Yesterday is history. Tomorrow is a mystery. Today is a gift.
*That's why it's called the present.**

– Eleanor Roosevelt, 1935 and Anonymous*

University of Alberta

Learning Multi-Agent Pursuit of A Moving Target

by

Jieshan Lu

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Jieshan Lu
Fall 2009
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

Vadim Bulitko, Computing Science

Russell Greiner, Computing Science

Duane Szafron, Computing Science

Michael Carbonaro, Educational Psychology, University of Alberta

To my parents and husband,
I would not be able to explore my capacity and potentials without their unconditional love and support.

Abstract

In this thesis we consider the task of catching a moving target with multiple pursuers, also known as the "Pursuit Game", in which coordination among the pursuers is critical. Our testbed is inspired by the pursuit problem in video games, which require fast planning to guarantee fluid frame rates. We apply supervised machine learning methods to automatically derive efficient multi-agent pursuit strategies on rectangular grids. Learning is achieved by computing training data off-line and exploring the game tree on small problems. We also generalize the data to previously unseen and larger problems by learning robust pursuit policies, and run empirical comparisons between several sets of state features using a simple learning architecture. The empirical results show that 1) the application of learning across different maps can help improve game-play performance, especially on non-trivial maps against intelligent targets, and 2) simple heuristic works effectively on simple maps or less intelligent targets.

Acknowledgements

I would like to thank my supervisors, Dr. Russell Greiner and Dr. Vadim Bulitko, for transforming me from a simply curious student to a mature, thoughtful Master of Science. Their advise and help have prepared me for all the ups and downs that I could encounter in life. I am also grateful for those fellow graduate students who shared their happiness and grief about how to be a graduate student. I sincerely thank their endless support and encouragement for the completion of my degree.

Most importantly, I would like to think my parents for their invaluable love and support, but they never ask for a return. Without them, I would not have enough courage to seek knowledge from half way around the world for over a decade. Without them, I would not see the the true value of family, which is priceless. Without them, I would not be able explore the full potential in myself in an independent platform of life. But with them, I survived and keep striving to succeed.

My husband, Bill, is yet another precious person for whom I am forever grateful. During these ten years of expedition outside my homeland, he is the only loyal companion who has seen my moments of tears and who has always been helping spread the joyful moments of life. For better or for worse, I am eternally grateful to him and wishing to turn every page of our life with our holding hands.

It has been a long, long haul in the pursuit of education and the finding of who Jieshan (Shanny) Lu really is. Finally, putting my name down on the thesis has become a reality. Tomorrow is another page of life. I wish all the best to who I love and who loves me, and to those whose path of life have intercepted with mine, even just once.

Table of Contents

1 Introduction **1**
1.1. Motivation . . . 2
1.1.1 . Animal Behaviors . . . 2
1.1.2 . Military Tactics . . . 3
1.1.3 . The Virtual World . . . 4
1.2. General Evaluation on Multi-Agent Moving Target Search . . . 6
1.3. Common AI Problems and Challenges . . . 7
1.4. Contribution . . . 9
1.5. Thesis Outline . . . 9

2 Problem Formulation **10**
2.1. Components . . . 10
2.2. The Real-Time Constraint . . . 12
2.3. Cooperation and Coordination . . . 12
2.4. Notation . . . 13

3 Related Work **15**
3.1. Graphical Analyses on the Pursuit Game . . . 15
3.2. Single Agent Search . . . 17
3.3. Applying Standard Heuristic Search Methods to Multi-Agent Pursuit Games 19
3.4. Cooperative Methods for Common Goals . . . 20
3.5. Reinforcement Learning Approaches . . . 22
3.6. Other methods . . . 23

4 Off-line Learning to Pursue **25**
4.1. The Training Phase . . . 26
4.1.1 . Computing Optimal Response Databases . . . 26
4.1.2 . Features . . . 28
4.1.3 . The Learning System . . . 33
4.2. Run-Time Phase . . . 36
4.2.1 . An Adversarial Approach . . . 36
4.2.2 . Multi-Agent Minimax . . . 37
4.3. Expected Performance . . . 40

5 Experiments and Evaluation **41**
5.1. Choice of Maps . . . 41
5.2. Frameworks . . . 43
5.3. Target Algorithms . . . 43
5.4. ANN-Training Experiments . . . 44
5.4.1 . Sensor Range . . . 45
5.4.2 . Training Data . . . 45
5.4.3 . Training and Preliminary Evaluation . . . 46
5.5. Run-time Experiments . . . 51

6 Limitations and Potential Improvements 58
6.1. Limitations . . . 58
6.2. Potential Improvements . . . 59

7 Conclusion 61

Appendices 63

A Minimax 63

B Empirical Results on Non-Optimal Targets 65
B.1. Training Results . . . 65
B.2. Run-Time Experimental Results on Full-Plies . . . 65
B.3. Run-Time Experimental Results Showing Fractional Plies . . . 65

Bibliography 73

List of Tables

5.1 The size of problems that are used in our study. 43
5.2 The set of features to be extracted on any pursuit instance. 44
5.3 Naming conventions on ANN and the maps that are used for training and testing (Hm = Hand-made 5×5, Em = Empty 5×5, R7 = Round-table 7×7, Rt = Round-table 9×9, Sp = Sparse 10×10). 46
5.4 RMSE comparison: Beam (Bm) versus Hill-climb (Hc), $R = 3$. 50

List of Figures

1.1 Examples of predatory animals engaged in cooperative pursuit behavior. . . 3
1.2 Examples of military tactics: in (a) and (b) the lighter clouds represent the divided offensive force and the arrows indicate their movement. 4
1.3 Examples of real-time strategy games (sources: (a) from [29], (b) to (d) from [17]). 5
1.4 An example of the round-table scenario. 8

2.1 Examples of grids and the corresponding graphs. 10

3.1 An illustration of a heuristic depression: heuristic values located at the top right corner of a cell are the Manhattan distances to the goal state T. The hill-climbing agent P tries to path-find from state s_6 to s_8. However, it is unable to reach s_8 due to the heuristic depressions at state s_6 and s_3. 18
3.2 The LRTA* algorithm. 18
3.3 The MTS algorithm. 19

4.1 The system architecture used in this study. 26
4.2 The OptimalResponse Algorithm. 27
4.3 Several examples of identical agent-topologies. 29
4.4 Examples of different delta scenarios on a 9×9 grid. 30
4.5 Mobility Computation: (a) and (b) compute for the target; (c) to (e) compute for pursuers. 31
4.6 The HillClimbSensor Algorithm. 33
4.7 ANN achitectures: (a) a general ANN feed-forward architecture with a single hidden layer for the pursuit game and (b) our ANN architecture for the pursuit game. 35
4.8 The MA-Minimax-Driver algorithm. 37
4.9 The MA-Minimax algorithm and the f function with an ANN evaluator. . . 39
4.10 Targeted performance of MA-Minimax with our ANN evaluators. 40

5.1 Illustrations of the toy maps used in study. 42
5.2 Feature weights with the beam sensor: in each subfigure, the chart on the left shows the weight-strength between each feature and a hidden neuron, whereas the much narrower chart on the right shows the weight-strength between the hidden neuron and the output neuron. 47
5.3 Feature weights with the hill-climb sensor. 48
5.4 RMSE from the tests against the ANN generated with the hill-climb sensor. 51
5.5 Run-time performance against the OPT target. 53
5.6 Dead-lock scenarios occurred during run-time: the dotted lines mean that there could be more grid cells in the direction of these lines (i.e., the presented terrain can be a portion of a large map). 54

A.1 The Minimax Algorithm. 63
A.2 Minimax with alpha-beta pruning: β represents target (or opponent in general) best choice of action. The subtree is discarded if α is worse than β. . . 63

A.3 An example of a minimax game tree for maximum depth $d = 3$ in the context of the pursuit game. A round state on the tree represent a joint state. The pursuers make a joint decision on a Max-ply, and predict the target's move on a Min-ply. By convention, the combination of a Max-ply and a subsequent Min-ply is considered as a full ply. 64

B.1 RMSE vs. the hill-climb sensor on SFL target. 67
B.2 RMSE vs. the hill-climb sensor on DAM target. 67
B.3 Run-time performance against the DAM target. P-values under each subfigure represent the statistic significance on paired comparison over the success rates at $d = 3$. 68
B.4 Run-time performance against the SFL target. P-values under each subfigure represent the statistic significance on paired comparison over the success rates at $d = 3$, except for Round9. 69
B.5 Run-time experiment against the OPT target with multiple hill-climb sensor ranges. 70
B.6 Run-time experiment against the DAM target with multiple hill-climb sensor ranges. 71
B.7 Run-time experiment against the SFL target with multiple hill-climb sensor ranges. 72

Chapter 1

Introduction

This dissertation addresses the challenge of building multiple agents that can capture a moving target in the context of video games. Our approach applies machine learning to produce effective heuristic search methods. Most search methods are guided by an estimated utility value with respect to the goal, called a *heuristic*. Motivated by real-time heuristic search and its applications, we are interested in using experience from one or more maps to learn strategies that apply to a novel map. Therefore, we study the feasibility of off-line learning to produce portable heuristics for solving the pursuit game.

The underline heuristic search method of the pursuit game is real-time Single Agent Search (SAS). Typical SAS approaches have a two-phase procedure: planning and execution. In the planning phase, the problem solver computes a solution for a single agent between a start state and a fixed goal state, while following some predefined optimization criterion and a heuristic function (or in short *heuristic*) that guides the search. Once a full solution is available, the agent executes the full solution by sequentially moving along the solution path. However, this two-phased approach cannot produce satisfying real-time performance in general because the time to compute a full solution could cause a significant delay in move execution; it also does not apply well to multi-agent pursuit games as the target is moving.

Many real-time applications restrict planning with a small constant time. This constraint forces the agent to interleave planning and execution until it reaches the goal, and decouples the planning complexity from the problem size [5, 7]. Hence, the agent appears to be more reactive regardless of the scale of the problem.

An example of a real-time heuristics search algorithm is Ishida and Korf's Moving Target Search (MTS) [22]. There are two types of agent in MTS: the pursuer and the target. Similar to SAS, the pursuer is an active agent who executes a partial solution that is based

on map-dependent distance-heuristics. The target, however, is mobile in MTS as opposed to the goal being static in SAS; and hence, complete search to one target location does not apply. The search terminates when the target is captured (i.e., the positions of the target and the pursuer coincide) or it escapes within a pre-specified time period. MTS assumes that 1) all agent are aware of each others' locations, 2) the target is slower than the pursuer, and 3) the two agents move in turns. MTS incrementally improves performance over multiple trials by learning the heuristics based on the target's locations. However, such learning is slow; and it is no longer sufficient in a pursuit game with multiple pursuers.

This dissertation explores a way of learning to produce an effective real-time search method for solving the multi-agent pursuit game. This is a challenging search problem because an effective pursuit requires cooperation among the pursuers. We first address the shortfalls of applying several existing SAS (learning) and multi-agent approaches to this problem. Typical learning search approach learns map-dependent heuristics instead of pursuit behavior. By contrast, our approach attempts to learn the pursuit behavior off-line by constructing a target-specific model based on a set of significant features of any pursuit game state. The resulting model is applied to a search method as an evaluation function to produce effective pursuit behavior against the target that was learned.

In the remaining sections, we first present several real-life motivations to our off-line learning approach. Then, a general evaluation of multi-agent moving target search is introduced, followed by a discussion of common problems and challenges in Artificial Intelligence (AI). This chapter closes with our contribution and the outline of this dissertation.

1.1. Motivation

The game of pursuit arises from various domains, including animal behaviors, military tactics and commercial video games. In these three exemplary domains, an effective pursuit requires cooperation among the pursuers, especially when the target has comparable or better mobility than the pursuers. However, there are differences in the quality of performance in nature, in man-power, and in the simulated world.

1.1.1. Animal Behaviors

Animals like wolves, African wild dogs, lionesses and tunas are naturally proficient in cooperative hunting behaviors driven by food [31, 36]. Most predatory animals have the ability to hunt for food independently. However, the success rate is low because the prey can sometimes outrun and/or outsmart a single predator. It has been observed that hunting

(a) Wolves (source: [48])

(b) African wild dogs (source: [39])

Figure 1.1: Examples of predatory animals engaged in cooperative pursuit behavior.

cooperatively with a group increases the success rate of capturing larger and stronger prey. Adult animals in a group teach their young to hunt their prey with the cooperative techniques of trap and chase, as shown in Figure 1.1. The youngling can learn from past experiences and practice.

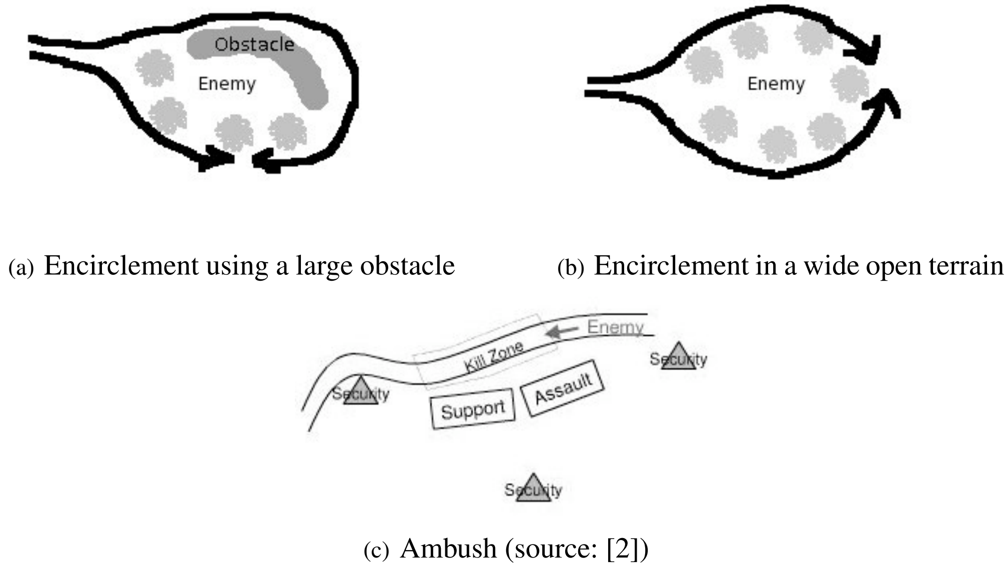
For example, adult wolves in a pack teach their young the art of hunting, where the young learn from the adults' experience and fine-tune their actions through practices [32]. In real hunting scenarios, pack members diversify their positions around the prey before a full-speed pursuit. During the pursuit, a wolf can cut corners to intercept a prey that had evaded another member in the pack.

African wild dogs are another example of predators that exhibit learning and cooperative hunting behaviors. Pack members use a divide-and-conquer approach, which is passed from generation to generation, to first isolate a vulnerable prey from the target group, and then pursue that single prey in joint efforts that is likely to lead to a kill [9]. In particular, some members can chase their prey to what they believe is the barriers of the current hunting ground, while other members disperse to intercept the prey.

In short, there is a close correlation between learning and cooperative pursuit in a wide range of predatory animals. However, hunting is a non-trivial process because the animals need to learn and hone their skill. Also, they need to identify similarities between the training environment and the current environment in order to apply appropriate skills for cooperative pursuit and capture.

1.1.2. Military Tactics

There have been numerous military support and profound motivation to computer research since World War II; approximately 60 to 80 percent of research funds were provided by



(a) Encirclement using a large obstacle

(b) Encirclement in a wide open terrain

(c) Ambush (source: [2])

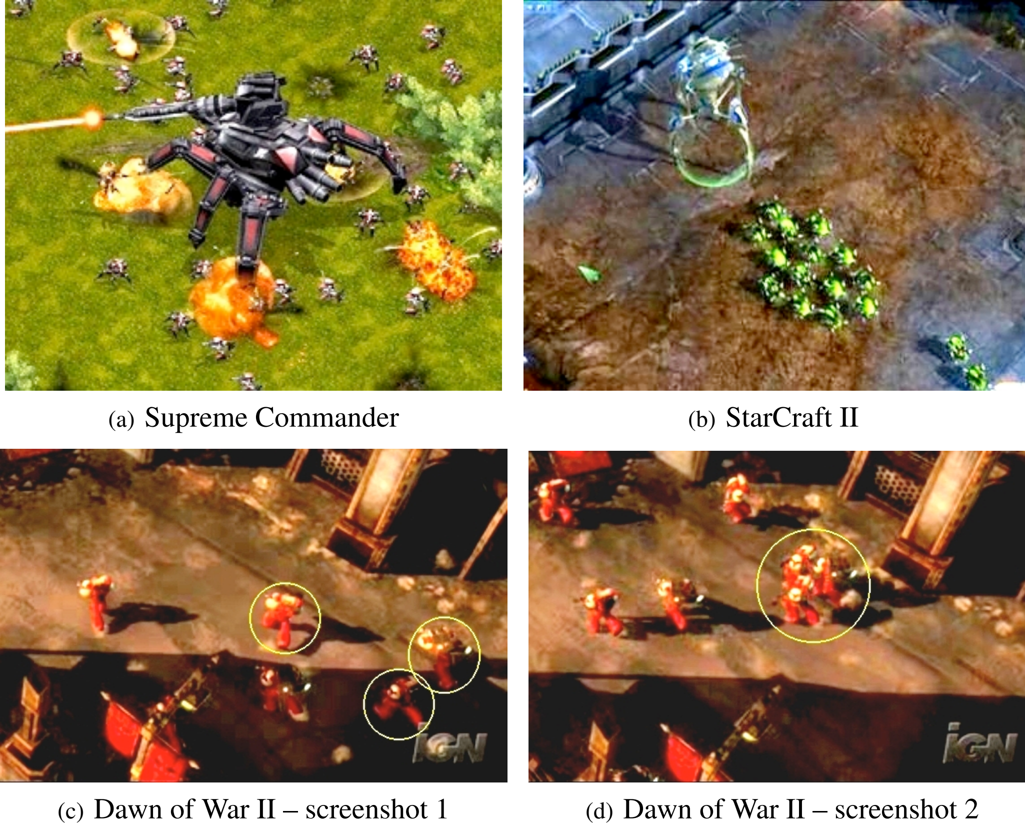
Figure 1.2: Examples of military tactics: in (a) and (b) the lighter clouds represent the divided offensive force and the arrows indicate their movement.

military-related agencies since 1960's [10]. Military tasks often require infantry training and may use supporting systems with AI that simulates the tactics. Military tactics that are relevant to the pursuit game inspire us with the technique of dividing up forces to surround the target with or without natural barriers in the environment.

Infantry use several tactics such as encirclement and ambush in pursuit of enemy forces. Encirclement is an aggressive form of cooperative attack [13]. After a sequence of pursuit, the offensive force performs a two-armed encirclement on an isolated enemy unit with (see Figure 1.2(a)) or without (see Figure 1.2(b)) the use of a large obstacle. The sole purpose of this act is to minimize the enemy's options for escape. Ambush is more tactical than encirclement, in terms of planning ahead before the arrival of the enemy. As shown in Figure 1.2(c), it requires 1) identifying a kill zone, 2) splitting up the attack squad around the kill zone and awaiting enemy's (target) arrival, and 3) ambushing the enemy [2]. The success of both tactics depends on theoretical understanding, field practice and terrain analysis.

1.1.3. The Virtual World

The need of the aforementioned military tactics often arises in computer video games when two groups of units are engaged due to a conflict of interest. These games are usually supported by AI for the ability to manage many units simultaneously. Game designers are confined to a conservative time limit (usually less than 100 milliseconds) of planning and



(a) Supreme Commander

(b) StarCraft II

(c) Dawn of War II – screenshot 1

(d) Dawn of War II – screenshot 2

Figure 1.3: Examples of real-time strategy games (sources: (a) from [29], (b) to (d) from [17]).

execution for all units in order to ensure a smooth frame-rate [7]. Learning is usually disallowed to keep game experience predictable and to simplify the AI code. If it appears at all, cooperation emerges immediately from the quality of path-finding, which is a fundamental operation of moving target search.

The AI in real-time strategy (RTS) games like *Supreme Commander* [12], *Dawn of War II* [40] and *StarCraft II* [3] has the ability to manipulate many units at once. However, there are often path-finding issues when the AI tries to relocate multiple units simultaneously. In Figure 1.3(a), the *Supreme Commander* system allows friendly units to overlap their positions during a battle with enemy units. They ignore each other in path-finding and do not maximize their control area, which will be formally defined in Section 3.4. A similar behavior also occur in *Dawn of War II*; the three units highlighted in Figure 1.3(c) bump into each other during a relocation, as shown in Figure 1.3(d). Path-finding issues as such reduce the range of attack and could leave enemy units a wide open space to escape.

Figure 1.3(b) illustrates another issue of coordinating path-finding in *StarCraft II* [3]. A platoon of Banelings engages to pursue a Colossus, which is a much larger, stronger unit.

The Banelings did not flank the Colossus with multiple units before it is too late. As a result, the Colossus flees, climbing onto a platform that is both inaccessible to the Banelings and a tactical advantage for the Colossus, which uses ranged attacks.

A system that does not efficiently coordinate its agents in path-finding could fail to achieve its goal and may suffer from severe consequences. Lack of coordination makes the units less cooperative; units that are unable to disperse make themselves an easy target of splash damage (i.e., indirect damage that is caused by an explosive impact at a location nearby). Proper coordination is prominent in a sense that it not only increases human player's satisfaction, but also improves the reputation of the game in terms of intelligence. Recent research on cooperative path-finding focuses on hand-coded strategies [23]. In this work, we attempt to derive cooperative behavior automatically, via machine learning.

1.2. General Evaluation on Multi-Agent Moving Target Search

This thesis studies ways to help multiple pursuers capture a single moving target in the settings with the following characteristics. All agents have the same speed. The pursuers and the target can move within a finite, fixed, fully observable grid map in a turn-based manner; all pursuers move in the current turn, then the target moves in the next turn, then all pursuers and so forth. The turn-based setting follows the classical moving target search approach, which ensures a pursuer's reactivity in real-time and the completeness of the search. This setting can also help compute the optimal pursuit policy (formally defined in Section 2.4) in response to the target policy. The target tries to evade as long as possible. Every pursuit game is restricted to a predefined time period, called *time-out*. When any one pursuer intercepts the target by moving onto the target's location before the time-out, it is called *a successful capture*. Otherwise, the game times out, and the target "wins".

We evaluate the run-time performance through a batch study on different maps with one hundred instances randomly generated on each map. Each instance consists of the starting locations of all agents, the pursuit policy, and the target's evasion policy. The following performance measures are chosen to evaluate the quality of pursuit in real-time simulations:

- the success rate of capture,
- the capture-travel and suboptimality, and
- planning effort per action.

The *success rate of capture* is the percentage of instances that end with a successful

capture over all simulated instances; it indicates the ability of the pursuers to solve the pursuit game. The *capture-travel* represents the average travelled time-steps that the pursuers take to capture the target over the same set of simulated instances. The *optimal response* to a specific evasion policy is computed and used as a ruler to measure *suboptimality* on corresponding instances. The *suboptimality* is precisely the ratio of run-time *capture-travel* over optimal (i.e., shortest possible) *capture-travel* on the same instance, assuming finite. Finally, *planning effort per action* presents an evaluation on step-wise computational cost. It provides a guideline to evaluate how long the problem solver "thinks" before taking an action in terms of the suitability to real-time applications. Our goal is to maximize the success rate and to minimize the average capture-travel and the planning effort.

