelessly until the heuristic depression is filled up completely.

move, it may be stuck in semi-closed regions for a long time because of the heuristic depression. A heuristic depression [15] is a set of states, which does not contain the goal state and has heuristic values less than or equal to those of a set of immediately and completely surrounding states. Thus it is a local maximum, whose heuristic values have to be filled up before the agent can escape from. The heuristic depression is illustrated on an example in Figure 2.3.

To reduce the effect of heuristic depression, RTA* can easily be extended to have any arbitrary look-ahead depth [28]. The structure of the algorithm is the same as RTA* except the computation of $h(x')$. Instead of computing the $h(x')$ of a neighbor cell x' only from the cell itself, $(n-1)$ level neighbors of x' are used, therefore the search space is expanded up to a predefined look-ahead depth (n). When the look-ahead depth is set to 1, the algorithm is the same as RTA*. Although the experiments showed that this improvement reduces the number of moves to reach the goal significantly, it becomes exponential in the look-ahead depth. Therefore, n-look-ahead depth with large n values is not preferred in practice.

Following the work of Korf, many variations of LRTA* and RTA* are proposed. Real-Time Horizontal A* (RTHA*) [46] is such an instance, which is only applicable to grid environments. Location and neighbor description of RTHA* is very different from common style, and the agent locations are not defined by the cell the agent is on, but defined as the row the agent is on. In addition, RTHA* does not accept the neighbor set as the immediate neighbor cells of the cell the agent is on, but accepts the neighbor set as the immediate neighbor rows of the row the agent is on. A row is

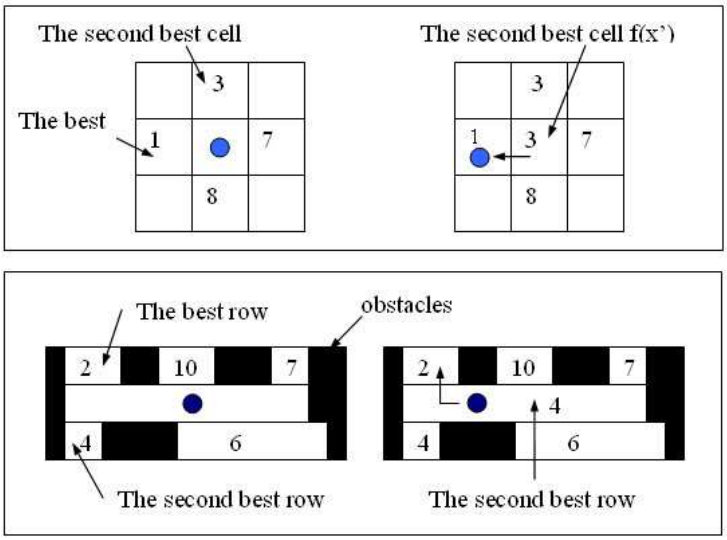

Figure 2.4: Comparison of RTA* (top) and RTHA* (bottom): RTA* estimates the costs of the neighboring cells of the agent whereas RTHA* estimates the costs of the neighboring rows of the agent [46].

described as a one dimensional horizontal chain of free cells, which is bounded from left and right sides by two obstacle cells. The difference between RTA* and RTHA* is illustrated on a typical example in Figure 2.4. Experimental results showed that RTHA* yielded better performance than RTA* with 1-look-ahead depth in complex environments. But a drawback was observed. The strategy realized by the algorithm was to usually move horizontally until coming almost the same column position as the target is on, then to move vertically. This was because of the heuristic function used, which underestimates the real-cost more in the horizontal axis than in the vertical axis.

Later on, Shimbo and Ishida introduced two other LRTA* variations, weighted LRTA* and upper bounded LRTA* [40], to control the amount of efforts to achieve a short-term goal (e.g., to safely arrive at a location in the current trial) and a long-term goal (to find better solutions through repeated trials). Weighted LRTA*, sacrifices the optimality property of LRTA* to reduce the overall amount of exploration, and to avoid intractability of large problem spaces. It has two versions, *ε-admissible* and *ε-additively admissible* weighted LRTA*. In *ε-admissible* weighted LRTA*, a constant $\varepsilon \geq 0$ is selected such that $h_{initial}(x) \leq (1+\varepsilon)h^*(x)$ for every state x in the problem space. Therefore, the solution costs do not exceed the optimal cost by more than a factor of $(1+\varepsilon)$. Similarly, in *ε-additively admissible* weighted LRTA*, a constant

$\varepsilon \geq 0$ is selected such that $h_{initial}(x) \leq h^*(x) + \varepsilon$ for every state x except the goal state in the problem space. In this case, the solution costs do not exceed the optimal cost by more than ε. On the other hand, upper bounded LRTA* provides an explicit upper bound to limit the amount of exploration and to reduce the memory usage in each trial meanwhile preserving the optimality and completeness. The algorithm introduces another heuristic function $u(x)$ along with $h(x)$, where $h(x)$ gives a lower bound and $u(x)$ gives an upper bound for exact cost $h^*(x)$. Initially, $u_0(x)$ is set to *zero* if x is a goal state, and infinite otherwise. As the initial upper bounds are infinite, the first trial is executed exactly the same as LRTA*. But in later trials, the upper bounds are learned and the amount of exploration is limited within the future trials. For state x, the upper bound of a neighbor y of x is updated as minimum of $u(y)$ and $k(x, y) + u(x)$, where $k(x, y)$ is the cost of moving from x to y.

Recently, Koenig proposed a new version of LRTA* that uses look-ahead depth more effectively to examine the local search space [21]. 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 until a point on the border of expanded states is reached or a previously unknown obstacle that blocks the way is discovered. Koening compared this LRTA* variation with D* Lite, which is an optimal incremental search algorithm. According to the experimental results, the planning time of his LRTA* variation is larger than the one of D* Lite for look-aheads of more than 45, and at that level, the path lengths are about 4.5 times longer than the ones produced by D* Lite. Therefore, it is only preferable if selected look-ahead is less than or equal to 45. Most recently, Koenig and Likhachev proposed a second version of this LRTA* variation called Real-Time Adaptive A* (RTAA*) [25]. 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. Although this modification reduces the quality of solutions for equal look-ahead depths, it increases the solution quality for equal planning times.

LRTA*(k) [13] is another recent version of LRTA*, which uses bounded propagation to update the heuristic estimates of up to k states, at the cost of extra com-

putation. Following the update of the heuristic value of the current state, LRTA*(k) propagates the change to the neighbor states, and continues with this propagation in a chain reaction until no change occurs in heuristics or k number of states are examined. In terms of first trial, convergence and solution stability, LRTA*(k) is reported to show a substantial performance improvement with respect to LRTA* and FALCONS [6], which is an older LRTA* version. Some other versions of LRTA* could be found in [45, 4, 7].

In the literature, there are also some probabilistic on-line search algorithms based on genetic algorithms [27], random trees [2] and probabilistic roadmaps [14] since a significant portion of the previous search data generated by these algorithms will be still valid after an environmental change, and can be used for deciding on the next move.

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) [15]. 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. MTS is described in Algorithm 2.

Algorithm 2 MTS Algorithm

When the problem solver moves:

1: Calculate $h(x', y)$ for each neighbor x' of x.
2: Update the value of h(x,y) with $max\{h(x, y), min_{x'}\{h(x', y) + 1\}\}$
3: Move to the neighbor x' with the minimum $h(x', y)$, i.e., assign the value of x' to x. Ties are broken randomly.

When the target moves:

1: Calculate $h(x, y')$ for the target's new position y'.
2: Update the value of h(x,y) with $max\{h(x, y), h(x, y') - 1\}$
3: Reflect the target's new position as the new goal of the problem solver, i.e., assign the value of y' to y.

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 [15]. 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.

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 these 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 [1], 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 [5], 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. The agents communicate their perception to their neighbors and use the communication information in their decision-making. As a result of the experiments, it was reported

that a satisfactory cooperative behavior is observed by employing 2LRL method in pursuit domain.

In [16], 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 [52], 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, and the action is executed. After receiving a reward (if any) from the environment based on the action taken, temporal difference formula is used to estimate the value of the next state, and the association of the current state and the chosen action is updated. The authors compared non-cooperative and cooperative (TD-FALCON) predator teams in a 16x16 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 [11, 12]. They employed strong typed genetic programming (STGP) to evolve

pursuit algorithms represented as Lisp S-expressions for predators and preys moving in a 30x30 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 [29].

Different from the work of Haynes and Sen, the generic algorithm is used by Yong and Miikkulainen [53] to evolve (and coevolve) neural network controllers, rather than program controllers. Coevolution 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 (coevolved) 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 100x100 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 [35], 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 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 the manhattan distance

to the prey, the predator is paid with a positive payment, and the reverse with a negative one. 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.

Another hand-crafted coordination strategy is proposed by Kitamura et al in [19]. Their coordination algorithm is build on a multi-agent version of RTA* called Multi-Agent Real-Time A* (MARTA*) [20], 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 120x120 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 [8], 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.

The last hand-crafted coordination strategy we will examine is the one proposed

by Kota et al [30]. 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 of the experiments, 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 [35, 16, 15, 8]. In the study of Ishida and Korf [15], 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 [16, 30], the prey is let to escape from the predators along a straight line or a circle. In [16], 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 [8], 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 a position that is not in line of sight of the predators and moving randomly, are used. There are also some studies focussing on evolving behavioral strategies [11, 12]. Since these reactive algorithms are not good enough to force our predator algorithms, we developed an off-line strategy, which is slow but more powerful.

CHAPTER 3

REAL-TIME EDGE FOLLOW

In this chapter, we introduce our first real-time path search algorithm, Real-Time Edge Follow (RTEF) [50, 47]. In the following sections, we state the problem description, and describe the algorithm in details. Later on, we examine the complexity of the algorithm, and introduce the search depth concept for bounding the complexity. Next, we prove the correctness of the algorithm, and finally present the experimental results.

3.1 Problem Description

In this section, we state the problem description in details. The objective is to search a static target in real-time with a single agent in a grid world environment. The assumptions of our domain are given as follows.

- The environment is a grid world, where any grid cell can either be free or obstacle.
- There is a single agent that aims to reach a static target.
- The agent and the target are randomly located far from each other in non-obstacle grid cells.
- The agent is expected to reach the target from a short path avoiding obstacles in real-time. The target's location is assumed to be known by the agent.
- The agent has limited perception, and is only able to sense the obstacles around him within a square region centered at the agent location. The size of the square is $(2v + 1)$x$(2v + 1)$, where v is the **vision range**. We used the term *infinite vision* to emphasize that the agent has unlimited sensing capability and knows the entire grid world before the search starts.

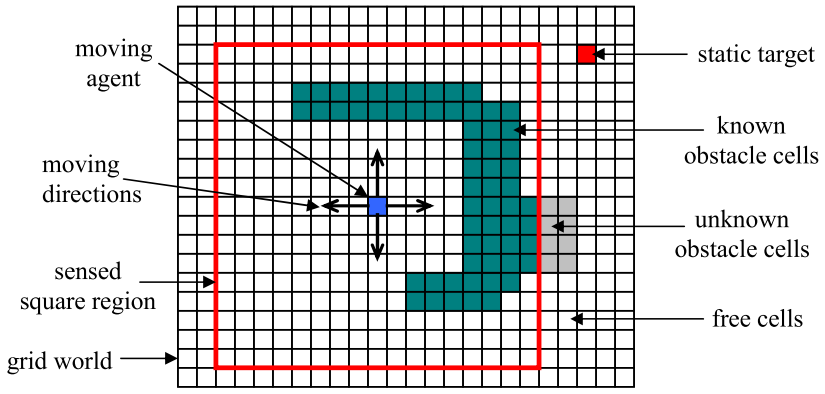

Figure 3.1: The problem description

- The unknown parts 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. When we say an obstacle, we refer to the known part of that obstacle (Figure 3.1).

- The agent can only perform four actions in each step; moving to a free neighbor cell in north, south, east or west direction. The environment is deterministic (that is, the effects of actions are all deterministic).

- The search continues until the agent reaches the target location.

An illustration of the problem description is shown in Figure 3.1.

3.2 The Search Algorithm

We developed Real-Time Edge Follow (RTEF) algorithm, which is able to search a path from an initial location to a target location in real-time. Although RTEF algorithm can be improved to handle moving targets, in this chapter, we assume that the target is static. The basic idea behind RTEF is to eliminate the closed directions not reaching the target point in order to determine which way to move. For instance, if the agent is able to realize that moving to north and east will not lead to the target, then he will prefer going to south or west. This sample case is illustrated in Figure 3.2.

In order to determine the closed directions and move accordingly, RTEF repeats the steps shown in Algorithm 3 until reaching the target or determining that the target is unreachable. RTEF internally uses the heuristic method, Real-Time Edge

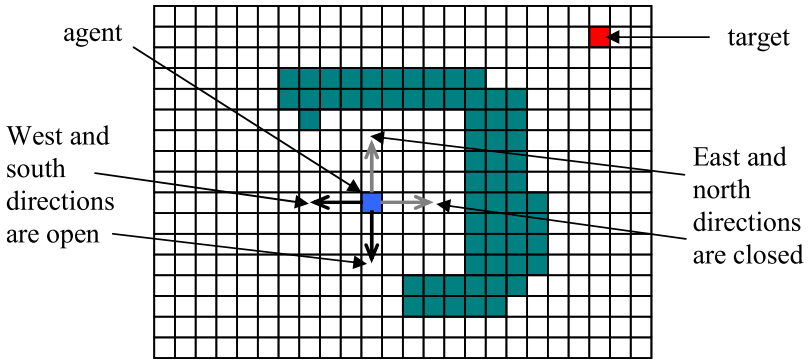

Figure 3.2: The problem description

Follow Alternative Reduction Method (RTEF-ARM), to find out open and closed directions and hence to eliminate non-beneficial movement alternatives. To avoid infinite loops and re-visiting the same locations redundantly, RTEF either uses *visit counts* or *history*, or both.

Algorithm 3 An Iteration of RTEF Algorithm

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

Definition 3.2.1 (Visit Counts) *The algorithm maintains the number of visits, visit count, to the grid cells. The agent moves to one of the neighbor cells in open directions with minimum visit count. If there exists more than one cell having minimum*

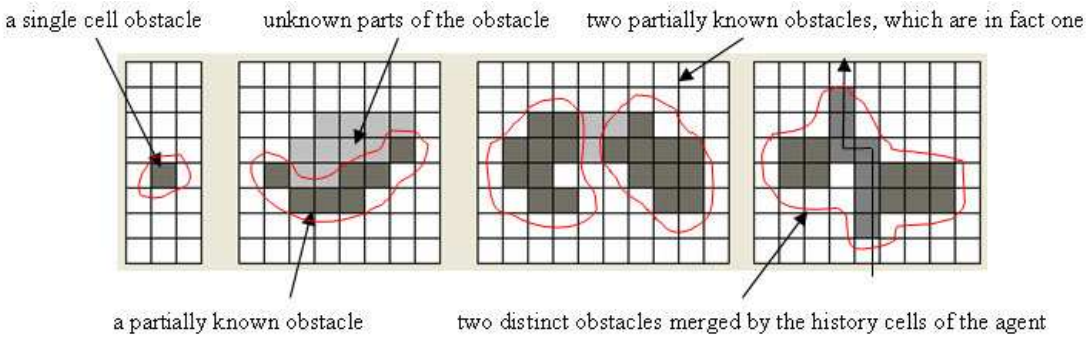

Figure 3.3: The obstacle samples

visit count, the one with the minimum Euclidian distance to the target is selected. If Euclidian distances are also the same, the ties are broken randomly.

Definition 3.2.2 (History) *The set of previously visited cells forms the history of the agent. History cells are treated as obstacles. During the exploration, if the agent discovers a new obstacle and realizes that the target became inaccessible due to history cells, the agent clears the history to be able to backtrack.*

Before describing the details of the algorithm, it is better to make clear what we mean by saying an obstacle. An obstacle in the mind of an agent is a set of neighboring grid cells (each either being marked as a real world obstacle cell or a history cell) that shape an object that has a border. In that respect, a single cell might also form an obstacle. Due the sensing limitation of the agent, sometimes just a part of a real obstacle might be discovered, in that case the known part will be considered as an obstacle, which may cause a single real world obstacle to be known as two different obstacles because of the missing environmental information. Furthermore, sometimes two or more real world obstacles can be merged and become a single obstacle because of history cells that neighbor the real objects. These cases are illustrated in Figure 3.3.

3.2.1 Real-Time Edge Follow Alternative Reduction Method

Real-Time Edge Follow Alternative Reduction Method (RTEF-ARM) is a sub-function of RTEF, which detects closed directions. The method sends four rays away from the agent in diagonal directions. The region between two adjacent rays forms a possible moving direction for the agent. Hence, the agent has four moving directions (north,

south, east and west). RTEF-ARM extracts the border of each obstacle hit by any ray and then analyzes the so-called regions to determine open and closed moving directions, as summarized in Algorithm 4.

Algorithm 4 RTEF-ARM Algorithm

1: Mark all the moving directions as open.
2: Propagate four diagonal rays.
3: **for** each ray hitting an obstacle **do**
4: Extract the border of the obstacle by starting from the hit-point and tracing the edges towards the left side until making a complete tour around the obstacle; and find out an island and an hit-point-island if exists.
5: Detect closed directions by analyzing the edges using the island, hit-point-island and the target.
6: If number of open directions is *zero*, stop with failure (target is unreachable).
7: **end for**

In RTEF-ARM, four diagonal rays splitting north, south, east and west directions are propagated away from the agent as shown in Figure 3.4. The rays go away from the agent until hitting an obstacle or maximum ray distance is achieved. The types of ray-hitting are exemplified in Figure 3.5. Four rays split the area around the agent into four regions. A region is said to be closed if the target is inaccessible from any cell in that region. If all the regions are closed then the target is unreachable from the current location. To detect closed regions, the borders of the obstacles that the rays hit are analyzed.

Definition 3.2.3 (Island) *If the edges on the border of an obstacle are traced by going towards left side starting from a hit-point (see Figure 3.6), we always return to the same point as illustrated in Figure 3.7. When a tour around the obstacle is completed, a polygonal area is formed as the border of the obstacle. We call this polygonal area an island (stored as a list of vertices forming the border of the obstacle). As shown in Figure 3.8, there are two kinds of islands:* **outwards-facing** *and* **inwards-facing islands**. *The target is unreachable from the agent location if the target is inside an outwards-facing island or outside an inwards-facing island.*

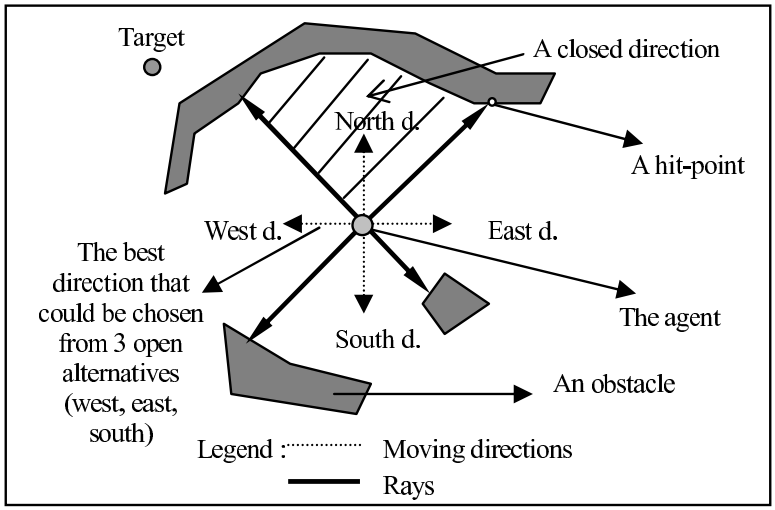

Figure 3.4: Propagating rays to split north, south, east and west directions

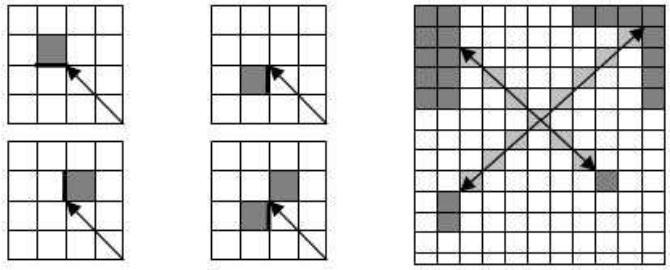

Figure 3.5: Ray propagation and hitting

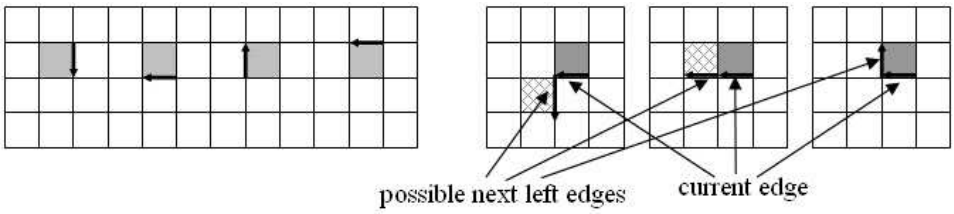

Figure 3.6: Finding next left edge: The figure on the left shows 4 possible current edges. The figures on the right illustrate all possible next states for a south current edge.

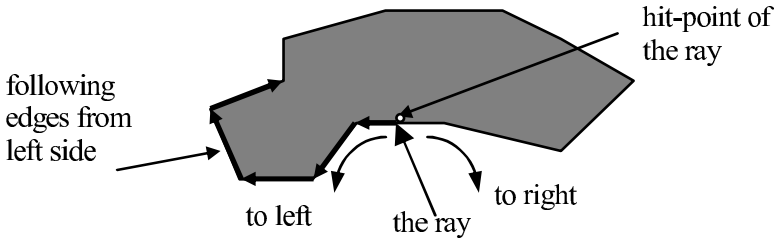

Figure 3.7: Identifying the obstacle border

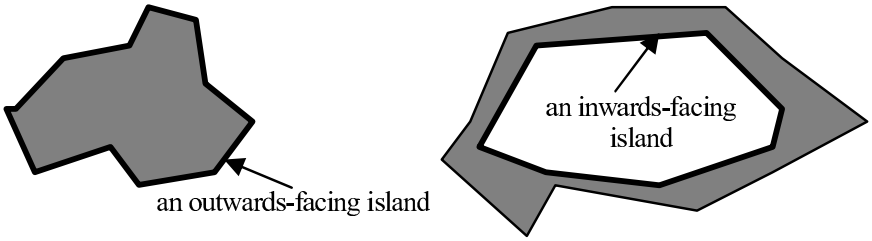

Figure 3.8: Island types: outwards facing (left), inwards facing (right)

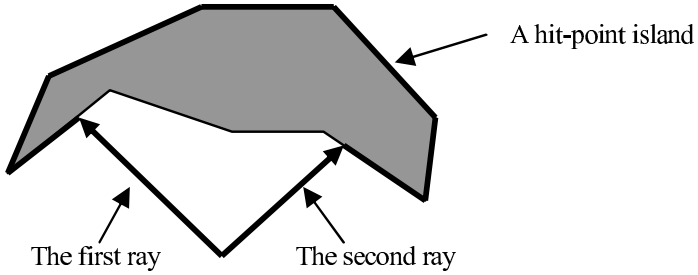

Figure 3.9: Two rays hitting the same obstacle at two different points form a hit-point island

Definition 3.2.4 (Hit-Point Island) *It is possible that more than one ray hit the same obstacle. As illustrated in Figure 3.9, an augmented polygonal area called hit-point-island is formed when we reach the hit-point of another ray on the same obstacle while following the edges. A hit-point-island borders one or more agent moving directions. If the target point is not inside the hit-point-island, all the directions that are bordered by the hit-point-island are closed; otherwise (the target is inside the hit-point-island) all the directions not bordered by the hit-point-island are closed. This is illustrated in Figure 3.10.*

Islands and hit-point-islands are stored as vertex lists and passed to the closed direction determination step shown in Algorithm 5. Note that function *isInside(x,y,p)*

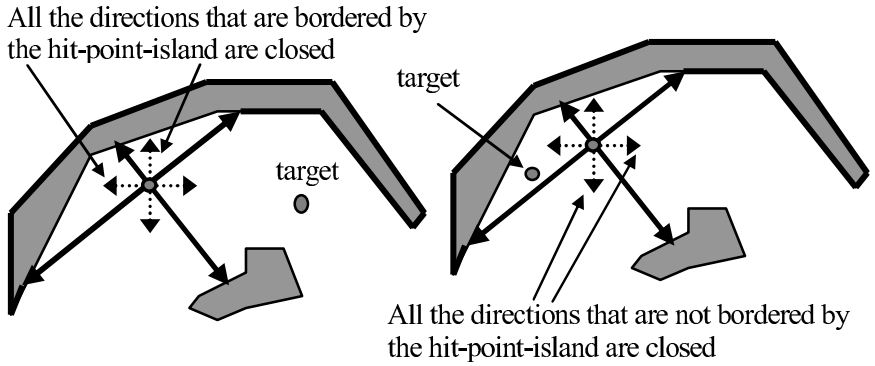

Figure 3.10: Analyzing hit-point islands and eliminating moving directions

returns *true* if coordinates (x, y) is inside polygon p and function *isClockwise(p)* returns *true* if the vertices of polygon p is ordered in clockwise direction (i.e., if polygon p is outwards facing with respect to the agent).

Algorithm 5 Determining Closed Directions

Require: (x, y) : coordinates of the target,

Require: i : the list of vertices forming the island border,

Require: h : the list of vertices forming the hit-point-island border,

1: **if** $isClockwise(i) = isInside(x, y, i)$ **then**
2: **Case 1:** Close the entire directions (the target is unreachable)
3: **else if** $|h| > 0$ **then**
4: **if** $isInside(x, y, h)$ **then**
5: **if** $isClockwise(h)$ **then**
6: **Case 2:** Close the directions between 1^{st} and 2^{nd} hit-points on i in counter clockwise direction
7: **else**
8: **Case 3:** Close the directions between 1^{st} and 2^{nd} hit-points on i in clockwise direction
9: **end if**
10: **else**
11: **if** $isClockwise(h)$ **then**
12: **Case 4:** Close the directions between 1^{st} and 2^{nd} hit-points on i in clockwise direction
13: **else**
14: **Case 5:** Close the directions between 1^{st} and 2^{nd} hit-points on i in counter clockwise direction
15: **end if**
16: **end if**
17: **end if**

In order to reduce the work load of closed direction determination step, unnecessary vertices are not inserted into the vertex lists of island and hit-point island during edge-tracing. Unnecessary vertices eliminated (exemplified in Figure 3.11) are the ones whose removal do not change the shape of the island or hit-point-island polygons.

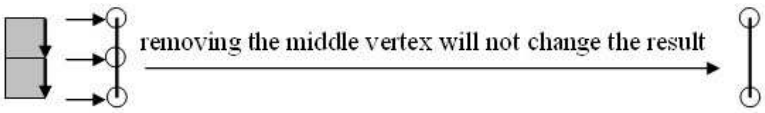

Figure 3.11: Eliminating the unnecessary vertices

3.2.2 Variations of Real-Time Edge Follow

The original RTEF algorithm uses visit counts and history together in order to guide the search, but there can be several other alternatives. In general, RTEF uses RTEF-ARM to find out open directions that possibly reach the target location. Later on, one of the open directions is selected for the next move. After performing the move, agent information is updated in order to prevent infinite loops. We developed several variations of RTEF by changing the technique used to select from open directions and update the agent information. These variations are described below:

1. *RTEF-History (RT-H) :* RTEF-History selects one of the open directions for the next move using the euclidian distance heuristic function. After moving to the next cell, the previous cell is inserted into the history. The history cells are considered as obstacles in RTEF-ARM, therefore the visited cells are never visited again unless the history is cleared. If the grid is fully known, the history is never cleared during the search and the target is reached (if a solution exists) without revisiting any previously visited cells. But if the grid is not fully known, some of the newly discovered obstacles may block the way the agent is going, so all or a part of the history is needed to be cleared in order to back track. The RTEF-History is the RTEF version, which clears all the history when the agent is completely blocked by a newly discovered obstacle.

2. *RTEF-History-BC (RT-HBC) :* RTEF-History-BC, where BC is an abbreviation for "Border Clear", is the same as RTEF-History except the history clearing method. When the agent is completely blocked by a newly discovered obstacle, RTEF-History-BC only clears the history cells encountered during the last edge-tracing phase. The reason behind this idea can be summarized as follows. Although the algorithm finds no path reaching the target, this does not mean the target is not reachable if the history is not empty yet, because the history cells may be blocking the way. If this is the case, failing of the last search must partially or completely be because of these history cells. Thus the algorithm

goes over all the edges followed in the last edge-tracing phase, and clears all the history cells on these edges. This edge-tracing and history-clearing loop must continue until the last history edge blocking the agent is cleared or no history cell could be found to clear.

3. *RTEF-Visit Count (RT-VC) :* This is a variation that does not use any history. To prevent infinite loops, number of visits is stored for each cell. Thus the algorithm selects the neighbor cell having the minimum visit count value. If there are more than one minimum, the algorithm uses the euclidian distance heuristic function. If they are also the same, one of them is selected randomly. After moving to the next cell, the visit count of the previous cell is increased by one.

4. *RTEF-Visit Count-History (RT-VCH) :* This is a mixture of RTEF-Visit Count and RTEF-History. The algorithm uses both visit counts and history together and clears all the history when the agent is completely blocked by a newly discovered obstacle.

5. *RTEF-Visit Count-History-BC (RT-VCHBC) :* This is a mixture of RTEF-Visit Count and RTEF-History-BC. The algorithm uses both visit counts and history together and only clears the history cells found in the last edge-tracing phase when the agent is completely blocked by a newly discovered obstacle.

6. *RTEF-RTA* (RT-RTA*):* This is the RTEF version, which performs RTA* integrated with RTEF-ARM. The algorithm uses the heuristic estimation update method of RTA* to guide the search and prevent infinite loops, and uses RTEF-ARM to reduce the possible moving directions. The closed directions found are considered as obstacles from the view point of the cell where the RTEF-ARM is executed, therefore a cell might be evaluated differently from the neighbors of that cell. This condition forces us to make a little change in the heuristic estimation update of RTA*. RTA* sets the second best heuristic estimation of the neighbors to the current cell before moving to another one. If there is just one alternative to select, the second best estimation will be infinite. If this is the case and there are some directions closed by RTEF-ARM, we should not set the second best (infinite) to the current cell, but set the best one, because there is

a two way communication with these closed cells. Although you may not move to these neighbors from the current cell, you may go these neighbors from some other cells and may need to move to the current cell in order to escape from the closed area.