1.3. Common AI Problems and Challenges

There exist many challenges for solving the pursuit game. These include theoretical challenges regarding the solvability of MTS problems, performance challenges wherein real-time response and solution quality are traded, the problems of applying SAS approaches in multi-agent settings, and the challenge of imperfect information about the problem domain.

Based on uniform speed and turn-base games, the number of pursuers necessary and sufficient to capture a moving target on grids is still unclear. It has been proven that *three* pursuers are always sufficient to capture a moving target in any planar graph [1]. As two-dimensional grids are a special class of planar graph, a recent study in heuristic search proves that two pursuers are in fact necessary and sufficient on empty finite grids (i.e., without obstacles) [18]. Their empirical results show that two pursuers might also be necessary and sufficient on a subset of grids that do not resemble a dodecahedron. There is, however, no associated theoretical proof.

It is difficult to balance real-time performance with success rate and capture-travel. Assuming that target is slower than the pursuer, classical single-agent MTS algorithms [20, 21, 27] guarantee real-time performance and learn incrementally to reduce capture-travel. They are, however, often misled by imperfect information about the map and the target's movement, resulting in infinite capture-travel (i.e., unsuccessful capture). One SAS approach [26] compromises real-time performance for a faster reduction rate of capture-travel and a higher success rate, but it become less applicable in real-time applications.

Typical SAS approaches assume that a slower target will eventually be *outrun* rather than being *outsmarted* [20, 21, 26, 27]. When these approaches are applied to agents of

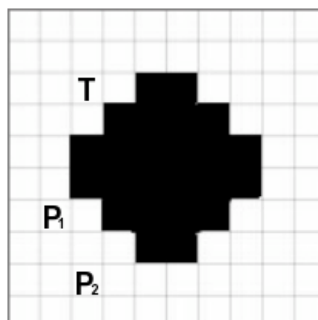


Figure 1.4: An example of the round-table scenario.

uniform speed, the target might only maintain a constant distance to the pursuer. Thus, the success rate decreases and capture-travel increases, regardless of the real-time performance.

Furthermore, it is non-trivial to solve a pursuit game with multiple pursuers, assuming that the target is at least as fast as the pursuers. The core problem lies in the direct application of SAS approaches to the multi-agent pursuit game. Pursuers often follow the same path, resulting in clumping together and/or blocking each other's path [37]. The performance degrades in terms of increasing capture-travel and decreasing success rate. Consider the round-table scenario on a grid (Figure 1.4) when the two pursuers (p_1 and p_2) try to capture a moving target (T). It is intuitive to secure a capture by having one pursuer travel clockwise and the other counter-clockwise. Existing SAS methods instead instruct both pursuers to travel clockwise or counter-clockwise together, resulting in an infinite pursuit around the table. This shows that merely applying SAS methods to the multi-agent pursuit game is insufficient to coordinate the pursuers' paths.

One intuitive way to impose true cooperation is to plan a fully joint path for all pursuers, so that they can maximize the control area. However, planning in joint fashion for multiple agents is subject to time complexity that is exponential in the number of agents. Such complexity in planning effort per action makes it less desirable in real-time applications.

It is also challenging to incorporate learning in real-time pursuit games without full information about the map. Success rate and capture-travel will not improve without learning the map. A possible solution is to apply learned cooperative behaviors to new maps. This way can promise a small planning effort per action and may improve capture-travel and success rate. However, this approach requires a set of matching features across different maps. Selection of such features is an open problem.

1.4. Contribution

The thesis established in this study is that off-line supervised learning over a set of features about any pursuit instance can improve the time to capture a target. As this work tries to tackle the last, most difficult challenge in Section 1.3, the contributions include: 1) the first application of off-line machine learning methods to multi-agent moving target pursuit; 2) in evaluating the approach, we compare several sets of features and several machine learning mechanisms; and 3) we compare the run-time performance against the precomputed optimal response policy with respect to a target policy.

Our approach decouples learning from run-time and makes the search in multi-agent pursuit more efficient in real-time. The results show that off-line learning can help capture a target that uses a specific evading strategy in non-trivial environments at run-time. It can also help planning in joint space while retaining the real-time property because the learning process is separated from run-time. Moreover, our results show that simple heuristic works effectively on trivial maps, where true distance-information is known. In fact, learning is never necessary depending on what is known. Scalability, however, appears to be dependent on the choice of training maps and the definition of features, which require a deeper investigation beyond the scope of this thesis.

1.5. Thesis Outline

Chapter 2 introduces a formal lexicon for the MTS problem, which includes performance measures and other technical terms used in a multi-agent pursuit game. The chapter will furthermore motivate the need for cooperation in multi-agent settings. In Chapter 3, we review previous research related to the task of MTS. They range from single-agent methods to multi-agent methods, as well as approaches that emphasize the effect of learning. Chapter 4 introduces the core of our approach, which applies learning to real-time pursuit applications. Experiment details and performance evaluation over a classical heuristic and our learned behavior models will be provided in Chapter 5. The limitations and potential improvements in Chapter 6. Chapter 7 summarizes the work and presents our conclusions.

Chapter 2

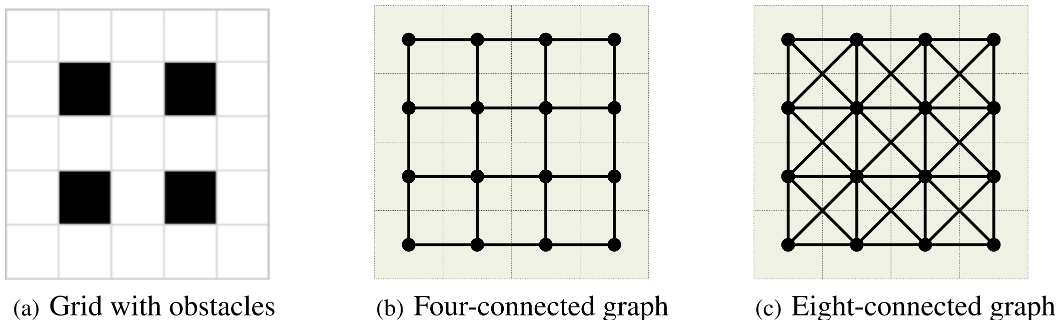
Problem Formulation

We focus on the task of $n \in \mathbb{N}$ pursuers attempting to capture a single target in a fully observable two-dimensional grid. Full observability is a simplifying assumption that is most frequently employed in video game AI. Future work will generalize our result onto partially observable cases.

2.1. Components

Grid. A game map is represented as a fixed, finite and fully observable two-dimensional *grid*. An empty grid consists of a matrix of open, traversable square cells of uniform size. Static geographical barriers are overlaid onto the grid to mark blocked cells. For example, Figure 2.1(a) shows a 5×5 grid with four pillars. Each pillar occupies one cell. Larger obstacles, such as walls, can occupy more than one cell. Any cell occupied by an obstacle is not traversable.

In path-finding, a grid is translated into an undirected graphs $G = \langle S, A \rangle$, where S is a finite set of traversable states (or *vertices*) and $A : S \rightarrow S$ is a finite set of actions (or *edges*) between them. A state $s \in S$ encodes its cell-location by x-y coordinates. An action $a \in A$



(a) Grid with obstacles (b) Four-connected graph (c) Eight-connected graph

Figure 2.1: Examples of grids and the corresponding graphs.

is an edge that connects a pair of adjacent states such that $s' = a(s)$, where s' is called a *successor* of s with respect to a. For undirected graphs, every action has a reverse action such that $s = a'(s')$. The weight of an action is defined by the cost function $c : A \rightarrow (0, \infty)$ with $c(s, s')$ being the travel cost in distance for taking action a. The set of actions available at any state $s \in S$ is denoted by $A_s = \{a \ : \ a(s) \neq \emptyset\} \subseteq A$. The size of the actions set at any state is called *branching factor*, or $|A_s|$.

Typically, grids are represented in either four or eight connected graphs. In four-connected graphs (e.g., Figure 2.1(b)), the maximum branching factor of a state is four with cardinal actions ($\rightarrow$, $\downarrow$, $\leftarrow$, and $\uparrow$). Such graphs are *planar graphs* since they do not contain intersecting edges other than on the endpoints of the actions. Eight-connected graphs (e.g., Figure 2.1(c)) are not planar graphs because a state could have diagonal edges that intersect each other. Any cardinal action has a unit distance cost of 1, or $c(s, s') = 1$. One special type of actions is called *stay*, such that $c(s, s) = 0$ at state s. However, taking any of the cardinal action or the *stay* action still accounts for *one* simulation time-step. Furthermore, the maximum branching factor of a state in a four-connected grid becomes five, and nine for eight-connected grids.

Agent. An *agent* is a moveable unit that can move from one state to another with a pre-defined constant speed and has an associated path-finding algorithm (or *policy*). The *speed* of an agent indicates the distance that the agent can traverse at each simulation time-step in terms of cumulative travel cost. In a four-connected graph with a uniform action cost of 1 unit distance, for example, a speed of 1.0 means that the agent can traverse to an adjacent state in exactly one simulation time-step. In single-agent moving target search, there are two types of agents using different policies: pursuer and target. A good pursuer policy can help a pursuer capture the target as quickly as possible. A good target policy allows the target to remain free as long as possible.

Movements. The pursuit is launched by deploying all agents to their initial states, and continues in a turn-based manner until the simulation is terminated by the *capture* (defined below) or the *time-out*. In particular to moving target search, a pursuer is allowed to take one action from its *current state* only after the target has committed an action, and so on. Any agent may plan for an immediate, reactive action just before its execution, or a sequence of cached actions by using an internal off-line search only to the last known target state before this off-line search starts. However, an agent can only execute one action at its own turn.

With the presence of multiple pursuers, all pursuers can take one action at a pursuer-turn, and they all stay at their current location at the target's turn.

Capture. The target is considered to be *captured* successfully when a pursuer makes its current state coincide with the target's current state before the simulation reaches the pre-specified time-out. Otherwise, the simulation is labeled "failure to capture" before time-out; and the target wins.

2.2. The Real-Time Constraint

The defining constraint of real-time performance is that the amount of planning effort per action is restricted to be a small constant amount of time regardless the size of the map [7]. This constraint allows a fast response from a search agent to adapt to a changing goal. One way to achieve real-time performance is that the agent plans its immediate action(s) by localizing the search space to the neighbourhood of current state s, denoted by $\mathbb{N}(s)$.

This real-time constraint requires fast "thinking" for two reasons, namely fluid frame-rates in video games and faster capture if all agents move asynchronously. The former is addressed in this dissertation to make "thinking" independent to the size of the game map. The latter is not included in our formulation, as it will be a subject of future research.

2.3. Cooperation and Coordination

The terms *cooperation* and *coordination* often appear in multi-agent scenarios; and to many, they seem to be synonymous. In this thesis, we draw the following distinction between the two. *Cooperation* specifies "how to pursue" in a pursuit game. It is a requirement that demands collaborative path-finding from the pursuers. *Coordination* specifies "how to cooperate" among the pursuers. It is an implementation of collaborative path-finding to achieve cooperation across the pursuers. The planning of each pursuer's action is required to take other pursuers' actions into account to capture a common target. Such planning approach is called, the *joint-action* plan.

To formally define the joint-action plan, let the set of the current states of n pursuers and a target be $\{s_1, \ldots, s_n, s_{target}\}$ for s_i, $s_{target} \in S$, $i = 1, \ldots, n$. A joint-action plan transform the set $\{s_1, \ldots, s_n, s_{target}\}$ to $\{s'_1, \ldots, s'_n, s'_{target}\}$. Such set of individual states is called a *joint state*, denoted by the shorthand notation, $j \in J$, where J is the set of all possible joint states given a map and the number of pursuers and the target, and is of the

size $|S|^{n+1}$. A joint action on j is denoted by $\kappa \in \mathbb{A}$ (read *joint*-A), which is the set of all joint actions of the size $|A|^{n+1}$, and the successor of a joint state j is $j' = \kappa(j)$. Similar to the definition of the action set at any singular state, the set of joint actions at any joint state j is denoted by $\mathbb{A}_j = \{\kappa : \kappa(j) \neq \emptyset\} \subseteq \mathbb{A}$.

2.4. Notation

Definition 2.5. Travel Distance: The travel cost of the shortest path between any two states s_1 and $s_2 \in S$, is called the *travel distance* and is denoted by $dist(s_1, s_2)$.

Definition 2.6. Heuristic Function: A heuristic function, $h(s_1, s_2)$, is an estimated travel distance between the two states. An admissible heuristic always underestimates the travel distance, such that $h(s_1, s_2) \leq dist(s_1, s_2)$. *Manhattan Distance* is an admissible heuristic that is often used on grids; and it ignores obstacles. It is the sum of differences of s_1 and s_2's corresponding coordinate components. When $h(s_1, s_2) = dist(s_1, s_2)$ for all state pairs, h is called a *perfect heuristic* and denoted by h^*.

Definition 2.7. Heuristic Depression: A heuristic depression is a local minimum of heuristic values, which occurs at a state s when $h(s, s_{goal}) \leq h(s', s_{goal})$, for all possible successors s' of s [27].

Definition 2.8. Capture-Travel (T) is the average number of simulation time-steps that a pursuer takes to capture the target.

Definition 2.9. Success Rate is the percentage of test instances that terminate with a successful capture over the total number of test instances in an experimental batch.

Definition 2.10. Optimal Response Database (ORD): Given a target policy, an *optimal response database* stores the lowest value of capture-travel of every possible joint state configuration on a map.

Definition 2.11. Suboptimality Given a target policy, *suboptimality* is the ratio of capture-travel under policy π, T^π, over the optimal capture-travel T^*, which is averaged over all test instances.

Definition 2.12. Sufficiency: The number of pursuers n is *sufficient* for a class of graphs $\mathbb{G}$ if and only if for any graph $G \in \mathbb{G}$ and any target policy π_t there exists a joint pursuit policy π for n pursuers, such that for any initial joint state j_o, π is guaranteed to capture the target in a finite number of simulation time-steps.

Definition 2.13. Necessity: The number of pursuers n is said to be *necessary* for a class of graphs $\mathbb{G}$ if and only if there exists a graph $G \in \mathbb{G}$, a target policy π and an initial joint state j_o such that no joint pursuit policy for fewer than n pursuers will capture the target in a finite number of simulation time-steps.

Definition 2.14. On-line Learning updates the estimate of a state value (e.g., heurisitics) immediately after selecting an action at that state. If the state is revisited before the end of the current trial, the new value will be considered instead.

Definition 2.15. On-policy Learning evaluates and updates the value of a policy for action selection. Learning is used to improve the current policy [51].

Definition 2.16. Portability describes the applicability of a single evaluation function that is trained on one map, onto another map. The training of such function follows the standard form of supervised learning; the function learns from a set of examples with its input and output. In our setting, the input is a set of features that describes a single pursuit instance, whereas the output is the corresponding optimal capture-travel.

Chapter 3

Related Work

Researchers from the mathematical and control communities have studied the "pursuit game" for a several decades. However, there are few research results in the context of multi-agent moving target search. This chapter first introduces several theoretical studies on the pursuit game, and then reviews the fundamental algorithms on single-agent heuristic search and single-agent moving target search. We also show how standard heuristic methods are applied to solve multi-agent pursuit games, followed by several solutions from cooperative methods. In addition, we show how other control methods are applied to address the pursuit game. The approaches we describe in this chapter have in common a focus on using learning to improve performance, which serves as a basis for our approach.

3.1. Graphical Analyses on the Pursuit Game

Graphical analysis of the pursuit game focuses on identifying theoretical solutions, limitations, and bounds under rigorous mathematical constraints. We first review a few studies on the necessity and sufficiency of the number of pursuers in arbitrary graphs, followed by a related study specifically on grid maps. We then show that the pursuit game is in general a challenging problem through the study on the search-time of pursuit games.

Goldstein et al. address the challenge of determining the number of pursuers that are necessary and sufficient to capture a target in arbitrary graphs, showing that this challenge is EXPTIME-complete [15]. Aigner and Fromme study the game on finite connected planar graphs with n pursuers and 1 target; all agents have uniform speed and play optimally [1]. They identify the pursuers as *cops* and the target as the *robber*, and accordingly partition planar graphs into *cop-win* graphs and *robber-win* graphs. By introducing the concept of a *pitfall* p, which is defined as that p and all p's successors overlap with the successors of a state $d \neq p$, a *cop-win* graph is one that can be reduced to a single vertex (or state) by

repeatedly removing pitfalls in any order. The authors prove that three pursuers are always sufficient to ensure a capture in any planar graphs by taking control of pitfalls. Two pursuers can control a region by controlling two distinct paths between two states, such that they can move to any state on these two paths faster than the target. The third pursuer expands this region by taking control of a pitfall in the target's region, freeing one of the two pursuers to subsequently expand their control region until the target cannot escape.

However, Aigner and Fromme's proof raises reasonable doubts about whether three pursuers are always necessary. Greiner and Isaza prove that two pursuers of uniform speed are in fact necessary and sufficient to capture a target with the same speed in any empty (i.e., no obstacles) finite four-connected grid, which is a robber-win planar graphs [18]. Assuming the grid is of size $R \times C$, this proof assigns one pursuer to match the state of the target column-wise, and the other row-wise. They show that the optimal solution takes a maximum of $max\{R, C\}$ steps for both pursuers to align with the target column-wise and row-wise, followed by another maximum of $R + C$ steps to corner the target for a capture.

For non-empty four-connected grids, Isaza's empirical results from examining the *OptimalStrategy* algorithm show that two pursuers are necessary and sufficient only on a subclass of such grids, but not all. However, there is no formal proof provided on the type of grids that always requires three pursuers. A grid maze that resembles a dodecahedral planar graph is an example in which three pursuers are necessary. In this grid, there are 20 major states with branching factor of 3 (i.e., T-intersections) and 30 major "edges" (i.e., hallways that are one-state wide). The length of cycle on each major state is no less than five times the length of each "edge". Therefore, the target can easily flee from one branch of an intersection while the two pursuers try to capture the target from the other two branches. In our study, however, this type of non-empty grids will also be a subject of future research.

Bonato et al. show that the search-time (i.e., capture-travel) of pursuit games depends on the number of pursuers when both the pursuers and the target play optimally [4]. Although the search-time for n pursuers is invariant of the structure of the graph, it is polynomial to the number of states. The upper bound of the search-time is half the number of states for a wide-range of graphs. Moreover, the decision of determining whether n pursuers can capture the evading target in a given time-out is NP-complete[1]. As a result, it is challenging to solve the pursuit game in general.

[1]NP stands for nondeterministic polynomial time. A decision problem is called NP if some algorithm can guess and then verify a solution for correctness in time polynomial in the input size [45]. NP-complete problems are the hardest problems in NP as no fast solution to this class of problems is known.

3.2. Single Agent Search

Single agent search (SAS) is a fundamental class of search methods for solving a problem with only one agent and a stationary goal. An SAS algorithm is often guided by a heuristic function to find a shortest solution path to the goal. There are two phases in any SAS algorithms, namely planning and execution. Given a graph $G = (S, A)$ as defined in Section 2.1, the heuristic value for state $s \in S$, denoted by $h(s)$, is the estimated travel cost to the fixed goal state s_g; and hence, the second argument of the h-function (see Definition 2.6) is ignored. During the planning phase, the problem solver computes a partial or full path depending on the imposition of the real-time property (Section 2.2). The problem solver subsequently moves the agent along the computed path toward the goal.

Well known algorithms for pathfinding problems are A*, and various versions of real-time heuristic approaches with or without learning. They are the underlying algorithms for moving target search algorithms, which differ from typical SAS algorithms by attempting to path-find to a changing goal. In this section, we will review these SAS algorithms that influence the algorithms for moving target search.

A widely applied search algorithm is A* [16], which computes an optimal solution from the start-state s_o to the single goal-state s_g. The search prioritizes state expansion by the evaluation function $f(s) = g(s) + h(s)$, where $g(s) = dist(s_o, s)$ and $h(s)$ is the estimated distance from s to s_g. A* allows a state to be re-opened when there exists a lower f-value through an alternative path; and its f-value is updated accordingly. If h is admissible (see Definition 2.6), then A* is guaranteed to return an optimal solution path. However, a naive implementation of A* is not suitable to real-time applications because 1) the run-time of A* depends on the size of the game map, which often causes serious delays in taking the initial action, and 2) the memory requirement is large, as A* maintains a list of expanded states and a list of opened states.

Real-time search, which plans in a local area (i.e., a neighbourhood of the current state) at a time, is one resolution that satisfies the real-time constraint and reduces memory requirements. It computes a partial solution to the goal and allows an agent to interleave planning and execution repeatedly until it reaches the goal. For example, the hill-climbing algorithm considers only the immediate successors of the current state without maintaining a list of expanded states or open states [44]. The agent selects and executes the action that leads to an adjacent state with the lowest cost, and repeats this process until it reaches the goal. The simplicity of this approach trades off to the ability of overcoming heuris-

Figure 3.1: An illustration of a heuristic depression: heuristic values located at the top right corner of a cell are the Manhattan distances to the goal state T. The hill-climbing agent P tries to path-find from state s_6 to s_8. However, it is unable to reach s_8 due to the heuristic depressions at state s_6 and s_3.

tic depression (see Definition 2.7) that often appears in non-trivial problems. As shown in Figure 3.1, an agent can be easily trapped in a heuristic depression as it "believes" in the inaccurate heuristics.

Korf introduces the trial-based algorithm, Learning Real-Time A* (LRTA*), to address the issue of heuristic depression [27]. LRTA* gradually updates a table of heuristic values with respect to the goal state during the course of search. The initial values need not be accurate as long as they are admissible and can be computed by functions as simple as Manhattan distance. The algorithm is given in Figure 3.2. For each state s being expanded, LRTA* considers the successors that are one action away (line 1) and selects the one with the lowest f-value (line 2). A heuristic update occurs when the f-value of the selected successor is greater then the h-value of the current state, followed by the execution of the corresponding action (line 3 and 4). The heuristics learned by LRTA* is proven to converge to the perfect heuristic (see Definition 2.6) of the states along every optimal path if the values are initially admissible.

LRTA* (s, s_g): $\{$** *uses and updates* $h(s, s_g)$ **$\}$

1: generate the list of successors of s
2: determine the best successor s' with the lowest $f(s') = dist(s, s') + h(s', s_g)$
3: if $f(s') > h(s, s_g)$, then $h(s, s_g) \leftarrow f(s')$
4: **return** the action a such that $s' = argmin_{a(s)}\{f(a(s))\}$

Figure 3.2: The LRTA* algorithm.

MTS (s_p, s_g, s'_g, $isPursuer$): {** *uses and updates* $h(s, s_g)$ **}

1: **if** $isPursuer$ **then**
2: generate the list of successors of s
3: determine the best successor s' with the lowest $f(s'_p, s_g) = c(s_p, s'_p) + h(s'_p, s_g)$
4: if $f(s'_p, s_g) > h(s_p, s_g)$, then $h(s_p, s_g) \leftarrow f(s'_p, s_g)$
5: **return** the action a such that $s'_p = argmin_{f(a(s_p))} a(s_p)$
6: **else**
7: $h'(s_p, s_g) \leftarrow h(s_p, s'_g) - c(s_g, s'_g)$
8: if $h'(s_p, s_g) > h(s_p, s_g$, then $h(s_p, s_g) \leftarrow h'(s_p, s_g)$
9: **end if**

Figure 3.3: The MTS algorithm.