7. *RTEF-RTA*-Penalty (RT-RTA*-p) :* This one is similar to RTEF-RTA* except a penalty extension. As mentioned before, the closed directions found are considered as obstacles from the view point of the cell where the RTEF-ARM is executed. So a cell might be evaluated as an obstacle from the view point of some of the neighbor cells but that may not be the case for some others. To prevent these non-promising cells from being visited from some other neighbors of the cell, the algorithm gives a penalty to the heuristic estimation of these cells according to a pre-defined rule. The main motivation of the rule comes from the fact that the heuristic estimation of a cell inside a closed region cannot be better than the minimum heuristic estimation computed among all the other neighbors, because the algorithm prefers the most-promising cell having the minimum heuristic estimation rather than the cell inside the closed region. Therefore the heuristic estimation of the neighbor cells inside the closed regions can be set to the minimum heuristic estimation of the open ones if these closed ones have smaller heuristic estimations. We call this technique as RTEF-RTA*-penalty-0 (RT-RTA*-p0). If we add a penalty value (n) to the minimum heuristic estimation, it is called RTEF-RTA*-penalty-n (RT-RTA*-pn).

3.3 Complexity Analysis

The most time consuming phases of RTEF are the *edge-tracing* and the *border-analyzing* phases. In the edge-tracing phase, the edges of at most four obstacles are traced in just one pass. For each edge point, the point is checked with the hit-points of three other rays in order to detect any other hit to the same obstacle. This reduces the efficiency in a constant manner. The point is than inserted into a list in constant time. Therefore the worst case complexity of this phase is proportional to the number of edges of the largest obstacle in the environment. In the border-analyzing phase, the vertices on the islands and hit-point islands are analyzed in several passes. Since the number of vertices of islands and hit-point-islands can be at most one plus the

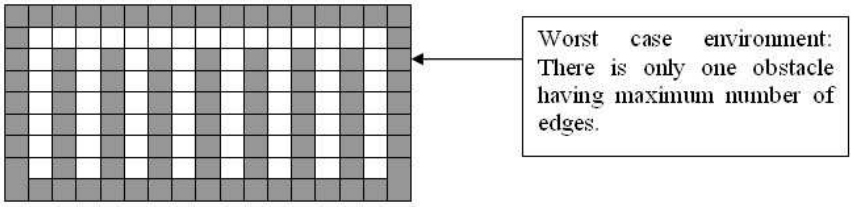

Figure 3.12: The worst case environment for a 17x9 sized grid: There is only one obstacle and it has the maximum number of edges that is possible.

number of edges of the largest obstacle in the environment, the worst case complexity of this phase is also proportional to the number of edges of the largest obstacle in the environment. Thus we can formulate the complexity of RTEF in a formal way if we can describe the number of edges of the largest obstacle in the environment. Having the assumption that the environment has a rectangle-shaped area, the largest obstacle could have at most $(w-1).(h-1)$ edges (exemplified in Figure 3.12), where w and h are the width and height of the rectangle respectively. Therefore the worst case complexity of RTEF for each move becomes $O(w.h)$.

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 obstacle size 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. Note that such worse cases are possible in 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 RTEF. A search depth is a rectangular area of size $(2d+1)$x$(2d+1)$ centered at agent location, which makes the algorithm treat the cells beyond the rectangle as non-obstacle. This extension can easily be obtained by just adding a single constraint check. With this limitation, the complexity of RTEF becomes $O(d^2)$.

RT-VC, RT-RTA* and RT-RTA*-p works fine with the search depth limit since they do not use history so there is no risk of blocking the gateways, and cost matrix of RTA* or the visit counts will never be cleared during the search. RT-VCH also works with this extension because the agent inserts every cell it has visited into the history. As a result, the free cells will be marked as history one by one, and at a particular time, the agent will eventually be able to detect that it is blocked, and clear all the history opening all the blocking cells.

RT-H and RT-HBC that use only history to prevent loops can go into an infinite loop with this extension. Without a depth limit, the history is only cleared when an unknown obstacle is detected. Therefore in the worse case the history clearing will continue until every unknown obstacle is learned. But when we include a depth limit, the history may need to be cleared not just because of an unknown obstacle but also because of a known large obstacle that does not fit into the depth limit rectangle as a whole. Additionally, RT-VCHBC does not work with this extension, because the algorithm may not be able to clear all the history-cells required to open a blocking cell. The search depth may prevent extracting the entire border of a blocking obstacle, thus the border-clear technique may not be able to detect that it is stuck in the area due to that obstacle, or if it detects, it will not be able to clear all the history.

3.4 Proof of Correctness

To show that the algorithm is correct, it will be enough to prove that RTEF-ARM only closes the directions not leading to the target. RTEF-ARM has four phases: *initialization*, *ray-sending*, *edge-tracing* and *border-analyzing*. In the initialization phase, all the directions are set to open.

In the ray-sending phase, four rays are propagated in four diagonal directions until hitting an obstacle or maximum ray distance is reached. If a ray does not hit an obstacle, it will not be used in the rest of the phases, therefore it is not possible to cause a direction to be closed. It is also valuable to show that whether the ray hits something or not, the agent can follow the ray from both sides of the ray until reaching the end of the ray (assuming that the environment does not change). In Figure 3.13, the narrowest corridor that a ray can pass through is illustrated and the routes reaching the end point of the ray are shown, which completes the proof that if a ray can pass through a corridor, than an agent can also pass through the same corridor from both sides of the ray.

In the edge-tracing phase, we must show that edge following is a finite process, an island could always be found, and a hit-point island can be detected if it exists. Indeed the proof is clear because every obstacle should have a finite size border even it is partially known. If it is partially known, then we just use its known border. Since every obstacle has a finite size border, if we trace the edges of the obstacle

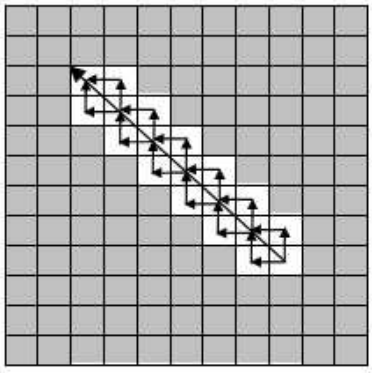

Figure 3.13: The narrowest corridor that a ray and an agent can pass through

in only left direction, we always reach the same point we started. Therefore, the process is bounded and an island could always be found, which forms the border of the obstacle. And if there exists a hit-point island (two or more rays hit the same obstacle), we always detect it because the algorithm follows every edge of the obstacle until returning to the same point and if another ray hits the same obstacle, the hit point must be on one of the edges that are followed. If there exits more than one hit-point island on the same obstacle, the algorithm will use only one of them. It does not matter whether we use the first one or the last one detected, but only one of the selection technique should be applied continuously in order not to miss some of the available information.

In the border-analyzing phase, we must show that the algorithm could find all the possible closed states and never decides to close a direction that should not be closed. Revisiting the Algorithm 5, we observe five cases that could close a direction. The rest of the cases that are not mentioned are all accepted as *Case 0* meaning that no conclusion is reached.

Case 1: If the ray hits the outer border of an obstacle, the agent must be outside of the obstacle and the island must be outwards facing (clockwise oriented). If the agent is outside the obstacle and the target is inside the obstacle, then it is clear that target cannot be reached from any of the directions. If the ray hits the inner border of an obstacle (counter clockwise oriented), the agent must be inside the obstacle. If the agent is inside the obstacle and the target is outside the obstacle, then this is also clear that target cannot be reached from any of the directions. These two cases are illustrated in Figure 3.14.

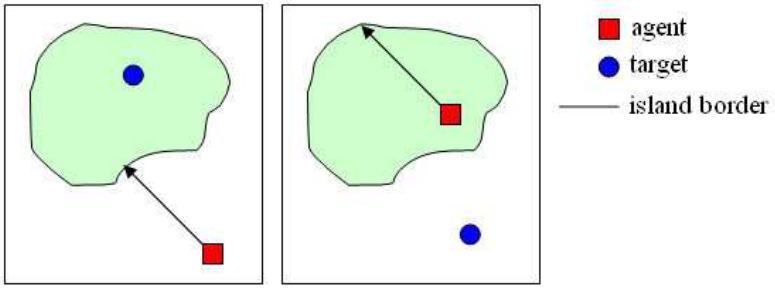

Figure 3.14: Case 1 - unreachable targets: Outwards-facing-island with a target inside (left), inwards-facing-island with a target outside (right)

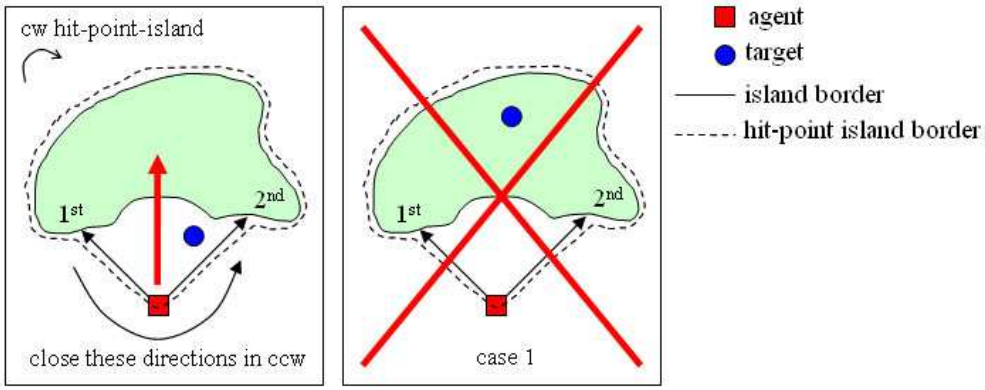

Figure 3.15: An illustration of Case 2 (left), an illustration seems to be Case 2 but already covered by Case 1 (right)

Case 2: The algorithm could only enter here if *Case 1* is not satisfied and there is a hit-point island found. To reach *Case 2*, the target must be inside the hit-point island and the hit-point-island must be clockwise oriented. Thus we find out that we must go into the region where the hit-point-island is bordering, which is illustrated in Figure 3.15. In this case, we must close the directions between the analyzed ray (ray of the first hit-point) and the other ray (ray of the second hit-point) in counter clockwise direction. Note that the case on the right of Figure 3.15 is not an example of this case because it is already covered by *Case 1*. From the figure, it is clear that target cannot be reached from any of the directions that are not inside the hit-point-island.

Case 3: The algorithm could only enter here if *Case 1* is not satisfied and there is a hit-point island found. To reach *Case 3*, the target must be inside the hit-point island and the hit-point-island must be counter clockwise oriented. Thus we find out that we must go into the region where the hit-point-island is bordering, which is illustrated

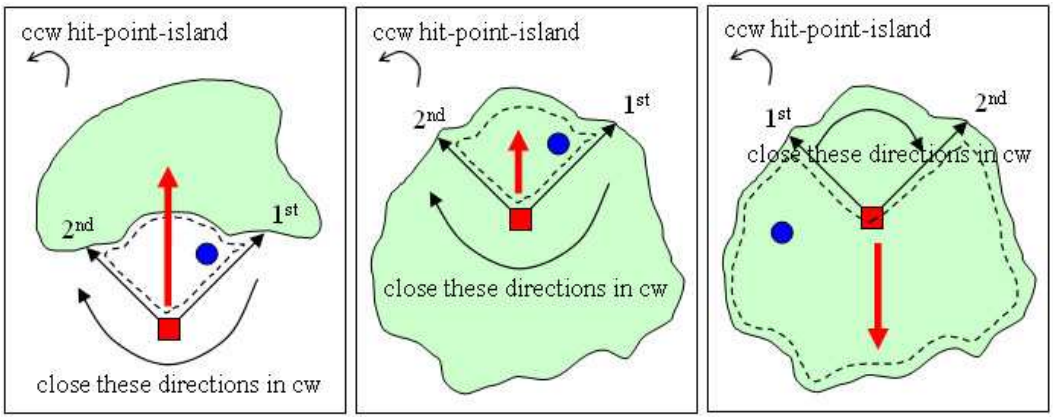

Figure 3.16: Illustrations of case 3

in Figure 3.16. In this case, we must close the directions between the analyzed ray (ray of the first hit-point) and the other ray (ray of the second hit-point) in clockwise direction. From the figure, it is clear that target cannot be reached from any of the directions that are not inside the hit-point-island.

Case 4: The algorithm could only enter here if *Case 1* is not satisfied and there is a hit-point island found. To reach *Case 4*, the target must be outside the hit-point island and the hit-point-island must be clockwise oriented. Thus, we find out that we must leave the region where the hit-point-island is bordering, which is illustrated in Figure 3.17. In this case, we must close the directions between the analyzed ray (ray of the first hit-point) and the other ray (ray of the second hit-point) in clockwise direction. From the figure, it is clear that target cannot be reached from any of the directions that are inside the hit-point-island.

Case 5: The algorithm could only enter here if *Case 1* is not satisfied and there is a hit-point island found. To reach *Case 5*, the target must be outside the hit-point island and the hit-point-island must be counter clockwise oriented. Thus, we find out that we must leave the region where the hit-point-island is bordering, which is illustrated in Figure 3.18. In this case, we must close the directions between the analyzed ray (ray of the first hit-point) and the other ray (ray of the second hit-point) in counter clockwise direction. From the figure, it is clear that target cannot be reached from any of the directions that are inside the hit-point-island. There are three other cases that look like *Case 5*, but in fact covered by *Case 1*. These cases are illustrated

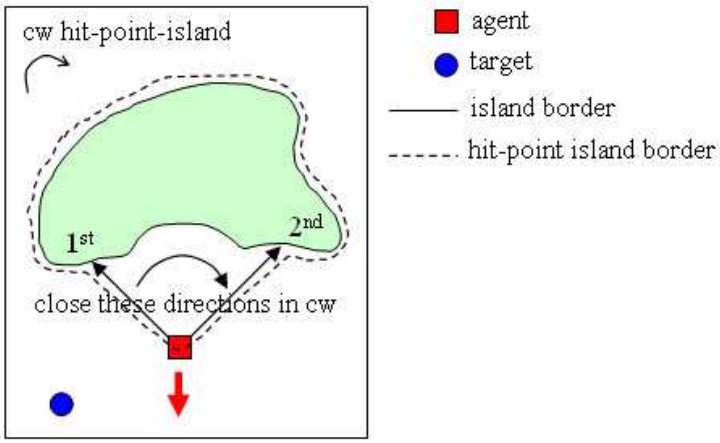

Figure 3.17: An illustration of case 4

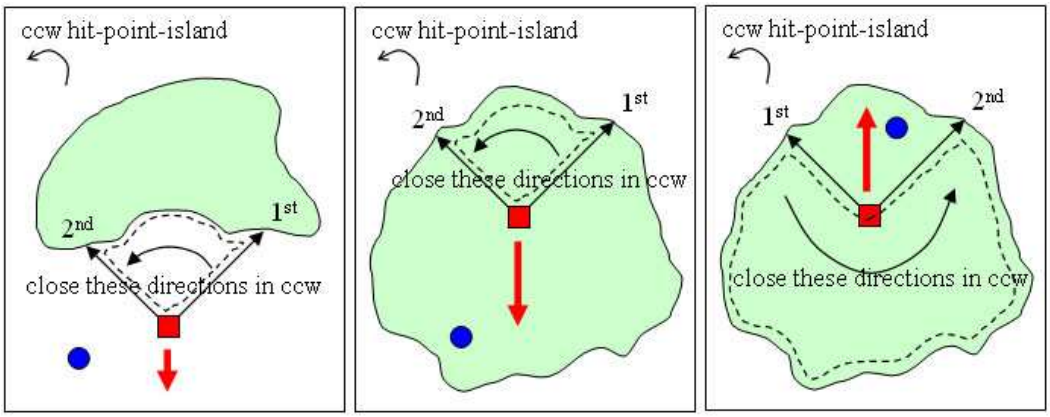

Figure 3.18: An illustration of case 5

in Figure 3.19.

In the above cases, we have shown that all directions that are closed by the algorithm are feasible. But we must also show that the algorithm could find all the possible closed directions. We enumerated all possible situations and determined the case each fits into. For this we constructed two tables. Table 3.1 contains four situations where no hit-point-island is found. As a result we can either conclude that the target is unreachable (*Case 1*) or say nothing (*Case 0*). Table 3.2 contains the situations where a hit-point-island is found. Note that some situations in the table have no real world interpretations (named "impossible" in the table); hence they are not considered by the algorithm. Thus the proof is completed.

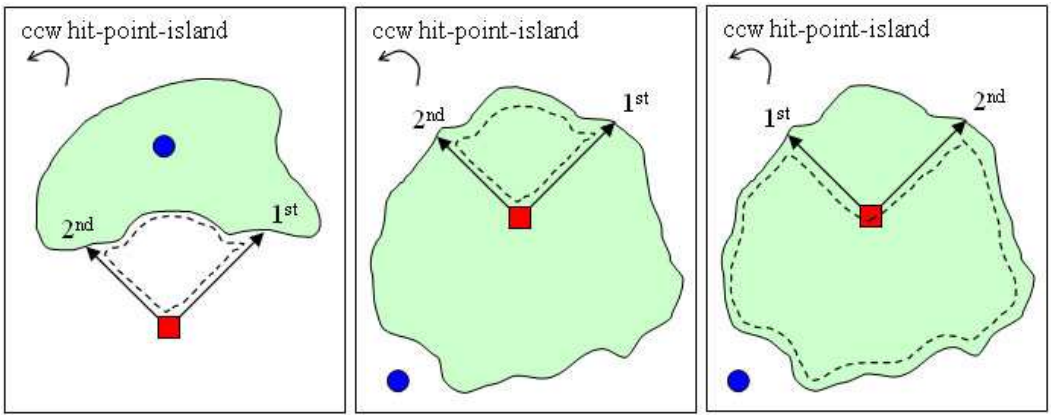

Figure 3.19: Illustrations seem to be case 5 but covered by case 1.

Table 3.1: The cases where no hit-point-island is found

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

Table 3.2: The cases where there is a hit-point-island

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

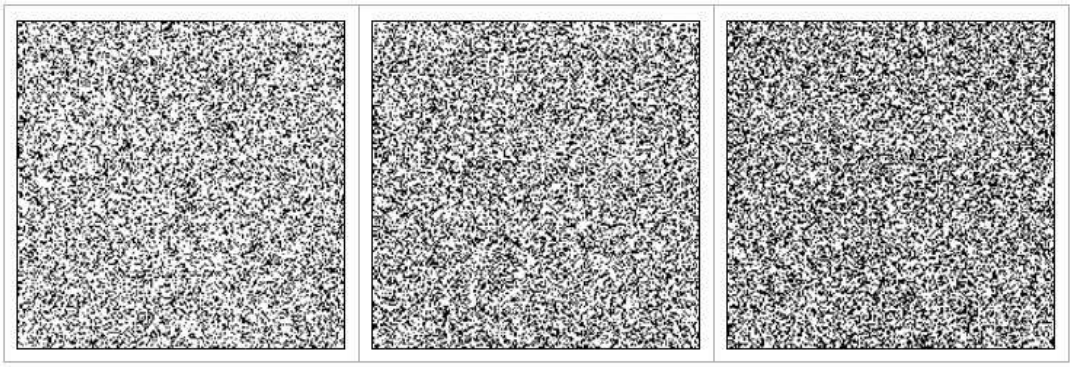

Figure 3.20: Random grids with 30% (left), 35% (middle) and 40%(right) obstacle ratios

3.5 Experimental Results

In this section, we report the comparison results of RTEF and RTA*. We used RTA* as the basis and evaluated performance of various RTEF algorithms on 3 different types of grid worlds (*random*, *maze* and *U-type*). We randomly generated 16 grids of size 200x200, and tested the algorithms on a Centrino 1.5 GHz laptop.

Random grids are generated randomly based on a specified obstacle ratio (the percentage of the obstacle cells). Figure 3.20 contains three random grids used in our experiments, which are generated with obstacle ratios 30%, 35% and 40%.

Maze grids are the ones where every two non-obstacle cells are always connected through a path (usually one path). Two parameters, obstacle ratio and corridor size (the minimum corridor width in the maze), are used to produce mazes. The corridor size effect is obtained by scaling small sized mazes (e.g., scaling a 50x50 maze by 4, we obtained a 200x200 maze). 9 different maze grids shown in Figure 3.21 are used in our experiments, which are generated using obstacle ratios 30%, 50% and 70%, and corridor sizes 1, 2 and 4.

U-type grids are created by randomly putting U-shaped obstacles of random sizes. Taking into consideration the number of U-type obstacles, minimum and maximum width and height of U-shaped obstacles, we generated 4 different U-type grids shown in Figure 3.22. The number of U-type obstacles 30, 50, 70 and 90, and minimum/maximum U-type obstacle sizes 5 to 50 are used in our experiments.

Initial locations for the agent and target pairs are randomly generated for each grid type such that the distance between them is at least half of the grid size. To ensure this, the grid is divided into 4 columns, and the agent coordinates are selected from the

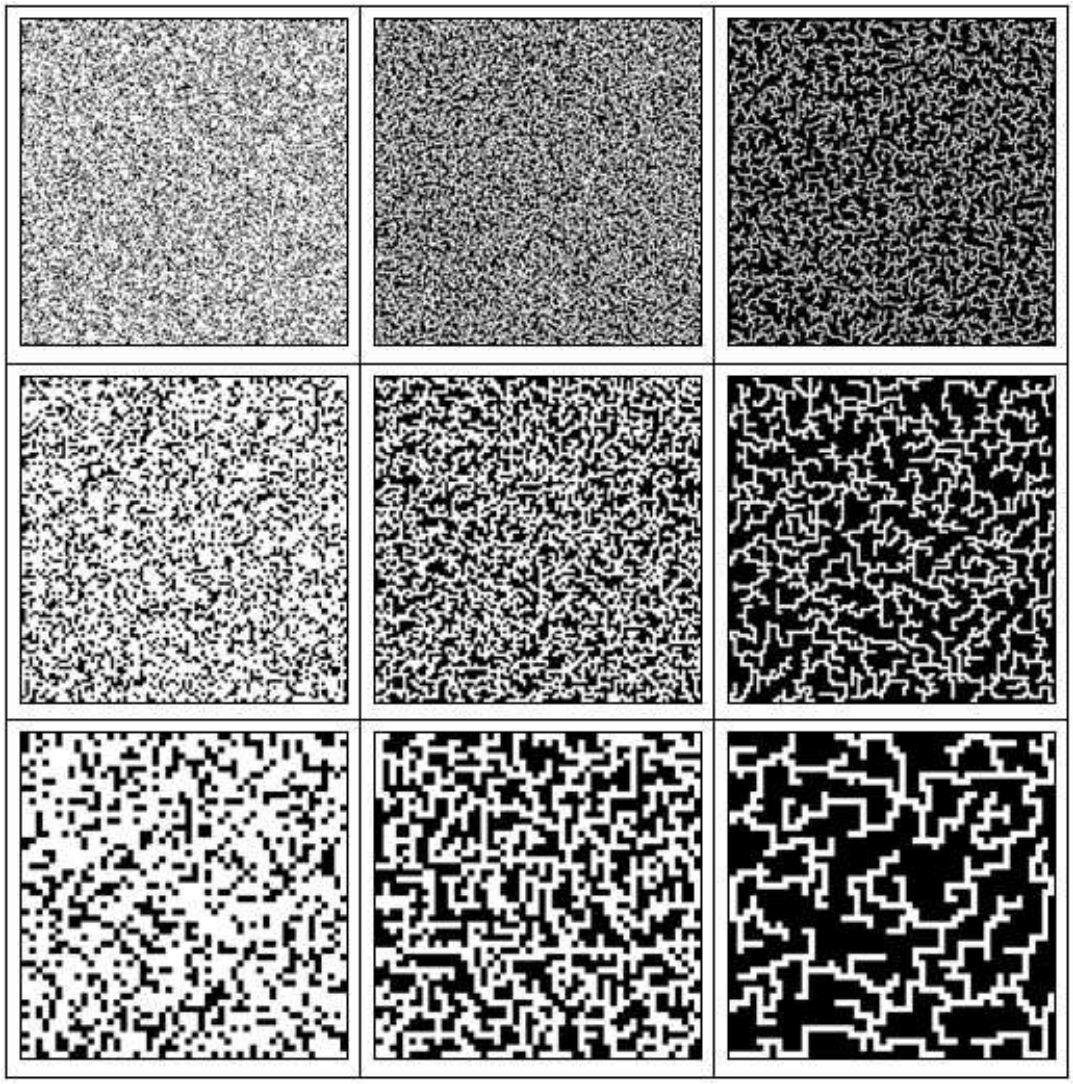

Figure 3.21: Maze grids with 30% (left column), 50% (middle column) and 70%(right column) obstacle ratios, and with 1 (top row), 2 (middle row) and 4 (bottom row) sized corridors

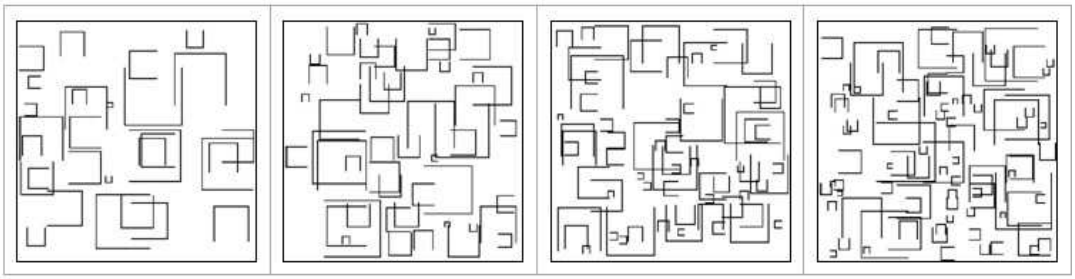

Figure 3.22: U-type grids with the number of U-type obstacles 30, 50, 70 and 90, respectively

most left column and the target coordinates are chosen from the most right column. These locations are kept the same in all the experiments with different configurations, for fairness.

The agents can perceive the environment up to a limited vision range (v). Therefore, being at its center, the agent can only sense the cells within a square area of size $(2v + 1)$x$(2v + 1)$. We also consider the cases that the agent has unlimited sensing capability (*infinite* vision range), which means that the agent knows the entire environment in advance. The vision ranges 10, 20, 40 and *infinite* are used throughout all the experiments.

3.5.1 Comparison of RTA* and RTEF

We tested 12 different algorithms (11 RTEF variations + RTA*) in 16 grids with 4 different vision ranges (10, 20, 40, infinite). Thus, 768 test configurations were generated, and 10 runs were performed for each configuration, making 7680 runs in total. Original RTA* with 1 look-ahead depth was used in all the experiments. The results demonstrate that RTEF finds much shorter paths in almost all the tested configurations. In terms of total execution time, RTEF seems to be better than RTA* in most of the tested configurations, although execution time per move is quite high.

With respect to path lengths, Figures 3.23 to 3.26 show the performance of RTEF algorithms compared to RTA* considering all the grids, the grids of different types, the agents with different vision ranges, and the maze grids with different corridor sizes, respectively. In the charts, the horizontal axis is the RTEF algorithms, and the vertical axis contains the ratio of improvement in the path length with respect to RTA* (the path length of RTA* divided by that of the compared algorithm).

The experiments showed that RTEF algorithms perform better than RTA* in all the grids. This was expected since RTEF is an improvement over RTA* without any drawbacks. Furthermore, the most beneficial improvement is obtained in maze, next in U-type and then in random grids. This is due to the fact that the difference between the sub-optimal solutions of RTA* and the optimal ones is the maximal in maze grids. Increasing the *vision range* yields better solutions (shorter paths) for RTEF over RTA* because RTEF is able to make use of environmental information. RTEF performs much better than RTA* when the *corridor size* increases since wider corridors increase the average branching factor and the area of heuristic depression

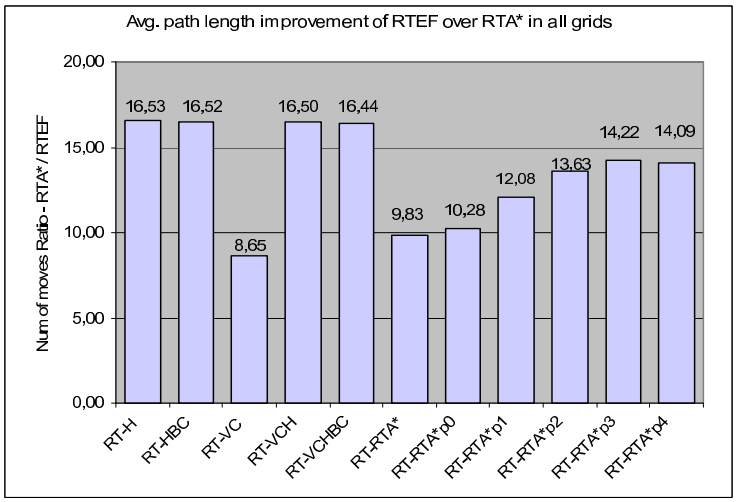

Figure 3.23: Ratio of improvement in the path length with respect to RTA* in all the grids.

to be filled up. When the branching factor is high, RTA* has lots of alternatives to pursue while the RTEF-ARM is able to classify the alternatives intelligently. RTEF algorithms integrated with history performs the best because history cells are merged with real obstacles yielding the larger obstacles that cause the agent to be more explorative.

For the variation RT-RTA*-p, penalty value 3 seems to be the best. RT-VC, RT-RTA* and RT-RTA*-p (penalty<2) perform the worst. The best improvement ratio (81.63) was encountered in maze grids with 50% obstacles and corridor size 2, due to the difficulty level of maze and the high branching factor. The worst improvement (1.08) was encountered in maze grids with 30% obstacles and corridor size 1. The second worst was 1.26 in random grids with 30% obstacles.