Moving target search (MTS) extends LRTA* to adapt to a changing goal and uses one pursuer to capture the moving target [22]. In lieu of learning one-dimensional heuristics with respect to a fixed goal, MTS learns pairwise heuristics and computes the shortest distance between pairs of states on a map. Assuming in a turn-based setting that the target is slower than the pursuer, MTS updates heuristics during both the pursuer's turn and the target's turn, as in Figure 3.3. However, this approach is subject to "thrashing" (i.e., moving back and forth) in heuristic depressions and "loss of information" when the target changes its current state [33, 34]. Ishida enhances MTS with the hand-tuned commitment and deliberation to reduce the effect of these problems. In particular, the degree of commitment targets the issue of "loss of information" by forcing the pursuer to focus on learning one target state without acquiring its current state at every step. The degree of deliberation, on the other hand, mediates the "thrashing" symptom by performing an off-line search that leads the pursuer outside the heuristic depression. Malex, nonetheless, shows that improvement of performance with the two enhancements is still not guaranteed [33, 34].

Koenig et al. also tackles the same issues by introducing the algorithm called Eager Moving-Target-Adaptive A* [26]. This method, however, is more similar to A* in that the memory requirement is linear in the size of state space, and the planning time depends on the map size. Therefore, it is less suitable to real-time applications.

3.3. Applying Standard Heuristic Search Methods to Multi-Agent Pursuit Games

Many existing approaches to multi-agent pursuit game directly apply SAS approaches to individual pursuers. Orkin emphasizes that applying SAS methods to multi-agent problems

causes the clumping behavior of the pursuers, which ultimately lack coordinated behavior [37]. In most cases, pursuers are not aware of each other and make plans independently. This defeats the advantage of having *multiple* pursuers to catch one target.

Korf introduces a simple non-learning solution to grid-based and hexagonal pursuit games, called the Multi-Agent Greedy (MA-Greedy) approach [28]. Under the same assumption of a slower target, each pursuer searches for the target greedily and independently by minimizing the evaluation on pairs of states with Equation 3.1.

$$f(s_p, s_g) = h(s_p, s_g) - wR, \tag{3.1}$$

where w is a constant weight in $[0, 1.0]$, and R is the repulsion (i.e., h-value from pursuer p to another nearest pursuer p'). The repulsion force is a naive way to prevent pursuers clumping together by the key component, w. A MA-Greedy pursuer behaves like a hill-climbing agent if w approaches 0, whereas the pursuers will cease to move in the direction of the target together if w approaches 1. The fixed magnitude of w prohibits the pursuers from traveling in close proximity, allowing the target to flee between them. However, an MA-Greedy pursuer is unable to solve the game in the round-table scenario (see Figure 1.4).

Multi-Agent Real-Time A*(MA-RTA*) [25] produces effects similar to MA-Greedy. It is based on a variation of LRTA* and hill-climbing, namely Real-Time A* (RTA*). RTA* "hill-climbs" from the current state s to the immediate (best) successor with the minimum f-value, but it differs from LRTA* as RTA* updates $h(s, s_g)$ with the f-value of its second best successor. If another pursuer considers the state s at a later time-step, the new value will be used and this pursuer will chose a different path than the previous pursuer's. MA-RTA* coordinates pursuers by sharing the same heuristic table only. On grid maps, it does not perform well due to a large number of ties [57]. Similar to MTS, MA-RTA* also requires trial-based learning to improve performance; The pursuers require on-line "training" on the map before they behave reasonably.

3.4. Cooperative Methods for Common Goals

For multi-agent pursuit games, the quality of cooperation is critical to the game-play performance, albeit highly difficult to achieve. Pursuers are required to capture the target as quickly as possible through joint effort due to the shared common goal. To date, only a limited number of published approaches that cope with cooperation. They either extend SAS algorithms with external control structures or invent new heuristics to impose cooperation.

Multiple Agent Moving Target Search (MA-MTS) is a distributed approach that equips a SAS method or an adversarial search algorithm[2] (e.g., Minimax) with the external structure called the *belief set*, which consists of information about other pursuers and the target [14]. If the target moves out of sight of a pursuer, then this pursuer keeps track of a set of possible states that the target could reach before it sees the target again. The belief set can help maximize the pursuers' control area to minimize the target's mobility. Each pursuer uses its own belief set as inference to select a distinct goal state.

Also, the size of the belief set must be reduced to a constant (e.g., records the location of the four corners of a region only) to satisfy the real-time property. When the target is not visible to a pursuer for an extended period, the non-reduced belief set of that pursuer could become a copy of the map, slowing down the process of selecting its subgoal. A reduced belief set could contain the same corners of the entire map. As a result, the pursuers with the same belief sets are confused and cease moving in the direction of the target. This unwanted behavior can be observed from the round-table scenario (see Figure 1.4).

Isaza et al. introduces a novel *cover* heuristic, which is incorporated in an new distributed multi-agent algorithm, called Cover with Risk and Abstraction (CRA) [19]. The cover heuristic evaluates a group of agents jointly by counting the number of states that this group of agents can reach faster than any opponent, through a bidirectional expansion over the entire map. A pursuer computes its cover by expanding the cover from its current state and other pursuers' current states. Each pursuer selects an action that maximizes its cover.

Let $|S|$ be the size of the map and b be the average branching factor. This naive greedy approach is subject to a $O(|S|^b)$-bounded computational overhead at each planning cycle. Naively maximizing the cover also causes a pursuer that is approaching to the target faster than the other pursuers, to hold back. In response to these issues, Isaza introduced a *risk* factor to make a pursuer less conservative, and used state abstraction as in PRA* [50] to combat the computational overhead of cover over the original state space. Among existing multi-agent moving target search methods, CRA demonstrates the best quality of cooperation and highest success rate to solve the round-table pursuit game. Yet, it does not satisfy the real-time property because the computation of cover is dependent on the map size.

[2] Adversarial search is used on problems that involve agents with two conflicting goals [43]. When an agent plans for an action, it must consider any possible actions that its opponent might take. This approach is usually applied in games such as chess; it is also applicable to the pursuit game as the two opposing types of agents are the pursuer(s) and the target.

3.5. Reinforcement Learning Approaches

Reinforcement learning (RL) is about making a sequence of decisions in an environment that is modeled as a finite Markov decision process[3]; it allows agent(s) to learn a policy (mapping each state to an action) that maximizes the long-term reward [52]. Typical approaches use some form of function approximator to encode (an approximation to) the policy. In this section, we will briefly describe how RL can be applied to the pursuit game, followed by RL's role in self-play learners (i.e., an agent that learns by playing games against itself). We will conclude this section with a discussion about the applicability of RL to computer games.

Typical multi-agent Reinforcement Learning approaches try to achieve cooperation with information shared on sensation, policies and joint tasks, and conduct on-line, on-policy learning (see Definition 2.14 and 2.15). For example, Kalyanakrishnan et al. tackle the problem called *half field offense* in RoboCup Soccer, where the offense team tries to maximize the number of goals it scores until the ball becomes the possession of a team of defenders or goes out of bound [24]. Their results show that adding inter-agent communication for sharing information helps the players learn more effectively, in that the learner's performance dominates an approach that does not use sharing, starting around 5000 episodes.

As another example, Wang and Xu introduce an adaptive algorithm called *POESALA* to four-pursuer pursuit games [56]. Using *Q-learning* [52], POESALA tries to learn an optimal joint pursuit strategy against a randomly moving target on four-connected grids. They show that POESALA outperforms a comparable heuristic algorithm and a distributed learner after 15,000 episodes. In games, however, enemy units often actively evade the pursuers. To date, determining an effective pursuit policy over an intelligent evading target remains an open problem.

Many RL systems encode the policy using some function approximators. For example, Tesauro's TD-Gammon combines a multi-layer neural network (see details in Section 4.1.3) and a TD approach, TD(λ), to evaluate its state-action pairs and to learn from the outcome of the game of backgammon [55]. In particular, the agent learns the policy through self-play games with randomness induced by rolling the dice. Besides the rules of the game, TD-Gammon does not require initial knowledge about the strategy at the onset of the game,

[3] A Markov decision process (MDP) is a discrete, stochastic decision making process defined by a set of states S and actions A and by the one-step dynamic of the environment with transition probabilities $P(s'|s,a) \in [0,1]$ and costs $R(s',s,a) \in \Re$, where $s, s' \in S$ and $a \in A$ [52].

as it back-progates the changes in state value (i.e., error) using TD(λ). The results show that TD-Gammon outperforms the Neurogammon learner, which is initialized using human expertise. However, the training process is time consuming, as early training episodes can take up thousands of time-steps and the best performance is obtained after training on 200,000 games.

Overall, RL approaches usually require substantial amount of training to some policy (i.e.,, not necessarily optimal), and moreover, RL is applied to problems that can be formulated in a MDP or partially observable MDP. In the pursuit game, if the target agent is also changing, the model of the problem becomes dynamic. Thus, the standard MDP does not apply because MDP is applicable to only stationary models. In particular to multi-agent pursuit game, Wang and Xu identify two major problems in learning the optimal pursuit strategies in multi-agent systems: 1) learning is not guaranteed to converge due to circulation among agents' actions (i.e., "thrashing" in Section 3.2), and 2) learned strategy is not optimal due to conflicts of actions (i.e., pusuers clumping together) [56]. Therefore, learning the multi-agent pursuit game in general and trying to formulate the pursuit game in a MDP are challenging.

Furthermore, as computer games require real-time performance, game designers disallow learning on-line to accommodate fast frame rate. Learning any game-play strategy will need to be decoupled from run-time before a good approximator can be exploited to produce reasonable actions on-line. The class of RL approaches on predictive state representation (PSR) could make generalized learning across different environments possible in the back-end. PSR uses features that are flexible, learnable, and portable to new maps [49, 54] to learn desired optimal strategies. Game developers can train a learner at the development phase, and exploit the resulting learner as an evaluation function in the final production of the game.

3.6. Other methods

A system, called Boids, which simulates real-time flocking motions and demonstrates various steering behaviors in continuous space, has been applied to moving target search [41]. Flocking units exhibit highly reactive coordination by eliminating collisions through novel definitions of separation, alignment and cohesion with flock-mates nearby. In the scenario of pursuing a moving object, the units travel as one big unit. They follow the target, which is comparable to a single-agent moving target search.

Transfer of learning is a well studied subject in educational psychology and economics [8]. Ormrod, researching in educational psychology, found a close correlation between transfer of learning and problem solving due to the general application of past experience to solve a problem in new situations [38]. This work indicates a potential application of transfer of learning in the context of AI. Sharma et al. use relative game-state information as portable features in adversarial Real-Time Strategy games. They show that transfer of learning can improve performance across various games in different areas on a map [47]. It is an on-line learning approach that combines case-base reasoning (CRB) and RL, which requires multiple training episodes to improve performance during run-time. In this approach, CRB serves as a function approximator for RL, while RL provides a temporal-difference based revision to CBR. This combination can cope with unseen related problems based on experience. Games that are run in the current batch-simulation are used as examples to learn the pursuit policy in a new game in the same simulation. In fact, this is a form of supervised learning in terms of machine learning techniques.

More recently, Samadi et al. applies a supervised learning architecture to solve problems such as the sliding-tile puzzles and Towers of Hanoi [46]. They use a simple artificial neural network to learn multiple heuristics from a set of training examples with various solution costs. The trained network serves as an evaluation function when it is applied in SAS methods. Their results show that this learned function can improve game-play performance by significantly reducing planning effort and being scalable to larger problems. Although their work applies learning to single-agent methods with static goals, it inspires our work for using supervised learning technique to derive an approximation of a optimal policy in the adversarial pursuit game.

In summary, it is challenging to strike a balance between real-time performance and the cost of learning to aid the quality of cooperation for multi-agent pursuit games. Many existing SAS methods lean toward the extreme of real-time performance by instructing individual pursuers to follow the shortest paths to the target, while cooperative methods require appropriate representation of knowledge to minimize computational overhead. Furthermore, on-line learning methods require substantial training before the pursuers exhibit reasonable behaviors in performance-critical applications, for example during a video game. It is also non-trivial for the agents to adapt learned behaviors in new maps. We address these issues by performing learning off-line and utilizing problem-independent features to increase portability of the learned policies.

Chapter 4

Off-line Learning to Pursue

The task of multiple agents pursuing a moving target is a difficult path-finding problem in that achieving cooperation is non-trivial. One way to achieve this task is to incorporate learning. Learning has shown its advantage of improving performance on various SAS methods, as in Section 3.2, for finding a shortest solution path.

We explore learning to pursue using two pursuers and one target on fully observable four-connected grid-maps in which two pursuers are sufficient; perfect heuristic (see Definition 2.6) is not always available on these maps. Most existing single-agent moving target search approaches apply to pursuers that can *outrun*, instead of *outsmarting*, the target with the assumption of a slower target (see Section 3.2). In the following study, all agents have the same speed, which makes the problem more interesting and challenging. Also, we make the following assumptions:

- All agents are aware of each other's locations at all times. Future work will extend our methods to partially observable problems.
- More than one agent can occupy the same state, although our desired approach is expected to learn to minimize such occurrences during pursuit.
- Four cardinal actions (defined in Section 2.1) and $stay$ are allowed at a state, making the maximum branching factor of a state five.

In our framework, we decouple learning from run-time by using the architecture as shown in Figure 4.1. The goal is to learn from a set of past experiences to produce reasonable, cooperative behaviors at the run-time of a pursuit game on a previously unseen map. There are two phases in this framework: training (off-line) and run-time (on-line). In the following sections, we examine each phase in detail.

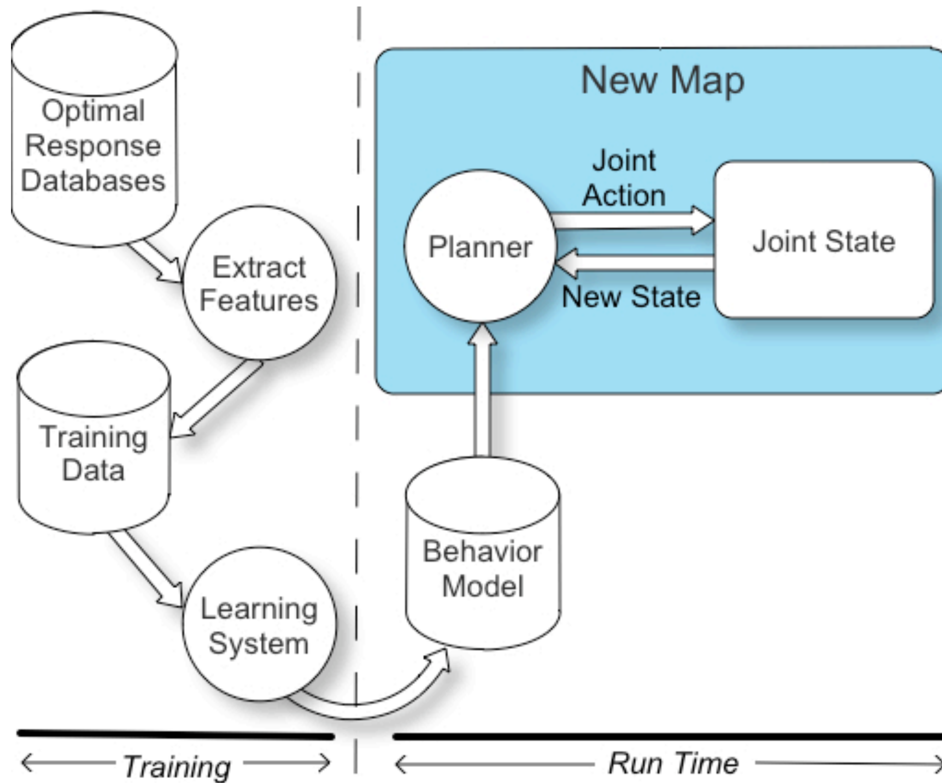


Figure 4.1: The system architecture used in this study.

4.1. The Training Phase

The training phase is an off-line process of supervised learning (see Definition 2.16). Given a target evasion policy and a map, we first compute the corresponding optimal response database (see Definition 2.10) for all possible initial joint state configurations. We then sample a set of instances randomly, and extract a set of portable features and the optimal value of capture-travel from each example to construct a set of training data. Finally, the training data serves as the input to a machine learning system for generating a behavior model, which can be independently applied in a search at run-time.

4.1.1. Computing Optimal Response Databases

The computation of an optimal response database for the pursuit game, which is carried out off-line, adopts the end-game analysis methods from board games. The input is comprised of the game map, agent information, and the target policy. In particular, agent information specifies the number of pursuers and the speed of each agent. In this study, we consider agents of a uniform speed, which is set at 1.0 as mentioned in Section 2.1.

TargetPolicy (map m, pursuers n, algorithm $evade_alg$):

1: generate the set J for all possible joint states, for map m and $n+1$ agents
2: **for** each j in J **do**
3: $\pi_{target}[j] \leftarrow evade_alg.next_action(j)$
4: **end for**
5: **save** π_{target}

OptimalResponse (map m, pursuers n, policy π_{target}):

1: generate the set J for all possible joint states, for map m and $n+1$ agents
2: **for** each j in J **do**
3: $h^*[j] \leftarrow \infty$
4: **if** j is a capture-state **then**
5: insert $\langle j, h = 0\rangle$ into priority queue q
6: **end if**
7: **end for**
8: **while** q is not empty **do**
9: $\langle j, h^*[j]\rangle \leftarrow q$.pop()
10: generate all possible joint action κ at j {** κ *is defined in Section 2.3***}
11: ***for** each joint action* κ *at* j ***do***
12: $j' = \kappa(j)$
13: ***if*** $\pi_{target}[j'] = j.getStateOfTarget()$ **AND** $h^*[j'] = \infty$ ***then***
14: $h \leftarrow h^*[j] + 1$
15: *insert* $\langle j', h\rangle$ *into* q
16: ***end if***
17: ***end for***
18: **end while**
19: **save** h^*

Figure 4.2: The OptimalResponse Algorithm.

As another input for the optimal response database, the target policy is computed a priori. We assume that the policy is deterministic and Markovian[1]. The target policy uses a pre-specified evading strategy to generate a move for every joint state configuration as shown in Figure 4.2. The result is stored in a table, so that it can serve as look-up table to compute its corresponding optimal pursuit response subsequently. This method is also a way to solve tie-breaking on the target's policy at run-time. There are many cases in which the target has multiple choice of actions with the same lowest cost. With randomness, the target might choose a different action from the one that is used to compute the optimal pursuit response database. This is problematic for computing the database because the target is not deterministic while the database does not currently handle uncertainty. Therefore, we

[1]The Markov property is defined as such that the likelihood of a given future state depends only on its present state, but not on any past states, at any given time [53].

solve tie-breaking on the target policy by storing the deterministic actions in a table.

In response to the target policy, the computation of the pursuers' optimal response starts from all possible joint state configurations where the target is captured (**OptimalResponse**: line 5 in Figure 4.2). The capture-travel for these configurations is clearly 0. Following the assumption of the target policy being deterministic and Markovian in any given joint state configuration j (defined in Section 2.3), the computation incorporates the bottom-up dynamic programming approach to back-propagate the value of capture-travel to all parent configurations (line 8 to 19). The computation stops when the values of capture-travel for *all* configurations are generated. During this process, the database keeps only the smallest value for each game-state configuration (line 13 and 14). Thus, the result is a optimal capture-travel for every joint state of the current map with regards to the target policy.

Let $|S|$ be the size of a map. The time complexity for both the target policy and the pursuers' optimal response database is $O(|S|^{n+1})$ for n pursuers and one target. Although this exponential complexity makes the computation tractable only for small maps and n, the resulting database can be used to derive a pursuit model that is efficient at run-time.

For the remaining body of this thesis, we refer to an optimal response database using the abbreviation ORD.

4.1.2. Features

An ORD contains the joint state configurations in a given map. To made its result portable to future application on different maps, it uses features to identify similar instances on different maps. This process is the fundamental operation of learning to pursue. As the joint state configuration is described in x-y coordinates (i.e., map-dependent), such set of features is not portable to different maps, making it difficult to identify similar instances. Instead of recording the coordinate of the agents' position, it is more sensible to record the agent-topology and the features about their local environment. In this section, we explore such set of portable features that could help a learner and/or a pursuit-planner identify similar instances across different maps.

Distance

Since the maps used in this study are four-connected grids and the cost in distance for each non-stay action is 1, we can use Manhattan distance (see Definition 2.6), $manH$ to estimate the travel distance between any pair of states. In particular, we record $manH$ for every pair of the current states of the three agents. Assuming that the pursuers are A and B, and the

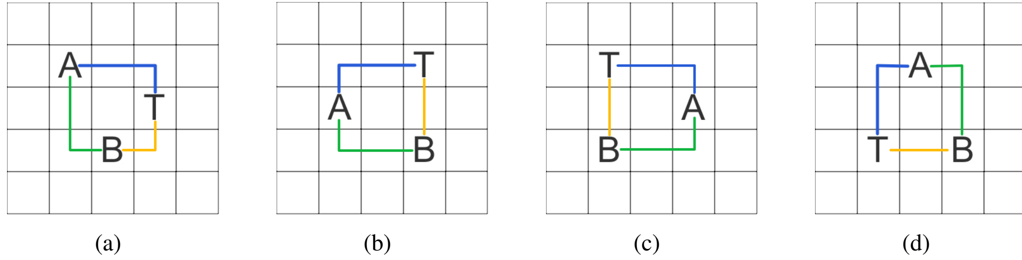


Figure 4.3: Several examples of identical agent-topologies.

target is T, the distance-features are:

- $manH(A, T)$: estimated distance between pursuer A and target T,
- $manH(B, T)$: estimated distance between pursuer B and target T, and
- $manH(A, B)$: estimated distance between pursuers A and B.

As in an sense of straight-line distance on four-connected grids, these features describe how far apart from the the target and from each other each pursuer is. They approximate the shape of the "triangular" topology over all agents on a grid, regardless the ordering of the agents. However, we label the pursuers systematically. Although there might be different sorting methods of the pursuers' positions, the pursuers here are sorted by their coordinates, such that pursuer A is always the one at the lowest y-coordinate and (then) the x-coordinate. This arrangement treats the topologies with the same $manH$ values as the same topology, as shown in Figure 4.3. This method is known as state aliasing or classes of equivalence over the state space.

Difference in Distance to Target

Another potentially useful distance feature is the difference in distance to target, which is $\delta = |manH(A, T) - manH(B, T)|$. It indicates whether the two pursuers are pursuing the target for similar distance or not. This feature is based on the assumption that both pursuers can close in to the target in a similar amount of time-steps.

Consider the scenario as in Figure 4.4(a), both pursuers are in-line with the target, such that one is closer to the target than the other. If the pursuers maintain this alignment during the pursuit, they allow the target to flee in the opposite direction, possibly indefinitely around an obstacle. On the other hand, if the pursuers spread out to cover more area of the map, the target could be forced to change direction in its movement, which could lead to a faster captures. In Figure 4.4(b), for example, both pursuers encircle the target using the

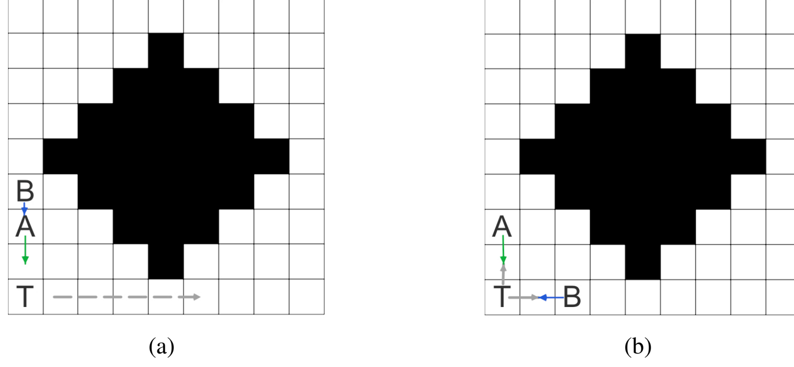


Figure 4.4: Examples of different delta scenarios on a 9×9 grid.

edges of the map while they are both maximum 2 steps away from the capture. We would like to give more credit to cases as in Figure 4.4(b) than in Figure 4.4(a). This feature is considered to be a supporting feature of the distance-features.

Mobility

As a component of the target's evasion algorithm in MA-MTS (Section 3.4), mobility is used to count the number of traversable states by looking in a small local search space with respect to the target. It helps the target maximize its chance to flee to an open space; the wider the better. In this thesis, we extend this term to both pursuer and target. To minimize the computational cost over all possible features, the computation is simplified from lookahead depth $d > 1$ to $d = 1$.

In general, the mobility, denoted by $m(s)$, is the number of distinct actions available at state s, or $m(s) = |A_s|$ (defined in Section 2.1: Grid). For the target, $m(s_T|s_A, s_B)$ excludes any action that leads to the current state of a pursuer. That is,

$$m(s_T|s_A, s_B) = |\left\{a(s_T) \ : \ a \in A_{s_T},\ a(s_T) \neq s_A,\ a(s_T) \neq s_B\right\}|, \tag{4.1}$$

where $a(s_T)$ is an action function that translates the state s_T to the successor state s'_T on action a. For example, Figure 4.5(a) shows that $m(s_T) = 5$ (including the $stay$ action) since none of the target's actions are blocked by any obstacle or a pursuer. In Figure 4.5(b), pursuer A is adjacent to the target; and hence, $m(s_T) = 3$.

In single-agent cases, $m(s_p) = |A(s_p)|$ for the pursuer p, including the action that leads to the current state of the target. In multi-agent cases, let P be the set of all pursuers, or $P = \{A, B\}$ in this study. The mobility of P, as in Equation 4.2, is the number of distinct joint actions κ available at the joint state j (defined in Section 2.3) with regard to the current state of any pursuer $p \in P$, which is jointly denoted by $j^P = \{s_A, s_B\} \subset j$. The definition

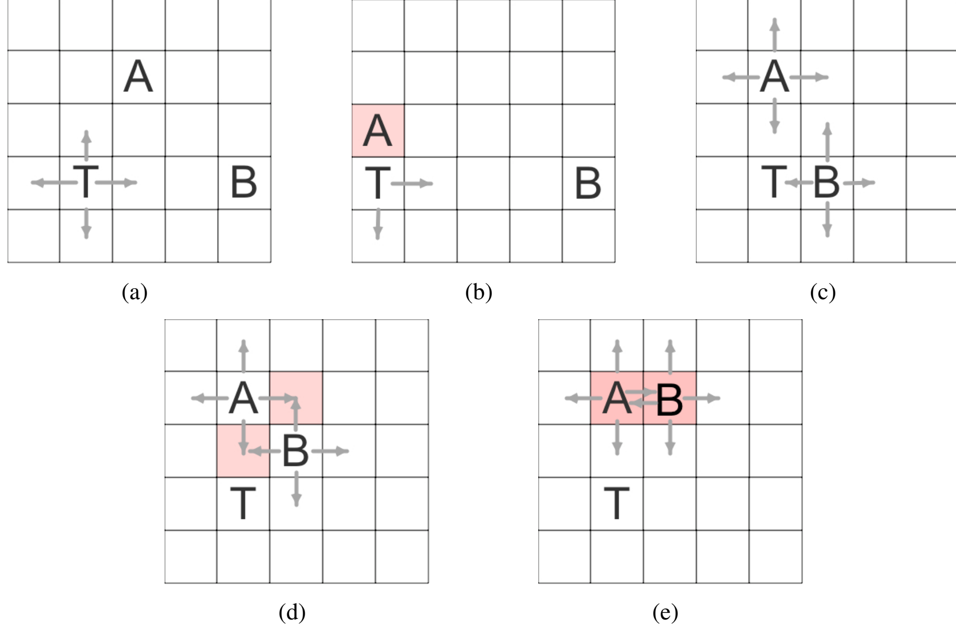


Figure 4.5: Mobility Computation: (a) and (b) compute for the target; (c) to (e) compute for pursuers.

of $\kappa(j^P)$ differs from the definition of $\kappa(j)$ only in that the target state is not considered. The same distinction also applies to the definitions of $\mathbb{A}_{j^P}$ and $\mathbb{A}_j$. The joint action that leads to either an alternative permutation of j^P (i.e., a swap of the pursuers' states), or a joint state that both pursuers coincide at the same state, are excluded in the computation of $m(j^P)$. Such computation follows the intuition of the *cover* heuristic (Section 3.4) on maximizing the cover area, but differs as it is independent of the map size.

$$m(j^P) = |\left\{\kappa(j^P) \,:\, \kappa \in \mathbb{A}(j^P),\, \kappa(j^P) \neq j^P,\, s'_A, s'_B \in \kappa(j^P),\, s'_A \neq s'_B\right\}|, \tag{4.2}$$

where s'_A and s'_B are a successor of s_A and s_B respectively.

Figure 4.5(c) illustrates the case where both pursuers have maximum amount freedom to move, even though the target is standing right next to B. In this case, $m(j^P) = 25$. Figure 4.5(d) shows the two cases where the successor states of pursuer A and B will coincide. Therefore, $m(j^P) = 23$. In addition to sharing the same pursuer state in a joint-state successor, swapping state is also undesirable since the agent-topology is the same as the current joint state. Thus, $m(j^P) = 22$ in Figure 4.5(e).

Sensing in Local Environment

As Manhattan distance is used to represent the agent-topology, it ignores obstacles in close proximity of any agent. There are, however, significant difference between a case that

occurs in an open area and a case that occurs in an area with static obstacles in the agent's way. To compactly describe map topology around an agent, we introduce virtual sensors on each agent for detecting obstacles or an opponent nearby. The range of sensor is limited to achieve the real-time property. We consider two types of light-weighed sensors on four-connected grid maps: *beam* sensor and *hill-climb* sensor.

The *Beam* sensor operates as a laser beam shooting outward in all four cardinal directions of an agent's current state. It is used to detect static obstacles on the map. Given a hand-tuned range limit R, the sensor returns four integer values, r_{east}, r_{south}, r_{west} and $r_{\text{north}} \in [0, R]$. Each beam emits through a state until it reaches an obstacle or the predefined range. The advantages of this sensor include its simplicity and the applicability to all agents while respecting to the real-time property (see Section 2.2). It is, however, a naive sensor because it misses the detection of any obstacle located diagonally to the sensor-origin. Also, it does not detect an opponent because it is difficult to justify what the value would be if the beam touches an opponent or an obstacle. Consider the case in which the agent tries to stay away from obstacles to avoid any potential heuristic depression while it attempts to approach the opponent. The learner would need to learn a weight with opposing signs for the two circumstances, which are embodied in the same feature.

The *hill-climb* sensor can distinguish the target from an obstacle. Since the target can use the mobility feature to detect open areas, this sensor is not necessary on the target, but it is applicable to the pursuers. Given a fixed number of state-expansions R (the range), the hill-climb sensor on a pursuer uses the heuristic $manH(s, s_{target})$ for any state s to hill-climb (see Section 3.2) toward the target. It indicates the distance that a pursuer can travel before its path is blocked by an obstacle or before it can reach the target within the pre-specified range. At any state s, the sensor value r is computed by the following equation:

$$r = \begin{cases} d/R & \text{if target is } \textit{not} \text{ detected within } d \leq R \text{ steps.} \\ 2 & \text{otherwise, target is seen within range.} \end{cases}$$

If an obstacle is detected within the range R, then r is a positive real number in $[0, 1)$. Otherwise, $r = 1$ indicates that the agent can freely travel R steps toward the target. If the target is detected within R steps inclusively, r takes a special value to distinguish the "sighting" of the opponent from an obstacle. This special value can be any constant greater than 1 because this sensor is designed to lead a pursuer directly toward the target if it falls within the pursuer's line-of-sight. We simply use 2.0 as the special feature value.

The corresponding algorithm is provided in Figure 4.6. First, it compares the current state s of a pursuer to the target state (line 2). If they coincide, it returns the capture value

HillClimbSensor (state s, state s_{target}): $\{$** *R is a pre-specifed non-zero constant* **$\}$

1: $captured \leftarrow 2.0$
2: **if** $s = s_{target}$ **then**
3: **return** $captured$
4: **end if**
5: $h \leftarrow manH(s, s_{target})$
6: $d \leftarrow 0$
7: $current \leftarrow s$
8: **while** $d < R$ **do**
9: expand $current$ to find the successor state s' = $argmin_{s'}\ manH(s', s_{target})$
10: $h' \leftarrow manH(s', s_{target})$
11: **if** $h' \geq h$ **then** $\{$** $current$ is in heuristic depression (i.e., detected an obstacle) **$\}$
12: **break**
13: **end if**
14: **if** $s' = s_{target}$ **then**
15: **return** $captured$
16: **end if**
17: $current \leftarrow s'$
18: increment d by 1
19: **end while**
20: **return** $\frac{d}{R}$

Figure 4.6: The HillClimbSensor Algorithm.

right away. Otherwise, it determines the next state s' within range by expanding s to s' with the lowest heuristic value (line 9). If s is in heuristic depression (see Definition 2.7), then the sensor has reached an obstacle and returns $\frac{d}{R}$. Also, if s' is where the target is, the algorithm returns the capture value. Otherwise, current state is updated to s' while a successful step has made (line 17 and 18).

In summary, these features provide information about the agent-topolgy, their mobility, and local search space. Agent-topology only captures the relative positioning information about each agent. Mobility can further describer how viable their immediate action plan is, while sensor information about the local space can help an agent recognize a local terrain. The combination of these features can be used to describe an instance over any grid map.

4.1.3. The Learning System

As different game maps may contain similar obstacle configurations, local features can describe a pursuit instance regardless the actual map from which they are extracted. These features enable the possibility of producing map-independent pursuit behaviors. We can construct a set of training data using these features labeled with the optimal capture-travel

T^* that is determined by the corresponding ORD. The training data can serve as the input to a learning system for generating a pursuit behavior model with two pursuers. Therefore, we can apply a supervised learning architecture based on artificial neural network (ANN), which is inspired by Samadi's approach that combines multiple heuristics to improve game-play performance (see Section 3.6). We choose supervised learning over reinforcement learning (RL) for two major reasons. First, when the target changes its location, the problem to be solved becomes dynamic. As the standard MDP does not apply because it only applies to stationary models, RL does not apply either. Secondly, the ORD already provides the optimal strategies that a learner can learn from. There is no learner that can outperform the optimal strategy. As a result, we use supervised learning to approximate the optimal strategies stored in the ORD.

An Overview of ANN

An ANN emulates the brain's information-processing circuitry by constructing a network of neurons, which are capable of collecting, processing, and disseminating electronic signals [44]. A link, which carries a weight $w_{i,j}$ can be established to propagate an activation value a_i (i.e., the signal) from neuron i to neuron j. The weight indicates the strength and the sign of this link. To produce the activation value a_j, the propagation process first computes the weighted sum of all of its input a_i for $i = 0, \ldots, n$. This sum is then sent to the activation function g_j at neuron j to produce the output value of a_j:

$$a_j = g_j \left(\sum_{i=0}^{n} w_{i,j} a_i \right).$$

The activation function g_j can take the form of one of many types of functions, such as linear regression, sigmoid function, and gaussian function, to name a few.

Types of ANN

There are two types of ANN structures in general: feed-forward networks and recurrent networks. In this study, we use the feed-forward network, which is a function that simply maps its input to output (i.e., an approximation of optimal capture-travel). However, the recurrent network, which feeds the output back into itself as an input, does not apply to our study. The main reasons are that, at run time, 1) the value of capture-travel is not available until the end of thes game, but it is used to evaluate the joint states at every planning step, and 2) the output is an estimated value to the optimal capture-travle. Feeding the estimated

capture-travel back into the network for tuning could confuse the learning system that intends to learn the optimal capture-travel. As this is an initial study that incorporate the ORD into the pursuit game, feed-forward network can approximate the optimal capture-travel of a given instance, which can help us analyze the correlation of the input features. Moreover, the resulting network, also called a *behavior model*, can be used directly as an evaluation function by a search algorithm at run-time.

Architecture

A network has three types of layers: input layer, hidden layer(s), and output layer. For the pursuit game, each feature about a pursuit instance is represented as a neuron in the input layer, whereas the corresponding T^* is represented as a single neuron in the output layer (see Figure 4.7(a)). The backpropagation algorithm is applied to train the network. In this study, we use a simple, three-layer, fully connected ANN architecture, which uses sigmoid and linear activation functions on the hidden neurons and the output neuron respectively. Using a single layer of hidden neurons that are activated by non-linear squashing functions can produce a simple function approximator of T^*, which can help analyze how each feature influences the output value. Using multiple hidden layers with nonlinear squashing function can produce a nonlinear function approximator, which makes it difficult to study the influence of individual features with respect to T^*; such nonlinear approximator will be a subject of future studies on the pursuit game.

Figure 4.7(b) illustrates a more precise, hand-tuned neural network architecture. It takes

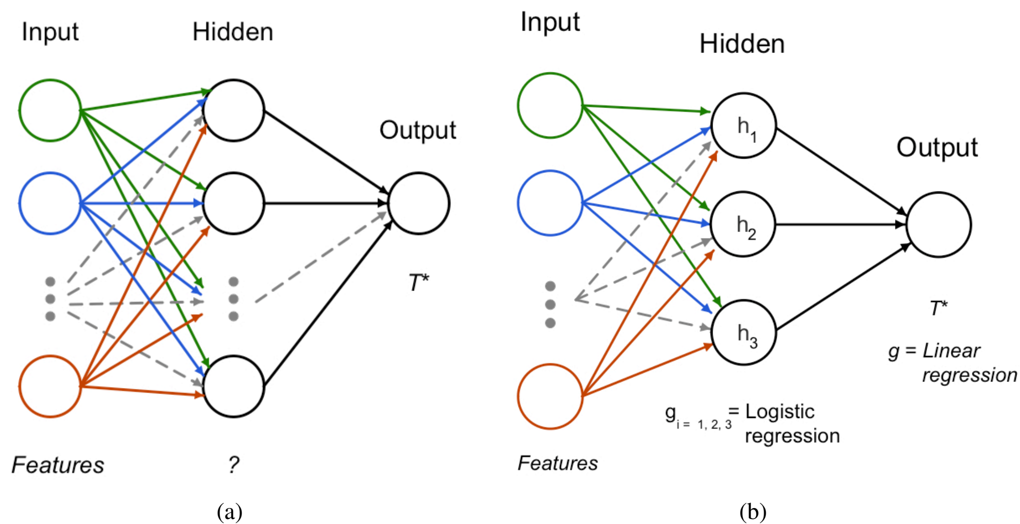


Figure 4.7: ANN achitectures: (a) a general ANN feed-forward architecture with a single hidden layer for the pursuit game and (b) our ANN architecture for the pursuit game.

the three-layer structure with three[2] neurons in the hidden layer. The activation function on every hidden neuron is a logistic regression, whose output signal is a real number between 0 and 1. Each hidden neuron applies the logistic regression on all input feature values; its output becomes an input to the output neuron. The output neuron then computes a linear combination on all input signals from the hidden neurons to produce the estimate of capture-travel in $(-\infty, +\infty)$.

4.2. Run-Time Phase

At the run-time phase, we test a behavior model that is generated by the learning system in a previously unseen map. The behavior model is incorporated into a real-time heuristic pursuit algorithm, in which it serves as an evaluation function f over a pursuit instance. As illustrated in Figure 4.1, the centralized action planner for the pursuers performs the following general steps:

1. The planner observes the joint state j over all agents.
2. It uses the behavior model (an ANN) to evaluate j.
3. It computes a joint path from j with a pre-specified length (or depth) and returns a joint action to the simulation.

The simulation executes the joint action, followed by a target's move, based on the turn-base setting. This process repeats until the target is captured, or the simulation times out.

4.2.1. An Adversarial Approach

The planner used by the pursuers opts for an appropriate real-time search algorithm, so that it can strike a balance between planning effort per action and pursuit quality. Since the pursuit game involves two opposing types of agents, we use an adversarial search algorithm to utilize a learned behavior model. The father of all adversarial search is the minimax algorithm. The pseudocode (Figure A.1) and a sample game tree (Figure A.3) are provided in the Appendix A respectively.

In terms of the pursuit game, the planner can apply this algorithm for all pursuers to evaluate a joint state based on the predicted movement of the target. It treats all pursuers as one central minded player and the target the other. At the pursuers' ply, they try to

[2] By observation, more hidden neurons in this architecture do not show any impact to the pursuit performance at run-time.

maximize the utility of a joint-state j, called $f(j)$. Since the pursuers, in fact, attempt to minimize the estimated capture-travel T_j, the utility of a joint-state j takes the negation of T_j, or $f(j) = -T_j$. The utility at the pursuers' ply is in turn based on the target's ply by assuming that the target is trying to minimize the utility of a successor joint state j'.

By convention, a full-ply in minimax include one Max-ply and one Min-ply. On four-connected grids, an agent can have branching factor $b = 5$ at a state, including $stay$. For a joint state with two pursuers and one target, one full-ply in minimax will need to examine $b^{3d} = 5^3 = 125$ successor joint states, 125^2 for two full plies, and 125^3 for three. Even without considering any internal computation cost for evaluating one state, three full-ply lookeahead requires an examination of almost two million joint states. This significant overhead brings a challenge in using ANN network at run-time.

4.2.2. Multi-Agent Minimax

To address the computational overhead for increasing number of plies, we modified the standard minimax algorithm with alpha-beta pruning by assigning one ply for each agent while preserving the joint state evaluation. At a pursuer's ply, the algorithm maximizes the f-value of a joint state j, whereas the algorithm tries to minimize the $f(j')$ at the target's ply, such that j' is a successor joint state of j by varying the target's state. For n pursuers and one target, each ply is $\frac{1}{n+1}$ of a full ply. We call this algorithm Multi-agent Minimax (or MA-Minimax), which relies on the zero-sum property of a two-player game.

As show in Figure 4.8, the planner must know the simulation time-out value, which is set to the total number of states $|S|$ on the current map. It also pre-loads an ANN for joint state evaluation. The MA-Minimax-Driver algorithm, which is initiated by the simulation, receives the current joint state and the lookahead depth. All agents are scheduled in a priority queue by their allowable $time\text{-}to\text{-}move$ according to the simulation clock. This

Global input:
1. set $max \leftarrow |S|$, the total number of states on the current map
2. load the neural network, ann

MA-Minimax-Driver (joint_state j, depth d):

1: initialize the global priority move queue q for each agent u by its next allowable $time\text{-}to\text{-}move$
2: **MA-Minimax** (j, $-max$, max, d)
3: execute the cached actions for the pursuers.

Figure 4.8: The MA-Minimax-Driver algorithm.

queue helps automate the process of iterating through subsequent successor joint states. With speed of 1.0 and the current simulation clock at time $\mathbb{C}$, the $time\text{-}to\text{-}move$ for all pursuers is $\mathbb{C} + 1.0$ and $\mathbb{C} + 2.0$ for the target initially based on the turn-based setting. This priority ensures that the pursuers consider their joint actions before predicting the targets whereabout. The driver then launches a full MA-Minimax, followed by the execution of one cached move for each pursuer simultaneously (line 2 and 3).

Figure 4.9 shows the detailed algorithm of MA-Minimax. Agent u is the first agent scheduled to move according to the priority queue (line 1 of the MA-Minimax algorithm). This algorithm then checks whether the current joint state j is a terminal state or it has reached the bottom ply (line 2 to 4). If so, its f-value is returned to the previous ply. Otherwise, it generates a portion of j's successors with regard to the current state of agent u. Its identity—a pursuer or a target—indicates whether this is a max-ply or a min-ply (line 5 to 7). If u is a pursuer, then MA-Minimax works on a max-ply; otherwise, it works a on min-ply. For each successor state s' of the current state s of u, MA-Minimax reconstructs a successor joint state j' with s' (line 11), and recursively calls itself on the next ply (line 12 to 16). Only the maximum return value $f(j')$ is recorded, and so as its corresponding action (line 19 and 20). Line 25 handles a potential cut-off of a subtree if the best value of the min-ply can not match up the best value of the max-ply. At last, the best action $a*$ for u is cached, and its value is returned to the previous ply.

The utility value is determined by the function $f(j) : |S|^{n+1} \rightarrow \Re$ for n pursuers on a map with $|S|$ number of states. To evaluate j using an ANN, a set of features must be extracted in the same manner as defined in this ANN (line 1 in Figure 4.9: the f-function). The ANN estimates the capture-travel e over this set features on j (line 2). Based on the zero-sum property, e is mapped to e_{norm} within the range $[-|S|, +|S|]$ (line 3 and 4). If j resembles a capture-state, then e approaches 0 and its e_{norm} approaches $-|S|$. Any value of e that is greater than $2|S|$ indicates that j is expected to result in a failure capture. Its norm e_{norm} is mapped onto $|S|$. Therefore, a pursuer's ply prefers a negative e_{norm} with small magnitude, whereas the target prefers a large positive value. To follow the max-ply manner, e_{norm} must be negated for a pursuer (line 6), as mentioned in Section 4.2.1.

MA-Minimax (joint_state j, value α, value β, depth d):

1: agent $u \leftarrow q.pop()$
2: **if** j is a capture-state or $d = 0$ **then**
3: **return** $\mathbf{f}(j, u)$
4: **end if**
5: **if** u is not a pursuer **then**
6: swap values of α and β
7: **end if**
8: $v^* \leftarrow -\infty$
9: **for** each successor s' of the current state s of u **do**
10: update or insert $\langle u, time\text{-}to\text{-}move + 1.0\rangle$ in q such that u will be at s'
11: $j' \leftarrow$ change only the current state of u from s to s' in j
12: **if** u is a pursuer **then**
13: $v \leftarrow$ **MA-Minimax** (j', α, β, $d - 1$)
14: **else**
15: $v \leftarrow$ -**MA-Minimax** (j', -β, -α, $d - 1$)
16: **end if**
17: reset the entry for agent u with the original $time\text{-}to\text{-}move$ in q such that u is at s
18: **if** $v > v^*$ **then**
19: $v^* \leftarrow v$
20: best action $a^* \leftarrow a$ such that $s' = a(s)$
21: **end if**
22: **if** $v^* > \alpha$ **then**
23: $\alpha \leftarrow v^*$
24: **end if**
25: **if** $v^* > \beta$ **then** {cut off!}
26: **break**
27: **end if**
28: **end for**
29: **if** u is not a pursuer **then**
30: $v^* \leftarrow -v^*$
31: **end if**
32: cache the best action a^* for u
33: **return** v^*

f (joint_state j, agent u):

1: extract the vector of $features$ of j as defined in the ANN used, ann
2: $e \leftarrow ann$.compute($features$)
3: clip the value of e between the range [0, $2max$] if e is out of this range
4: $e_{norm} \leftarrow e - max$ {** map e to the range $[-max, max]$ **}
5: **if** u is a pursuer **then**
6: **return** $-e_{norm}$
7: **else**
8: **return** e_{norm}
9: **end if**

Figure 4.9: The MA-Minimax algorithm and the *f* function with an ANN evaluator.