In terms of total execution times, Figures 3.27 to 3.30 show the speed-up obtained by RTEF algorithms compared to RTA* considering all the grids, the grids of different types, the agents with different vision ranges and the maze grids with different corridor sizes, respectively. In the charts, the horizontal axis is again the RTEF algorithms, and the vertical axis contains the ratio of improvement in the total execution time with respect to RTA* (the total execution time of RTA* divided by that of the compared algorithm).

According to the experiments, we can conclude that RTEF algorithms perform much better than RTA* in U-type grids. This is due to the decrease in both path lengths and the cost of RTEF-ARM (U-type grids have shorter obstacle borders).

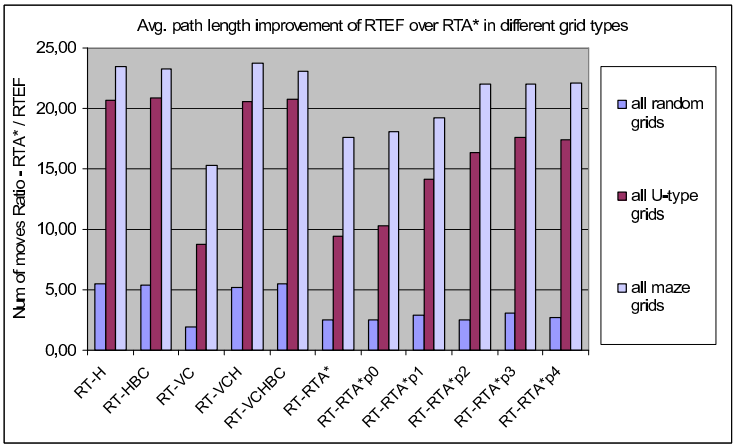

Figure 3.24: Ratio of improvement in the path length with respect to RTA* in maze, random and U-type grids.

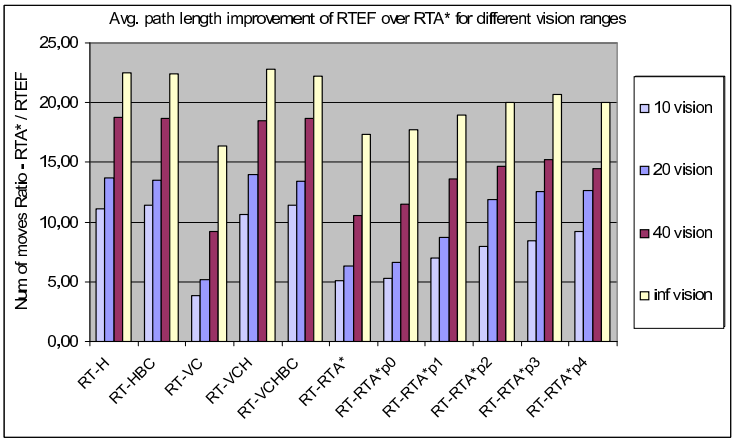

Figure 3.25: Ratio of improvement in the path length with respect to RTA* using vision ranges: 10, 20, 40 and infinite.

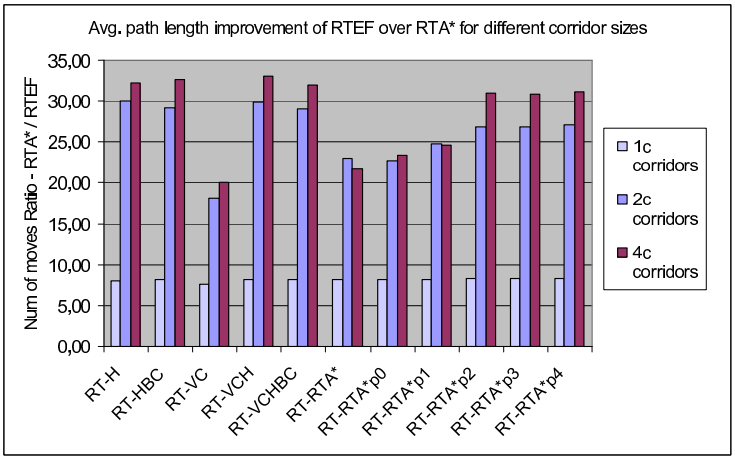

Figure 3.26: Ratio of improvement in the path length with respect to RTA* in grids with corridor sizes: 1, 2 and 4. Note that *c* stands for cell in the legend.

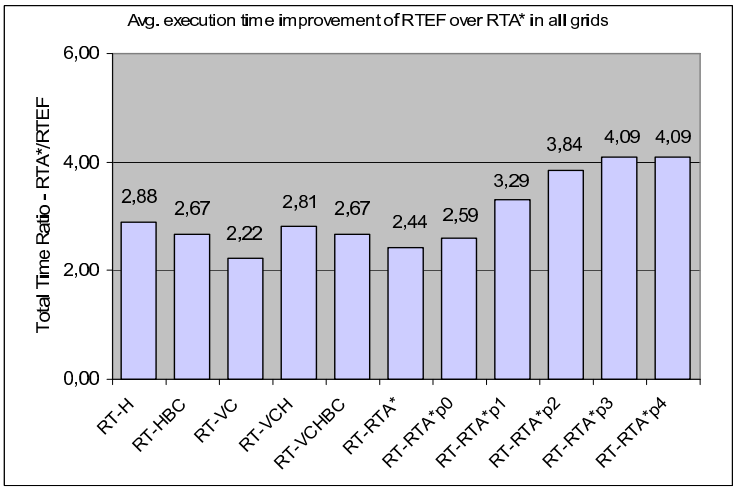

Figure 3.27: Ratio of improvement in the total execution time with respect to RTA* in all the grids.

Increasing the vision range is beneficial up to a point; and having the complete grid information (infinite vision) does not bring more efficiency because knowing the entire obstacle borders, which may not be useful all the time, makes RTEF-ARM costly. This can be easily seen in the experiments reported in Figure 3.29. When the corridor size gets larger, the performance difference of RTEF over RTA* gets also larger since RTEF-ARM becomes less costly, and the average branching factor increases making RTA* worse. On the average, RTEF algorithms perform better than RTA* in maze and U-type grids, which are the most difficult ones. Since random grids are the easiest grids for RTA*, RTEF algorithms generally perform worse than RTA* with respect to total execution time.

The variation RT-RTA*-p3 seems to be one of the most efficient RTEF variation and also returns acceptably good solution paths. Penalties greater than 3 did not bring any performance improvement, but even a reduction in some cases. Although history computations are costly, RTEF algorithms with history take less execution time due to their ability to return the shortest solution paths. The RT-VC, RT-RTA* and RT-RTA*-p0 are the most inefficient algorithms. The best average speed-up (15.26) was encountered in U-type grids with vision range 40. And the worst speedup (0.11) was in random grids with infinite vision range since random grids were easy for RTA* and knowing the entire obstacle borders due to infinite vision increases the RTEF-ARM cost.

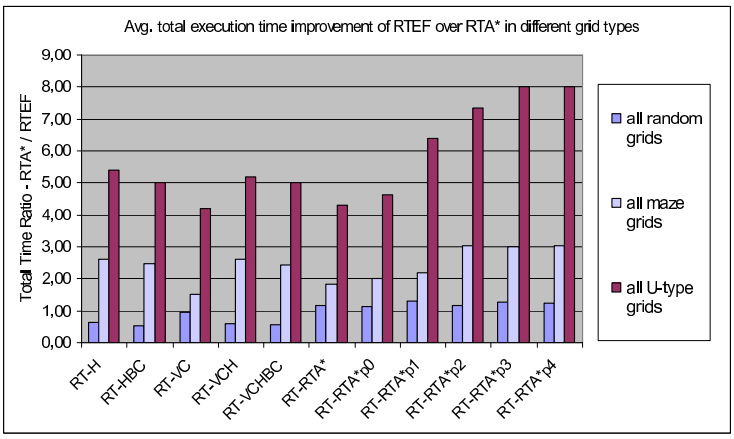

Figure 3.28: Ratio of improvement in the total execution time with respect to RTA* in random, maze and U-type grids. Note that the ones below 1 are not an improvement for RTEF.

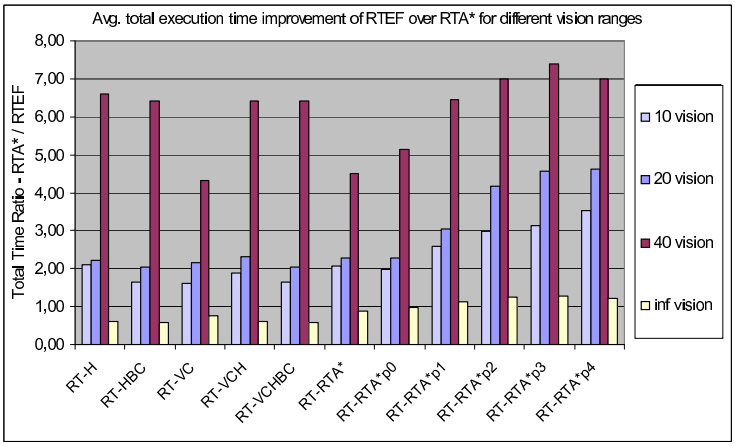

Figure 3.29: Ratio of improvement in the total execution time with respect to RTA* with 10, 20, 40 and infinite vision ranges. Note that the ones below 1 are not an improvement for RTEF.

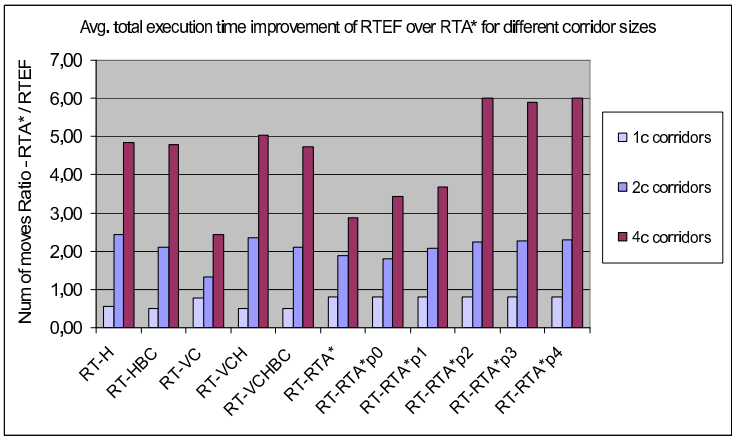

Figure 3.30: Ratio of improvement in the total execution time with respect to RTA* with 1, 2 and 4 cell corridor sizes in mazes. Note that the ones below 1 are not an improvement for RTEF.

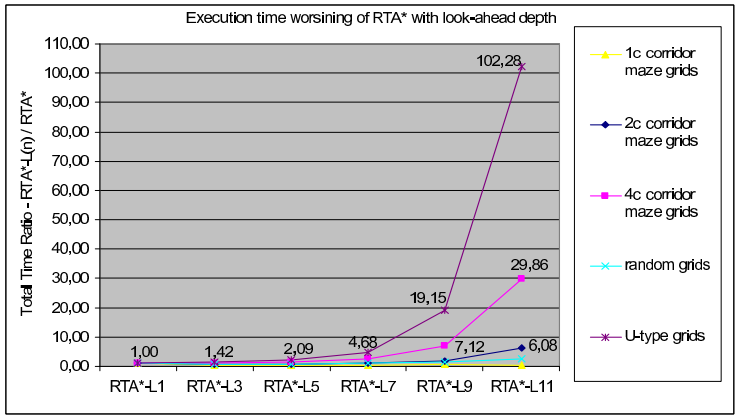

Figure 3.31: The increase in step execution times using look-ahead depths: 3, 5, 7, 9, 11, with respect to RTA* using look-ahead depth 1. Note that L stands for look-ahead depth, and lower values are better.

3.5.2 Analysis of Look-ahead Depth

In order to see the path length improvement of RTA* with higher look-ahead depths, a number of experiments were conducted in 16 grids with 5 different look-ahead depths (3, 5, 7, 9 and 11) and 4 different vision ranges (10, 20, 40 and infinite). Thus 320 test configurations were generated, and 10 runs were performed for each configuration, making 3200 runs in total.

The results showed that the path length improvement of RTA* with reasonable look-ahead depths is insignificant compared to the RTEF algorithms. The time per move and total execution time of RTA* with large look-ahead depths are too high because the time complexity is exponential in the size of the look-ahead depth. In terms of step execution time, Figure 3.31 shows the decrease in performance with higher look-ahead depths. In the chart, the horizontal axis contains RTA* variations with look-ahead depths 1, 3, 5, 7, 9 and 11; and the vertical axis is the increase in execution times of these RTA* variations with respect to RTA* using look-ahead depth 1. The sharpness of increase in the execution time highly depends on the grid type, which affects the average branching factor and the area of heuristic depression needed to be filled up. The increase is too high in grids with wide corridors, and low in grids with narrow corridors.

The average path length improvement of RTA* using higher look-ahead depths with respect to original RTA* is shown in Figure 3.32, and the comparison of RTA* with RT-VCH and RT-RTA*-p3 can be seen in Figure 3.33. We reduced the number

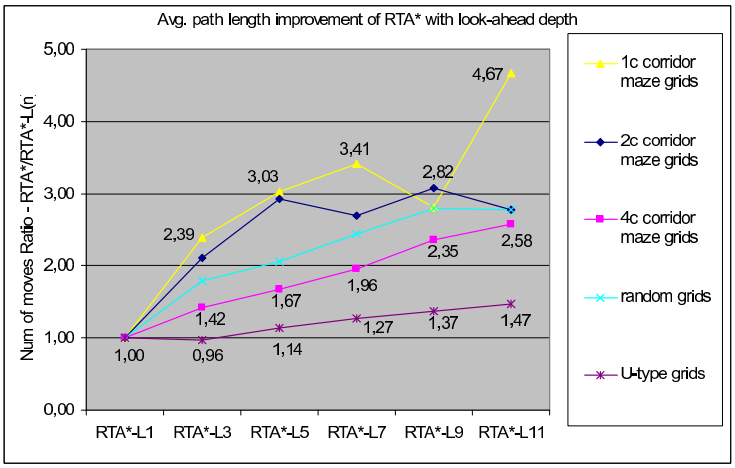

Figure 3.32: The improvement in the path lengths of RTA* using look-ahead depths 3, 5, 7, 9, 11, with respect to RTA* using look-ahead depth 1

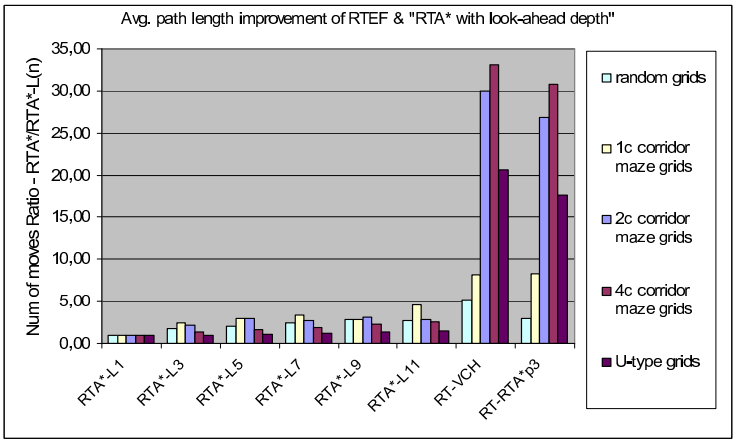

Figure 3.33: The improvement in the path lengths of RT-VCH, RT-RTA*-p3 and RTA* (using look-ahead depth 3, 5, 7, 9, 11), with respect to RTA* using look-ahead depth 1

of RTEF variations compared to 2, which includes RT-VCH and RT-RTA*-p3 since these were the best ones in the previous experiments, and also methodologically the most different variations.

As can be easily seen from Figure 3.32, increasing the look-ahead depth does not always improve the solution, although we expect shorter paths. This case is observed in the results of maze grids with 1-cell and 2-cell corridors. Since the results seem to be strange at first, we examined the test runs in details, and find out the problem, which was also mentioned in [10]. The reason was to choose a wrong alternative at a very critical decision point because of stopping the search at an immature depth guiding a local optimal. This is exemplified using one of our problematic runs shown

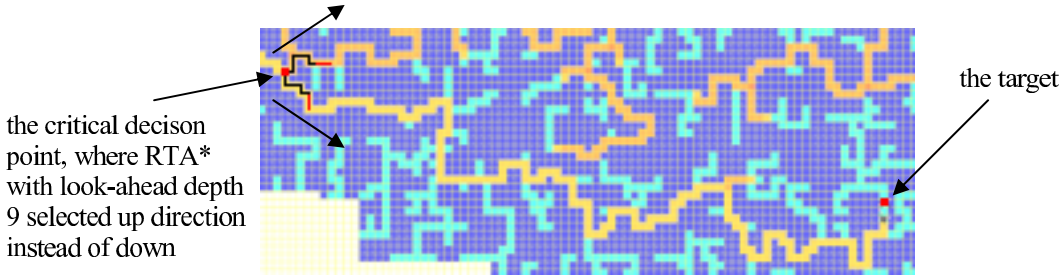

Figure 3.34: Critical decision point of RTA* with look-ahead depth 9

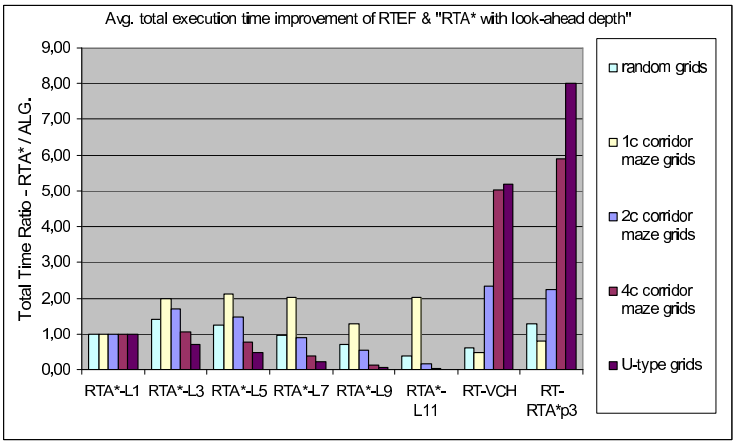

Figure 3.35: The improvement in the total execution times of RTA* (using look-ahead depth 3, 5, 7, 9, 11), RT-VCH and RT-RTA*-p3 with respect to RTA* using look-ahead depth 1. Note that ratios below 1 means the compared algorithm is worse than the original RTA*.

in Figure 3.34. In the example, although RTA* with look-ahead depth 7 could easily reach the target, RTA* with look-ahead depth 9 selected a wrong direction at a very early stage, and had to travel almost the entire maze meanwhile exploring. Therefore instead of 576 moves, it took 11704 moves to reach the target.

With respect to total execution time, Figure 3.35 shows the improvement of RTA* using higher look-ahead depths, and additionally the comparison with RT-VCH and RT-RTA*-p3. According to the results, we can conclude that using higher look-ahead depths is not sufficient for RTA* to over-perform RTEF in U-type grids and maze grids having wide corridors. However, RTA* becomes better than RTEF with small look-ahead depths in maze grids with 1-cell corridors, and in random grids.

3.5.3 Analysis of Search Depth

In order to see how search depth affects RTEF Algorithms, we performed a number of experiments with 4 different search depths (10, 20, 40 and 80) and 4 different vision

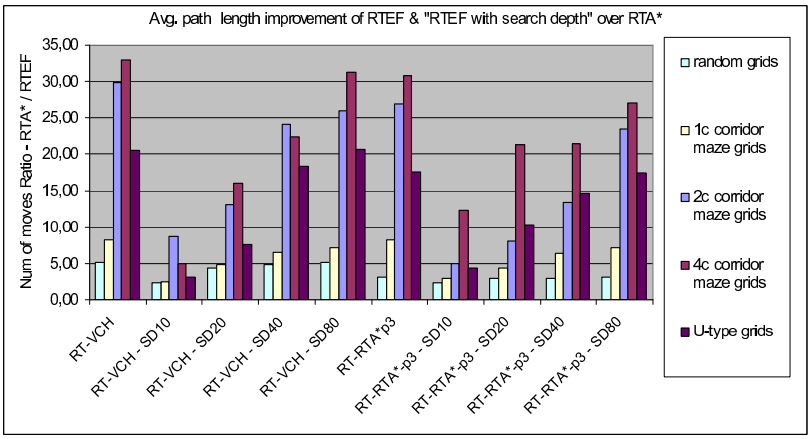

Figure 3.36: The improvement in the path lengths of RT-VCH and RT-RTA*-p3 using 10, 20, 40 and 80 search depths, with respect to RTA*. Note that SD stands for search depth.

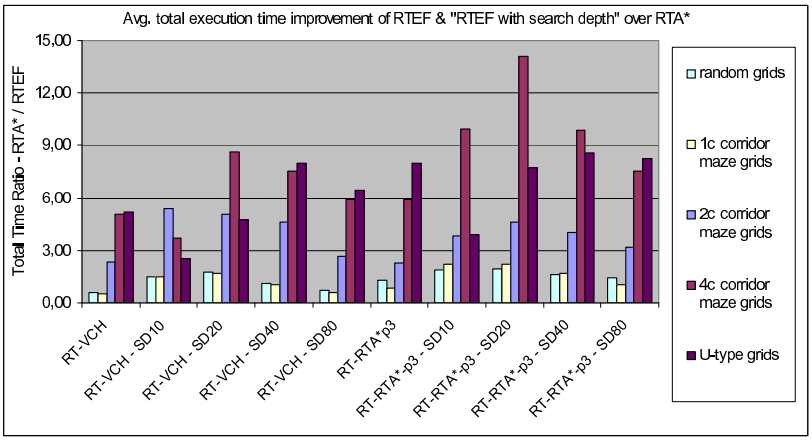

Figure 3.37: The improvement in the total execution times of RT-VCH and RT-RTA*-p3 using 10, 20, 40 and 80 search depths, with respect to RTA*

ranges (10, 20, 40, infinite) in 16 grids, and used 2 RTEF variations (RT-VCH and RT-RTA*-p3). 512 test configurations were generated, and 10 runs were performed for each configuration, making 5120 runs in total. The results of path lengths and total execution times with various search depths are shown in Figures 3.36 and 3.37.

The results showed that when a search depth is specified, the path lengths get longer, but the time spent per move decreases significantly. Therefore, if a very small depth is not used, the path lengths do not increase sharply, and the total execution time usually decreases. In conclusion, if the execution time per move and total execution time are important, it is better to use a reasonably small search depth, but if the path lengths are more important, it is better to use a large search depth to have a bounded complexity, or even not to use any.

3.5.4 Comparison with Optimal Solution Paths

Finally, we have conducted a number of experiments to compare the path lengths of RTEF and RTA* with the optimal path lengths in 16 grids. We implemented the off-line path planning algorithm A* [38] to compute the optimal path lengths. We assumed that the mazes are fully known (infinite vision range) by the agents, and computed the ratio of path lengths of each of these algorithms to that of optimal solutions to clearly see the proximity of solutions to the optimals (see Figure 3.38).

We observed that the path lengths obtained by RTEF algorithms are very close to the optimal ones. On the other hand, the solutions produced by RTA* are very far away from the optimals. The best performance is obtained by RT-VCH. Its solutions are only 1.501 times longer than the optimal ones on the average, and the standard deviation is 1.068. The worst performance is obtained by RTA*. The solutions are 33.022 times longer than the optimal ones on the average, and the standard deviation is 50.417, which is unacceptably high in practice. When we closely look at the results from the view point of different types of grids, we see that RTEF algorithms and RTA* show opposite behaviors most of the time. When the obstacle ratio increases and grids become complicated, RTEF algorithms almost converge to the optimal (e.g., the results are exactly optimal in maze grids with 70% obstacles), on the contrary RTA* gets far away from the optimal solutions (e.g., the results are 124 times longer than the optimal ones in maze-grids with 2-cell corridors and 50% obstacles).

		RT-H	RT-HBC	RT-VC	RT-VCH	RT-VCHBC	RT-RTA*	RT-RTA*p0	RT-RTA*p1	RT-RTA*p2	RT-RTA*p3	RT-RTA*p4	RTA*
Random 30%	Avg	1,272	1,281	1,256	1,283	1,269	1,214	1,205	1,189	1,192	1,191	1,200	1,588
	Std	0,083	0,086	0,188	0,074	0,076	0,088	0,093	0,103	0,097	0,095	0,100	0,254
Random 35%	Avg	1,643	1,628	1,889	1,628	1,638	1,589	1,620	1,693	1,653	1,499	1,575	3,676
	Std	0,579	0,586	0,666	0,586	0,582	0,554	0,662	0,667	0,649	0,389	0,483	1,922
Random 40%	Avg	1,384	1,384	7,587	1,383	1,383	6,185	6,724	4,428	5,243	4,024	4,231	26,332
	Std	0,259	0,260	4,464	0,258	0,258	5,121	5,422	2,552	2,930	3,353	2,361	22,261
Maze 1c x 30%	Avg	2,924	3,860	1,515	2,948	2,974	1,525	1,460	1,503	1,440	1,461	1,455	2,194
	Std	3,266	4,005	0,304	3,263	3,236	0,258	0,264	0,295	0,235	0,276	0,248	0,684
Maze 1c x 50%	Avg	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005	21,418
	Std	0,004	0,004	0,004	0,004	0,004	0,004	0,004	0,004	0,004	0,004	0,004	9,491
Maze 1c x 70%	Avg	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	20,640
	Std	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	18,747
Maze 2c x 30%	Avg	1,568	1,507	1,580	1,550	1,567	1,600	1,329	1,220	1,223	1,200	1,212	2,121
	Std	0,318	0,305	0,240	0,327	0,317	0,466	0,356	0,169	0,140	0,125	0,131	0,571
Maze 2c x 50%	Avg	1,001	1,001	1,000	1,001	1,001	1,000	1,000	1,000	1,000	1,000	1,000	124,921
	Std	0,002	0,002	0,000	0,002	0,002	0,000	0,000	0,000	0,000	0,000	0,000	47,574
Maze 2c x 70%	Avg	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	11,961
	Std	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	8,534
Maze 4c x 30%	Avg	1,560	1,555	8,601	1,588	1,561	7,626	6,547	5,784	3,921	3,894	1,942	18,126
	Std	0,485	0,474	4,334	0,457	0,484	5,023	4,709	3,141	2,495	2,752	1,010	12,570
Maze 4c x 50%	Avg	1,017	1,017	1,000	1,017	1,017	1,000	1,000	1,000	1,000	1,000	1,000	91,888
	Std	0,044	0,044	0,000	0,044	0,044	0,000	0,000	0,000	0,000	0,000	0,000	87,633
Maze 4c x 70%	Avg	1,005	1,005	1,000	1,005	1,005	1,000	1,000	1,000	1,000	1,000	1,000	46,929
	Std	0,014	0,014	0,000	0,014	0,014	0,000	0,000	0,000	0,000	0,000	0,000	55,477
U-type 30	Avg	1,600	1,540	4,728	1,621	1,600	3,890	2,767	2,143	1,485	1,708	1,751	19,286
	Std	0,618	0,618	4,460	0,649	0,618	2,816	1,891	1,714	0,646	0,884	0,931	35,029
U-type 50	Avg	1,893	2,008	10,523	1,800	1,925	3,356	8,054	2,067	1,538	1,493	1,744	59,066
	Std	0,894	1,153	19,732	0,915	0,899	2,829	18,523	1,671	0,576	0,462	1,213	76,739
U-type 70	Avg	1,247	1,292	9,416	1,285	1,452	13,900	8,596	3,032	1,497	1,701	1,735	19,888
	Std	0,303	0,346	10,379	0,340	0,570	20,480	10,445	3,285	0,579	0,970	1,256	18,865
U-type 90	Avg	3,828	3,964	17,201	2,907	3,593	23,072	13,358	10,274	4,884	4,212	4,663	58,325
	Std	2,053	2,523	18,463	0,991	1,403	17,358	8,931	11,000	3,136	2,795	2,720	51,438
		RT-H	RT-HBC	RT-VC	RT-VCH	RT-VCHBC	RT-RTA*	RT-RTA*p0	RT-RTA*p1	RT-RTA*p2	RT-RTA*p3	RT-RTA*p4	RTA*
Global Average		1,559	1,628	4,394	1,501	1,562	4,373	3,604	2,459	1,880	1,774	1,719	33,022
Global Stdev		1,239	1,499	8,555	1,068	1,162	8,927	6,859	3,828	1,848	1,700	1,487	50,417

Figure 3.38: The average ratio of RTEF algorithms and RTA* solution path lengths over optimal path lengths, and their standard deviations.

CHAPTER 4

REAL-TIME MOVING TARGET EVALUATION SEARCH

In this chapter, we introduce our second real-time path search algorithm, Real-Time Moving Target Evaluation Search (MTES) [49, 48]. First, we describe the details of the algorithm. Then, we examine its complexity and correctness. We go on with the introduction of our prey algorithm, Prey-A*, and finalize with the results of experiments.

4.1 Problem Description

Agents that use less informed heuristics such as Euclidian distance cannot precisely evaluate the cost differences of neighbor states and hence usually make wrong decisions in selecting their next moves towards the target. Although RTEF attempts to solve this problem to some extend by detecting closed directions correctly, it is poor in estimating real cost because it uses Euclidian distance heuristic to select the moving direction from open ones. Figure 4.1 shows the route an agent follows guided by RTEF. Initially the agent has two open (north and south) directions. Due to the Euclidian distance heuristic, the agent prefers the north direction leading to a very long route to the target. If the agent had selected to move south, the route would have been much shorter.

The problem of determining the right moving direction from the open alternatives was one of the motivations behind our new algorithm, Real-Time Moving Target Evaluation Search (MTES). The effectiveness of MTES is illustrated on the previous example in Figure 4.2. Here, the algorithm identifies three possible moving directions, and evaluated that the middle one (which we name as the *inner right most direction*)

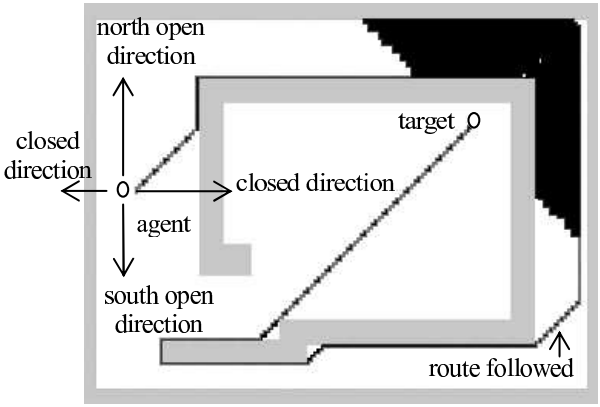

Figure 4.1: RTEF can detect which directions are open, but cannot evaluate the cost differences successfully just using Euclidian distance heuristic

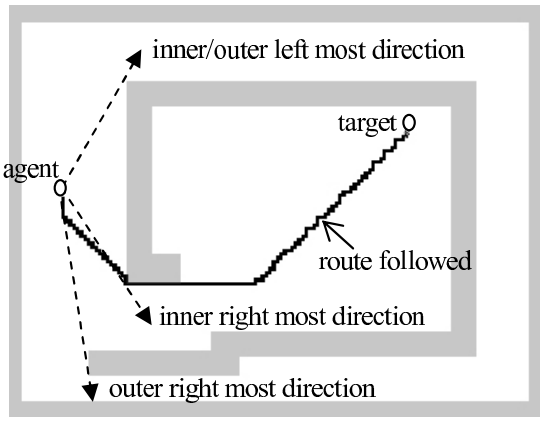

Figure 4.2: MTES chooses inner right most direction because it seems to be the shortest in possible alternatives

is approximately the shortest. In this study, we also extend both of our path search algorithms (RTEF and MTES) to handle moving targets. The assumptions of our domain different from the previous chapter are summarized below.

- There is a single agent (predator) that aims to reach a static or moving target (prey).
- The predator is expected to reach the prey from a short path avoiding obstacles in real-time. The prey is either static or escaping from the predator, and its location is assumed to be known by the predator all the time.
- The prey has unlimited perception and knows all the grid world and the location of the predator all the time.
- The search continues until the predator reaches the prey.

4.2 The Search Algorithm

MTES makes use of a heuristic, Real-Time Target Evaluation (RTTE-h), which analyzes obstacles and proposes a moving direction that avoids these obstacles and leads to the target through shorter paths. To do this, RTTE-h geometrically analyzes the obstacles nearby, tries to estimate the lengths of paths around the obstacles to reach the target, and proposes a moving direction. RTTE-h works in continuous space to identify the moving direction, which is then mapped to one of the actual moving directions (north, south, east and west). MTES repeats the steps in Algorithm 6 until reaching the target or detecting that the target is inaccessible. In the first step, MTES calls RTTE-h heuristic function, which returns a moving direction and the utilities of neighbor cells according to that proposed direction. Next, MTES selects one of the neighbor cells on open directions with the minimum *visit count* (see Definition 3.2.1). If there exists more than one cell having the minimum visit count, the one with the maximum utility is selected. If utilities are also the same, then one of them is selected randomly. After the move is performed, the *visit count* of the previous cell is incremented and the cell is inserted into the *history* (see Definition 3.2.2). If no alternative could be determined to move and the *history* is not empty, MTES clears the *history* to be able to backtrack. If the *history* is also empty, it is concluded that the target is unreachable.

In moving target search problem, the target may sometimes pass through the cells the agent previously walked through. In such a case, there is a risk that the history blocks the agent to reach the target since history cells are assumed to be obstacles and may close some gateways required to return back. If this situation occurs at some point, the agent will surely be able to detect this at the end, and clear the history opening all the closed gateways. Therefore, the algorithm is cable of searching moving targets without any additions. As a matter of fact, the only drawback of the history is not the possibility that it can block the way to the target entirely, but it can sometimes prevent the agent to reach the target through shorter paths by just closing some of the shortcuts. To reduce the performance problems of this side effect, the following procedure is applied. Assuming that (x_1, y_1) and (x_2, y_2) are the previous and newly observed locations of the target, respectively, and R is the set of cells the target could have visited in going from (x_1, y_1) to (x_2, y_2), the algorithm clears the history along

Algorithm 6 An Iteration of MTES Algorithm

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

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). In this chapter, we have integrated this extension to both RTEF and MTES.

4.2.1 Real-Time Target Evaluation Heuristic

Real-Time Target Evaluation heuristic (RTTE-h) given in Algorithm 7 propagates four diagonal rays away from the agent location, and analyzes the obstacles these rays hit to find out the best direction to move. If a ray hits an obstacle before exceeding the maximum ray distance, the obstacle border is extracted by tracing cells on the border starting from the hit-point. Concurrently, we also find the point on the border

which is closest to the target. This point will be used in calculating the estimated path lengths. Next, the closed directions are determined. Then the border is re-traced from both left and right sides to determine the geometric features that will be described in the next section. The obstacle features are evaluated and a moving direction to avoid the obstacle is identified. After all the obstacles are evaluated, the results are merged in order to propose a final moving direction.

Algorithm 7 RTTE-h Algorithm

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

In RTTE-h, ray-sending, edge-tracing and closed direction detection steps are almost the same as RTEF-ARM. There is only a small difference in edge-tracing step. In order to reduce the work load of closed direction detection step, RTEF-ARM does not insert unnecessary vertices (the ones that does not change the shape of the obstacle) into the vertex lists of island and hit-point island during edge-tracing, but RTTE-h does, because RTTE-h deals with the directions of these vertices, and missing vertices will effect the process. Additionally, RTTE-h performs three more steps shown in lines 6, 7 and 9 in Algorithm 7 for extracting additional geometric features and estimating the moving direction that minimizes the path length to the target. Details of these steps are given in the following sections.

4.2.2 Analyzing an Obstacle Border

When a ray hits an obstacle, its border is extracted and then analyzed. Border analysis (line 6 in Algorithm 7) 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 4.2.1 to 4.2.8 (see Figure 4.3 for illustrations):

Definition 4.2.1 (Outer left most direction) *Relative to the ray direction, the largest cumulative angle is found during the left tour on the border vertices. In each step of the trace, we move from one edge vertex to another on the border. The angle between the two lines (TWLNS) starting from the agent location and passing through these two following vertices is added to the cumulative angle computed so far. Note that the added amount can be positive or negative depending on whether we move in counter-clockwise (ccw) or clockwise (cw) order, respectively. This trace (including the trace for the other geometric features) continues until the sum of the largest cumulative angle and the absolute value of smallest cumulative angle is greater than or equal to 360. The largest cumulative angle before the last step of trace is used as the outer left most direction.*

Definition 4.2.2 (Inner left most direction) *The direction with the largest cumulative angle encountered during the left tour until reaching the first edge vertex where the angle increment is negative and the target lies between TWLNS. If such a situation is not encountered, the direction is assumed to be* $0 + \varepsilon$*, where* ε *is a very small number (e.g., 0.01).*

Definition 4.2.3 (Inside of left) *True if the target is inside the polygon whose vertices starts at agent's location, jumps to the outer left most point, follows the border of the obstacle to the right and ends at the hit point of the ray.*

Definition 4.2.4 (Inside of inner left) *True if the target is inside the polygon that starts at agent's location, jumps to the inner left most point, follows the border of the obstacle to the right and ends at the hit point of the ray.*

Definition 4.2.5 (Behind of left) *True if the target is in the region obtained by sweeping the angle from the ray direction to the outer left most direction in ccw order and the target is not inside of left.*

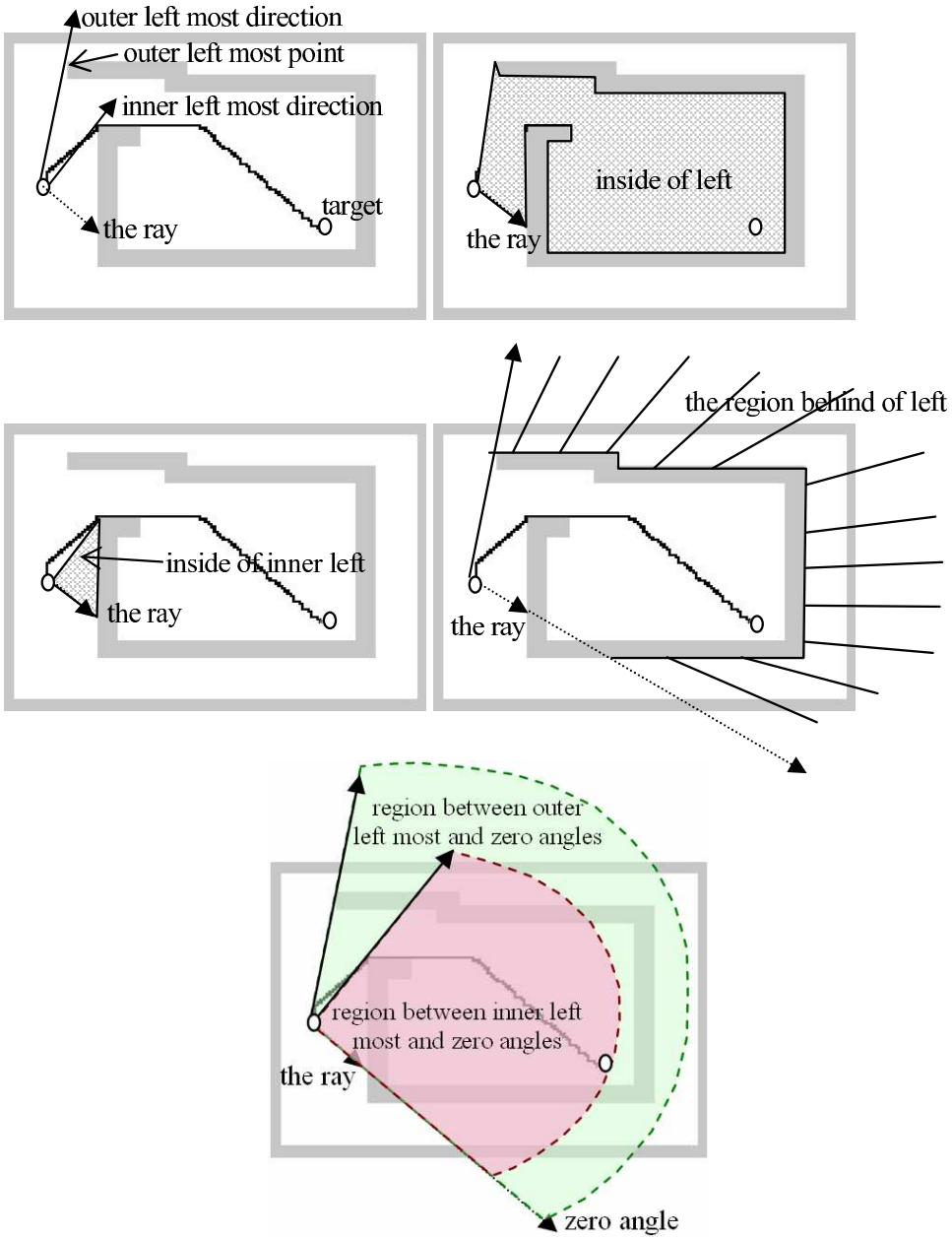

Figure 4.3: Geometric features of an obstacle: Outer left most and inner left most directions (left-top), Inside of left (right-top), Inside of inner left (left-middle), Behind of left (right-middle), Outer-left-zero angle blocking and Inner-left-zero angle blocking (bottom)

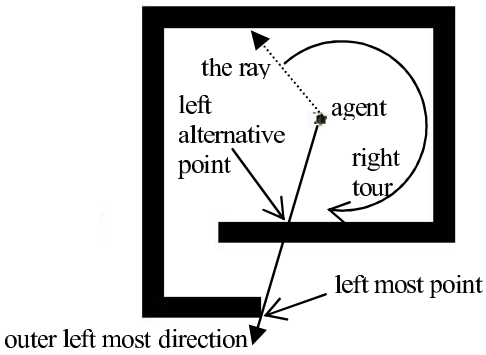

Figure 4.4: Left alternative point

Definition 4.2.6 (Outer-left-zero angle blocking) *True if target is in the region obtained by sweeping the angle from the ray direction to the outer left most direction in ccw order.*

Definition 4.2.7 (Inner-left-zero angle blocking) *True if target is in the region obtained by sweeping the angle from the ray direction to the inner left most direction in ccw order.*

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

Definition 4.2.8 (Left alternative point) *The last vertex in the outer left most direction encountered during the right tour until the outer right most direction is determined (see Figure 4.4).*

4.2.3 Evaluating Individual Obstacle Features

In individual obstacle evaluation step (line 7 in Algorithm 7), 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 8, which requires the path length estimations given in Definitions 4.2.9 to 4.2.11 in addition to the acquired geometric features of the obstacle:

Definition 4.2.9 (d_{left}) *The approximated length of the path which starts from the agent location, jumps to the outer left most point, and then follows the path determined by Algorithm 9 (see Figure 4.5).*

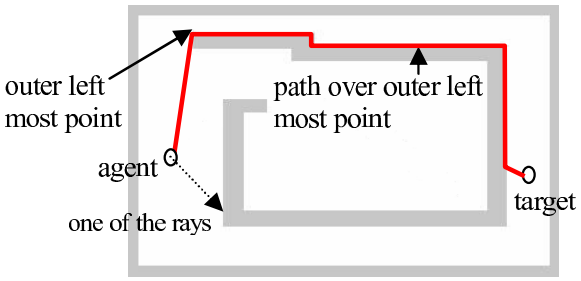

Figure 4.5: Exemplified d_{left} estimation

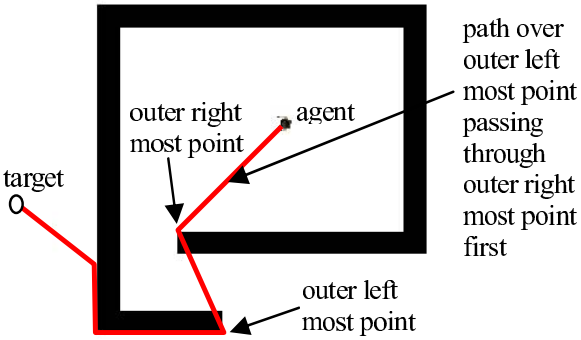

Figure 4.6: Exemplified $d_{left.alter}$ estimation

Definition 4.2.10 ($d_{left.alter}$) *The approximated length of the path which starts from the agent location, jumps to the outer right most point, and then to the outer left most point, and finally follows the path determined by Algorithm 9 (see Figure 4.6).*

Definition 4.2.11 ($d_{left.inner}$) *The approximated length of the path passing through the agent location, the inner left most point, and the target (see Figure 4.7).*

Algorithm 9 is internally used in computations of d_{left} and $d_{left.alter}$, and the sub-function *isoutwardsfacing* is called for detecting if a border segment, whose both ends touch the line passing through the *outer left most point* and the target point, is

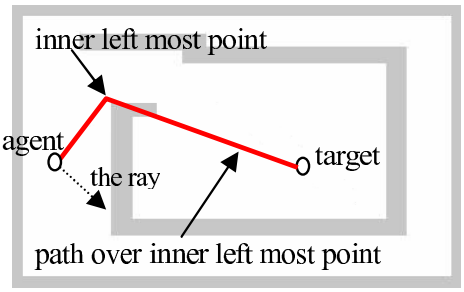

Figure 4.7: Exemplified $d_{left.inner}$ estimation

Algorithm 8 Individual Obstacle Evaluation Step

if (*behind of left* and not *inside of right*) or (*behind of right* and not *inside of left*) **then**
  Case 1:
  if *outer left most angle* + *outer right most angle* ≥ 360 **then**
    Case 1.1:
    if distance from agent to *outer left most point* $<$ distance from agent to *left alternative point* **then**
      Case 1.1.1: Let estimated distance be $min(d_{left}, d_{right.alter})$, and propose *outer left most direction* as moving direction
    else
      Case 1.1.2: Let estimated distance be $min(d_{left.alter}, d_{right})$, and propose *outer right most direction* as moving direction
    end if
  else
    Case 1.2:
    if $d_{left} < d_{right}$ **then**
      Case 1.2.1: Let estimated distance be d_{left}, and propose *outer left most direction* as moving direction
    else
      Case 1.2.2: Let estimated distance as d_{right}, and propose *outer right most direction* as moving direction
    end if
  end if
  Mark obstacle as blocking the target
else if *behind of left* **then**
  Case 2:
  if Target direction angle $\neq 0$ and *outer-right-zero angle blocking* **then**
    Case 2.1: Let estimated distance be d_{left}, and propose *outer left most direction* as moving direction
  else
    Case 2.2: Let estimated distance be $d_{right.inner}$, and propose *inner right most direction* as moving direction
  end if
  Mark obstacle as blocking the target
else if *behind of right* **then**
  Case 3:
  if Target direction angle $\neq 0$ and *outer-left-zero angle blocking* **then**
    Case 3.1: Let estimated distance be d_{right}, and propose *outer right most direction* as moving direction
  else
    Case 3.2: Let estimated distance be $d_{left.inner}$, and propose *inner left most direction* as moving direction
  end if
  Mark obstacle as blocking the target
else
  Case 4:
  if (*inside of left* and not *inside of right*) and (*inner-left-zero angle blocking* and not *inside of inner left*) **then**
    Case 4.1: Let estimated distance be $d_{left.inner}$, and propose *inner left most direction* as moving direction
    Mark obstacle as blocking the target
  else if (*inside of right* and not *inside of left*) and (*inner-right-zero angle blocking* and not *inside of inner right*) **then**
    Case 4.2: Let estimated distance be $d_{right.inner}$, and propose *inner right most direction* as moving direction
    Mark obstacle as blocking the target
  end if
end if

Algorithm 9 Path length estimation used for d_{left} and $d_{left.alter}$

Require: t : target point

Require: s : outer left most point

Require: n : the nearest point to the target

Require: $+$: next border point (left of)

Require: $-$: previous border point (right of)

Require: $insert(p)$: inserts a point to the estimated path

Require: $clasify(p_1, p_2, p_3)$: if the edge formed by the points p_1, p_2, p_3 does a left turn then returns true, else returns false

Require: $isoutwardsfacing(side, p_1, p_2)$: see Algorithm 10

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

Algorithm 10 The function $isoutwardsfacing(side, p_1, p_2)$

Require: t : target point

Require: s : outer left most point

Require: $len(n_1, n_2)$: returns distance between points n_1 and n_2

Require: $positive(m)$: if $len(m,t) \leq len(s,t)$ or $len(m,t) \leq len(s,m)$ then returns true, else returns false

Require: $slen(m)$: if $positive(m)$ then returns $+len(s,m)$ else returns $-len(s,m)$

1: **if** ($side$ and $slen(p_1) < slen(p_2)$) or (not $side$ and $slen(p_1) > slen(p_2)$) **then**

2: **return** true

3: **else**

4: **return** false

5: **end if**

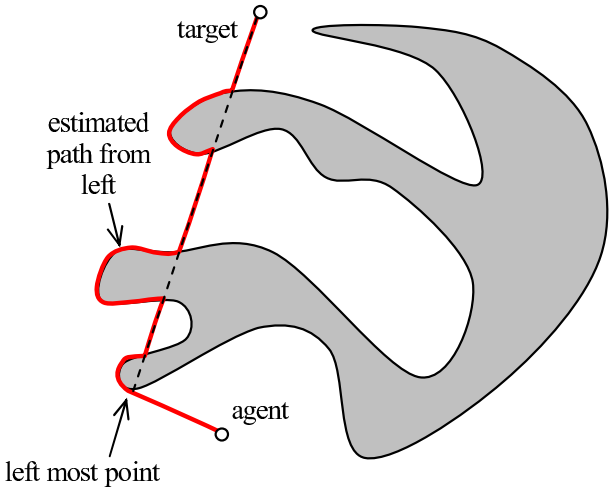

Figure 4.8: Exemplified path length estimation

outwards facing (see Figures 4.8 and 4.9). 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 $d_{right.inner}$.

In Algorithm 7, lets consider four top-level if-conditions in lines $1, 19, 27$ and 35, which correspond to *Cases 1, 2, 3* and *4* respectively. The algorithm enters *Case 1* only if the target is certainly behind the obstacle and the target can be reached by either going around the obstacle through the *outer left most point* or the *outer right most point*. The if-condition preceding *Case 1* consists of two disjuncted sub-conditions. The first one, *"behind of left and not inside of right"*, is satisfied when the

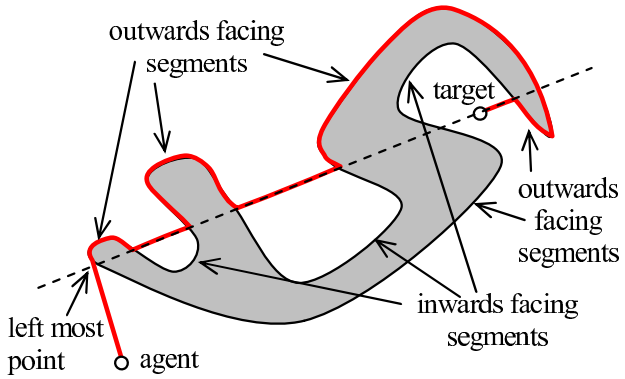

Figure 4.9: Outwards and inwards facing segments

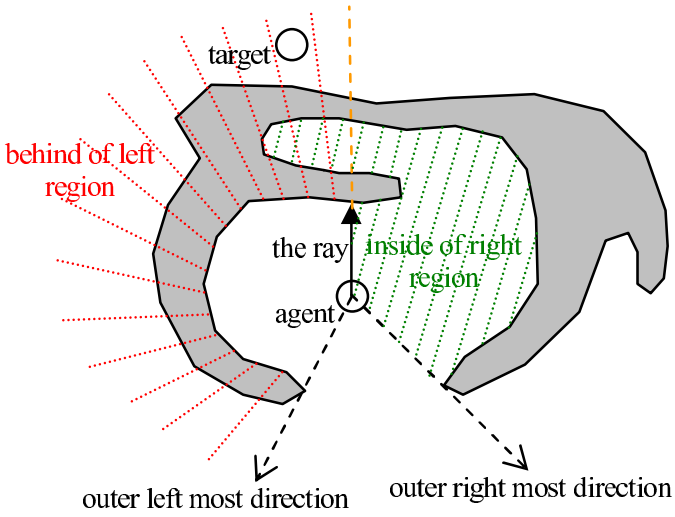

Figure 4.10: Case 1: Target is behind of left region and not inside of right region (means not inside of the overlap area of behind of left and inside of right regions).

target is behind the left side of the obstacle and we are sure that we cannot go to the target from the inner right region since the target is not *inside of right*. Hence, we need to go around the obstacle to reach the target. This case is exemplified in Figure 4.10. The second sub-condition is symmetric to the first one.

Case 1 has two second-level if-conditions in lines 3 and 10, which cover *Case 1.1* and *Case 1.2* respectively. The if-condition preceding *Case 1.1* checks if the sum of the *outer left most angle* and the *outer right most angle* is greater than or equal to 360 degree. The condition is satisfied if the swept angles from left and right to opposite orientations meet each other and some angle overlap occurs. This means that the agent is surrounded by the obstacle in all directions, the target is outside, and the agent needs to go out from the nearest exit. In this case, if the nearest exit is determined as the corner of the *outer left most edge*, *Case 1.1.1* otherwise *Case 1.1.2* is executed.

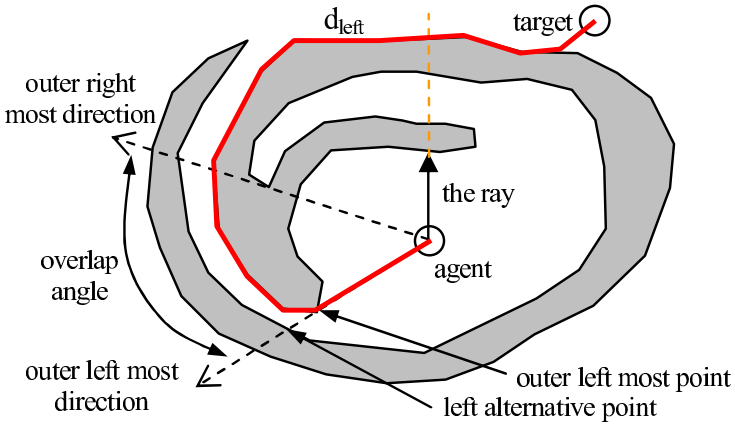

Figure 4.11: Case 1.1.1: Since outer left most angle + outer right most angle ≥ 360 and outer left most point is nearer to the agent than left alternative point, the outer left most direction will be proposed.

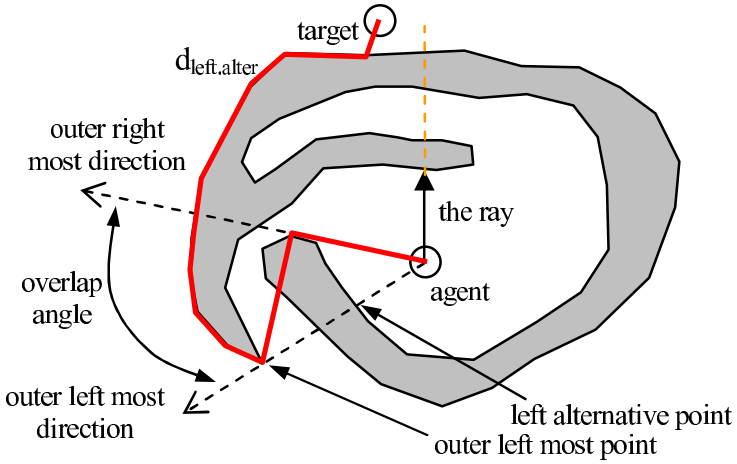

Figure 4.12: Case 1.1.2: Since outer left most angle + outer right most angle ≥ 360 and left alternative point is nearer to the agent than outer left most point, the outer right most direction will be proposed.

The cases are illustrated in Figures 4.11 and 4.12. If the if-condition preceding *Case 1.1* is not satisfied, *Case 1.2* is executed, which means there is no angle overlap and the agent is only surrounded by the obstacle in some directions but not all. In this case, the edge minimizing the route distance to the target is determined, and either *Case 1.2.1* or *Case 1.2.2* is executed depending on the value of d_{left} and d_{right}. This case is exemplified in Figure 4.13.

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

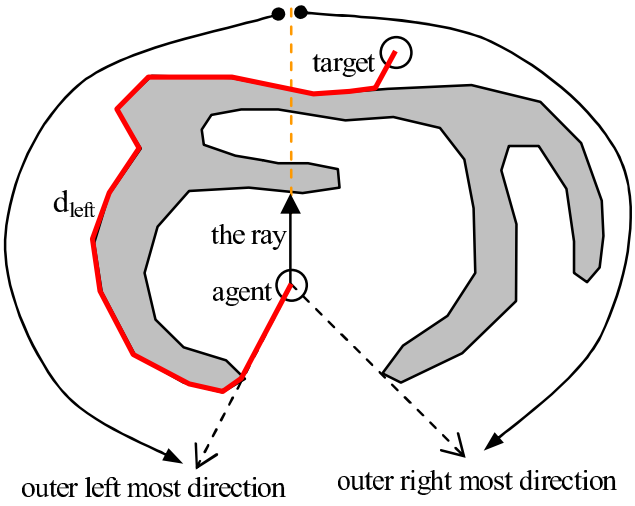

Figure 4.13: Case 1.2.1: Since outer left most angle + outer right most angle < 360 and the estimated path length to the target through the outer left most point is shorter than the right one, the outer left most direction will be proposed.

most edge and the *outer right most edge* (see Figure 4.14), and the target inside that region can be reached by going through the corner of the *outer left most edge*. The second one handled in *Case 2.2* lies in the inner part of the obstacle (see Figure 4.15), and the target inside this region can be reached by going through the corner of the *inner right most edge*. We will not go into the details of *Case 3* since it is symmetric to *Case 2*.

The algorithm enters *Case 4* if none of the previous top-level if-conditions are satisfied. *Case 4* has two second-level if-conditions in lines 37 and 40, which cover *Case 4.1* and *Case 4.2* respectively. The if-condition preceding *Case 4.1* consists of two conjuncted sub-conditions. The first one, *"inside of left and not inside of right"*, is satisfied when the target is inside of the left but not right region, thus we are sure that we need to enter the left region but we don't know yet if the flying direction to target is feasible. The second sub-condition, *"inner-left-zero angle blocking and not inside of inner left"*, is satisfied if the target is behind the *inner left most edge*, thus the flying direction to target is not feasible. If both sub-conditions hold, we known that the agent needs to enter the left region through the corner of the *inner left most edge*. This case is illustrated in Figure 4.16. We will not examine *Case 4.2* since it is also symmetric to *Case 4.1*.

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

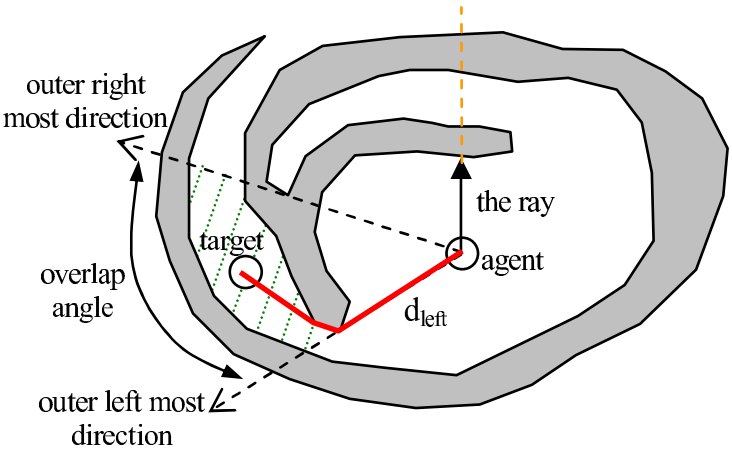

Figure 4.14: Case 2.1: Since the target is at the direction that falls into the overlap angle, the outer left most direction will be proposed.