4.3. Expected Performance

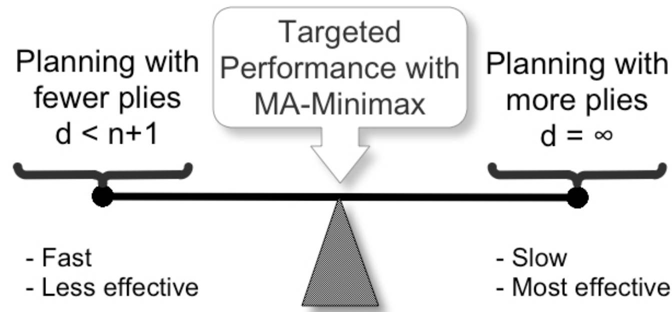


Figure 4.10: Targeted performance of MA-Minimax with our ANN evaluators.

This approach transforms a full ply into $n + 1$ plies, so that we can verify the effectiveness of a learned ANN on more values of ply d. When $d \neq x(n + 1)$ where x is the number of full plies, the algorithm allows a pursuers to plan for one more action than the other. This could be beneficial as one pursuer may appears to be more aggressive than the other. Although alpha-beta pruning reduces the number of states examined from $O(b^{3d})$ to $O(\sqrt{b^{3d}})$ in the best case on expanding in full plies, pruning is not guaranteed since the states are not ordered by values in advance. Nevertheless, any alpha-beta pruning could help reduce the planning cost, regardless of full plies or fractional plies, to further satisfy the real-time property.

The targeted performance of MA-Minimax is illustrated in Figure 4.10. On the left side of the scale, we have planning with shallow depth of plies d. Any d-value less than a full ply with a small n ($n \leq 2$) can produce short planning effort, resulting in fast action response time. However, the target is considered to be static during planning, which will increase the chance of a failure capture. On the other side of the scale, it can plan as deep as possible in a simulation without a time-out. The response time is potentially indefinitely slow, but the infinite depth guarantees the algorithm to reach a capture-state, assuming that our problem is a win for the pursuers. Our goal is to find the setting for this algorithm combined with the use of a trained ANN to strive a balance between response time and effectiveness.

Chapter 5

Experiments and Evaluation

To achieve the expected performance described in Section 4.3, we conduct various experiments using two pursuers and one target, all of uniform speed, to help us understand whether the fundamental pursuit behavior given a target strategy is learnable. We use two different frameworks: a simulation framework to generate training data and testing run-time performance, and a learning framework to train the behavior models. Due to the overhead of computing an optimal response database (ORD), we selected several small maps with less than 100 states. Given a target algorithm and the optimal response database, we use two sets of features that include different types of sensor information to learn a pursuit behavior: the beam sensor and the hill-climb sensor. Also, we use a simple ANN as a lightweight behavior model to evaluate the joint states at run-time testing. We analyze the learned behaviors through run-time simulations. The planning algorithm is MA-Miniax, which is used to compare the performances of two different types of evaluation function, whereas their result is evaluated by the aforementioned measures in Section 1.2.

5.1. Choice of Maps

The use of small maps makes the computation of an optimal pursuit response tractable, as well as helps us understand the characteristics of features and the scalability of a learned behavior over small maps. We aim to construct a behavior model using small problems that reflect a wide range of terrain configurations to induce sensible performance on different maps, and perhaps larger maps. Regarding the test domain, we select five toy maps, as shown in Figure 5.1, with increasing complexity in obstacle layout.

Empty 5×5 is a trivial grid map. Any agent can freely maneuver at any state (branching factor $b = 5$), except those along the border of the map. In contrast, the maneuverability of an agent is greatly restricted due to the four pillars in Hand-made 5×5. Only the center

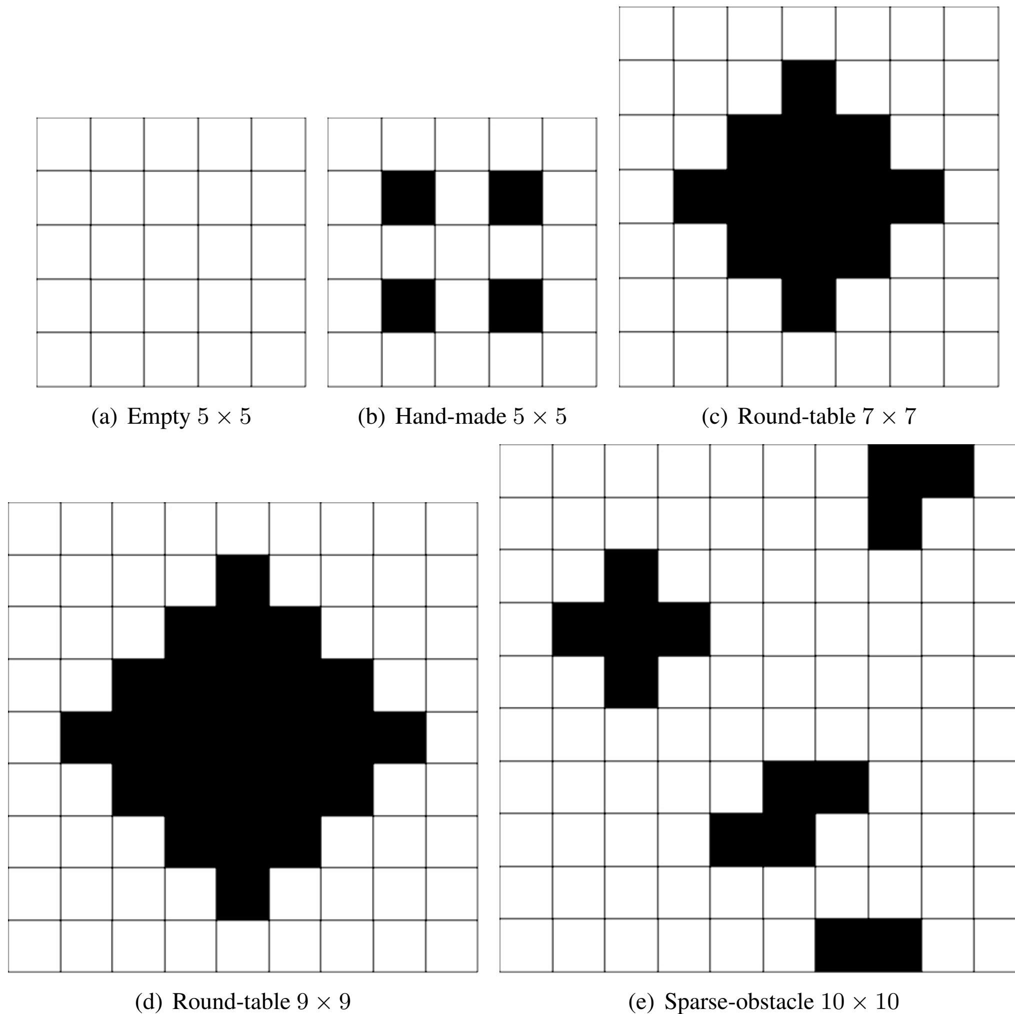


Figure 5.1: Illustrations of the toy maps used in study.

state (an intersection) has a branching factor of 5; the other states have between 2 and 4 successors. This map resembles a maze environment with no dead-ends. Each pillar stands as the smallest round-table that can be placed on a grid map. Pursuers can capture a target only if they can corner the target together. Figure 5.1(c) shows a larger round-table—the only obstacle—in the map. This map is known to be difficult to capture a target with a small amount of cooperative pursuers [19]. Once a behavior model is constructed over Round-table 7×7, we observe whether this model can derive reasonable performance on Round-table 9×9 (Figure 5.1(d)) in terms of a quick capture. Sparse-obstacle 10×10 (Figure 5.1(e)) is a test of scalability. It contains small obstacles that are sparsely placed on the map, where the effect of learning over the other four smaller maps is evaluated.

The actual problem size is provided in Table 5.1. The second column presents the various percentage of obstacle occupations. The third column shows the total number of

Map	Obstacles (%)	$\lvert S\rvert$ in original space	$\lvert S\rvert^{2+1}$ in joint space
Hand-made 5×5	16	21	$9,261$
Empty 5×5	0	25	15,625
Round-table 7×7	27	36	46,656
Round-table 9×9	31	56	175,616
Sparse 10×10	14	86	636,056

Table 5.1: The size of problems that are used in our study.

passable grid cell on the map (i.e., state size), whereas the last column presents the size of the joint space for two pursuers and one target. It is the total number of joint state configurations on the corresponding map.

5.2. Frameworks

We use two separate frameworks to perform learning and apply learning. We first use a real-time simulation framework to generate the optimal pursuit response databases. This framework is created in C++ by Isaza [19] and used for the development of the CRA algorithm (see Section 3.4). Isaza's framework can also simulate various path-finding approaches, including cooperative ones. Therefore, we also use this framework for testing the learned behavior models with the MA-Minimax Algorithm at run-time.

We use the Fast Artificial Neural Network Library (FANN) [35] for the learning system during the training phase. FANN is an open source library that implements multilayer artificial neural networks and provides a large, robust set of training algorithms and methods to manipulate the training data. As it is portable to any C++ platform, we also use this library to port a trained ANN into Isaza's framework for run-time simulations.

5.3. Target Algorithms

To study performance on a variety of pursuit tasks, we consider the following target strategies:

- **OPT**: The OPT target uses the **opt**imal evading policy to flee from its pursuers, which it assumes uses an optimal pursuit policy. The computation of the optimal pursuit policy is similar to our OptimalResponse algorithm, except that the target's policy is not pre-computed. Instead, the computation of the target's optimal policy interleaves with the computation of the pursuers' optimal policy in an adversarial fashion. It follows the OptimalStrategy algorithm in Isaza's study [18].

- **SFL**: The SFL target uses the **S**imple **Fl**ee algorithm. This algorithm randomly places twenty beacons on the states of the map at the onset of the game [19]. If the number of states on a map is less than twenty, then it places exactly one beacon per state and no extra beacons. It applies Path-refinement A* (PRA*) [50] that is based on distance-heuristic and state abstraction to compute a path between the target and the beacon that is furthest from any pursuer. Once the path is available at a predefined level of state abstraction, it is then refined to the ground level for execution. The target executes three moves of the complete path before it re-plans for the new state-configuration of all agents.

- **DAM**: The DAM target also uses state abstraction in the **D**ynamic **A**bstract **M**inimax algorithm [6]. It starts with a minimax search at the top level of abstraction and seeks an escape path from both of the pursuers by using **cu**m**ul**ative distance (also called CML) to all pursuers. If such path is found, the abstract path is refined to the ground level for execution. If not, the search continues at a lower abstraction level until it reaches the ground level. We fix lookahead depth in the minimax algorithm to be 5 fractional plies for this study.

5.4. ANN-Training Experiments

A preprocessing step prior to training is feature extraction. With the same ORD, we extraction two separate sets of features that differ only in the sensor type used (see Table 5.2).

No.	Feature Name
0	$manH_x(A,T)$
1	$manH_y(A,T)$
2	$manH_x(B,T)$
3	$manH_y(B,T)$
4	$manH_x(A,B)$
5	$manH_y(A,B)$
6	δ
7	$m(j^P)$
8	$m(j^T)$

No.	Feature Name	
	Beam Sensor	Hill-climb Sensor
9	r_{east_T}	r_A
10	r_{south_T}	r_B
11	r_{west_T}	
12	r_{north_T}	
13	r_{east_A}	
14	r_{south_A}	
15	r_{west_A}	
16	r_{north_A}	
17	r_{east_B}	
18	r_{south_B}	
19	r_{west_B}	
20	r_{north_B}	

Table 5.2: The set of features to be extracted on any pursuit instance.

Suppose the set of pursuers P is $\{A, B\}$, and the target is T. We extract the distance features between A and T, B and T, as well as A and B (features 0 to 5). The distance features are decomposed into x-components and y-components. Feature 6 is the difference in distance (in both x and y components) to target, δ. Regarding the representation of current joint state j, features 7 and 8 computes the mobility of the pursuers and the target respectively by looking around the immediate neighbours of the current state. The following features convey sensor information at the current state of each agent. Using the beam sensor as an example, feature 9 to 12 represent the sensor data in all four compass directions from the current state of the target. When the hill-climb sensor is used, only the sensor data on the pursuers are computed, which reduces the feature set by half.

5.4.1. Sensor Range

To meet the real-time constraint, the sensing range does not reach every edge of the map. However, it is still undetermined about what range value a sensor should take, without compromising performance to the real-time property. Since the beam sensor can only detect static obstacles, such as walls and fixed objects, it is considered to be a naive sensor for determining how far a pursuer is to the nearest obstacle within range. Scanning the map with a deep, straight range can potentially result in recording the boundaries of the map, making the sensor data less scalable to larger maps. For example, a beam sensing range of at least 4 steps will detect all edges of the 5×5 maps. Alternatively, if the sensing range for all beam sensors are set to 6, no 5×5 map will expose the agents to a training example in which the beam range exceeds 4. Instead trying to determine an adequate range value, we focus on verifying whether one sensor is superior than the other in solving pursuit problems by comparing both sensors at range 3. We then examine the ranges of the hill-climb sensor at 5, 7, 10 to determine the effect of increasing the sensing ranges.

5.4.2. Training Data

To study the applicability of a learned behavior model on a new map, we use the leave-one-out cross-validation method to construct the training data and the test data as shown in Table 5.3. In particular, we choose 4 out of the 5 toy maps for training, and then we test the resulting network on the remaining map at run-time. Hence, we have 5 categories of resulting networks, each of whose name is prefixed with one of the five names shown in the first column of Table 5.3. To construct the training data for each network, we randomly sample 5,000 joint states in each map, whose total number of joint state is provided

ANN Name	Training Map Combination	Test Map
ann-Empty5	Hm, R7, Rt, Sp	Empty 5×5
ann-Handmade5	Em, R7, Rt, Sp	Hand-made 5×5
ann-Round7	Em, Hm, Rt, Sp	Round-table 7×7
ann-Round9	Em, Hm, R7, Sp	Round-table 9×9
ann-Sparse10	Em, Hm, R7, Rt	Sparse 10×10

Table 5.3: Naming conventions on ANN and the maps that are used for training and testing (Hm = Hand-made 5×5, Em = Empty 5×5, R7 = Round-table 7×7, Rt = Round-table 9×9, Sp = Sparse 10×10).

in Table 5.1. As a result, each set of training data contains 20,000 instances, each of which includes a list of features and the optimal capture-travel.

5.4.3. Training and Preliminary Evaluation

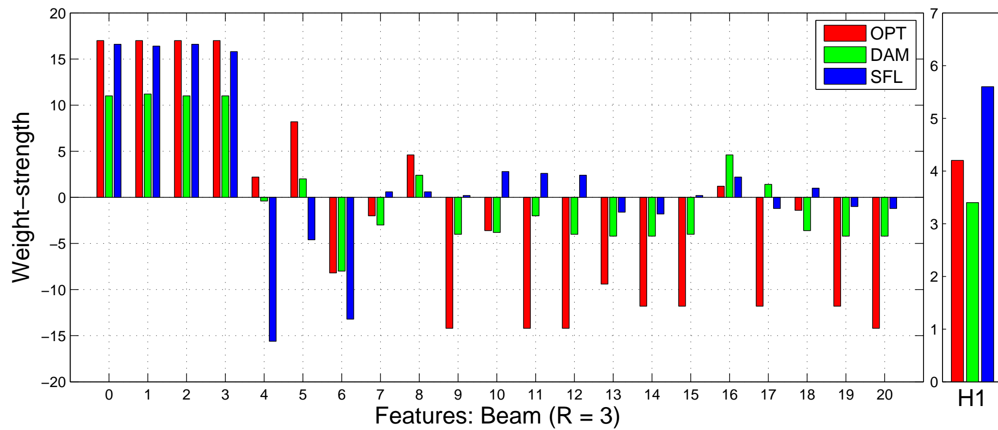
We use the resilient back-propagation (RPROP[1]) algorithm to drive the training process on different sets of data (5 maps $\times$ 2 sets of features $\times$ 3 target policies). For each map M (i.e., the hold-out map), each target policy, and each set of features, we train one behavior model over 20,000 samples from the other four maps (5,000 each). We then combine training with 10-fold cross-validation[2] on these samples. Although the training process runs through the entire 10-fold cross-validation, the model (i.e., network) is saved at the lowest validation error value over the samples. The network will be then tested over a set of sample pursuit instances on the hold-out map M at run-time. Before testing the resulting networks at run-time, however, we evaluate the networks by 1) determining the influence of each feature under different sensor types for estimating optimal capture-travel, and 2) determining the portability of a network by the average root mean squared error (defined in Equation 5.1) against the corresponding hold-out map.

Using the beam sensors, Figure 5.2 shows the resulting feature weight-strength. Each subfigure illustrates the feature correlations to a hidden neuron H1, H2, or H3 respectively. Each group of bars corresponds to the feature weight-strength from the network learned against the OPT, DAM, or SFL targets respectively. The connection weight between a feature and a hidden neuron is averaged over all five networks under the same target, whereas each bar shows the magnitude of such average weight-strength, which is automatically quantized between -26 and 26 by FANN.

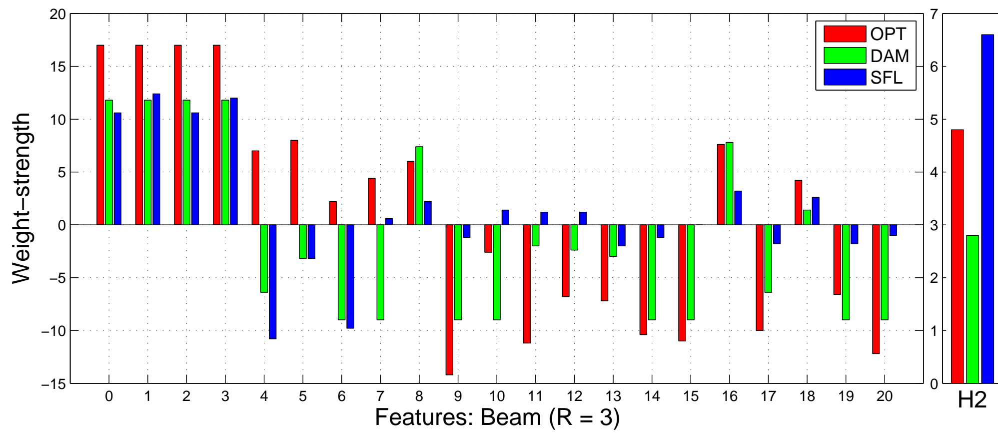
For learning to pursue the OPT target with two pursuers, H3 shows the strongest weight-

[1] RPROP is a weight update algorithm that is faster than the standard back-propagation algorithm [42] .

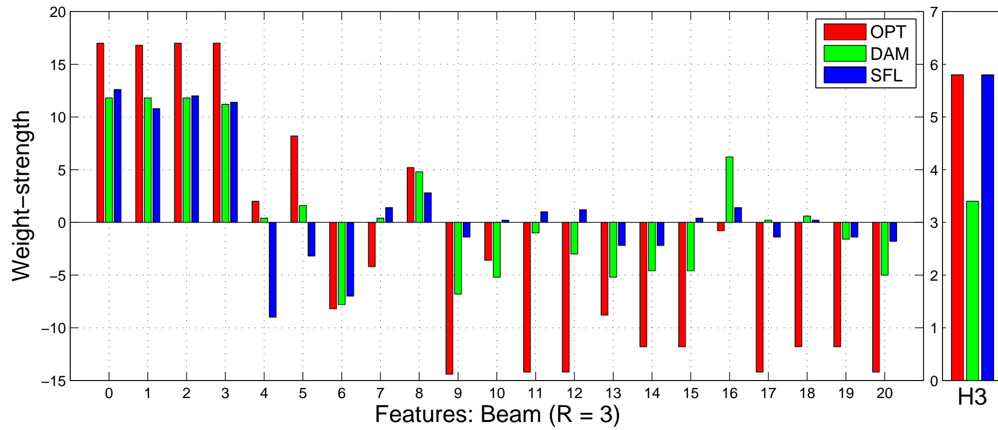
[2] We found no noticeable impacts on the RMSE as we varied the k in k-fold cross-validation.



(a)

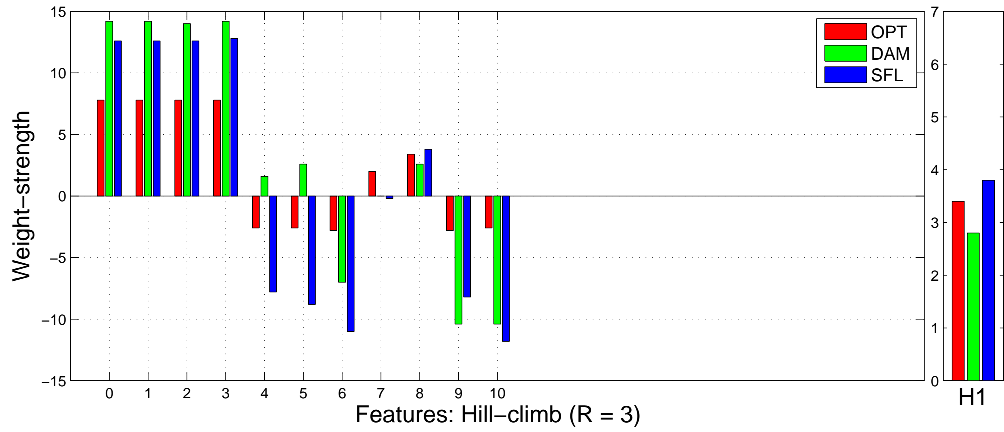


(b)

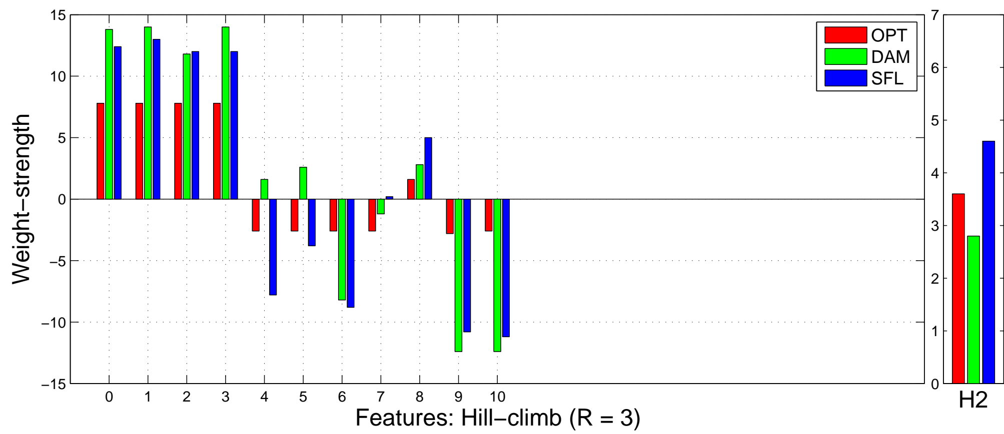


(c)

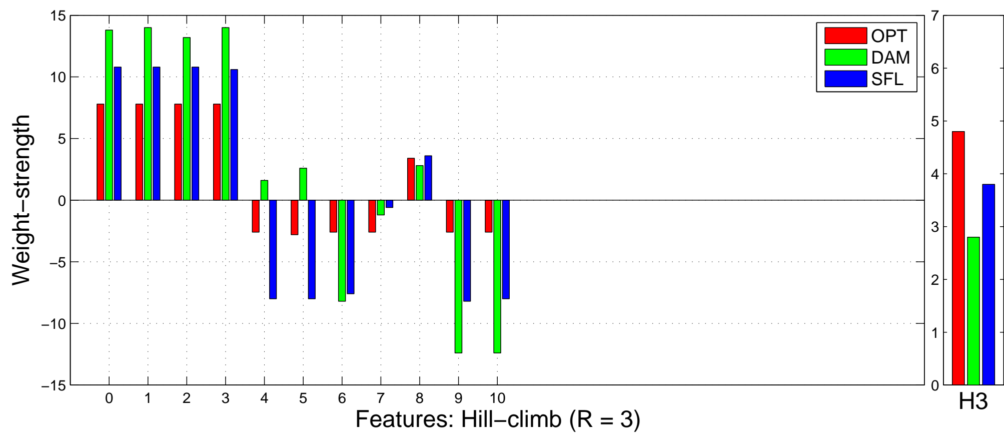
Figure 5.2: Feature weights with the beam sensor: in each subfigure, the chart on the left shows the weight-strength between each feature and a hidden neuron, whereas the much narrower chart on the right shows the weight-strength between the hidden neuron and the output neuron.