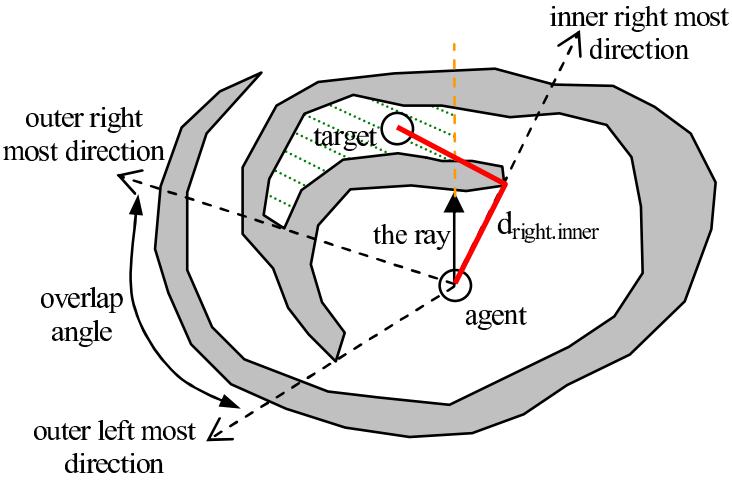

Figure 4.15: Case 2.2: Since the target is not at the direction that falls into the overlap angle, the inner right most direction will be proposed.

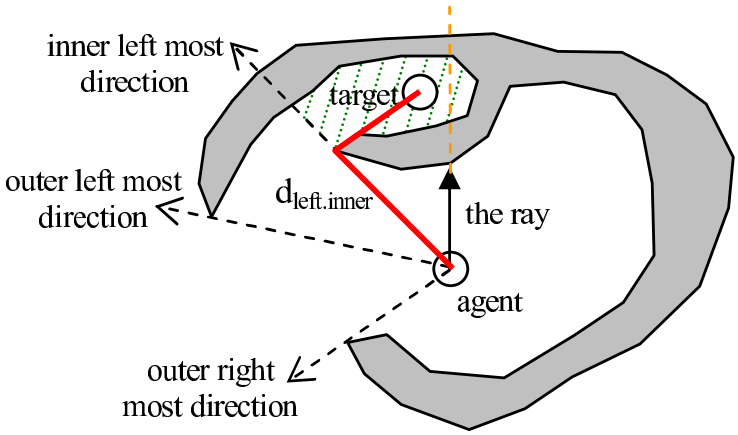

Figure 4.16: Case 4.1: Since target is inside of left region, but not right, and inner-left-zero angle blocking and not inside of inner left, the inner left most direction will be proposed.

to move.

4.2.4 Merging Entire Results

In the result merging step (line 9 in Algorithm 7), 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 6) for final decision. The merging algorithm is given in Algorithm 11.

Algorithm 11 Merging Phase

1: **if** all the directions to neighbor cells are closed **then**
2: propose no moving direction and halt with failure
3: **end if**
4: Select the obstacle (*most constraining obstacle*) that is marked as blocking the target and maximize the distance to the target, if there exists one
5: **if** most constraining obstacle exists **then**
6: identify a moving direction that gets around the most constraining obstacle avoiding the remaining obstacles
7: **else**
8: select the moving direction as the direct flying direction to the target
9: **end if**
{Compute utility of each neighbor cell}
10: **for** each neighbor cell **do**
11: **if** direction of the neighbor cell is closed **then**
12: set utility to *zero*
13: **else**
14: set utility to $(181 - dif)/181$, where dif is smallest angle between the proposed moving direction and the direction of the neighbor cell
15: **end if**
16: **end for**

The most critical step of the merging phase is to compute the moving direction to get around *the most constraining obstacle*. The reason why we determine the moving direction based on the most constraining obstacle is the fact that it might be blocking

the target the most. We aim to get around the most constraining obstacle and to do this we have to reach its border. In case there are some other obstacles on the way to the most constraining obstacle, we need to avoid them and determine the moving direction accordingly. Our algorithm works even if we ignore the intervening obstacles but we employ the following technique in order to improve solution quality with respect to path length.

Let the final direction to be proposed by the algorithm considering ray r be pd_r. Initially pd_r is set to the direction dictated by the most constraining obstacle o_r hit by ray r. Assume that pd_r is computed in the left tour. Note that the pd_r was determined during the counter clockwise (ccw) tour started from the hit point of ray r. If pd_r is blocked by some obstacles, pd_r can be changed by sweeping pd_r in clockwise direction until pd_r is not blocked by any obstacle or pd_r becomes the direction of ray r. By definition, we know that r is guaranteed to reach the border of obstacle o_r before hitting any other obstacle. In order to determine intervening obstacles, we check obstacles (not equal to o_r) hit by the other rays fall into ccw angle between r and pd_r. If an obstacle o_s hit by ray s has *outer left most direction* outside ccw angle between ray s and pd_r, and has *outer right most direction* inside ccw angle between r and s, then the obstacle o_s blocks pd_r and proposed direction should be swept to *outer left most direction* of obstacle o_s. Using this information we compute the direction nearest to pd_r between r and pd_r and not blocked by the intervening obstacles. The method is exemplified in Figure 4.17. 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 RTTE-h heuristic can be seen in Figure 4.18. In the sample, there exist three obstacles, two of which are blocking the target (obstacle A and B). Therefore, the final proposed moving direction is determined by examining these two obstacles.

4.3 Complexity Analysis

In each move, MTES performs steps similar to RTEF. In MTES, the number of passes over obstacle borders is greater than that of RTEF and in each pass more time is consumed. As a result, at each step MTES is slower than RTEF. Although MTES

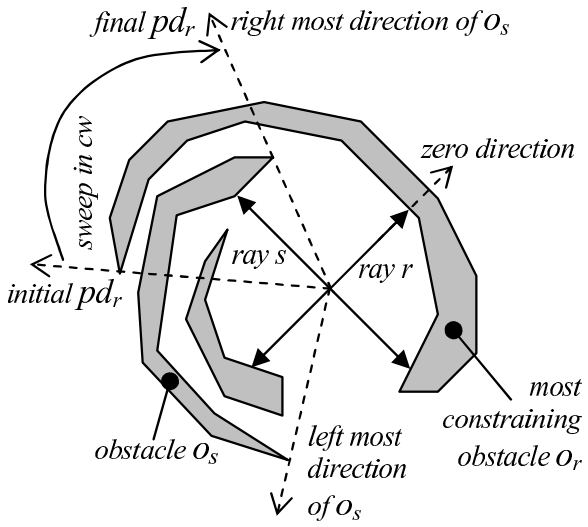

Figure 4.17: An example of avoiding the intervening obstacles

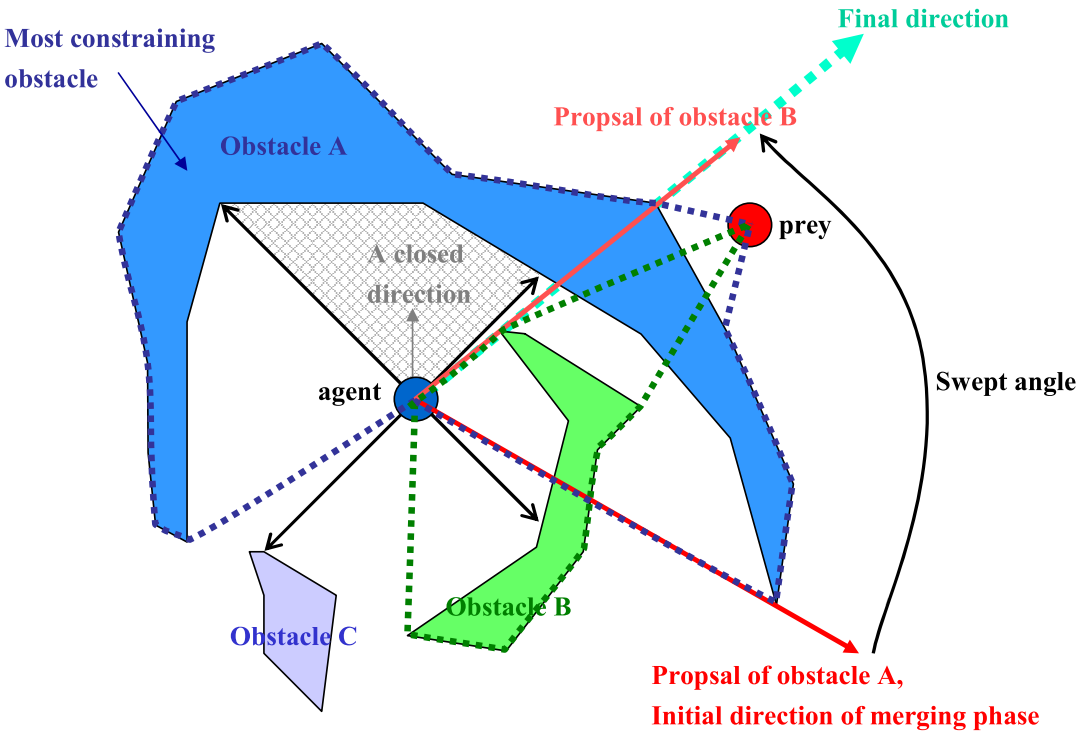

Figure 4.18: A sample illustrating the entire process of RTTE-h heuristic

seems to be less efficient than RTEF, its worst case complexity is the same as that of RTEF, which is $O(w.h)$ per step, where w is the width and h is the height of the grid. Since increasing the grid size decreases the efficiency, a search depth (d) can be introduced similar to RTEF in order to limit the worst case complexity of MTES. With this limitation, the complexity of MTES becomes $O(d^2)$.

4.4 Proof of Correctness

A single iteration of MTES given in Algorithm 6 is similar to RTEF shown in Algorithm 3, which is proved to be correct. MTES uses visit counts and history to prevent infinite loops, and eliminate closed directions in order not to enter dead-ends, which are the same as RTEF. The difference is in selection of moving directions, which will only effect the solution quality, but not the completeness. RTEF selects an open direction minimizing the Euclidian distance to the target; on the other hand MTES selects an open direction maximizing the utility computed by RTTE-h heuristic which measures the actual distance to the target more precisely than Euclidian distance. The algorithm is complete in the sense that if the target is accessible, the agent will surely find its way to the target without entering any infinite loop.

4.5 Prey Algorithm

To test our predator algorithm, we developed a deliberative off-line prey algorithm, Prey-A*, which is powerful but not very efficient. To prevent the side effects caused by the efficiency difference, the predator and the prey algorithms are executed alternately in performance tests. Prey-A* performs the steps given in Algorithm 12 in each iteration. The algorithm 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.

Algorithm 12 An Iteration of Prey-A* Algorithm

1: Generate $costs_{predator}$, which is a grid, each cell of which maps to a cell of the environment and contains the number of moves required for the nearest predator to reach the cell.

2: Generate $costs_{prey}$, which is a grid, each cell of which maps to a cell of the environment and contains the number of moves required for prey to reach the cell.

3: Check all the nearby cells of the prey within a limited search window centered at the prey location, and find out the cell (*destination*), which maximizes the number of moves for the predators to reach (the cell maximizing the cost in $costs_{predator}$) meanwhile ensuring that the prey will not be caught by any predators during the travel to the *destination* on the shortest path generated by A*. The safety of the shortest path is guaranteed by checking all the cells on the shortest path. If $time_{caught}$ for all the cells on the way is greater than *zero*, we are sure the predators will not catch the prey during its travel to the (*destination*). $time_{caught}$ for a cell is computed as:
$costs_{predator}[cell] - \alpha.costs_{prey}[cell]$
where α is the speed ratio of the predator and the prey found by the formula $speed_{predator}/speed_{prey}$.

4: The move to the first cell that is on the way of the shortest path to the *destination* is taken as the next step. Note that prey may choose not to move if *destination* is the cell the prey is on.

4.6 Experimental Results

In this section, we present our experimental results on MTS-c, MTS-d, RTEF (RT-VCH), MTES and A* (predators) against static and moving targets (preys). As being an off-line algorithm, we executed A* in each step from scratch. For the test runs, we used 9 randomly generated sample grids of size 150x150. Six of them were the *maze* grids (see Figure 4.19), and three of them were the *U-type* grids (see Figure 4.20). The *maze* grids were produced with the constraint that every two non-obstacle

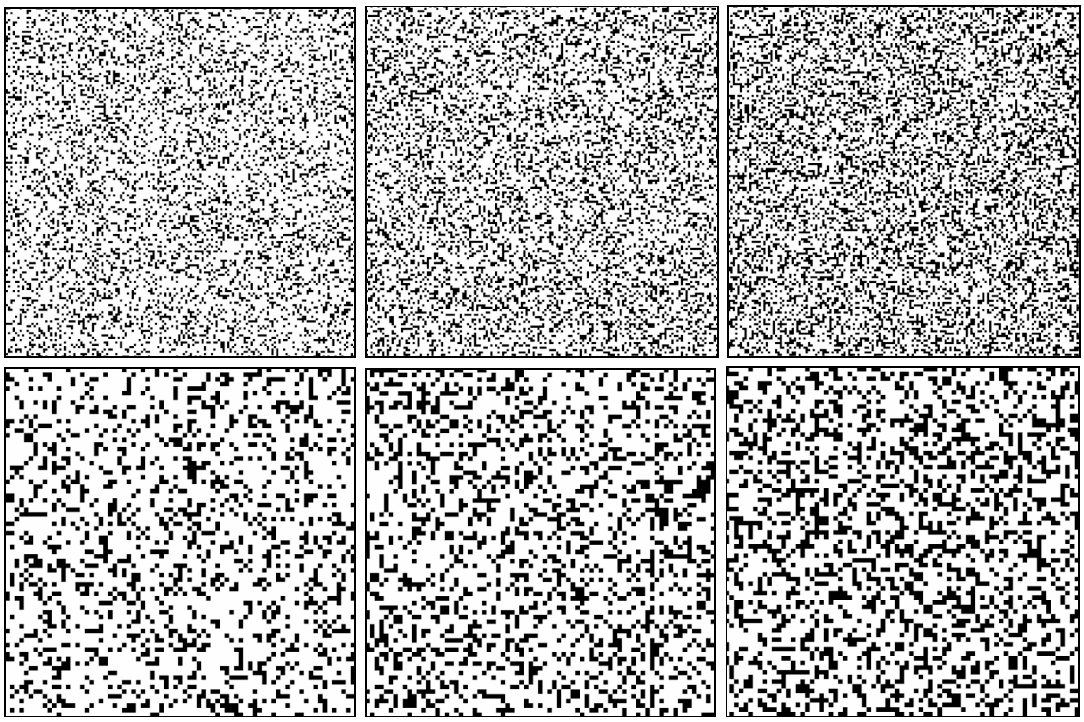

Figure 4.19: Maze grids with 25% (left column), 30% (middle column) and 35% obstacles (right column), and maze grids with 1-cell (top row) and 2-cell corridors (bottom row).

cells are always connected through a path. Two parameters, obstacle ratio (25%, 30% and 35%) and corridor size (1 and 2 cell corridors), were used to produce the mazes. 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, we produced 15 different randomly generated predator-prey location pairs, and made all the algorithms use the same pairs for fairness. The location of predators were randomly chosen from left or right sides of the grids, and the prey locations were chosen randomly from middle part of the grids. This effect is obtained by dividing the grids into 5 columns, and selecting the predator locations from the most left and right columns, and the prey locations from the middle column.

To test the algorithms against a moving target, we used the deliberative off-line prey algorithm, Prey-A* (see Algorithm 12) with 161x161 sized search window. In the experiments, we assumed that the prey knows the entire grid world and the location of the predator all the time, and the predator always knows the location of the prey, but perceives the grid world up to a predefined *vision range*. Our tests were performed with 10, 20, 40 and *infinite* vision ranges and search depths. Additionally, we assumed

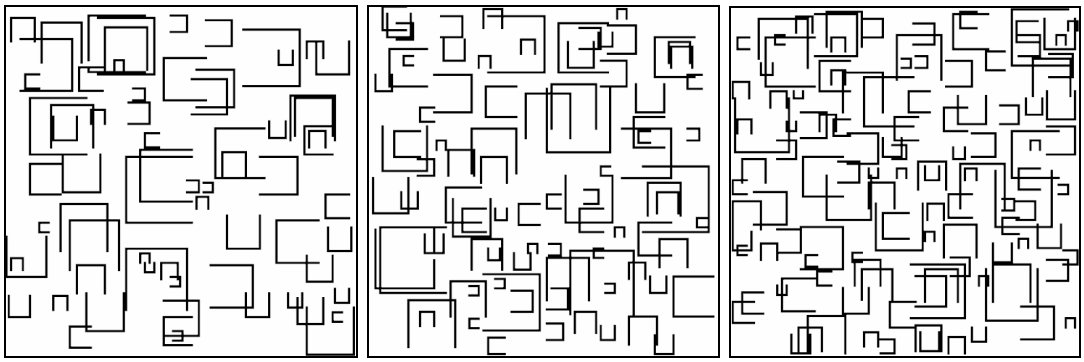

Figure 4.20: U-type grids with 70 (left), 90 (middle) and 120 (right) u-shaped obstacles

that the prey is slower than the predator, and skips 1 move after each 7 moves.

4.6.1 Analysis of Static Targets

In this section, we present the performance of MTS-c, MTS-d, RTEF, MTES and A* against a static target. We conducted experiments with different vision ranges, search depths and maze types. With respect to vision ranges and search depths, the averages of path lengths on maze grids are given in Figures 4.21 and 4.22, and the averages of path lengths on U-type grids are given in Figures 4.23 and 4.24, respectively. In the charts, the horizontal axis is either the vision range or the search depth, and the vertical axis contains the ratio of improvement in the path length with respect to MTS-c (the path length of MTS-c divided by that of the compared algorithm).

According to the results, we can conclude that MTES performs significantly better than MTS-c, MTS-d and RTEF, and usually offers solutions near to optimal ones produced by A*. RTEF and MTS-d perform head to head, and MTS-c is the worst. When we examine the results with respect to vision range, we see that vision range does not effect the solutions significantly in maze grids with 25% and 30% obstacles because the obstacle sizes are not very large. Similarly, search depth also does not effect the solutions much in these grids for the same reason.

4.6.2 Analysis of Moving Targets

In this section, we present the performance of MTS-c, MTS-d, RTEF, MTES and A* against a moving target guided by Prey-A*. We conducted experiments with different vision ranges, search depths and maze types. With respect to vision ranges and search depths, the averages of path lengths on maze grids are given in Figures 4.26 and 4.27,

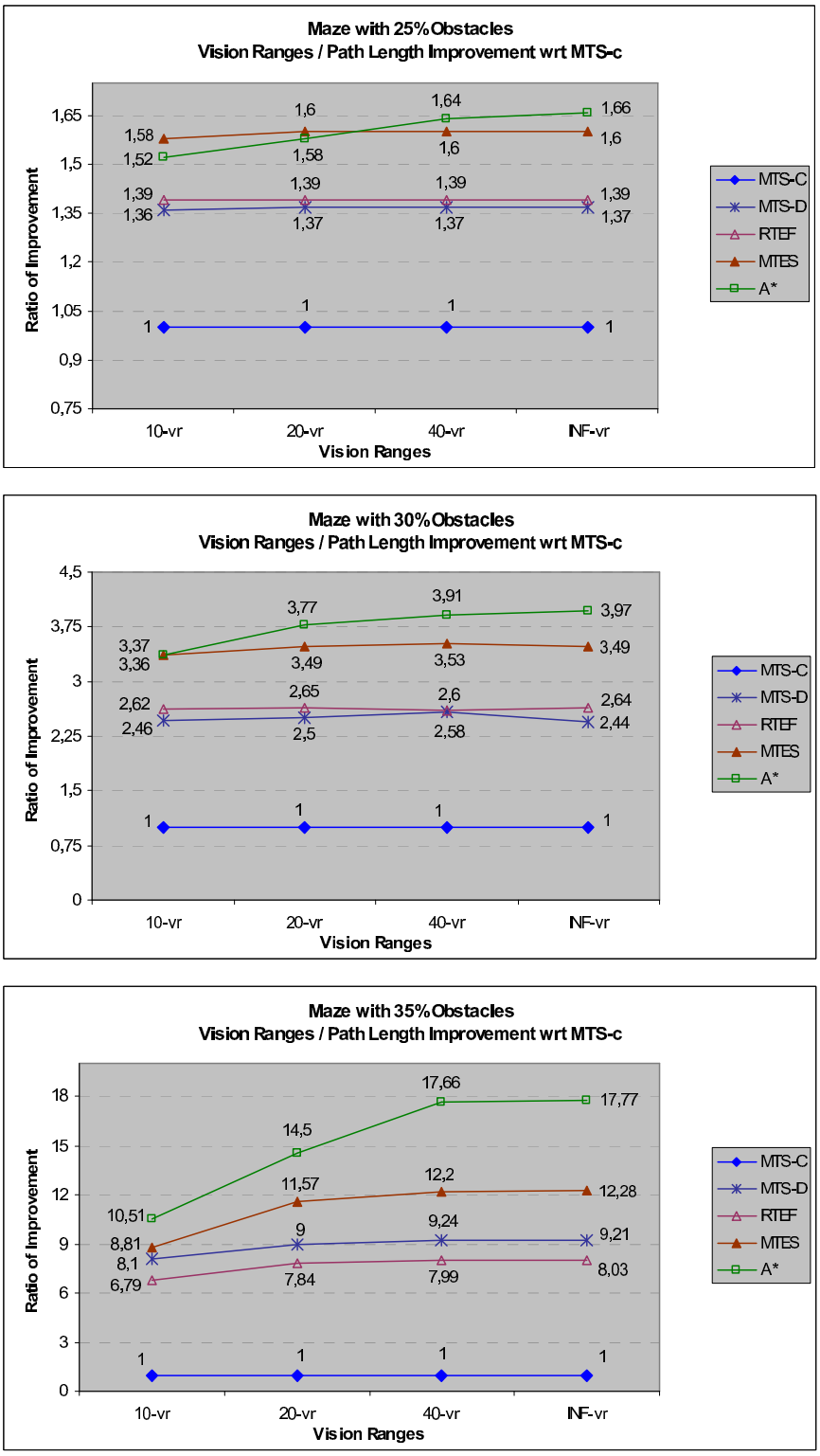

Figure 4.21: Average of path length results of maze grids (25% obstacles (top), 30% obstacles (middle), 35% obstacles (bottom)) for increasing vision ranges against a static target

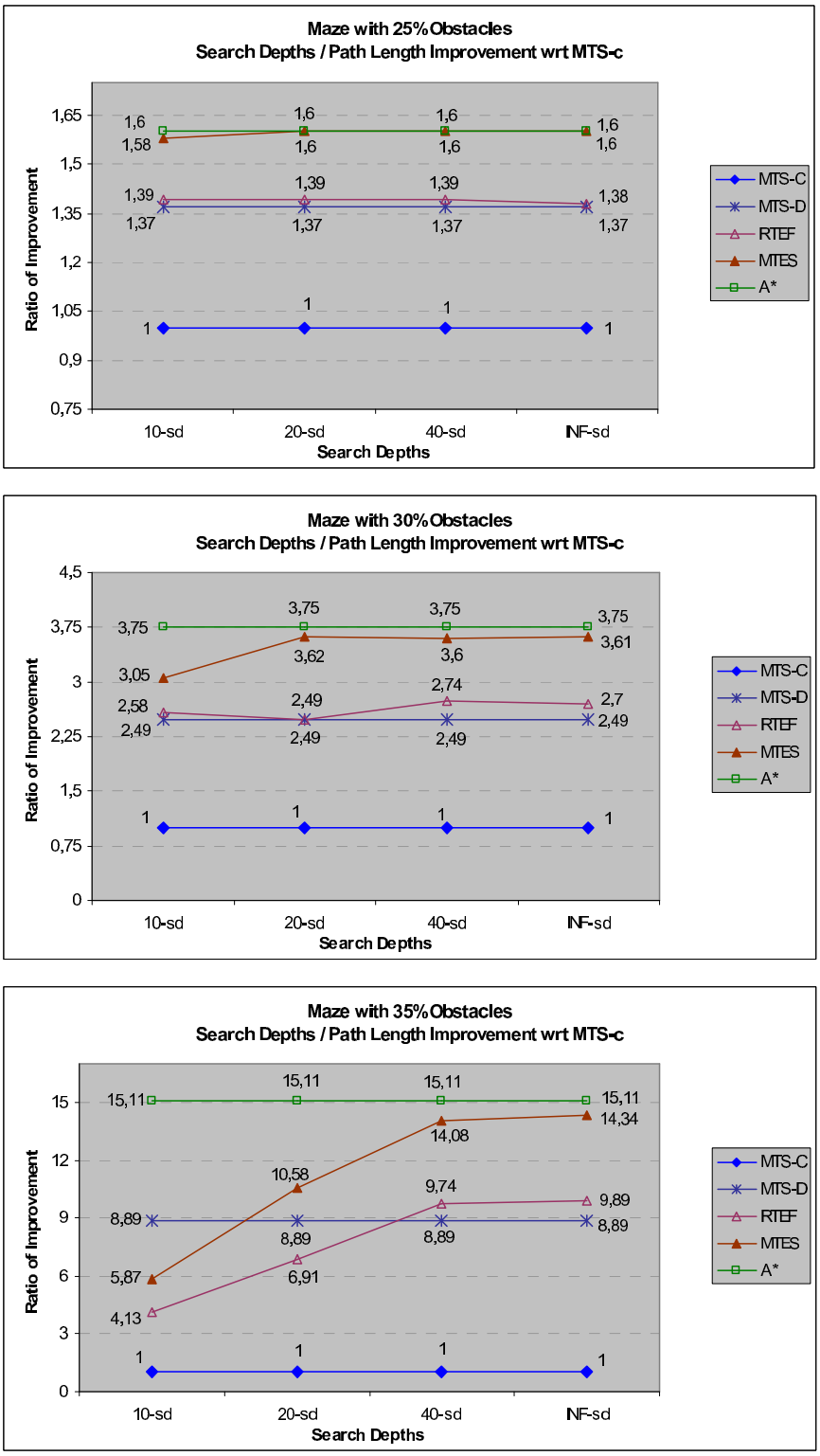

Figure 4.22: Average of path length results of maze grids (25% obstacles (top), 30% obstacles (middle), 35% obstacles (bottom)) for increasing search depths against a static target

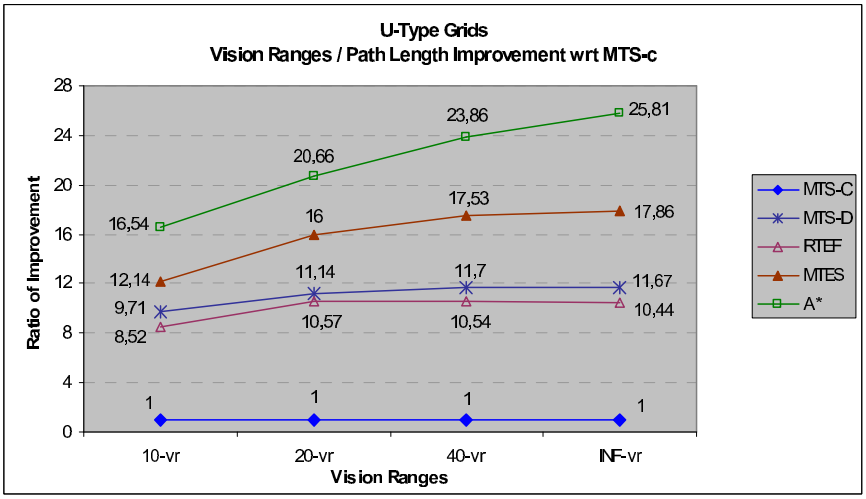

Figure 4.23: Average of path length results of U-type grids for increasing vision ranges against a static target

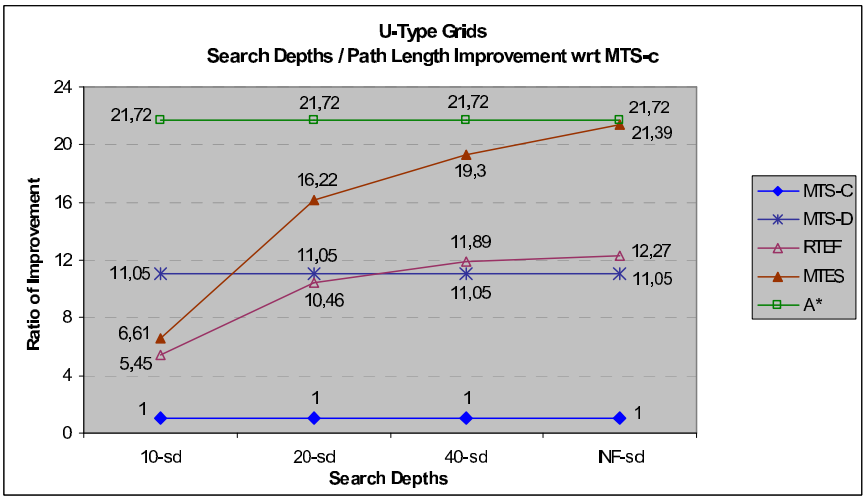

Figure 4.24: Average of path length results of U-type grids for increasing search depths against a static target

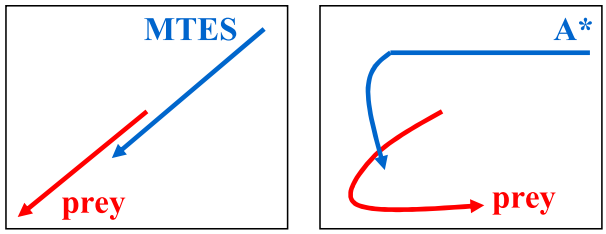

Figure 4.25: MTES prefers performing diagonally shaped manoeuvres (left) and A* prefers performing L-shaped manoeuvres (right) for approaching targets located in diagonal directions.

and the averages of path lengths on U-type grids are given in Figures 4.28 and 4.29, respectively. In the charts, the horizontal axis is either the vision range or the search depth, and the vertical axis contains the ratio of improvement in the path length with respect to MTS-c (the path length of MTS-c divided by that of the compared algorithm).

The results showed that MTES performs significantly better than RTEF, MTS-d and MTS-c, and usually offers near optimal solutions that are almost as good as the ones produced by A*. Next, RTEF, MTS-d and MTS-c follow MTES. In U-type grids, MTES mostly outperforms A*. When we examined this interesting result in details, we observed that they behave very differently in sparse parts of the grid. MTES prefers performing diagonally shaped manoeuvres for approaching targets located in diagonal directions, whereas A* prefers performing L-shaped manoeuvres in such cases (see Figure 4.25). Since the agents are only permitted to move in horizontal and vertical directions, these two manoeuvre patterns have equal path distances to a fixed location. Although, there is nothing wrong with these manoeuvres for fixed targets, this is not the case for moving targets since the strategy difference significantly affects the behavior of the prey in U-type grids, which sometimes makes A* worse than MTES.

When we examine the results in terms of vision range, we see that vision range does not effect the solutions significantly except in U-type grids since the obstacle sizes in these grids are the largest. Similarly, search depth also does not effect the results much except in maze grids with 35% obstacles and U-type grids. Another important fact we observed about the search depths is that even with very small depths, MTES always performs better than MTS-c and MTS-d, and almost always performs better than RTEF.

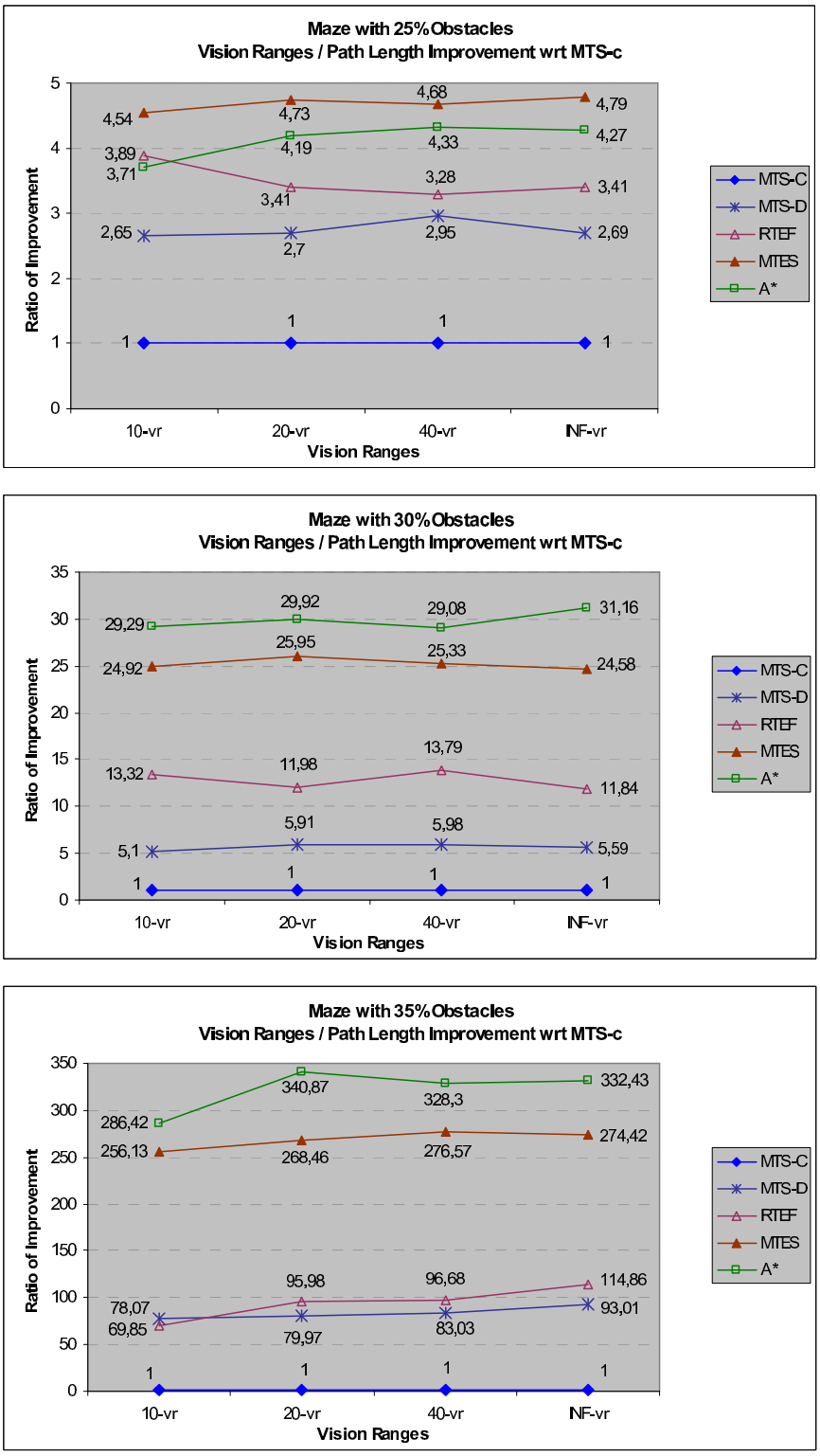

Figure 4.26: Average of path length results of maze grids (25% obstacles (top), 30% obstacles (middle), 35% obstacles (bottom)) for increasing vision ranges against a moving target

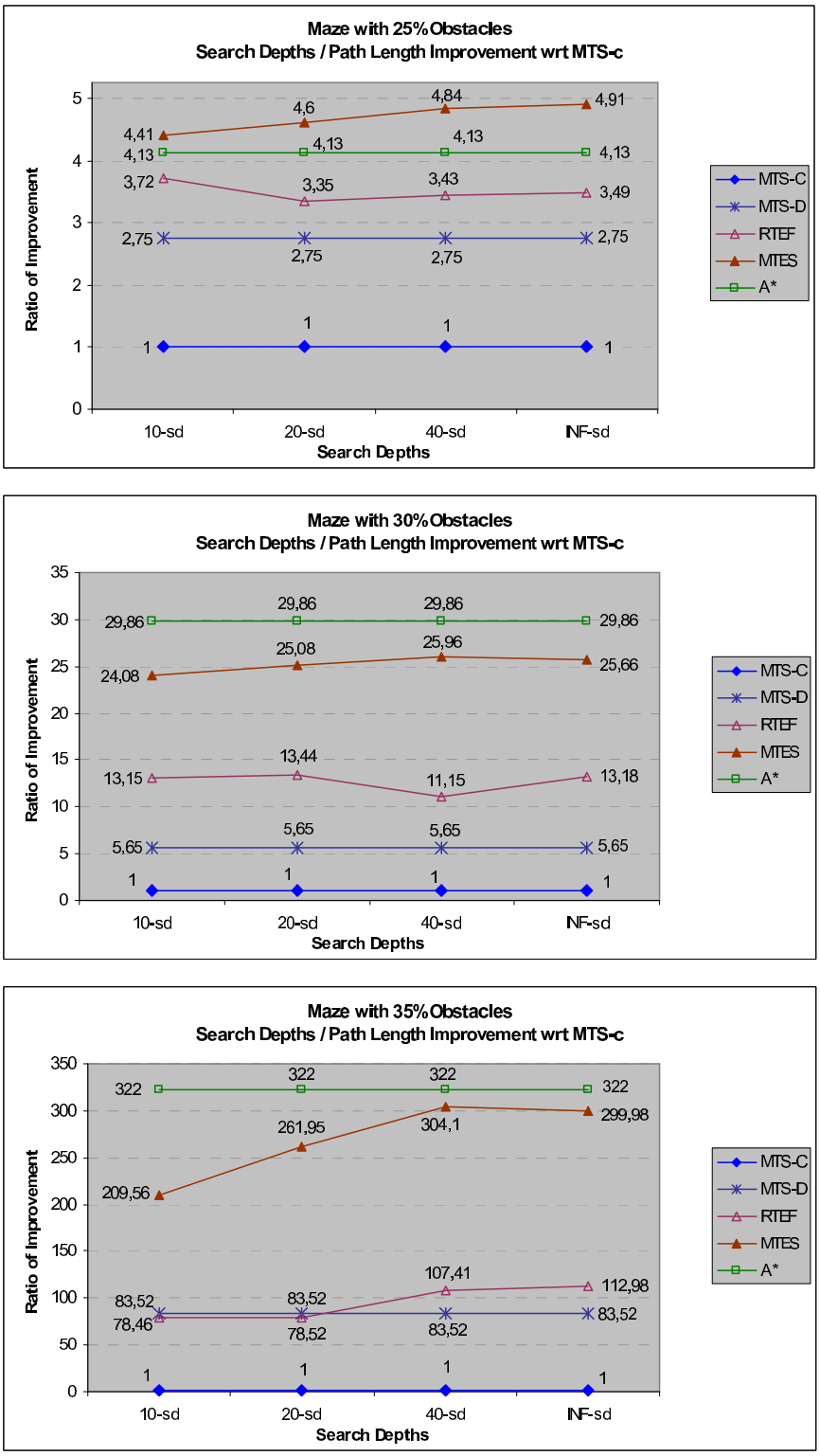

Figure 4.27: Average of path length results of maze grids (25% obstacles (top), 30% obstacles (middle), 35% obstacles (bottom)) for increasing search depths against a moving target

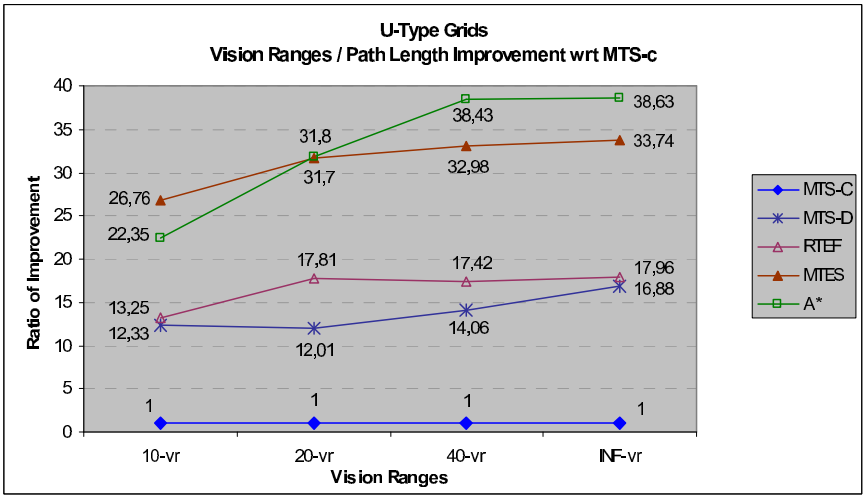

Figure 4.28: Average of path length results of U-type grids for increasing vision ranges against a moving target

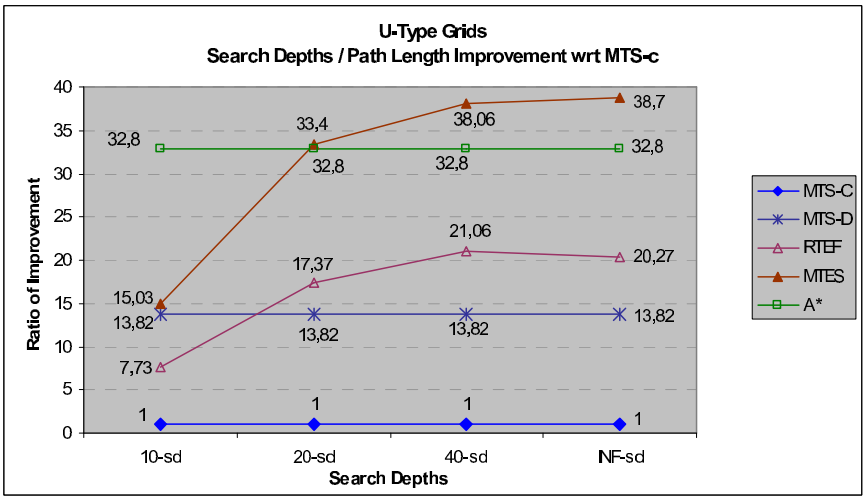

Figure 4.29: Average of path length results of U-type grids for increasing search depths against a moving target

4.6.3 Analysis of Step Execution Times

We examined the step execution times of the algorithms running on an AMD Athlon 2500+ desktop computer. In Table 4.1, the minimum and the average number of moves executed per second in maze and U-type grids are shown. The rows are for the compared algorithms and the columns are for the search depths. According to the results, we can conclude that MTS-c and MTS-d have low and almost constant step execution times whereas the efficiency of MTES and RTEF is tied to the search depth and obstacle ratio, and hence the appropriate depth should be chosen according to the required efficiency and grid type. With respect to worse case performance, A* seems to be the worst as expected, but in terms of average case performance, there is one exception. When an infinite search depth is used, A* outperforms MTES and RTEF in mazes with 35% obstacle ratio. But when we use a search depth, MTES and RTEF becomes better again almost without loosing any solution quality. This result also indicates that a reasonable search depth gains more than it takes.

Table 4.1: The minimum and average number of moves per second for increasing search depths

maze grids with 25% obstacles

S.Depth	10 Depth		20 Depth		40 Depth		INF Depth	
Algorithm	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.
MTS-c	1612	2456	1612	2456	1612	2456	1612	2456
MTS-d	1562	2324	1562	2324	1562	2324	1562	2324
RTEF	1096	1834	1063	1550	537	1282	110	1040
MTES	1063	1324	641	984	292	709	84	585
A*	17	203	17	203	17	203	17	203

maze grids with 30% obstacles

S.Depth	10 Depth		20 Depth		40 Depth		INF Depth	
Algorithm	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.
MTS-c	1562	2698	1562	2698	1562	2698	1562	2698
MTS-d	1483	2445	1483	2445	1483	2445	1483	2445
RTEF	1483	1687	641	1212	188	679	71	377
MTES	793	1073	400	649	152	358	50	211
A*	13	198	13	198	13	198	13	198

maze grids with 35% obstacles

S.Depth	10 Depth		20 Depth		40 Depth		INF Depth	
Algorithm	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.
MTS-c	1063	2874	1063	2874	1063	2874	1063	2874
MTS-d	937	2469	937	2469	937	2469	937	2469
RTEF	1000	1625	400	936	118	309	32	81
MTES	531	907	212	444	82	174	23	55
A*	20	168	20	168	20	168	20	168

U-type grids

S.Depth	10 Depth		20 Depth		40 Depth		INF Depth	
Algorithm	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.
MTS-c	1063	2855	1063	2855	1063	2855	1063	2855
MTS-d	1063	2498	1063	2498	1063	2498	1063	2498
RTEF	1063	1955	641	1319	188	665	78	406
MTES	793	1257	400	747	133	348	57	233
A*	8	104	8	104	8	104	8	104

CHAPTER 5

MULTI-AGENT REAL-TIME PURSUIT

In this chapter, we introduce our multi-agent coordinated path search algorithm, Multi-Agent Real-Time Pursuit (MAPS), for catching a moving target. The details of the algorithm is described in the following section. After the description of the algorithm, we present its complexity analysis and the experimental results.

5.1 Problem Description

In this section, we present the problem description in details, which offers the environment for testing our algorithm that can pursue a moving target in real-time with multiple agents in coordination. The assumptions of our domain different from the previous chapters are summarized below.

- There are multiple agents (predators) that aim to reach a static or moving target (prey) in coordination.
- The predators and the prey are randomly located far from each other in non-obstacle grid cells. The predators are positioned at different cells.
- The predators are expected to reach the prey in coordination as soon as possible avoiding the obstacles in real-time. The prey is either static or escaping from the predators using Prey-A* (see Algorithm 12), and its location is assumed to be known by the predators all the time.
- The unknown parts of the grid world is assumed to be free of obstacle by each predator, until it is explored. Each predator maintains its own tentative map, which holds the known part of the grid world, and updates it as he explores the environment.

- 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 5.1.1) of the diagonal neighbor cell and the current cell is also free (see Figure 5.1 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 1x1 unit, and the coordinates of the agents are stored in continuous space (i.e., in floating numbers). All the moves are performed as exactly 1 unit steps 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.

- And finally, the prey is assumed to be caught when the prey and any of the predators are in the same cell.

Definition 5.1.1 (Common Neighbors) *Common neighbors of two cells (source cells) are defined as the cells that are neighboring both source cells at the same time. For instance, if we select the first source cell as the one located in* (X, Y)*, and the second source located in* $(X + 1, Y + 1)$*, then the common neighbors of these source cells will be the ones located in* $(X + 1, Y)$ *and* $(X, Y + 1)$*.*

5.2 The Pursuit Algorithm

Multi-Agent Real-Time Pursuit (MAPS) is a multi-agent real-time moving target search algorithm, which offers two different coordination strategies: *blocking escape directions* (BES) and *using alternative proposals* (UAL). The predators may selectively

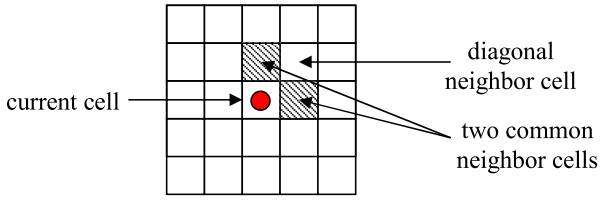

Figure 5.1: Common neighbors of the diagonal cell and the current cell

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 algorithm, Advance Real-Time Target Evaluation (ARtte), the second strategy is not generic and only applicable to our path planner. MAPS given in Algorithm 13 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 13). The *blocking location* (see Definition 5.2.1) is used as the target coordinate to be reached in the path search algorithm.

Definition 5.2.1 (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 5.2.2), which passes through that point.*

Definition 5.2.2 (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 13 An Iteration of MAPS Algorithm

1: Call BES method in Algorithm 14 to determine the *blocking location* (tx, ty) this predator will aim to
2: Call ARtte method in Algorithm 16 with the target (tx, ty) to acquire the set st containing the closed directions, and the set pr containing the best and the second best (if exists) proposals for the next move
3: **if** the set pr is empty **then**
4: **if** *history* is not empty **then**
5: Clear all the *history*.
6: Jump to 2.
7: **else**
8: Since the prey is unreachable, stop search with failure.
9: **end if**
10: **end if**
11: Call UAL method in Algorithm 19 with the parameter pr to compute the final proposed moving direction dir
12: Call the method in Algorithm 20 with the parameters (st, dir) to determine the set of utilities $ut[8]$ for the neighbor cells
13: Select mov as the neighbor cell with the highest utility in ut from the set of neighbors with non-zero utility and smallest visit count.
14: **if** mov is not empty **then**
15: Perform one step move to the selected neighbor cell mov.
16: **if** the cell location of this predator is changed after the move **then**
17: Increment the visit count of previous cell by one.
18: Insert the previous cell into the history.
19: **end if**
20: **else**
21: Clear all the *history* to be able to search for an alternative way.
22: Since the way is temporarily blocked by another predator, stop search and wait for one iteration.
23: **end if**

For performing path search, a modified version of RTTE-h algorithm, called Advance Real-Time Target Evaluation (ARtte), is used (lines 2-10 in Algorithm 13). ARtte algorithm 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 11). According to the final moving direction, the utilities of the eight neighbor cells are computed next (line 12). Then, a move is selected considering the utilities of the neighbor cells (line 13). Finally, the move (if there exists one) is performed and the agent information is updated for the next step (lines 14-23).

In order to avoid infinite loops and re-visiting the same locations redundantly, MAPS uses *visit counts* and *history* together (see Definitions 3.2.1 and 3.2.2). 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.

If the agent selects a cell to move from the eight neighbors, he performs a one-step move towards the direction of that cell. 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 keep and maintain its real coordinate in continuous space. In the following sections, some of the phases mentioned above are described in more details.

5.2.1 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 14. This phase is optional and only executed unless *escape direction blocking* is disabled (checked by line 1 in Algorithm 14). If it is disabled or there is only one predator, the current location of the prey is returned as the *blocking location* (line 2). To compute the *blocking location* of the predator, the algorithm first needs to determine n possible *escape directions* (lines 8-9), 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 balanced 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 Figure 5.2 for illustration). Thus, we share 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 Figure 5.3) 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 *blocking location* of the predator is computed 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 13-14). The validation is done by checking if there is 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. To find this location, RTA* algorithm is executed 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*.

In order to compute *blocking location* given the *escape direction*, 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

Algorithm 14 Determining the *Blocking Location*

Require: n : the number of predators

Require: hx, hy : the coordinate of this predator

Require: β : path search limit multiplier (e.g., 2)

if (n=1) or (escape direction blocking is disabled) **then**
  return the prey location as the *blocking location*
else
  Select *nearest* as the nearest predator to the prey
  if this predator is *nearest* **then**
    return the prey location as the *blocking location*
  else
    Select *escape_1st* as the direction from the prey to *nearest*
    Select *escapes* as the set of *escape directions*, which starts with *escape_1st* and contains n directions whose angle differences from its two neighbor directions are equal
    Assign predators to *escapes* such that *nearest* is always assigned to the first *escape direction*, and the total distance from predators to their *blocking locations* is minimized.
    Select *esc* as the *escape direction* assigned to this predator
    Call the method in Algorithm 15 to compute the *blocking location* (tx, ty) of this predator such that *esc* is blocked
    Execute RTA* to search a path from (hx,hy) to (tx,ty) until (tx,ty) is reached or the number of moves executed exceeds $\beta.(abs(hx - tx) + abs(hy - ty))$
    Return the coordinate nearest to (tx,ty), which is found during the path search
  end if
end if

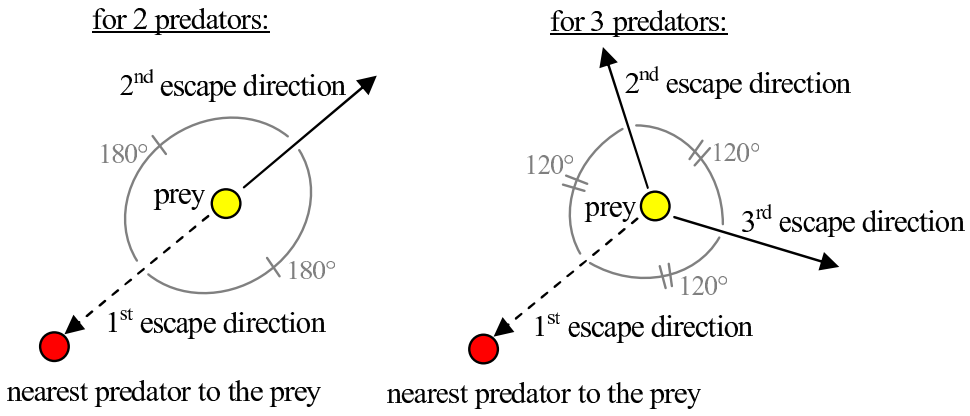

Figure 5.2: Determining the escape directions

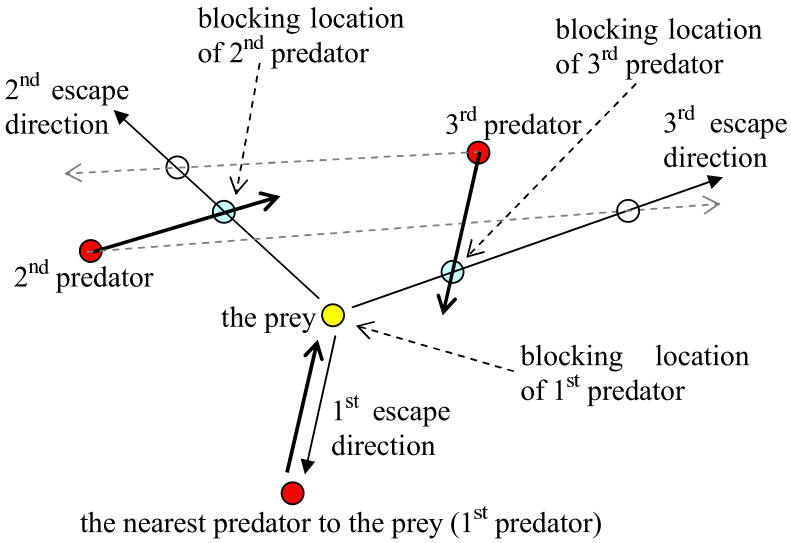

Figure 5.3: Assigning escape directions to predators minimizing the total walking cost to the blocking locations

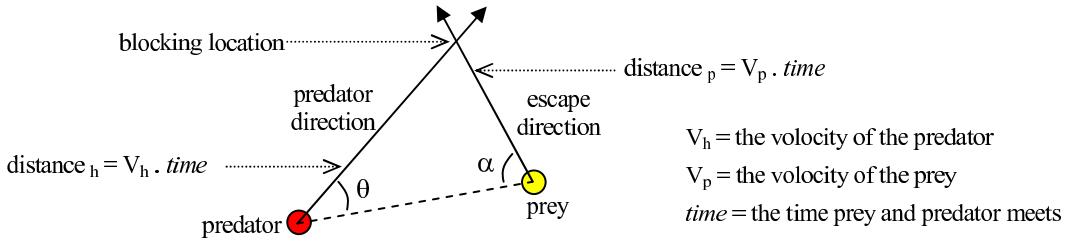

Figure 5.4: Determining the blocking location

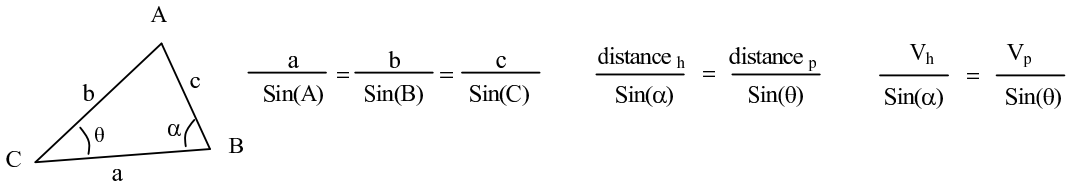

Figure 5.5: Sinus theorem

for avoiding obstacles and dead-ends. The *blocking location* is chosen as the coordinate on the way of the prey's *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 Figure 5.4.

To compute θ given in Figure 5.4, we base our formula on sinus theorem shown in Figure 5.5, and according to the sinus theorem we get the following formula:

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

The θ formula 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 15.

If α is close to 0 or 180 degrees (line 1 in Algorithm 15), we select the *blocking location* as the location of the prey (line 18) since this will make the *predator direction* (see Figure 5.4) aim to the prey location and be almost parallel to the *escape direction* causing errors in floating point computations. Otherwise, we additionally check if $\sin(\alpha).(1 + \varepsilon).V_p/V_h$ is less than or equal to 1 (line 2) 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 $\arcsin(\sin(\alpha).(1+\varepsilon).V_p/V_h)$ (line 3). And last, we have to do a test to see if θ is less than $180 - \alpha - \epsilon$ (line 4) since sinus theorem is unreliable 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.

Algorithm 15 Computing the *Blocking Location*

Require: α : the small angle between the *escape direction* and the direction from the prey to the predator

Require: V_h : the velocity of the predator

Require: V_p : the velocity of the prey

Require: ϵ : a small number (e.g., 0.5)

Require: ε : a very small number (e.g., 0.05)

Require: d_{max} : the maximum permitted distance between the *blocking location* and the prey (e.g., 100)

1: **if** $\epsilon < \alpha < 180 - \epsilon$ **then**
2: **if** $\sin(\alpha).(1 + \varepsilon).V_p/V_h \leq 1$ **then**
3: Let θ be $\arcsin(\sin(\alpha).(1 + \varepsilon).V_p/V_h)$
4: **if** $\theta < 180 - \alpha - \epsilon$ **then**
5: Let *bl* be the intersection point of lines passing through the *escape direction* and the *predator direction* computed using θ (see Figure 5.4)
6: **if** distance from the prey to *bl* $\leq d_{max}$ **then**
7: **Return** *blocking location* as *bl*
8: **else**
9: **Return** *blocking location* as the point with distance d_{max} from the prey in the *escape direction*
10: **end if**
11: **else**
12: **Return** *blocking location* as a far point with distance d_{max} from the prey in the *escape direction* since it is not possible to catch the prey
13: **end if**
14: **else**
15: **Return** *blocking location* as a far point with distance d_{max} from the prey in the *escape direction* since it is not possible to catch the prey
16: **end if**
17: **else**
18: **Return** *blocking location* as the location of the prey since the direction to the prey is almost parallel to the *escape direction*
19: **end if**

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* (lines 5-10). But we limit the distance of the *blocking location* to the prey with distance d_{max} in order to reduce the computational cost of the *blocking location* validation (see line 13 in Algorithm 14). 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 12 and 15).

5.2.2 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 7) called Advanced Real-Time Target Evaluation (ARtte) given in Algorithm 16. In ARtte, all the phases are the same as RTTE-h except *evaluating individual obstacle features* (line 7 in Algorithm 16) and *merging entire results* (line 9).

Algorithm 16 The algorithm *ARtte*

1: Mark all the moving directions as open.
2: Propagate four diagonal rays.
3: **for** each ray hitting an obstacle **do**
4: Extract the border of the obstacle by starting from the hit-point and tracing the edges towards the left side until making a complete tour around the obstacle.
5: Detect closed directions.
6: Analyze the border to extract geometric features of the obstacle.
7: Evaluate the results and determine the best and the second best (if exists) directions to avoid the obstacle.
8: **end for**
9: Merge individual results, compute the best and the second best (if exists) directions to move, and their estimated path lengths
10: **Return** the closed directions, and the best and the second best proposals for the next move

We have modified the method in *evaluating individual obstacle features* phase (see

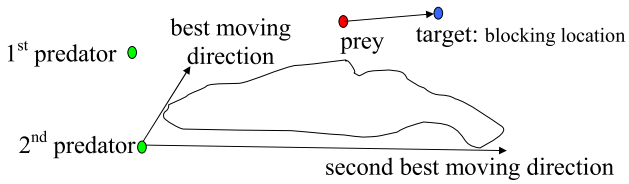

Figure 5.6: The best and second best proposed moving directions

Algorithm 8) to determine both the best and the second best proposals for individual obstacles (see Figure 5.6 for an illustration). First, we have assumed that all the previous proposals in Algorithm 8 were pointing to the best direction and its estimated path length. Next we have modified *Case 1.2* as shown in Algorithm 17 in order to propose two alternatives, the best and the second best directions and their estimated path lengths.

Algorithm 17 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**

We have also modified the method in *merging entire results* phase (see Algorithm 11) to determine the best and the second best proposals considering all the obstacles. The new method given in Algorithm 18 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 20 from now on. The details of the moving

direction determination for getting around the *most constraining obstacle* is the same as the previous algorithm.

Algorithm 18 Modified Merging Phase

1: **if** all the directions to neighbor cells are closed **then**
2: propose no moving direction and halt with failure
3: **end if**
4: Select the obstacle (*most constraining obstacle*) that is marked as blocking the target and maximize the distance to the target, if there exists one
5: **if** most constraining obstacle exists **then**
6: Determine and propose the best and the second best (if exists) moving directions that gets around the most constraining obstacle from left and/or right sides avoiding the remaining obstacles.
7: **else**
8: Propose flying direction and distance to the target as the best direction and its estimated path length, respectively.
9: **end if**

5.2.3 Selecting the Final Moving Direction

Following the selection of proposals, the best and the second best (if exists) moving directions, the algorithm needs to decide on a *final proposed moving direction*. 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 19), 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.