(a)



(b)



(c)

Figure 5.3: Feature weights with the hill-climb sensor.

strength to the output (see Figure 5.2(c)) although there is no great difference in weight-strength to the other two hidden neurons. According Table 5.2 and the goal of minimizing the capture-travel, we make the following observations:

- The features of distance-to-target (0 – 3) are most positively influential to minimizing the capture-travel, implying that short distance-to-target are favourable.
- The smaller positive strength on feature 4 and 5 implies that the separation between the pursuers should not be too great in the attempt of capturing the OPT target.
- However, the moderately negative strength of the difference in distance-to-target (Feature 6: δ) shows that the classifier prefers one pursuer to be closer to the target while the other is further away during a pursuit. It is surprising that this feature does not support equal distance to target for a pincer movement. Instead, it prefers one pursuer to be more aggressive in chasing the target while the other one holds back for a potential trap. While surprising, this result is in line with observed animal behaviors in cooperative hunting (see Section 1.1.1), where some group members herd their prey toward other members as a trap.
- In terms of mobility, the pursuers prefer a joint state that gives them more choices of actions (by a negative weight on Feature 7) but gives the target lower mobility (by a positive weight on Feature 8). These two weights can be related to the best capture scenario where both pursuers corner the target against an obstacle.
- The negative weight-strengths on the beam sensors (Features 9 to 20) are mostly consistent in preferring large, positive sensor data; and hence, these features prefer all agents in an open area by straight-line distances. This result makes sense on the pursuers, but not so on the target. Features 9 to 12 contradict with the target mobility (Feature 8), which could cause some undesirable behaviors that cause the performance to degrade during run time.

The feature correlation to H1 and H2 also exhibit similar characteristics in terms of pursuing the OPT target.

The trained networks on the suboptimal targets, DAM and SFL respectively, bear similarity to the networks trained against OPT in terms of the distance-to-target, δ, mobility, and sensor features' weights. They differ in the pursuer-separation features, such that the pursuers should keep greater separation from each other. However, such speculations cannot be verified until the run-time performance is shown.

Network	OPT		SFL		DAM	
	Bm	Hc	Bm	Hc	Bm	Hc
ann-Empty5	17.38	17.38	**19.74**	19.89	20.62	**20.59**
ann-Handmade5	**15.13**	18.08	**20.93**	21.97	**20.77**	20.79
ann-Round7	**31.87**	37.32	42.15	**41.00**	43.40	**41.96**
ann-Round9	**61.63**	61.64	**71.06**	71.43	**70.35**	72.22
ann-Sparse10	**91.91**	91.94	**90.91**	92.79	92.09	**91.73**

Table 5.4: RMSE comparison: Beam (Bm) versus Hill-climb (Hc), $R = 3$.

Compared to the feature correlations in Figure 5.2, Figure 5.3 shows more consistent correlation between the features and the hidden neurons in the networks using the hill-climb sensors, especially for both the OPT target and the SFL target. The most positively dominant features are still the distance-to-target (preferring small values), while the most negatively dominant features are the sensors (preferring larger values) because the sensor feature of a pursuer spikes if the target moves within the sensing range. The mobility features of both pursuers and the target also agree with the networks using beam sensors. Moreover, the pursuer separation and δ are promoted by the negative weight-strength when pursuing the OPT and SFL target. For the DAM target, however, separation seems less important. Again, studies in run-time performances will reveal the actual behaviors induced by the features.

As another preliminary study prior to the run-time evaluations, testing trained networks against instances from a previously unseen map can help us further understand the sensor data. The ANN-predicted capture-travel is evaluated by the root mean squared error (RMSE) (as in Equation 5.1) against the optimal capture-travel from the corresponding ORD. Given a set of joint states samples J_o, RMSE(J_o) is the average of the absolute difference between ORD(j) and the estimated ANN(j) for every joint state $j \in J_o$. The smaller the value of the RMSE(J_o), the more accurate the trained network is.

$$\text{RMSE}(J_o) = \sqrt{\frac{1}{\mid J_o \mid} \sum_{j \in J_o} [\text{ANN}(j) - \text{ORD}(j)]^2} \tag{5.1}$$

Due to the limited options for sensing range on small maps, we present the RMSE on networks with sensing range $r = 3$ for the pursuit of all three types of targets. Table 5.4 illustrates the RMSE when the networks are tested against their unseen maps. Within this myopic range, there is no obvious differences in using the beam sensor or the hill-climb sensor. However, beam sensor shows lower RMSE in 80% of the cases for pursuing the OPT target and the SFL target respectively. The results on DAM targets are mixed.

To determine whether the sensing range of the hill-climb sensor matters to the estimated capture-travel, Figure 5.4 shows the RMSE on capture-travel computed by the networks using the hill-climb sensor with the range of $R = 3, 5, 7, 10$ respectively. For the OPT target, the sensor range does not demonstrate any impact on the RMSE, except that the error is high overall. The lowest capture-travel error is about 14 steps by testing the neural net, *ann-Handmade5*, with the sensor range of 7. The resulting RMSE on the two largest maps, Sparse 10×10 and Round-table 9×9, produce the highest errors respectively. The order of the curves indicates that RMSE is positively proportional to the map size. Qualitatively, the tests against the DAM and the SFL target also show similar, consistent RMSE trends (see Figure B.1 and B.2). Nevertheless, it is undetermined at this stage whether there is direct correlation between RMSE and the run-time performance.

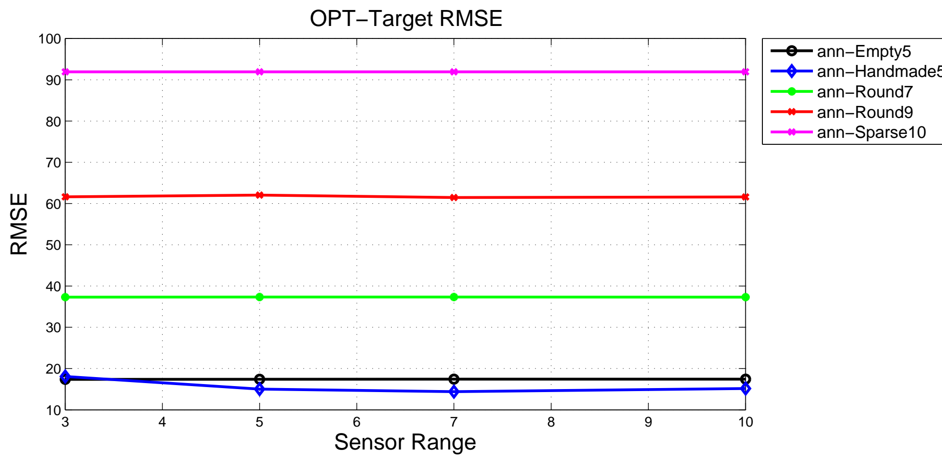


Figure 5.4: RMSE from the tests against the ANN generated with the hill-climb sensor.

5.5. Run-time Experiments

The main goal of this study is learning to pursue instead of determining an appropriate pursuit algorithm. In a run-time simulation, all agents have the same speed. For the ease of evaluation, we use speed of 1.0, as defined in Chapter 2. Each state allows the maximum four cardinal action and the *stay* action, which costs 1 time-step. The time-out of a simulation is set to be 10 times the maximum dimension of the grid map being tested on. For each combination of training maps and the type of target (OPT, DAM and SFL), the corresponding learned behavior model is tested on 100 instances on a new map.

The pursuers are driven by the following pursuit approaches:

- **MA-Minimax (d) with ANN**: An ANN is used as the evaluation function in MA-

Minimax with a depth of d plies.

- **MA-Minimax (d) with CML**: CML is an evaluation function that computes the cumulative Manhattan distance heuristic between the pursuers and the target. It is the sum of Manhattan distance between every pursuer and the target. This heuristic is used to provide a base-line performance as compared to using an ANN evaluator.

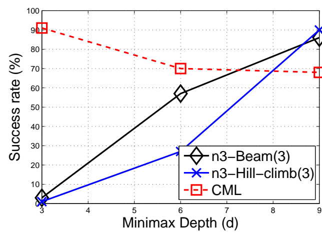
The lookahead depth ranges from 3 to 9, and range values r include 3, 5, 7, and 10. However, only the results on full-ply lookahead depths ($d = 3, 6, 9$) and $R = 3$ are presented in this section. The results showing the performance of fractional plies and more range values are arranged in Appendix B.3.

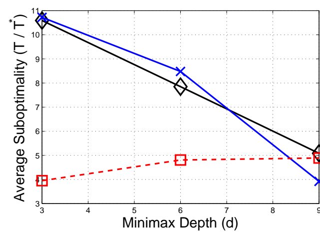
We first focus on the run-time results in Figure 5.5 for pursuing the OPT target. Each row of subfigures represents the three performance measures, namely success rate, average capture-travel, and average suboptimality, on the networks to be tested on the corresponding unseen map. Each subfigure illustrated three curves, which represent three different evaluation functions. The curve with a diamond shaped marker represents the performance of the model using three hidden neurons (n3) and the beam sensor with range of 3 (i.e., *beam model*), whereas the curve with a cross shaped marker represents the one using the hill-climb sensor under the same network architecture (i.e., *hill-climb model*). The curve with a square shaped marker shows the base-line performance of CML function. We refer to the models with the short-hand name *beam* and *hill-climb* when statistic significance are presented with the corresponding measures.

Our observation over the run-time results is statistically supported by *paired t-tests*. Each t-test is conducted over any two different evaluation functions. The *p-value* indicates the statistic significance of the relevant observation.

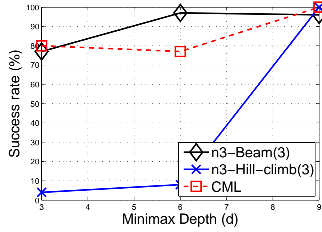
As illustrated in Figure 5.5(a), the performance of CML on the empty map surpasses both the beam model (with $p = 8.9 \times 10^{-67}$) and the hill-climb model (with $p = 7.9 \times 10^{-75}$) at $d = 3$ by a large margin. On empty maps (i.e., finite, trivial, four-connected graphs without obstacles), the Manhattan heuristic is optimal. CML with a myopic lookahead depth leads the pursuers to aggressively corner the OPT target without worrying about separation between each other. In contrast, both models show learning from more complicated maps can be misleading in the application on simply open terrains. They overly deliberate on the decision of separation on empty maps.

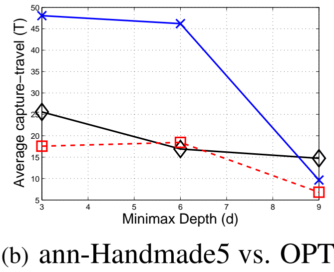
Interestingly enough, CML's success rate decreases as d increases. Deeper planning makes the CML pursuers more conservative in pursuing the smart OPT target, and therefore,

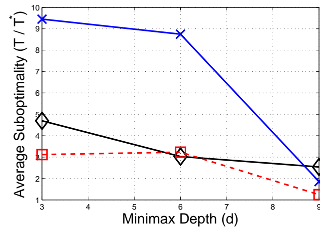




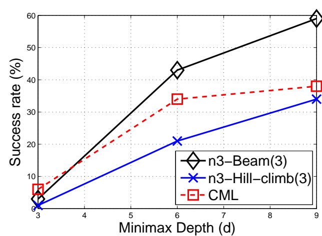
(a) ann-Empty5 vs. OPT

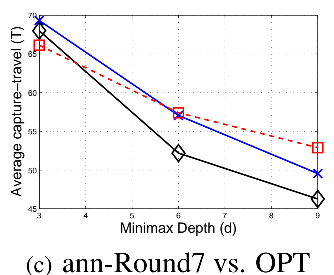


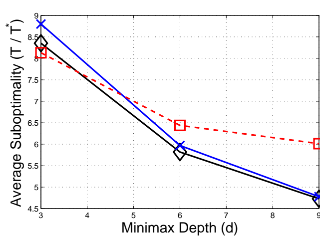




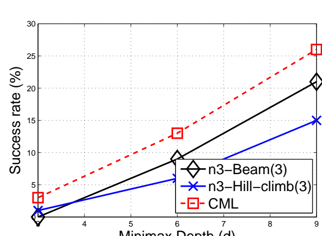
(b) ann-Handmade5 vs. OPT

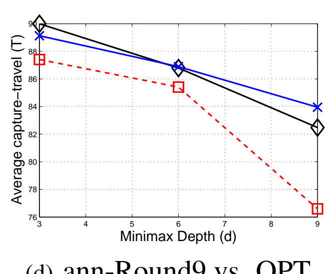


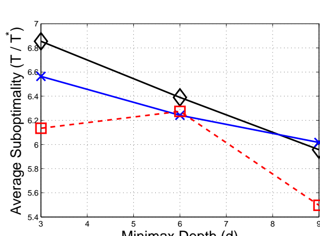




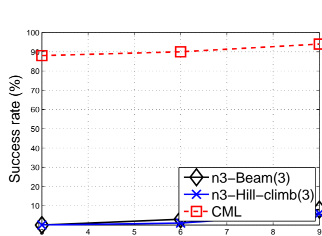
(c) ann-Round7 vs. OPT

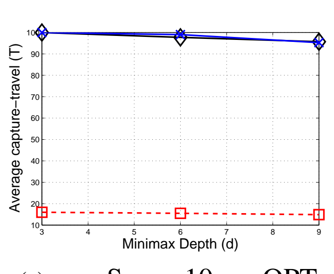






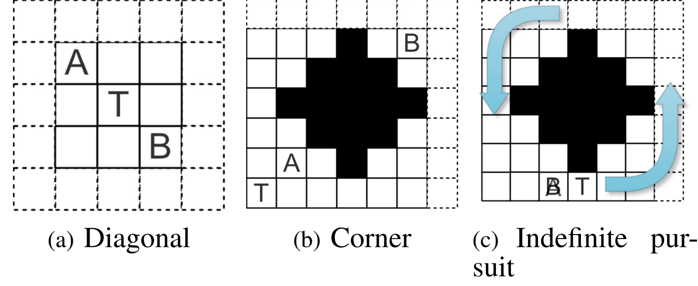
(d) ann-Round9 vs. OPT





(e) ann-Sparse10 vs. OPT

Figure 5.5: Run-time performance against the OPT target.



(a) Diagonal (b) Corner (c) Indefinite pursuit

Figure 5.6: Dead-lock scenarios occurred during run-time: the dotted lines mean that there could be more grid cells in the direction of these lines (i.e., the presented terrain can be a portion of a large map).

allowing the OPT target to lead the pursuers into the diagonal dead-lock scenario, as shown in Figure 5.6(a). Also with the increasing lookahead depth, both models catch up with CML and finally overtake the success rate at $d = 9$: beam $>$ CML with $p = 1.2 \times 10^{-3}$ and hill-climb $>$ CML with $p = 5.5 \times 10^{-5}$. In comparison, the beam model solves more cases than the hill-climb sensor at shallower depth ($p = 6.1 \times 10^{-6}$ at $d = 6$), while hill-climb crosses both CML ($p = 5.5 \times 10^{-5}$) and the beam model at $d = 9$. The results in terms of suboptimality and capture-travel show that the hill-climb sensor becomes more effective in solving more difficult cases as d increases; the suboptimality of the hill-climb model at $d = 9$ decreases to the same level CML at $d = 3$, while its average capture-travel at $d = 9$ is still greater than CML at $d = 3$.

The mobility of an agent is intrinsically limited by the characteristics of the map as seen in Hand-made 5×5 (Figure 5.1(b)). However, Manhattan heuristic is also optimal on this map, whereas separation is physically imposed upon the pursuers. These characteristics give pursuit advantages to CML. Therefore, CML shows superior performance overall as in Figure 5.5(b), except at $d = 6$; by the success rate,

- at $d = 3$, CML $>$ beam and CML $>$ hill-climb with $p = 8.7 \times 10^{-41}$, and
- at $d = 9$, CML $>$ beam while both CML and hill-climb have 100% success rate.

Depth 6 seems to be a pathological lookahead value for CML pursuers (CML $<$ beam with $p = 9.5 \times 10^{-6}$) because they can easily fall into the indefinite pursuit around the obstacle (see Figure 5.6(c)). The hill-climb model is also subject to the same dead-lock scenario (hill-climb $<$ CML with $p = 8.0 \times 10^{-31}$ at $d = 6$) because the sensor encourages individual pursuers to move in toward the target. On the other hand, the beam model solves just a few cases shy of CML at $d = 3$; and it surpasses CML at $d = 6$ with a success rate of 97%, which appears to be indicated by the lowest RMSE in Table 5.4. This observation

shows that that learning on other larger, more complicated maps with imperfect heuristics can help pursue the OPT target in a smaller, obstacle-filled environment.

The round-table scenarios are fundamental motivating examples in this study. The Manhattan heuristic is no longer perfect. Pursuers must find ways to propel away from each other so that they can corner the target together from opposing sides of the table. In Figure 5.5(c), the success rates of all three types of pursuers drop drastically at $d = 3$, showing that learning cannot combat the shallow lookahead in a difficult map. The beam model declares a clear victory from depth at and beyond $d = 6$ by the success rate,

- beam $>$ CML , and
- beam $>$ hill-climb with $p = 3.9 \times 10^{-4}$,

but the success rate cannot exceed 60% even at $d = 9$. CML is subject to all three dead-lock cases shown in Figure 5.6 without the assistance of true distance-heuristics. The beam model is able to avoid the diagonal case and the case of indefinite pursuit because learning encourages separation between the pursuers alone as well as the difference in distance-to-target. However, the corner case is the most difficult dead-lock to overcome. In Figure 5.6(b), pursuer A is able to hold the target at the corner. Pursuer B, positioned on the diagonally opposite side of the table, cannot escape the depression in the joint space without A or T changing their current state. The hill-climb sensor shows consistent conservativeness in separation; clumping pursuers give a greater chance for the target to run free.

While learning on Round-table 9×9 and Sparse 10×10 can improve the solution of smaller problems, as in Round-table 7×7, it is less scalable to larger problems in applying learned behaviors from Round-table 7×7 in Round-table 9×9. The beam model can only solve around 20% of the cases, while the hill-climb model can solve only 15% (see Figure 5.5(d)). The winner, CML, is also only able solve at most 26% of cases (at $d = 9$, CML $>$ beam and CML $>$ hill-climb with $p = 2.7 \times 10^{-2}$) most of which are likely simpler cases as indicated by the high values of suboptimality.

The scalability is even poorer as in Sparse 10×10, which contains multiple small heuristic depressions (see Figure 5.5(e)). Although this map is largest in our study, the obstacle topologies and the large open areas are not similar to those on the smaller maps that were used in training. It is difficult to test the model on an instance of which it have no experience. CML performs unexpectedly well in both success rate and suboptimality, for example, at $d = 9$:

- by success rate, CML $>$ beam with $p = 4.1 \times 10^{-60}$ and CML $>$ hill-climb with $p = 3.1 \times 10^{-66}$;
- by suboptimality, CML $<$ beam with $p = 1.6 \times 10^{-21}$ and CML $<$ hill-climb with $p = 4.9 \times 10^{-20}$.

The pursuit in open areas resembles the behavior on the empty map.

Either the beam model or the hill-climb model on pursuing the DAM target shows dominating success rates of at least 90% over CML (see Figure B.3 with p-values), except for the trivial map, Hand-made 5×5. The main reason for this is that DAM is a surrendering target. When it is pushed against at least one edge while a pursuer in its immediate neighbourhood, a capture is guaranteed. In fact, learning shows that one greedy pursuer and one laid-back pursuer (i.e., further away from the target) can help minimize the optimal capture-travel. The greedy pursuer can herd the target into a corner, which the laid-back pursuer provides an extra barrier to limit the mobility of the target. It is also noteworthy that the run-time performance against the DAM target is also indicated by the corresponding RMSE entry in Table 5.4. As both sensors show mixed RMSE results on different maps, such that one has lower RMSE value than the other on certain maps, but not all, all three performance measures in Figure B.3 show similar mixed results at run-time.

The SFL target is much easier to capture due to self-defeating behavior. in particular, it often plans an "escape" path toward a selected beacon through the interception course of the pursuers. Therefore, pursuers that uses a greedy evaluation like CML can easily succeed in capturing the target on simple small maps without learning (see Figure B.4(a) and B.4(b) with p-values). On more difficult maps as in Figure B.4(c) to B.4(e), the results support the hill-climb model over the beam model because the hill-climb sensor can draw a pursuer toward the target, while the target may meet the pursuer half way.

In terms of planning effort per time-step, the dominating component lies in the expansion of the search tree, as described in Section 4.3. Since all three types of pursuers use the same planning algorithm, the only difference in planning effort is the process of extracting the features. CML uses a scalar heuristic, whose computation is in small constant time. Similarly, the distance features in a model can be computed in small constant time, whereas the computation of the mobility features and the sensor data are linear to the local search space. With a prespecified range, both mobility and sensor data can be obtain in constant time as well. Thus, the use of these models still satisfies the real-time property: planning effort per time-step is a small constant.

Overall, learning can help improve performance in pursuing an intelligent target in a terrain that resembles a previously experienced one. In particular, beam sensor demonstrates better combined ability in pursuer separation and in cornering the target. On the other hand, there are several scenarios in which a greedy approach can directly corner the target with shallow planning depth (i.e., without too much "thinking"). First, trivial, open-area maps with an accurate initial heuristic can provide easy-to-compute, sufficient heuristic to guide the pursuit. Secondly, if the pursuers cannot recognize the obstacles on a new map, there is a higher chance of successful capture by being a risk-taker than an uncertain thinker. Also, if the target's escape strategy is highly suboptimal, the pursuers can simply apply the distance heuristics instead of over-thinking the pursuit of a trivial target. In terms of scalability, it is still inconclusive on what causes the inability for the models to scale up to larger problems.

In hindsight of the run-time evaluation on the learned models, there exists a correlation between RMSE (see Table 5.4) and the run-time performance measures in the pursuit games of chasing a less optimal target (see Figure B.6 and B.7). A model with a lower RMSE shows higher success rate in solving the pursuit instances, and hence, reducing the average capture-travel. Also, a smaller RMSE value implies the higher accuracy of the learned model. Therefore, more difficult cases can be solved while reducing the overall suboptimality.

Chapter 6

Limitations and Potential Improvements

In this study, we examine the multi-agent pursuit game from a microscopic view on small toy maps. Out intent is to gain a better understanding of how cooperation can be achieved through two pursuers. Any additional agent can be thought of as an augmentation from the discussed approach. There are still many limitations to overcome and improvements to be done for this specific problem.