To compute the *final proposed moving direction*, the algorithm first determines n possible *escape directions* (lines 8-9), 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* (see Section 5.2.1). The first *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 4-6) since we would like to have someone following the prey from a short path, but the others may decide

Algorithm 19 Selecting the Final Moving Direction

Require: n : the number of predators

Require: pr : the set containing the best and the second best (if exists) proposals for the next move

if (n=1) or (using alternative proposals is disabled) or (there is no second best proposal in pr) **then**
  return the best direction proposed in pr
else
  Select *nearest* as the nearest predator to the prey
  if this predator is *nearest* **then**
    return the best direction proposed in pr
  else
    Select *escape_1st* as the direction from the prey to *nearest*
    Select *escapes* as the set of *escape directions*, which starts with *escape_1st* and contains n directions whose angle differences from its two neighbor directions are equal
    Assign predators to *escapes* such that *nearest* is always assigned to the first *escape direction*, and the total angle difference from predators (prey-predator vectors) to their *escape directions* is minimized.
    Select *edir* as the *escape direction* assigned to this predator
    Let dif as the smallest angle between *edir* and prey-predator vector.
    Compute attraction factor $afactor$ as $0.5 + dif/360$
    if the clockwise angle from prey-predator vector to *edir* is smaller than the counter clockwise one **then**
      Compute attraction direction *adirection* as 90 degree left of predator-prey vector
    else
      Compute attraction direction *adirection* as 90 degree right of predator-prey vector
    end if
    Let $angle1$ be the small angle between *adirection* and the best direction proposed in pr
    Let $angle2$ be the small angle between *adirection* and the second best direction proposed in pr
    Let $distance1$ be the estimated path length of the best direction in pr
    Let $distance2$ be the estimated path length of the second best direction in pr
    Compute $angle\_factor$ as $(angle1/180 + 1)/(angle2/180 + 1)$
    Compute $utility\_factor$ as $afactor.angle\_factor$
    Compute $utility\_of\_alternative$ as $utility\_factor.(distance1/distance2)$
    if $utility\_of\_alternative \geq 1$ **then**
      return the second best direction proposed in pr
    else
      return the best direction proposed in pr
    end if
  end if
end if

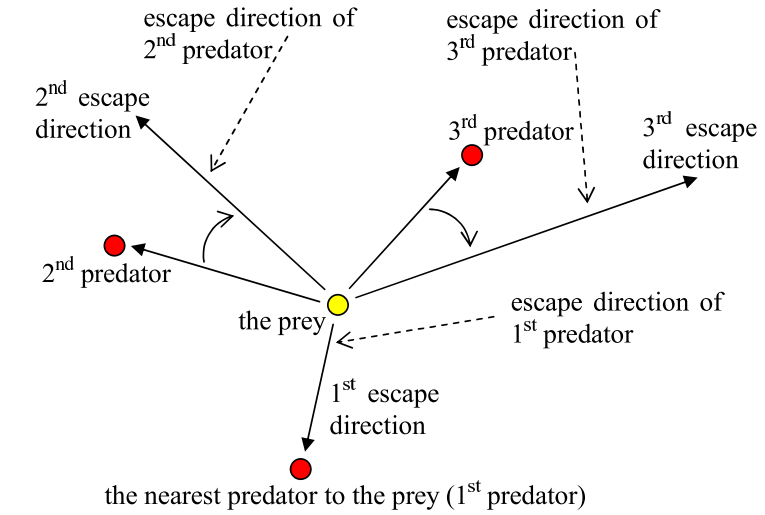

Figure 5.7: Assigning escape directions to predators minimizing the total angle difference

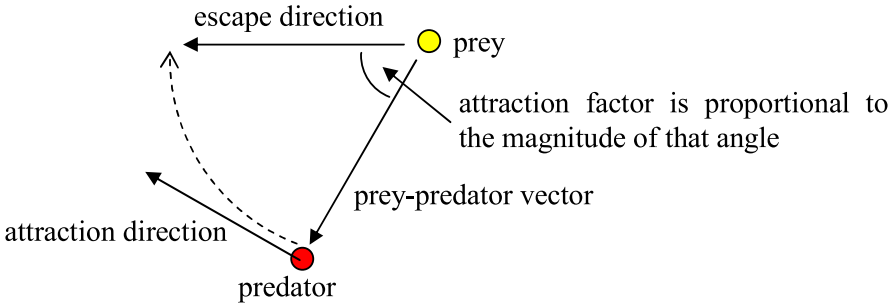

Figure 5.8: Computing attraction direction

going from longer routes for the sake of better coordination. After selecting the *escape directions*, the algorithm assigns *escape directions* to predators optimally (see Figure 5.7 for illustration) such that the total angle difference between the directions from the prey to the predators and their assigned *escape directions* is minimized and the first *escape direction* is assigned to the nearest predator to the prey (line 10). Then we compute the *attraction factor* and the *attraction direction* as given in lines through 11 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 Figure 5.8 for illustration). Finally we compute the utility of the second best alternative (lines 19-25), 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 26-30). The utility formula is determined such that it selects the second best alternative routes that are at most two times longer than the best one.

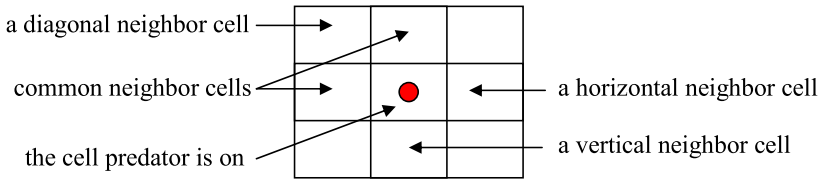

Figure 5.9: Neighbor cells of the predator

5.2.4 Computing the Utilities of Neighbor Cells

After determining the final proposed moving direction, we compute the utilities of eight neighbor cells (see Algorithm 20) 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*. Otherwise we branch 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 5.1.1) of the diagonal cell and the cell the predator is on (see Figure 5.9 for illustration). If common neighbors are both not obstacles, and at least one of the directions to these cells are not closed by the ARtte algorithm, then we set the utility of the diagonal cell to $(181 - dif)/181$, where dif is the smallest angle between the final proposed moving direction and the direction of the diagonal neighbor cell. Otherwise, we set the utility of the diagonal cell to *zero*. If the neighbor cell is a horizontal or vertical one, then we set the utility to $(181 - dif)/181$ if the direction to that cell is not closed, else we set the utility to *zero*.

A complete sample illustrating the entire process of MAPS is given in Figure 5.10. 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 in to 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.

Algorithm 20 Computing the Utilities of Neighbor Cells

Require: st : the set containing the closed directions, which may be north, south, east or west

Require: dir : the final proposed direction

```
for each neighbor cell do
  if the neighbor cell is not an obstacle and not blocked by another predator then
    if this is a diagonal neighbor cell then
      Determine the set com, which contains two common neighbors of the current
      neighbor cell and the cell this predator is on
      if (the cells in com are not obstacles) and (at least one of the directions
      of cells in com is not marked as closed in st) then
        set utility of the neighbor cell to (181−dif)/181, where dif is the smallest
        angle between dir and the direction of the neighbor cell
      else
        set utility of neighbor cell to zero
      end if
    else
      if the direction of the neighbor cell is not marked as closed in st then
        set utility of the neighbor cell to (181−dif)/181, where dif is the smallest
        angle between dir and the direction of the neighbor cell
      else
        set utility of the neighbor cell to zero
      end if
    end if
  else
    set utility of the neighbor cell to zero
  end if
end for
return utilities of the neighbor cells
```

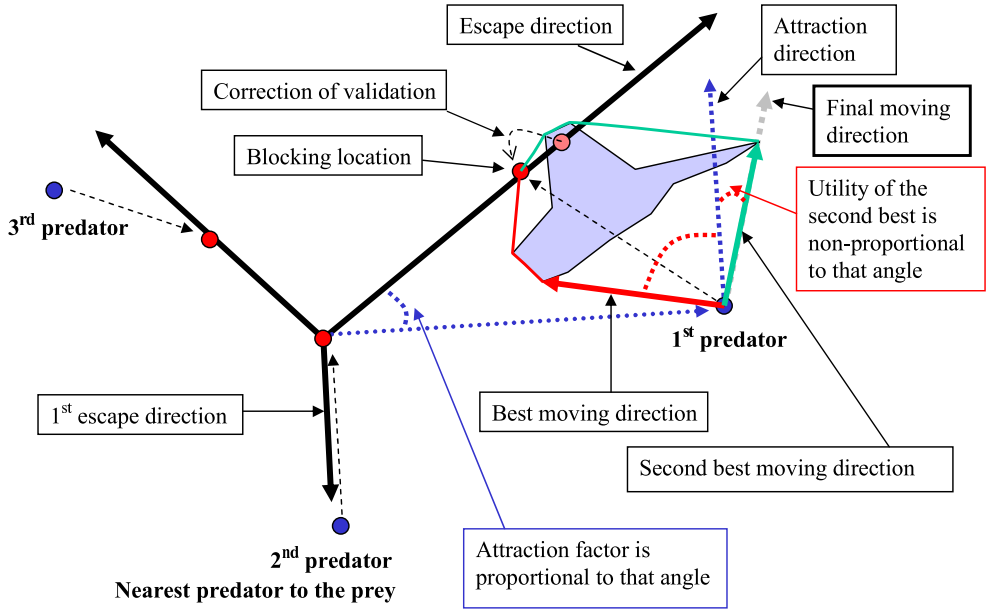

Figure 5.10: A complete sample illustrating the entire process of MAPS

5.3 Complexity Analysis

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. The complexity of the path search is the same as MTES, which is $O(w.h)$ per step, where w is the width and h is the height of the grid. 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.

5.4 Experimental Results

In this section, we present the experimental results of our coordinated pursuit algorithm against static and moving preys guided by Prey-A* (see Algorithm 12). 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). In Prey-A*, we used 161x161 sized search window. In each iteration, the predators and the prey are executed alternately in order to prevent the side effects caused by the difference in efficiency of the algorithms.

For the test runs, we used 9 randomly generated sample grids of size 150x150. Six of them were the *maze* grids (see Figure 5.11), and three of them were the *U-type* grids (see Figure 5.12). 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 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. 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.

5.4.1 Analysis of Static Targets

In this section, we examine the effect of coordination against static targets, and present the results of experiments conducted with different configurations. In Figure 5.15, the

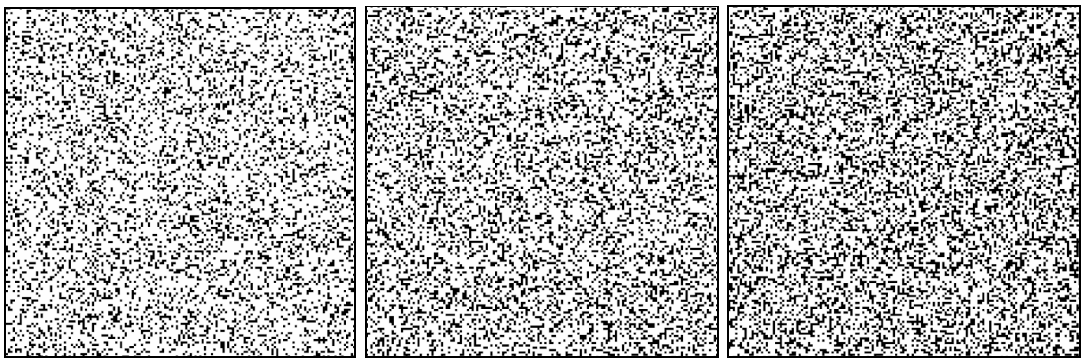

Figure 5.11: Maze grids with %25 (left), %30 (middle) and 35% (right) obstacles

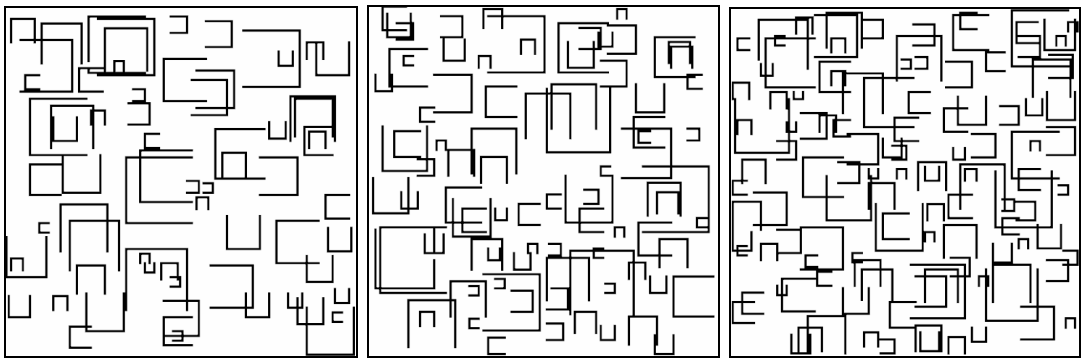

Figure 5.12: U-type grids with 70 (left), 90 (middle) and 120 (right) u-type obstacles

results for various team sizes, vision ranges and initial locations of the predators are seen, and in Figure 5.14, the results in terms of maze and U-type grids are given. 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.

According to the experimental results, the coordination does not seem to offer any performance increase against a static target, even it sometimes makes the results slightly worse since the path lengths of the predators get longer for the sake of coordination, as easily seen in Figure 5.15. The predators using coordination strategies try to surround the prey in all directions assuming that the prey may start escaping in any time. But, since the prey will not escape till the end and the nearest predator to the prey directly aims to the current location of the prey without taking into consideration the locations of the other predators, the number of moves to reach the prey is usually determined by that nearest predator to the prey, whether the coordination is enabled or not.

The results also showed that the number of predators involved in the search or the vision range used do not effect the results very significantly. The average number of

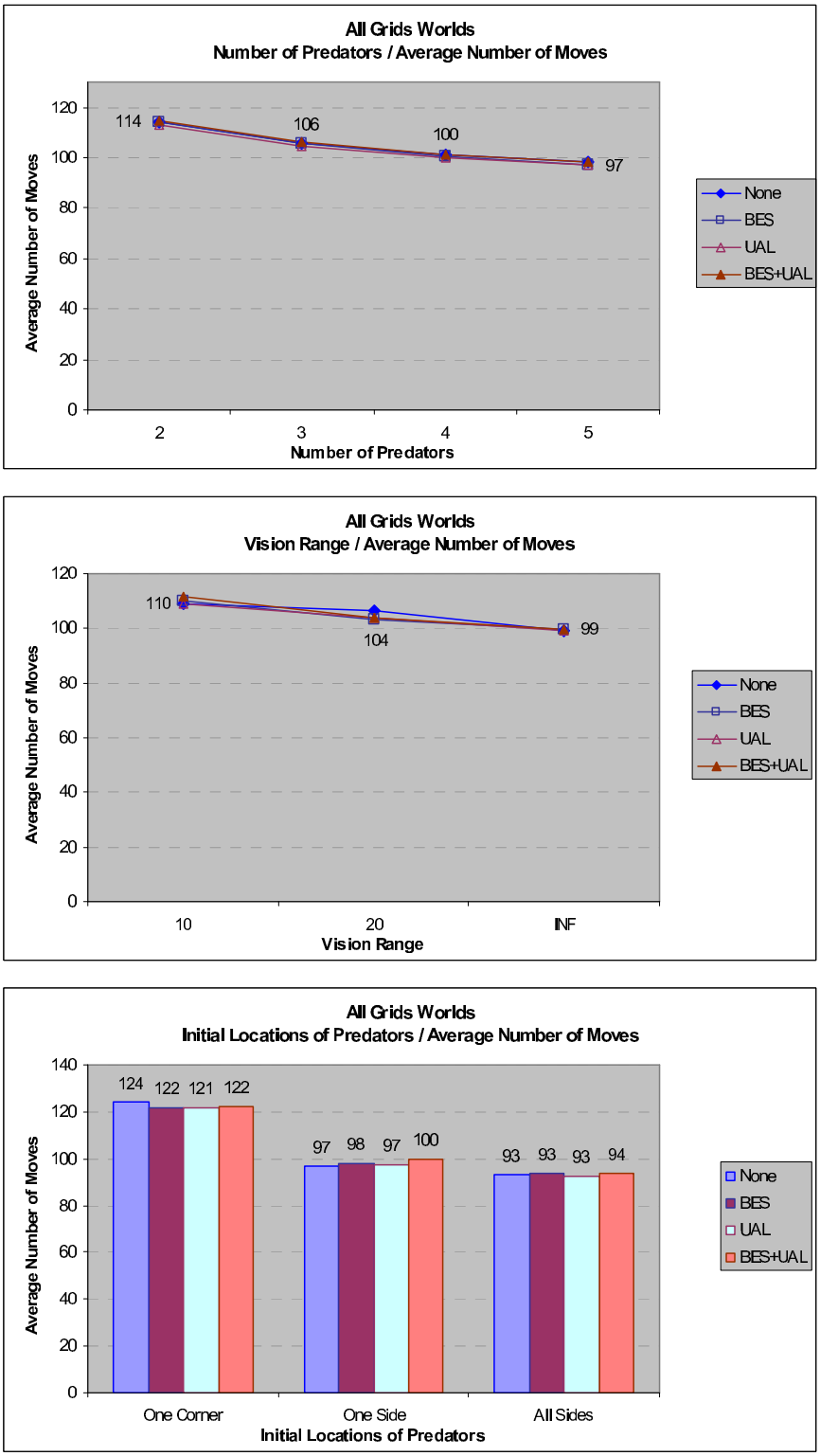

Figure 5.13: Average number of moves to reach a static prey for different number of predators (top), vision ranges (middle) and initial locations of predators (bottom)

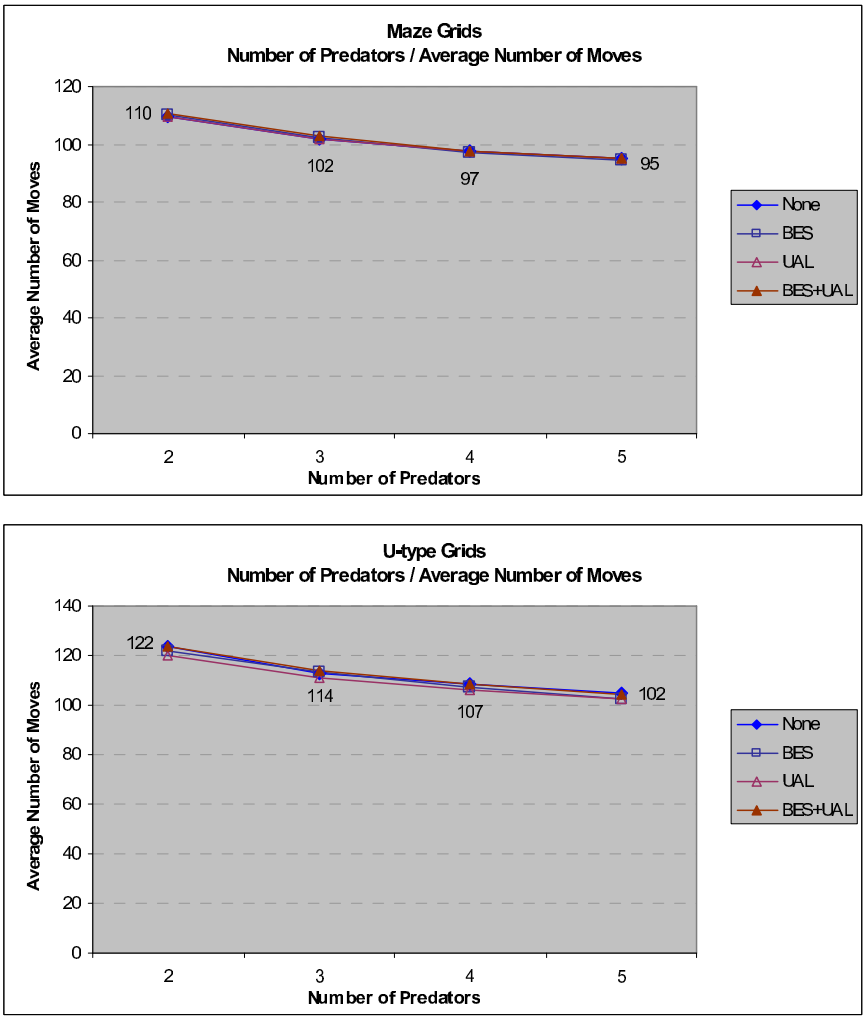

Figure 5.14: Average number of moves to reach a static prey in maze (top) and U-type (bottom) grids

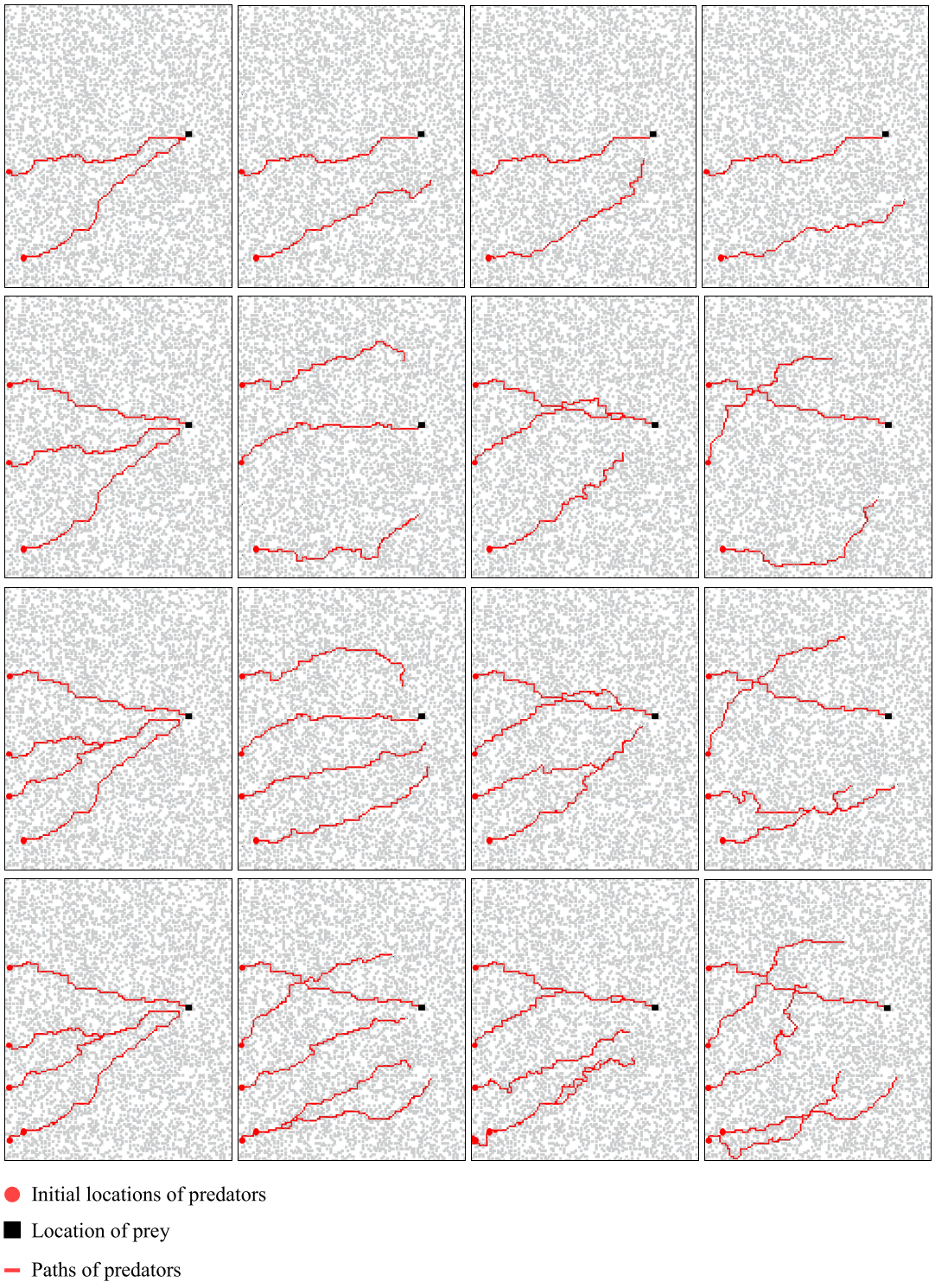

Figure 5.15: 2, 3, 4 and 5 predators (in rows) starting from left side against a static prey: No coordination (first column), coordination with BES (second column), UAL (third column) and BES+UAL (fourth column)

moves to reach the target is 114 with 2 predators, and 97 with 5 predators. Similarly, the average number of moves to reach the target is 110 with 10 vision range, and 99 with infinite vision range. This is not very surprising since with our strategy in locating agents, the average euclidian distance between randomly located predators and preys will be about 72 in a grid world with 150x150 size, and by increasing the number of predators, we just increase the probability to have a shorter distance between the prey and the nearest predator to the prey, but not more than that. And since our grid worlds are no very complicated (obstacle percentage is less than or equal to 35), increasing the vision range does not gain much.

And finally, when we examine the effect of the strategy in selecting the initial locations of the predators, we observe that locating the predators in one of the corners of the grid world increase the number of moves to reach the prey since the corners are farthest from the target.

5.4.2 Analysis of Moving Targets

In this section, we examine multi-agent coordination against moving preys guided by Prey-A*. In Figures 5.16, we present the experimental results with respect to team size, vision range and initial locations of the predators, and in Figure 5.17, 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 prey behind, blocking the second escape direction (laying in front of the prey) is a hard

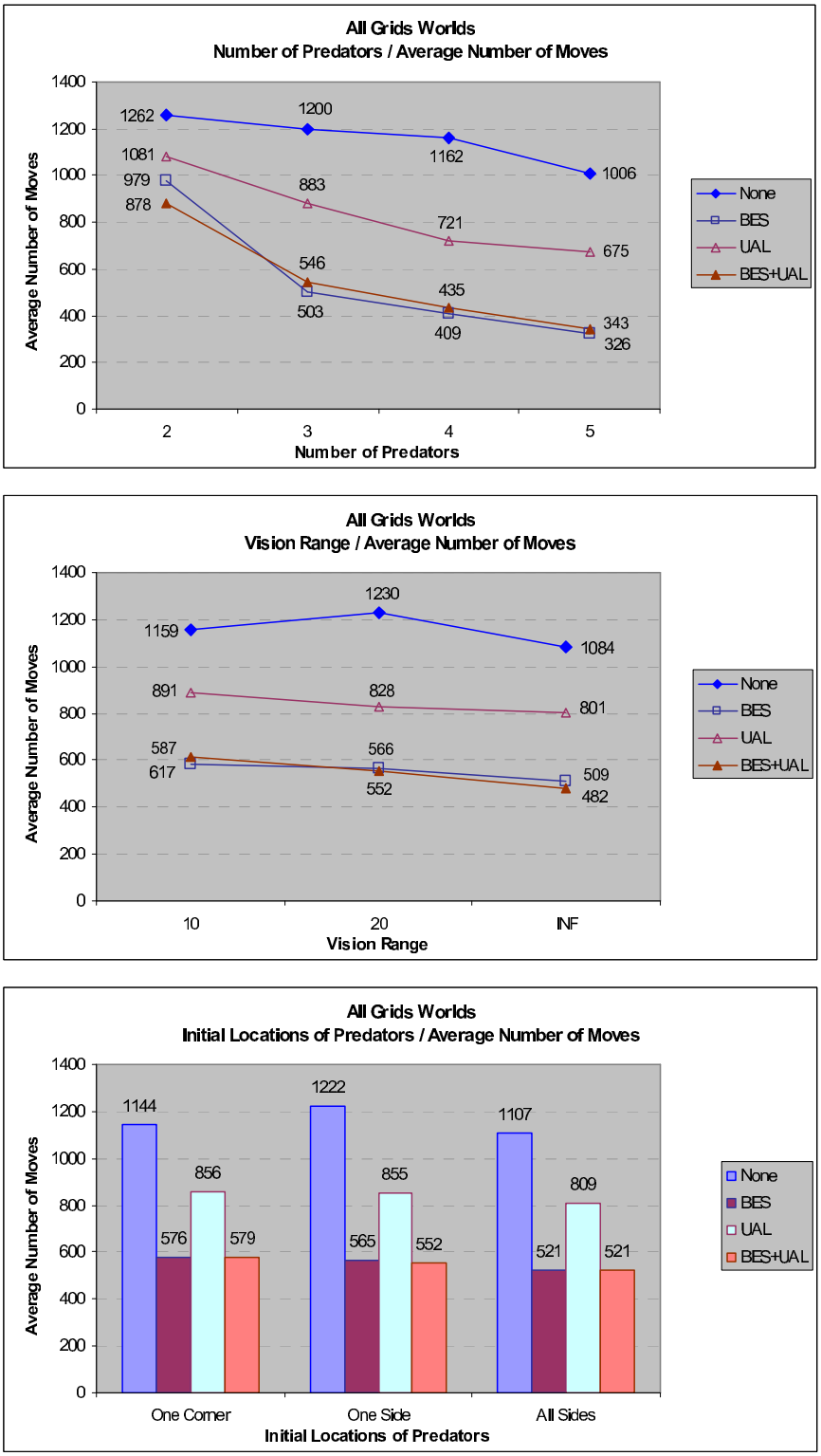

Figure 5.16: Average number of moves to reach a moving prey for different number of predators (top), vision ranges (middle) and initial locations of predators (bottom)

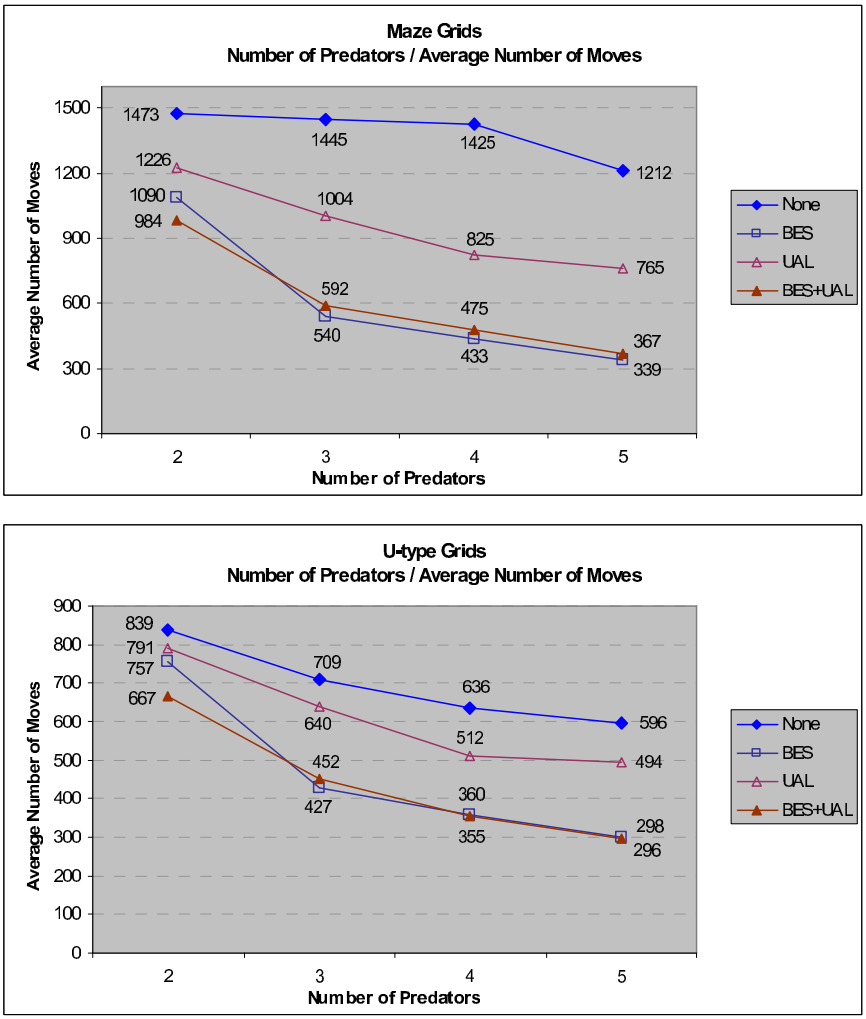

Figure 5.17: Average number of moves to reach a moving prey in maze (top) and U-type (bottom) grids

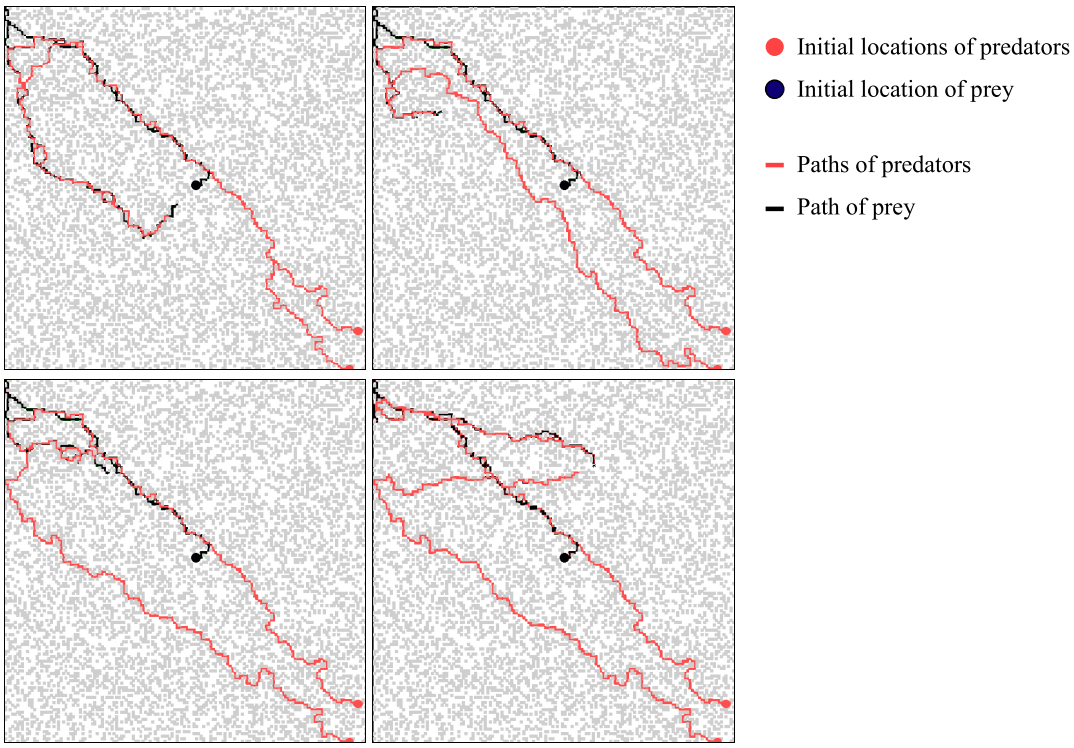

Figure 5.18: 2 predators against a moving prey: No coordination (top-left), coordination with BES (top-right), UAL (bottom-left) and BES+UAL (bottom-right)

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 Figures 5.18 to 5.21, 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 performing a quick manoeuvre last 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. We also observe that BES+UAL strategy spread the predators the most.

With respect to vision range, we see similar results that are observed in static targets. Increasing the range does not effect the performance much since the grids have low obstacle ratio, therefore they are not very challenging. But in contrast with

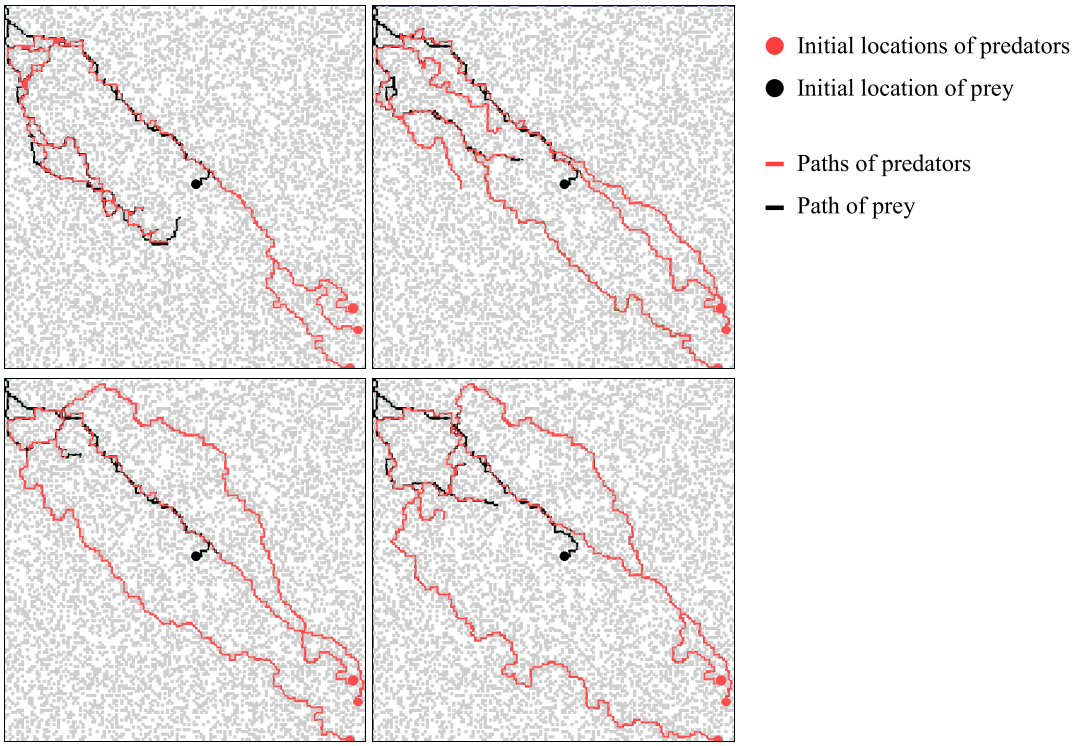

Figure 5.19: 3 predators against a moving prey: No coordination (top-left), coordination with BES (top-right), UAL (bottom-left) and BES+UAL (bottom-right)

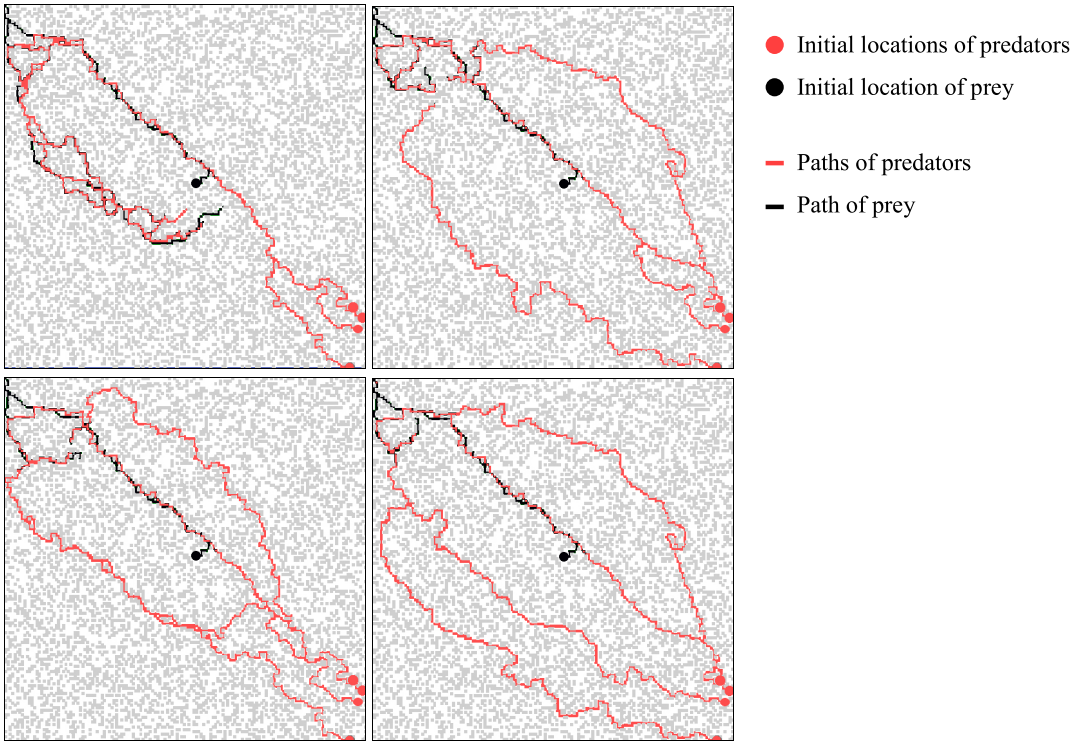

Figure 5.20: 4 predators against a moving prey: No coordination (top-left), coordination with BES (top-right), UAL (bottom-left) and BES+UAL (bottom-right)

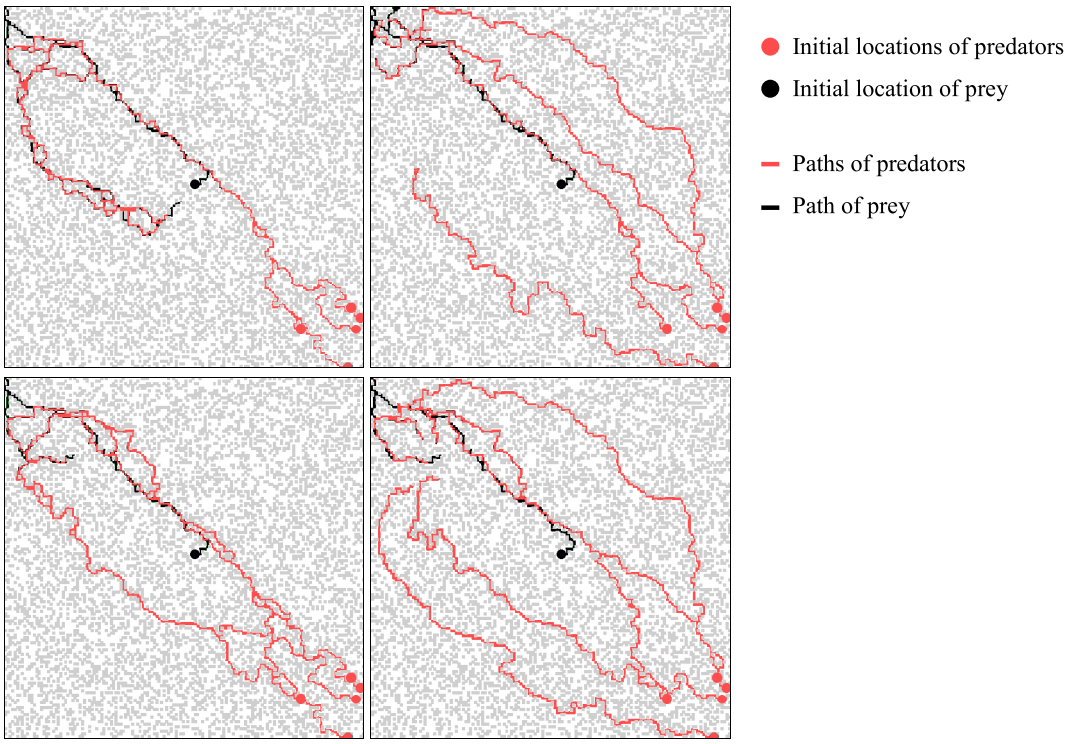

Figure 5.21: 5 predators against a moving prey: No coordination (top-left), coordination with BES (top-right), UAL (bottom-left) and BES+UAL (bottom-right)

the static targets, the initial locations of the predators are also insignificant, and slightly changes the results.

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 5.1 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 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 although this was not the case for static targets. 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.

5.4.3 Analysis of Step Execution Times

We 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 5.2, the average number of moves executed per second per predator in maze and U-type 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 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 5.1: 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

All grids

Number of Predators	None		BES		UAL		BES+UAL	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
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

Number of Predators	None		BES		UAL		BES+UAL	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
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

Number of Predators	None		BES		UAL		BES+UAL	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
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

Number of Predators	None		BES		UAL		BES+UAL	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
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

Number of Predators	None		BES		UAL		BES+UAL	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
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

Table 5.2: The average number of moves per second per predator for different coordination strategies and predator team sizes

maze grids with 25% obstacles

Algorithm	2 Predators	3 Predators	4 Predators	5 Predators
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

Algorithm	2 Predators	3 Predators	4 Predators	5 Predators
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

Algorithm	2 Predators	3 Predators	4 Predators	5 Predators
None	216	210	206	207
BES	191	180	169	169
UAL	211	210	199	198
BES+UAL	187	177	166	166

U-type grids

Algorithm	2 Predators	3 Predators	4 Predators	5 Predators
None	547	534	507	492
BES	408	388	376	375
UAL	550	522	504	447
BES+UAL	418	389	377	376

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we have focused on real-time search in partially observable grid worlds, and realized three algorithms namely Real-Time Edge Follow (RTEF), Real-Time Moving Target Evaluation Search (MTES) and Multi-Agent Real-Time Pursuit (MAPS).

First, we have described RTEF in details, stated its complexity analysis and proof of correctness clearly, and presented the results of the experiments. The experiments showed that RTEF is able to make use of environmental information acquired during the search successfully, and brings significant performance improvement over RTA* with respect to path length in all types of grids. Especially, the improvement is the highest in grids with wide corridors (e.g., U-type grids and maze grids with corridor size greater than 1) since these grids are difficult for RTA* because of their high branching factor and large heuristic depression area to be filled up. The tests also showed that the path length improvement of RTA* with reasonable look-ahead depths is still insignificant compared to RTEF. With respect to execution time, we observed that the total time to reach the goal for RTEF is usually better than RTA* in maze and U-type grids although the time per move of RTEF is very high compared to RTA*. This improvement is due to much shorter paths found by RTEF. But in random grids, the total execution time of RTEF is worse than RTA* because random grids are not very challenging for both RTEF and RTA*. We also observed that small look-ahead depths (e.g., 2-5) do not improve the execution time of RTA* significantly, and large look-ahead depths make the execution times unreasonably high. Finally, we have introduced a search depth to RTEF guaranteeing a constant complexity. The test results clearly demonstrated that search depth significantly decreases the execution time per step, but this advantage is balanced with longer paths, and therefore the total execution time generally becomes more or less the same on the average.

Later on, we have examined the problem of pursuing a moving target in grid worlds, and introduced our single agent real-time search (predator) algorithm, MTES, and our off-line moving target (prey) algorithm, Prey-A*. We have presented the comparison results of RTEF, MTES, MTS-c, MTS-d and A* against a moving target controlled by Prey-A*. Since Prey-A* is an off-line algorithm, it was very successful in escaping from the predators. As a matter of fact, it is not easy to use Prey-A* in practice because the execution time per move is very high as being an off-line algorithm. With respect to path lengths, the experimental results showed that MTES performs significantly ahead of RTEF, MTS-c and MTS-d, and competes with A*, especially in U-type grids. In the test runs, we have also observed that the two MTS versions are significantly different from each other. Although, MTS-d performs acceptably good and competes with RTEF, MTS-c almost never offers good solutions. In terms of step execution times, we observed that MTS-c and MTS-d are the most efficient algorithms, and almost spend constant time in each move. But their solution path lengths are usually very long. RTEF and MTES follow MTS respectively, and their efficiency is inversely proportional to the increase in the obstacle density. Finally, A* is always the worst except in highly dense grids. In such grids, the efficiency of MTES and RTEF sometimes drops below A* unless a limited search depth is used, which seems to be worth since, with reasonable search depths, we significantly gain efficiency almost without loosing solution quality.

Finally, we have presented our 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 offers no improvement against a static target, but significantly reduces the number of moves to reach 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 this thesis, we have proposed a pursuit algorithm, which estimates the escape directions of the prey analytically without 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 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. Additionally, we have developed a deliberative prey algorithm in order to place a powerful rival against the predators. Although this algorithm is strong enough most of the time, it is slow, and we think the algorithm can be improved furthermore in terms of both escape capability and execution time efficiency.

REFERENCES

[1] M. Benda, V. Jagannathan, and R. Dodhiawalla. On optimal cooperation of knowledge sources. *Technical Report No.BCS-G2010-28, Boeing Advanced Technology Center*, 1986.

[2] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. *In Proceedings of Int'l Conf. on Intelligent Robots and Systems*, pages 2383–2388, 2002.

[3] P. Cheng and S. M. LaValle. Resolution complete rapidly-exploring random trees. *In Proceedings of IEEE Int'l Conf. on Robotics and Automation*, pages 267–272, 2002.

[4] S. Edelkamp and J. Eckerle. New strategies in real-time heuristic search. *In Proceedings of the AAAI-97 Workshop on On-Line Search*, pages 30–35, 1997.

[5] G. Erus. 2lrl: A two-level multiagent reinforcement learning algorithm with communication. M.S. Thesis in Cognitive Sciences Program of Middle East Technical University, 2002.

[6] D. Furcy and S. Koenig. Speeding up the convergence of real-time search. *In Proceedings of AAAI*, pages 891–897, 2000.

[7] D. Furcy and S. Koenig. Combining two fast-learning real-time search algorithms yields even faster learning. *In Proceedings of the 6th European Conference on Planning*, 2001.

[8] M. Goldenberg, A. Kovarsky, X. Wu, and J. Schaeffer. Multiple agents moving target search. *Int'l Joint Conf. on Artificial Intelligence, IJCAI*, pages 1536–1538, 2003.

[9] J. Gutmann, M. Fukuchi, and M. Fujita. Real-time path planning for humanoid robot navigation. *Int'l Joint Conf. on Artificial Intelligence IJCAI-05*, pages 1232–1237, 2005.

[10] B. Hamidzadeh and S. Shekhar. Dynoraii: A real-time path planning algorithm. *Int'l Journal on Artificial Intelligence Tools*, 2(1):93–115, 2005.

[11] T. Haynes and S. Sen. Evolving behavioral strategies in predators and prey. *Springer Book on Adaptation and Learning in Multiagent Systems*, 1996.

[12] T. Haynes and S. Sen. The evolution of multiagent coordination strategies. *Adaptive Behavior*, 1997.

[13] C. Hernandez and P. Meseguer. Lrta*(k). *Int'l Joint Conf. on Artificial Intelligence IJCAI-05*, pages 1238–1243, 2005.

[14] D. Hsu, R. Kindel, J. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *Int. J. Robotics Research*, 21(3):233–255, 2002.

[15] T. Ishida and R. Korf. Moving target search: A real-time search for changing goals. *IEEE Trans Pattern Analysis and Machine Intelligence*, 17(6):97–109, 1995.

[16] Y. Ishiwaka, T. Sato, and Y. Kakazu. 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, 2003.

[17] I. Kamon, E. Rivlin, and E. Rimon. A new range-sensor based globally convergent navigation algorithm for mobile robots. *In proceedings of the IEEE Int'l Conf. on Robotics and Automation*, 1:429–435, 1996.

[18] L. Kavraki and J. Latombe. *Probabilistic Roadmaps for Robot Path Planning.* In Practical Motion Planning in Robotics: Current and Future Directions. Addison-Wesley, 1998.

[19] Y. Kitamura, K. Teranishi, and S. Tatsumi. Organizational strategies for multiagent real-time search. *In Proceedings of Int'l Conference on Multi-Agent Systems (ICMAS-96)*, pages 150–156, 1996.

[20] K. Knight. Are many reactive agents better than a few deliberative ones? *In Proceedings of the 10th Int'l Joint Conference on Artificial Intelligence*, pages 432–437, 1993.

[21] S. Koenig. A comparison of fast search methods for real-time situated agents. *AAMAS 2004*, pages 864–871, 2004.

[22] S. Koenig and M. Likhachev. D* lite. *In Proceedings of the National Conference on Artificial Intelligence*, pages 476–483, 2002.

[23] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. *In Proceedings of the Int'l Conf. on Robotics and Automation*, 2002.

[24] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *Transactions on Robotics*, 21(3):354–363, 2005.

[25] S. Koenig and M. Likhachev. Real-time adaptive a*. *5th Int'l Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 281–288, 2006.

[26] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine*, 2004.

[27] A. Konar. *Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of Human Brain.* CRC Press LLC, 2000.

[28] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[29] R. Korf. A simple solution to pursuit games. *In Working Papers of the 11th International Workshop on Distributed Artificial Intelligence*, pages 183–194, 1992.

[30] R. Kota, S. Braynov, and J. Llinas. Multi-agent moving target search in a hazy environment. *IEEE Int'l Conf. on Integration of Knowledge Intensive Multi-Agent Systems*, pages 275–278, 2003.

[31] J. Kuffner and J. Latombe. Goal-directed navigation for animated characters using real-time path planning and control. *In Proceedings of CAPTECH '98: Workshop on Modeling and Motion Capture Techniques for Virtual Environments*, pages 26–28, 1998.

[32] J. Kuffner and J. Latombe. Fast synthetic vision, memory, and learning models for virtual humans. *In Proceedings of Computer Animation '99*, pages 118–127, 1999.

[33] S. LaValle and J. Kuffner. Randomized kinodynamic planning. *In Proceedings of the IEEE Int'l Conf. on Robotics and Automation (ICRA'99)*, 1999.

[34] S. M. LaValle and J. J. Kuffner. *Rapidly-exploring random trees: Progress and prospects*, pages 293–308. Algorithmic and Computational Robotics: New Directions. A K Peters, Wellesley, MA, 2001.

[35] R. Levy and J. Rosenschein. A game theoretic approach to the pursuit problem. *11th Int'l Workshop on Distributed Artificial Intelligence*, 1992.

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

[37] A. Mudgal, C. Tovey, S. Greenberg, and S. Koenig. Bounds on the travel cost of a mars rover prototype search heuristic. *SIAM Journal on Discrete Mathematics*, 19(2):431–447, 2005.

[38] S. Russell and P. Norving. *Artificial Intelligence: a modern approach.* Prentice Hall, Inc., 1995.

[39] G. Sanchez, F. Ramos, and J. Frausto. Locally-optimal path planning by using probabilistic roadmaps and simulated annealing. *Proceedings IASTED Robotics and Applications International Conference*, 1999.

[40] M. Shimbo and T. Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1–41, 2003.

[41] A. Stentz. Optimal and efficient path planning for partially-known environments. *In Proceedings of the IEEE Int'l Conf. on Robotics and Automation*, 1994.

[42] A. Stentz. The focussed D* algorithm for real-time replanning. *In Proceedings of the Int'l Joint Conference on Artificial Intelligence*, 1995.

[43] K. Sugihara and J. Smith. 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, 1997.

[44] A. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., 1996.

[45] P. Thorpe. A hybrid learning real-time search algorithm. *Master's Thesis, Computer Science Department, University of California at Los Angeles*, 1994.

[46] C. Undeger. Real-time mission planning for virtual human agents. M.Sc. Thesis in Computer Engineering Department of Middle East Technical University, 2001.

[47] C. Undeger and F. Polat. Real-time edge follow: A real-time path search approach. *IEEE Transaction on Systems, Man and Cybernetics, Part C, (In press).*

[48] C. Undeger and F. Polat. Rttes: Real-time search in dynamic environments. *Applied Intelligence (In press).*

[49] C. Undeger and F. Polat. Real-time target evaluation search. *5th Int'l Joint Conf. on Autonomous Agents and Multiagent Systems, AAMAS-06*, pages 332–334, 2006.

[50] C. Undeger, F. Polat, and Z. Ipekkan. Real-time edge follow: A new paradigm to real-time path search. *The Proceedings of GAME-ON 2001*, 2001.

[51] L. V.J. and T. Skewis. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algoritmica*, 2:403–430, 1987.

[52] D. Xiao and A. Tan. Cooperative cognitive agents and reinforcement learning in pursuit game. *In Proceedings of 3rd Int'l Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS'05)*, 2005.

[53] C. Yong and R. Miikkulainen. Cooperative coevolution of multi-agent systems. *Technical Report: AI01-287, University of Texas at Austin*, 2001.

CURRICULUM VITAE

PERSONEL INFORMATION

Surname, Name	: Ündeğer, Çağatay
Date and Place of Birth	: 7 April 1975, İzmit
Nationality	: Turkish (T.C.)
Mobile	: +90-532-6649015
e-mail	: cagatay@undeger.com

EDUCATION

Degree	Institution	Year of Graduation
Ph.D.	METU, Computer Engineering	2007
M.Sc.	METU, Computer Engineering	2001
BSc.	KOU, Computer Engineering	1998

WORK EXPERIENCE

Year	Place	Enrollment
2001-	STM Tic.A.Ş.	Modeling & Simulation Expert
1999-2001	METU, Computer Engineering	Research Assistant
1996-1999	D-Bilgisayar Market Tic.Ltd.Şti.	Software Developer
1995-1996	TV 41 Tic.A.Ş.	Software Developer
1994-1995	Metropol TV Tic.A.Ş.	Software Developer

PROJECT EXPERIENCE

Year	Place & Project	Enrollment
2007-	NATO, NMSG-050 Task Group	Turkish Representative
2006-	NATO, IST-065 Task Group	Turkish Representative
2002-	Turkish General Staff, MÜHATEM	Manager
2003-2005	NATO, IST-038 Task Group	Turkish Representative
1999-2001	TUAF-METU MODSIMMER, Sensim	Assistant Manager

FOREIGN LANGUAGES

Advance English

PUBLICATIONS

1. Cagatay Undeger and Faruk Polat. "Moving Target Search in Grid Worlds". 6th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS-07, Hawai, 2007.

2. Cagatay Undeger and Faruk Polat. "RTTES: Real-Time Search in Dynamic Environments". Applied Intelligence (In Press).

3. Cagatay Undeger and Faruk Polat. "Real-Time Target Evaluation Search". 5th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS-06, 332-334, Japan, 2006.

4. Cagatay Undeger and Faruk Polat. " Real-Time Edge Follow: A Real-Time Path Search Approach". IEEE Transaction on Systems, Man and Cybernetics, Part C (In Press).

5. E.Kapusuz, S.Girgin, M.Tan, E.Çilden, M.Balcı, E.Göktürk, V.Koç, A.Coşar, A.Yavaş, Ç.Ündeğer and F.Polat. "Küçük Ölçekli Harekatın Modellenmesi ve Simülasyonu". USMOS 2005, ODTÜ, Ankara, Türkiye, 2005.

6. M.Fatih Hocaoğlu, Cüneyd Fırat, Faruk Sarı, Nurşen Sarı, Şeref Paşalıoğlu, Savaş Öztürk, Ramazan Cengiz, Erdem S.İlhan, Ahmet Arif Ergin, Aşkın Erçetin and Çağatay Ündeğer. "Eğitim, Mimari Tasarım, Optimizasyon ve Konsept Geliştirmede C4ISR Simülasyon Aracı". USMOS 2005, ODTÜ, Ankara, Türkiye, 2005.

7. Joachim Biermann, LtCol Louis de Chantal, Reinert Korsnes, Jean Rohmer, Cagatay Undeger. " From Unstructured To Structured Information in Military Intelligence - Some Steps To Improve Information Fusion". NATO Systems Concepts and Integration Panel Conference on SCI Methods and Technologies for Defense Against Terrorism, London, United Kingdom, 2004.

8. Cagatay Undeger, Askin Ercetin and Ziya Ipekkan. "Modeling Command & Control Centers". NATO Modeling and Simulation Group Conference on C3I and Modelling and Simulation (M&S) Interoperability, Antalya, Turkey, Oct 2003.

9. Cagatay Undeger, Murat Balci, Sertan Girgin, Volkan Koc, Faruk Polat, Sukru Bilir and Ziya Ipekkan. "Sensor Platform Optimization and Simulation for Surveillance of Large Scale Terrains".Proceedings of I'ITSEC 2002 Conference on Modeling and Simulation, Orlando, Florida, 2002.

10. Cagatay Undeger, Faruk Polat, and Ziya Ipekkan. "Real Time Edge Follow: A New Paradigm To Real-Time Path Search".Proceedings of GAME-ON 2001, London, England, Dec 2001.

11. Cagatay Undeger. " Real-Time Mission Planning For Virtual Human Agents".M.S. Thesis in Computer Engineering Department of Middle East Technical University, Jan 2001.

12. Cagatay Undeger, Veysi Isler, and Ziya Ipekkan. "An Intelligent Action Algorithm for Virtual Human Agents".Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, May 2000.