6.1. Limitations

There are limitations in both training phase and the run-time phase. In the training phase, they are:

- The computation for the ORD is subject to the "curse of dimensionality". Therefore, our study is restricted in learning from small maps, which in turn, limits certain complicated terrain representation which could be useful for learning to pursue.
- The off-line learning is currently using one machine learning approach. We can also try support vector machines (if time permits), and various reinforcement learning approaches that are capable of coping with uncertainty. The precondition on selecting these methods is that the result model must be portable to run-time simulations. Particularly, fast execution of the model is preferred in real-time applications.
- Although the current feature sets exhibit some advantage of utilizing learning in the pursuit games and they are easy to compute, it is an on-going investigation into discovering features with more sophisticated correlations.

- Considering the feature correlations as in both Figure 5.2 and 5.3, the feature weights follow similar connection distribution toward each hidden neuron, independent of the type of the target. Although the hidden neurons are activated by the sigmoid function, this function can be seen as a squashing function that translate each input value between zero and one. Also, using a one single hidden layer, the similar weight strength on the connections between a hidden neuron and the output neuron indicates that model is linear in the feature space. Using deep nets can produce a nonlinear function to approximate the optimal capture-travel. However, it is not yet known whether the use of deep nets can improve pursuit performance.

The limitation in the run-time phase lies in the choice of planning algorithm. Comparing the learned models with simple heuristics inside the same minimax algorithm is subject to exponential expansion in the search tree. Regarding to real-time applications, a desired method must strike a balance between the quality of the pursuit and the planning effort. We can try utilizing the models in different multi-agent real-time search approaches, or a combination of these approaches. However, the literature on this specific topic is also limited to date.

6.2. Potential Improvements

To echo the motivation of the real-time property in Section 2.2 with respect to the assumption on fully observable problems, it will be interesting to consider the pursuit games in an asynchronous environment, which is also partially observable. However, before we can move onto such future studies, several potential improvements can be made to our current approach, ranging from the choice of maps, to feature selection, to training approaches, and to the run-time approach.

Learning over the five toy maps is insufficient to conclude that a learned model will scale. One improvement can be training a model over a few more maps of different sizes, but with similar shapes of obstacle, so that a familiar obstacle can be recognized on a new map; the pursuit behaviors from this new map can be derived from the known maps. Alternatively, we can take advantage of state abstraction and compute the ORD on each abstraction level as each abstract graph is a reduced version of its child graph.

Furthermore, reinforcement learning (RL) is an alternative for building pursuit models over large maps, in which ORD becomes intractable to compute. Especially, large maps that contain a wide variations of terrains can provide a rich set of examples to the learner.

However, the premise of applying RL on the pursuit game is that time is not an issue.

With the same assumptions about the map, the agent and their movement as in our approach, it will be interesting to use learning to approximate the value of the cover heuristic (see Section 3.4). However, such a study is non-trivial as the shape of the cover can be completely different within one step. There exist technologies that can help decompose the state space to make reinforcement learning more effective. Graph Laplacian is an example of such technology, as it uses Laplacian matrix to represent spanning trees on a graph [30]. However, making it into a feature that is independent to the map size remains an open problem.

Until the development of a better suited features set for the pursuit game, it is difficult to determine the appropriate training method for this particular problem. The number of parameters for turning an ANN is already an overhead. We can use the cascade-correlation [11] approach to automatically determine the best net work architectures instead of using the simple, hand-tuned neural network architecture, The advantage of using cascade networks is that the network has more freedom in choosing what it needs for learning the instances. The disadvantage, however, lies in the difficulty in analyzing the effect of each features by their weights, and the tendency of overfitting.

During run-time simulations, we can incorporate online-learning to fine-tune the behavior model for the current map to determine whether the current performance can be further improved. On another token, we can augment this two-pursuer approach to more pursuers. Similar to the idea of state abstraction, one can designed a binary command hierarchy consisting of abstract units for making decisions and the ground units for executing the commanders' decisions.

Chapter 7

Conclusion

The multi-agent pursuit game often arises in real life, ranging from animal and human behaviors to computer video games. In all three real-life domains, there exist a common process of learning from others' experience prior to the one's own real practice. In computer simulations, however, learning off-line is often difficult to achieve due to the computation overhead on the training instances, the construction of portable feature sets, and the challenge of finding of an appropriate training method.

This thesis presented a survey on various related work to the classical single-agent moving target search approaches, which incrementally learn about the environment to help improve performance online. These approaches are subject to "loss of information" and the "thrashing" behavior due to the restriction of running in real-time. When they are applied in multi-agent scenarios, the pursuers become uncoordinated as they attempt to capture the moving target. Furthermore, the learned information depends on the structure of the current map only, neglecting the existence of any teammates or any previously known pursuit behaviors in similar environments.

To make learning to pursue possible, we proposed an off-line learning framework to gather pursuit instances from different maps and to model a general pursuit behavior across these maps. To do so, we precomputed the optimal response database for different targets, and defined two different sets of features containing relative distance metric about all agents, local mobility evaluation, and local sensor data. These two sets differ in the approach of collect sensor data (i.e., the beam sensor and the hill-climb sensor).

Preliminary training results in terms of root mean squared error to the optimal capture-travel showed that beam sensor slightly outperformed hill-climb sensor in the pursuit of the optimal target due to the moderate level of separation between the pursuers. The same results also showed that varying the sensing range do not have any impact on the accuracy

of estimating the capture-travel.

The application of the learned models in run-time are driven by the multi-agent minimax algorithm, which puts planning for each agent on a separate ply. Due to the exponential expansion of the minimax search tree, the run-time testing beyond 9 steps ahead becomes intractable. Therefore, we examined the learned behaviors on the full-ply of 3, 6, and 9. The empirical results showed that learning is required to help improve performance on non-trivial maps (e.g., round-tables) when pursuing an intelligent target. These results also reveal that learning is unnecessary when the map is trivial or the target is suboptimal.

Moreover, these empirical results demonstrated correlation between the root mean squared error and the run-time performances. The greater the error, the less successful a model is in capturing the target. This error also correlates to the map size, showing that the model applied on smaller maps is more efficient, in terms of success rate, than applying on larger maps. Such correlation indicates the poor scalability of the current learning framework. Future work is necessary in improving the feature definition, and well as the training process.

Appendix A

Minimax

Minimax (joint_state j, depth d):

1: **if** j is a capture-state or $d = 0$ **then**
2:     **return** $f(j)$
3: **end if**
4: $v \leftarrow -\infty$
5: **for** each successor joint state j' **do**
6:     $v \leftarrow \max\{v, -$**Minimax**$(j', d-1)\ \}$
7: **end for**
8: **return** v

Figure A.1: The Minimax Algorithm.

AlphaBeta (joint_state j, depth d, α, β)

1: **if** j is a capture-state or $d = 0$ **then**
2:     **return** $f(j)$ {** f is an arbitrary evaluation function **}
3: **end if**
4: **for** each successor joint state j' **do**
5:     $\alpha \leftarrow$ max(α, -**AlphaBeta**(j', $d-1$, $-\beta$, $-\alpha$))
6:     **if** ($\beta \leq \alpha$) **then** {** Beta cut-off **}
7:         **break**
8:     **end if**
9: **end for**
10: **return** α

Figure A.2: Minimax with alpha-beta pruning: β represents target (or opponent in general) best choice of action. The subtree is discarded if α is worse than β.

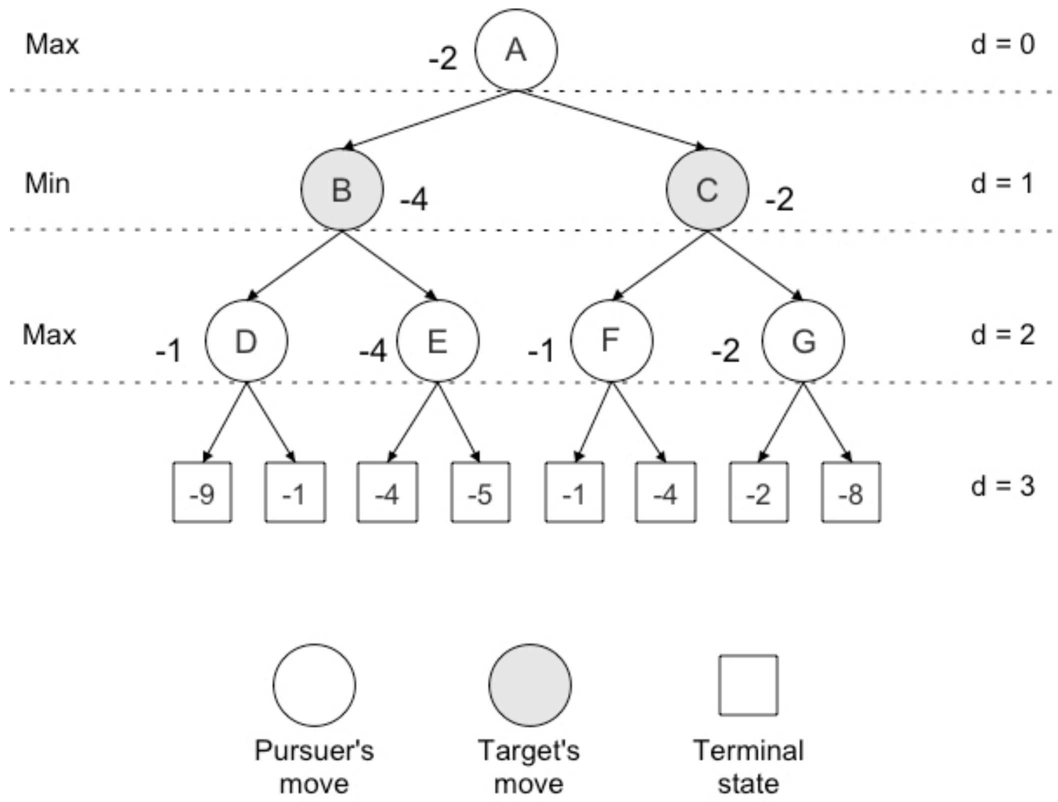


Figure A.3: An example of a minimax game tree for maximum depth $d = 3$ in the context of the pursuit game. A round state on the tree represent a joint state. The pursuers make a joint decision on a Max-ply, and predict the target's move on a Min-ply. By convention, the combination of a Max-ply and a subsequent Min-ply is considered as a full ply.

Appendix B

Empirical Results on Non-Optimal Targets

B.1. Training Results

As mentioned in Section 5.4.3, RMSE is dependent to the map size but independent to the sensor range. Here, this claim is reinforced by the tests against non-optimal targets. Both type of sensors yield to similar RMSE value as the sensor range increases.

B.2. Run-Time Experimental Results on Full-Plies

Figure B.3 and Figure B.4 demonstrate the run-time performance of the DAM target and then SFL target respectively. For more detailed discussion on these two figures, please refer to Section 5.5.

B.3. Run-Time Experimental Results Showing Fractional Plies

In this section, we present the run time performance with $d = 3, 4, \ldots, 9$, including fractional plies. Also, we include the performance multiple sensing ranges at $R =$ 3, 5, 7, 10 for the hill-climb sensor. This result can help us further understand the effect of sensing range and the fractional ply lookahead during run-time.

For the OPT target, neither longer sensing range for hill-climb sensor or the fractional ply changes previous observation in Section 5.5 on every map, except for Hand-made 5×5. On this map, however, increasing the range for the hill-climb sensor improves pursuit performance. In Figure B.5(b), both models at $R = 10$ and $R = 7$ can achieve an success rate about 97% to 98%. Although there exist pathology at $d = 4$ and 5, the model at $R = 7$ recovers 100% success rate at $d = 6$, while the one at $R = 10$ reaches 100% at $d = 7$ with

a smoother curve. on this simple map, these two models demonstrate high competence against CML.

For the DAM target, the observation on the two 5×5 maps still does not vary. However, the hill-climb models at $R = 5$ supersedes CML on Round-table 7×7; and at $R = 5$ and $R = 7$, the hill-climb models join the beam model in the superior league of pursuit game solver on Round-table 9×9. On Sparse 10×10, the hill-climb model at $R = 5$ with $d = 4$ exhibits the best performance among all models. The favourable fractional ply of 4 indicates that allowing one pursuer to plan an extra move to the target that is one time-step out dated constitutes a team of greedy and laid-back pursuers, which seems to be the best combination behavior in chasing the DAM target as mentioned in Section 5.5.

For the SFL target, the run-time performance shows that every hill-climb model is an efficient candidate in pursuing the SFL target, except for the two 5×5 maps. Consistently for the SFL target and DAM target, $d = 4$ appears to be the magic ply for the pursuers because at depth 4, the pursuers are greedy toward the target in general, while the one pursuer planning an extra move has the ability of changing the joint state continuously. This behavior makes the other pursuer re-plan an action toward the target as well.

In summary, sensing range with the combination of planning with fractional plies on separate non-trivial maps do show an improvement in performance. Especially, hill-climb models with range of at least 5 generally demonstrate highly efficient performance in terms of success rate and suboptimality. All hill-climb models are in general better suited for pursuing less optimal targets.

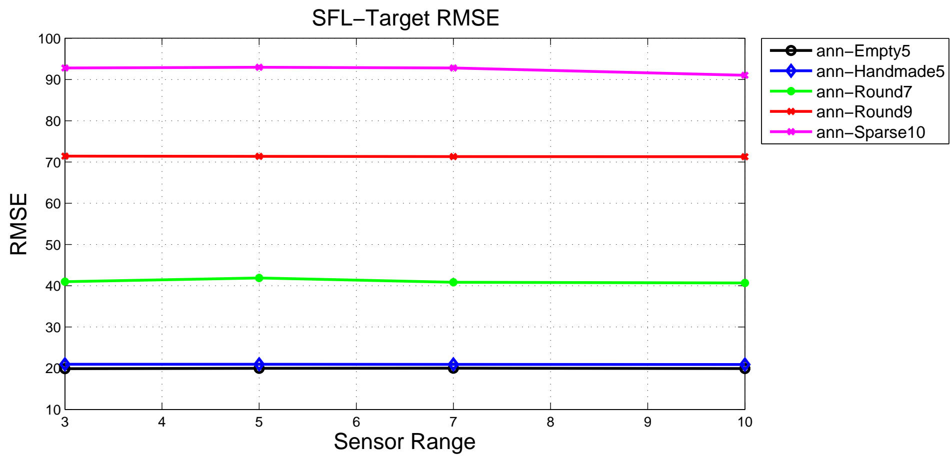


Figure B.1: RMSE vs. the hill-climb sensor on SFL target.

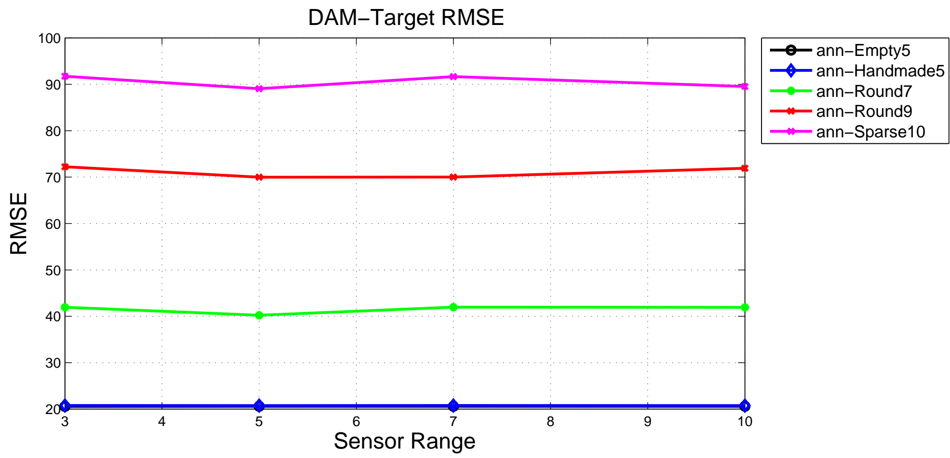
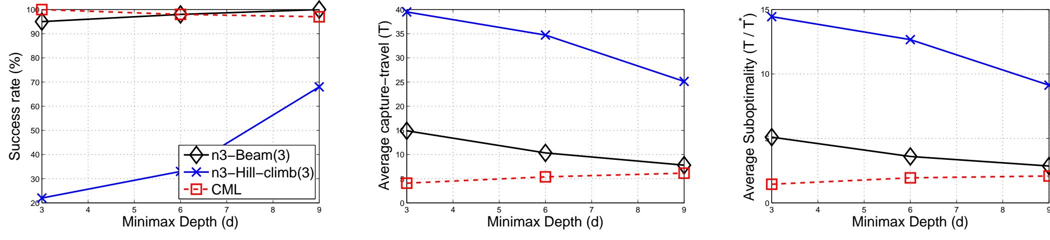
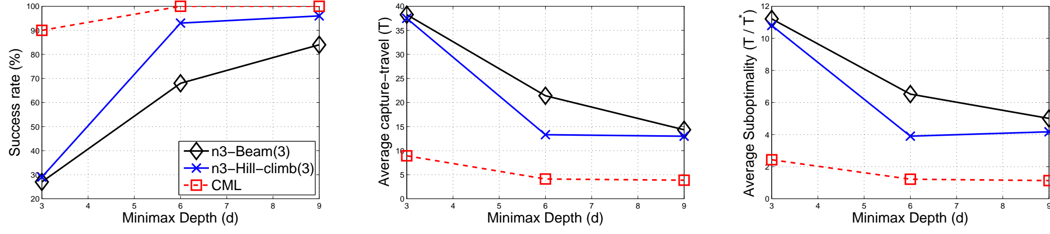


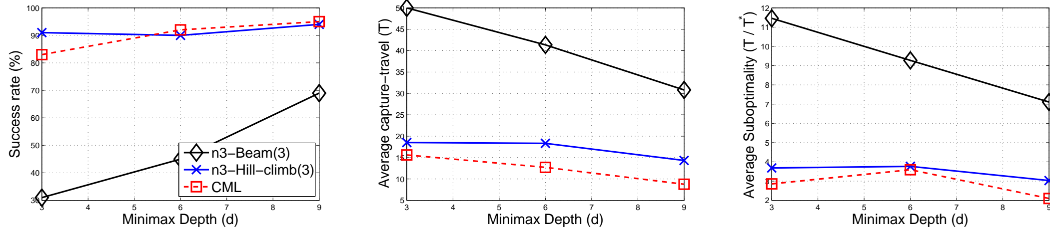
Figure B.2: RMSE vs. the hill-climb sensor on DAM target.



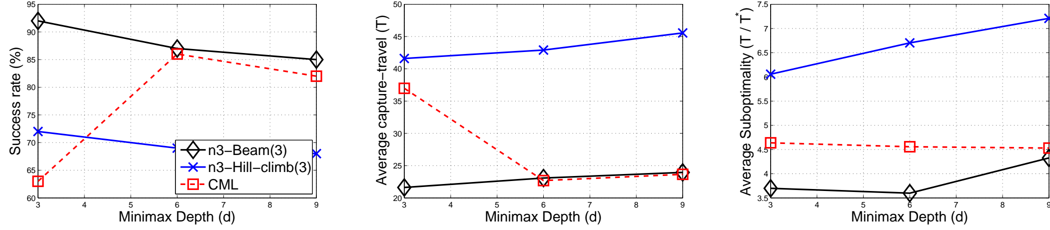
(a) Empty5: CML > beam ($p = 1.2 \times 10^{-2}$) and CML > hill-climb ($p = 5.0 \times 10^{-46}$)



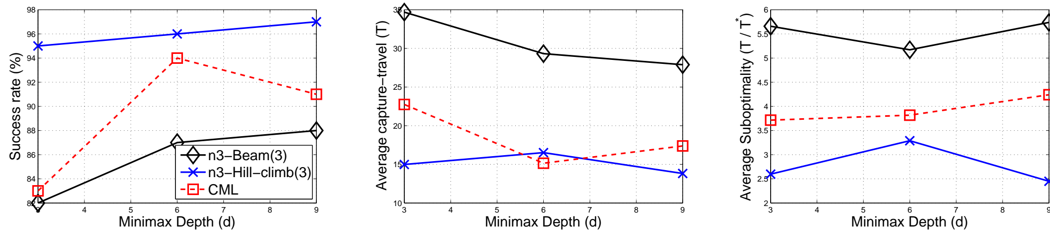
(b) Handmade5: CML > hill-climb ($p = 4.8 \times 10^{-23}$) and CML > beam ($p = 1.1 \times 10^{-24}$)



(c) Round7: hill-climb > CML ($p = 4.7 \times 10^{-2}$) and hill-climb > beam ($p = 1.7 \times 10^{-22}$)

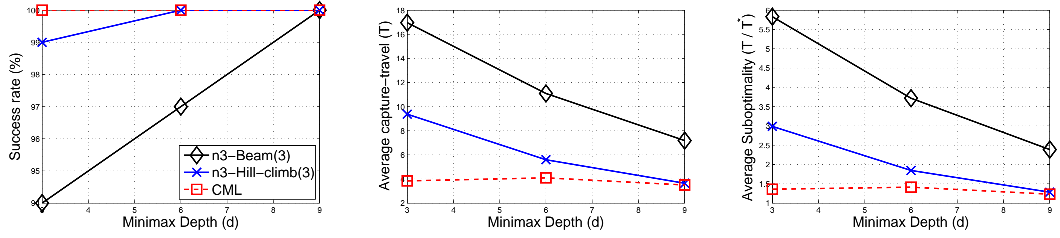


(d) Round9: beam > hill-climb ($p = 9.9 \times 10^{-5}$) and beam > CML ($p = 2.4 \times 10^{-7}$)

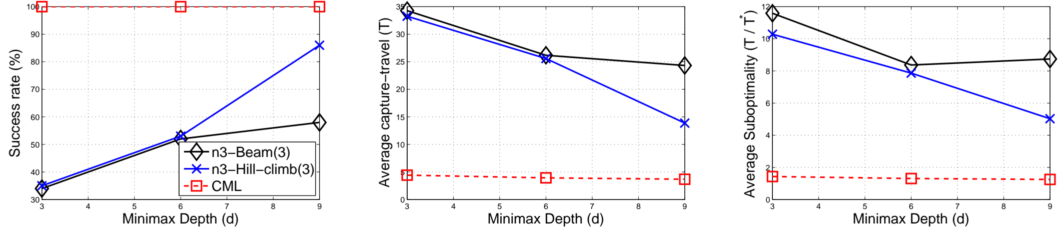


(e) Sparse10: hill-climb > CML ($p = 3.3 \times 10^{-3}$) and hill-climb > beam ($p = 1.9 \times 10^{-3}$)

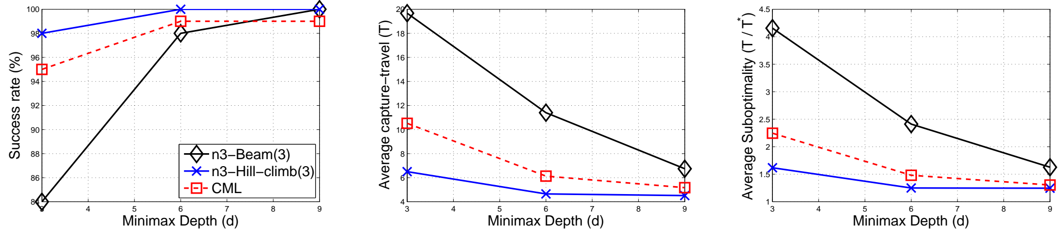
Figure B.3: Run-time performance against the DAM target. P-values under each subfigure represent the statistic significance on paired comparison over the success rates at $d = 3$.



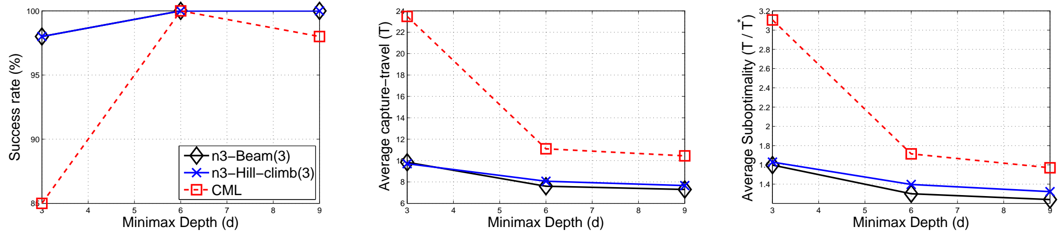
(a) Empty5: CML > hill-climb ($p = 1.6 \times 10^{-1}$) and CML > beam ($p = 6.4 \times 10^{-3}$)



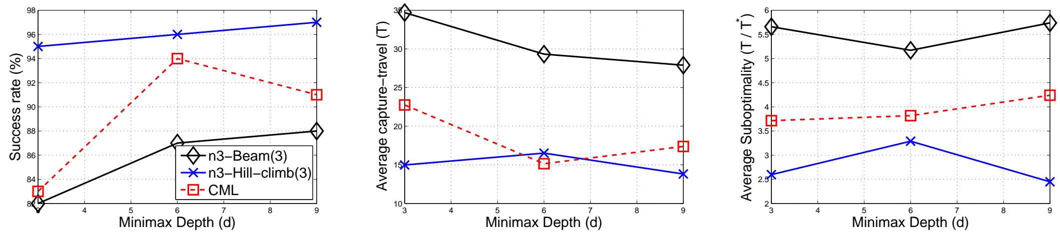
(b) Handmade5: CML > hill-climb ($p = 2.3 \times 10^{-30}$) and CML > beam ($p = 2.7 \times 10^{-31}$)



(c) Round7: hill-climb > CML ($p = 1.3 \times 10^{-1}$) and hill-climb > beam ($p = 2.4 \times 10^{-4}$)

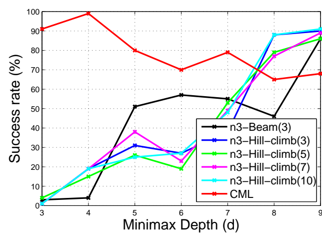


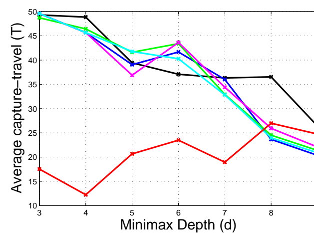
(d) Round9: for suboptimality at $d = 9$, beam < hill-climb ($p = 4.4 \times 10^{-1}$) and beam < CML ($p = 1.8 \times 10^{-6}$)

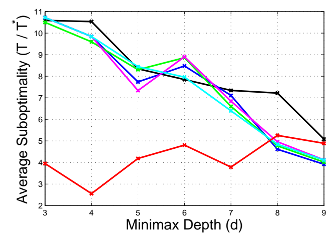


(e) Sparse10: hill-climb > CML ($p = 3.3 \times 10^{-1}$) and hill-climb > beam ($p = 7.2 \times 10^{-1}$)

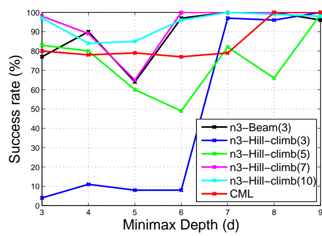
Figure B.4: Run-time performance against the SFL target. P-values under each subfigure represent the statistic significance on paired comparison over the success rates at $d = 3$, except for Round9.

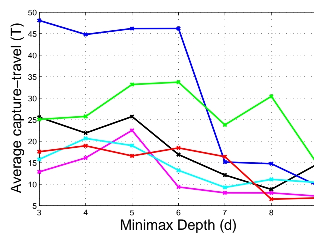


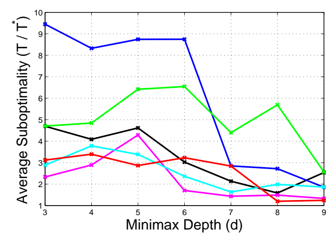




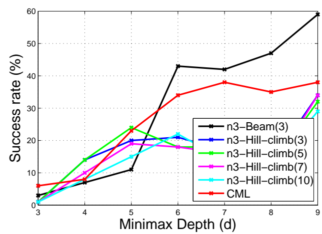
(a) ann-Empty5 vs. OPT

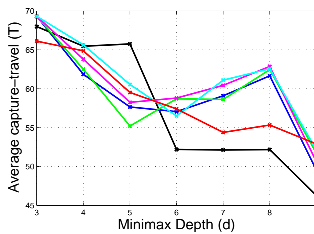


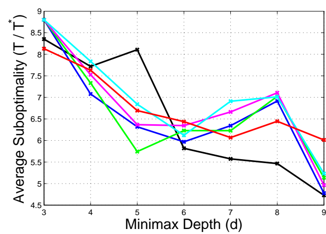




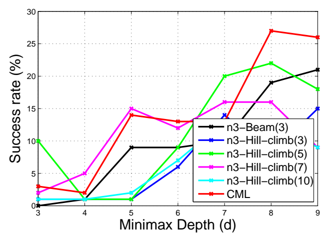
(b) ann-Handmade5 vs. OPT

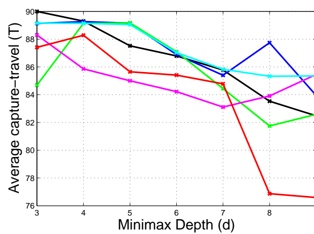


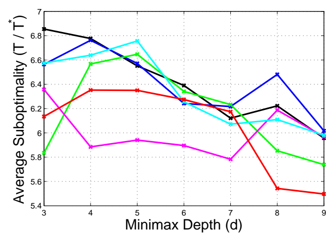




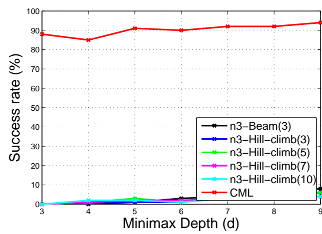
(c) ann-Round7 vs. OPT

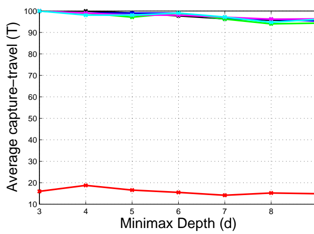


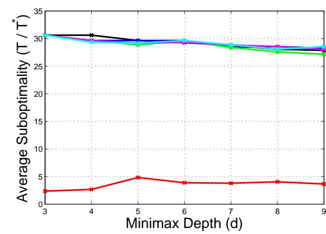




(d) ann-Round9 vs. OPT

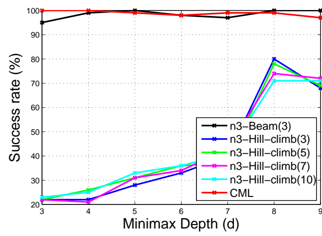


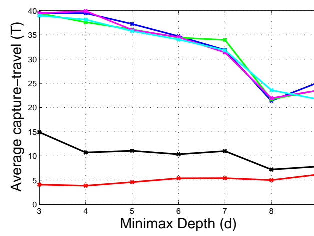


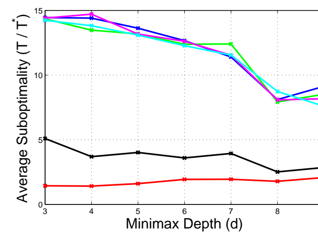


(e) ann-Sparse10 vs. OPT

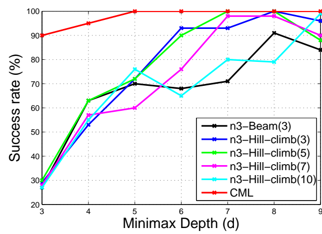
Figure B.5: Run-time experiment against the OPT target with multiple hill-climb sensor ranges.

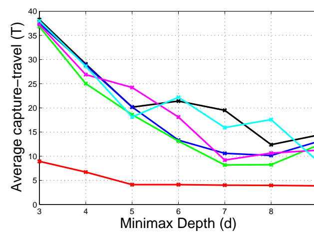


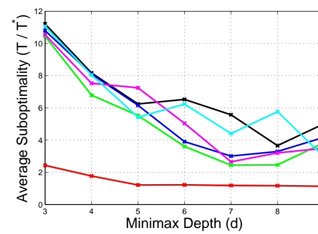




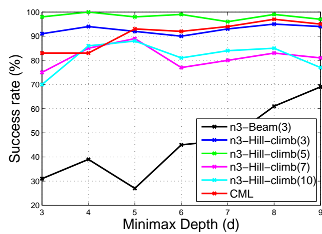
(a) ann-Empty5 vs. DAM

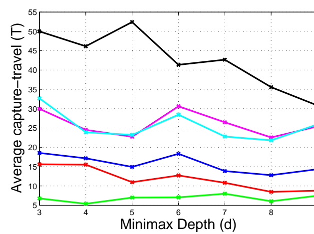


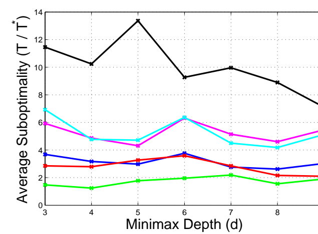




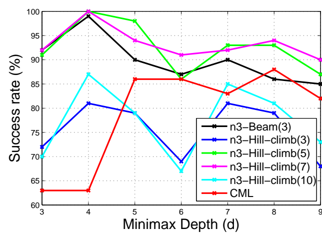
(b) ann-Handmade5 vs. DAM

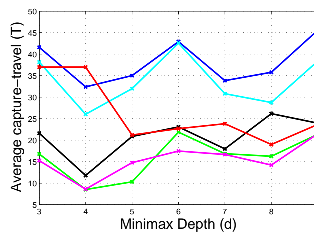


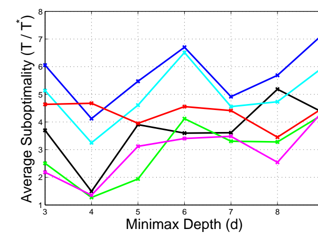




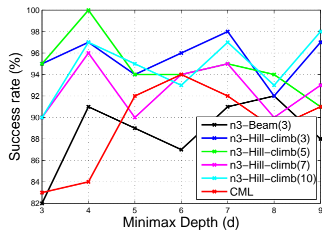
(c) ann-Round7 vs. DAM

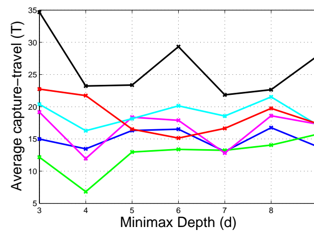


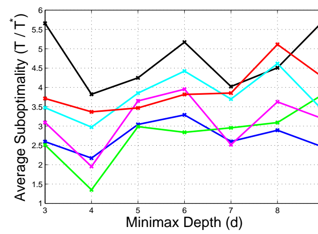




(d) ann-Round9 vs. DAM

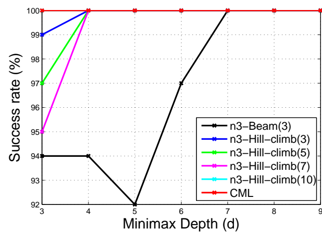


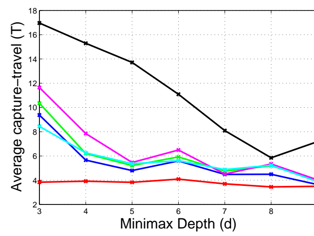


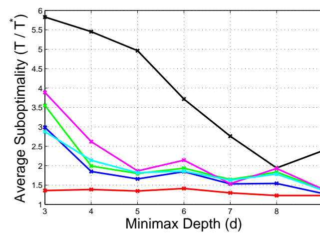


(e) ann-Sparse10 vs. DAM

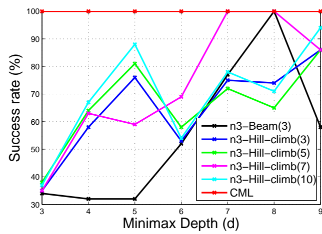
Figure B.6: Run-time experiment against the DAM target with multiple hill-climb sensor ranges.

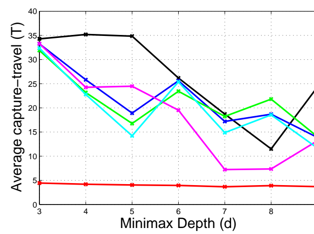


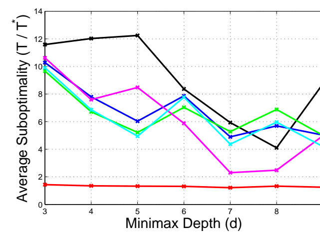




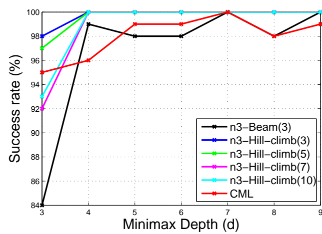
(a) ann-Empty5 vs. SFL

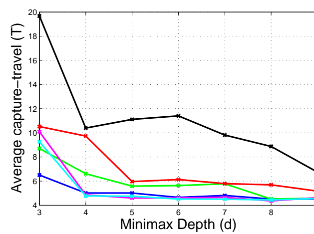


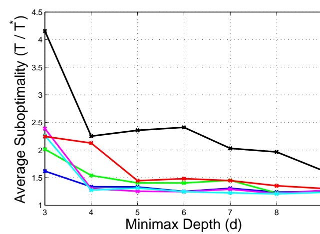




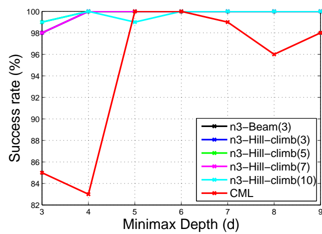
(b) ann-Handmade5 vs. SFL

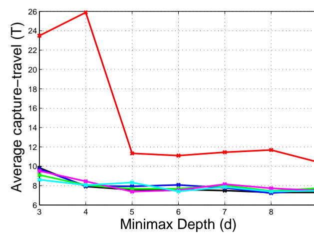


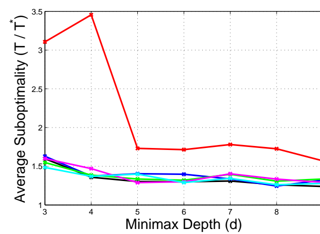




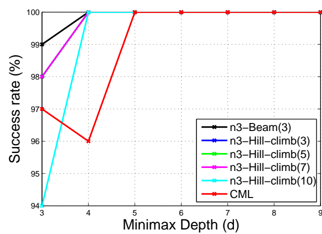
(c) ann-Round7 vs. SFL

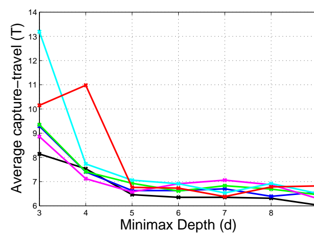


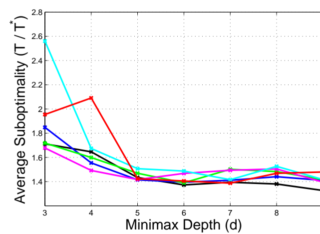




(d) ann-Round9 vs. SFL







(e) ann-Sparse10 vs. SFL

Figure B.7: Run-time experiment against the SFL target with multiple hill-climb sensor ranges.

Bibliography

[1] Martin Aigner and Michael Fromme. A game of cops and robbers. *Discrete Applied Mathematics*, 8(1):1–12, 1984.

[2] United States Army. *US Army Ranger Handbook (ebook)*, chapter 5-14. United States Army Infantry School, Fort Benning, Georgia, 2000. `http://www.scribd.com/doc/100897/ebook-US-Army-Ranger-Handbook`.

[3] Blizzard Entertainment. StarCraft II, 2009. `http://www.starcraft2.com`.

[4] Anthony Bonato, Gena Hahn, Peter Golovach, and Jan Kratochvil. The search-time of a graph. *Mathematics Subject Classification*, 1991.

[5] Vadim Bulitko and Greg Lee. Learning in real time search: A unifying framework. *Journal of Artificial Intelligence Research (JAIR)*, 25:119 – 157, 2006.

[6] Vadim Bulitko and Nathan Sturtevant. State abstraction for real-time moving target pursuit: A pilot study. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, pages 72–79, Boston, Massachusetts, 2006.

[7] Vadim Bulitko, Nathan Sturtevant, Jieshan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 30:51 – 100, 2007.

[8] David Cooper and John Kagel. Learning and transfer in signaling games. *Economic Theory*, 34(3):415–439, March 2008.

[9] Scott Creel and Nancy Marusha Creel. *The African Wild Dog: Behavior, Ecology, and Conservation*. Princeton University Press, Princeton, New Jersey, USA, 2002.

[10] Paul N. Edwards. *The Closed World*, chapter 2, pages 60–73. The MIT Press, Massachusetts, 1997.

[11] Scott Fahlman and Christian Lebiere. The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems*, 1990.

[12] Gas Powered Games. Supreme Commanders, 2007. `http://www.supremecommander.com`.

[13] GlobalSecurity.org. Encirclement operations. `http://www.globalsecurity.org/military/library/policy/army/fm/3-90/appd.htm`.

[14] Mark Goldenberg, Alex Kovarksy, Xiaomeng Wu, and Jonathan Schaeffer. Multiple agents moving target search. *International Joint Conference on Artificial Intelligence (IJCAI) - Poster*, pages 1538–1538, 2003.

[15] Arthur S. Goldstein and Edward M. Reingold. The complexity of pursuit on a graph. *Theoretical Computer Science*, 143(1):93–112, 1995.

[16] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths". *IEEE Transactions on Systems Science and Cybernetics*, ssc-4(2):100–107, 1968.

[17] IGN Entertainment. GameSpy, 2004. `http://www.gamespy.com/`.

[18] Alejandro Isaza. A heuristic-based approach to multi-agent moving-target search. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, 2008.

[19] Alejandro Isaza, Jieshan Lu, Vadim Bulitko, and Russell Greiner. A cover-based approach to multi-agent moving target pursuit. In *The Fourth Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Stanford, California, 2008. AAAI Press, AAAI Press.

[20] T. Ishida. Moving target search with intelligence. In *Proceedings of the National Conference on Artificial Intelligence*, pages 525–532, 1992.

[21] T. Ishida. A moving target search algorithm and its performance improvement. *Journal of Japanese Society for Artificial Intelligence*, 8(6):760–768, 1993.

[22] Toru Ishida and Richard E. Korf. Moving target search. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210, 1991.

[23] Renee Jansen and Nathan Sturtevant. Direction maps for cooperative pathfinding. In *The Fourth Arftificial Intelligence and Interactive digital entertainment conference (AIIDE poster)*, 2008.

[24] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Artificial Intelligence*, pages 72–85. Springer Verlag, Berlin, 2007.

[25] Kevin Knight. Are many reactive agents better than a few deliberative ones? In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 432–437, 1993. `http://dli.iiit.ac.in/ijcai/IJCAI-93-VOL1/PDF/061.pdf`.

[26] Sven Koenig, Maxim Likhachev, and Xiaoxun Sun. Speeding up moving-target search. In *the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, page To appear, 2007.

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

[28] Richard E. Korf. A simple solution to pursuit games. In *The 11th International Workshop on Distributed Artificial Intelligence*, pages 183–194, Glen Arbor, Mich., Feb 1992.

[29] Dave Kosak. Preview: Supreme commander, 2005. `http://pc.gamespy.com/pc/supreme-commander/632026p1.html`.

[30] Sridhar mahadevan and Mauro Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *machine Learning Research*, 8:2169–2231, 2007.

[31] Christie A. McCloud. Influences on group size and population decline in african wild dogs. *Animal Behavior Bulletin*, 2(4), 1997. `http://www.indiana.edu/~animal/archive/ABB/1997_2(4).html`.

[32] L. David Mech and Luigi Boitani. *Wolves: Behavior, Ecology, and Conservation*. University of Chicago Press, 2003.

[33] Stan Melax. New approaches to moving target search. In *Proceedings of the AAAI Fall Symposium*, 1993.

[34] Stan Melax. New approaches to moving target search. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, 1993.

[35] Steffen Nissen. Fast artificial neural network library (fann). http://leenissen.dk/fann/.

[36] Encyclopædia Britannica Online. Cooperative hunting, 2008. http://www.britannica.com/EBchecked/topic/136356/cooperative-hunting.

[37] Jeff Orkin. *Simple Techniques for Coordinated Behavior*, chapter 3.2, pages 199–206. Charles River Media, Inc., Massachusetts, 2004.

[38] Jeanne Ormrod. *Human Learning*. Pearson, Upper Saddle River, New Jersey, USA, 4th edition, 2004.

[39] Michael Poliza. Wild dog hunting at Dunmatau camp. http://www.fisheaglesafaris.com/cust_south-african-safaris.htm.

[40] Relic Entertainment. Warhammer 40,000 Dawn of War II, 2009. http://www.dawnofwar2.com/us/home.

[41] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphic (SIGGRAPH '87 Conference Proceedings)*, volume 21(4), pages 25–34, 1987.

[42] Martin Riedmiller. Rprop – implementation details. Technical report, University of Karlsruhe, January 1994.

[43] Stuart Russell and Peter Norvig. *Adversarial Search*, pages 161–171. Pearson Education, Inc., New Jersey, USA, 2 edition, 2003.

[44] Stuart Russell and Peter Norvig. *Informed Search and Exploration*, pages 94–134. Pearson Education, Inc., New Jersey, USA, 2 edition, 2003.

[45] Stuart Russell and Peter Norvig. *Mathematical background*, pages 978–979. Pearson Education, Inc., New Jersey, USA, 2 edition, 2003.

[46] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristic. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 357–362, 2008.

[47] Manu Sharma, Michael Homes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. Transfer learning in real time strategy games using hybrid cbr/rl. *Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[48] Doug Smith. Mollies pack wolves baiting a bison. http://www.nps.gov/archive/yell/tours/thismonth/feb2005/doug/molliesbison.htm.

[49] Martin Stolle and Chris Atkeson. Knowledge transfer using local features. In *Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007.

[50] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings AAAI-2005*, July 2005.

[51] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*, chapter 5.4, pages 122–124. The MIT Press, Cambridge, Massachusetts, 1998.

[52] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[53] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*, chapter 3.6, pages 66–67. The MIT Press, Cambridge, Massachusetts, 1998.

[54] Brian Tanner, Vadim Bulitko, Anna Koop, and Cosmin Paduraru. Grounding abstractions in predictive state representations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1077–1082, Hyderabad, India, 2007.

[55] Gerald Tesauro. Temporal difference learning and td-gammon. *Communicaitons of the ACM*, 38(3), 1995. `http://www.research.ibm.com/massive/tdl.html`.

[56] Yue-Hai Wang and Chi Xu. Adaptive algorithm for multi-agent learning optimal cooperative pursuit strategy based on markov game. In *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, volume 5, pages 2973–2978, August 2004. `http:://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=1378542`.

[57] Makoto Yokoo and Yasuhiko Kitamura. Multiagent real-time-A* with selection: Introducing competition in cooperative search. In *In Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96*, pages 409–416, 1996.