University of Nevada, Reno

EVOLVING EFFECTIVE MICRO BEHAVIORS FOR REAL-TIME STRATEGY GAMES

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of Doctor of Philosophy in

Computer Science and Engineering

by

Siming Liu

Dr. Sushil Louis / Dissertation Advisor

May 2015

© 2015 Siming Liu

ALL RIGHTS RESERVED

UNIVERSITY OF NEVADA RENO

THE GRADUATE SCHOOL

We recommend that the dissertation prepared
under our supervision by

SIMING LIU

entitled

EVOLVING EFFECTIVE MICRO BEHAVIORS

FOR REAL-TIME STRATEGY GAMES

be accepted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

Sushil Louis, Ph. D. – Advisor

Sergiu Dascalu, Ph. D. – Committee Member

Monica Nicolescu, Ph. D. – Committee Member

Zong Tian, Ph. D. – Committee Member

Larry Dailey, M. S. – Graduate School Representative

David Zeh, Ph. D. – Dean, Graduate School

May 2015

ABSTRACT

Real-Time Strategy games have become a new frontier of artificial intelligence research. Advances in real-time strategy game AI, like with chess and checkers before, will significantly advance the state of the art in AI research. This thesis aims to investigate using heuristic search algorithms to generate effective micro behaviors in combat scenarios for real-time strategy games. *Macro* and *micro* management are two key aspects of real-time strategy games. While good macro helps a player collect more resources and build more units, good micro helps a player win skirmishes against equal numbers of opponent units or win even when outnumbered. In this research, we use influence maps and potential fields as a basis representation to evolve micro behaviors. We first compare genetic algorithms against two types of hill climbers for generating competitive unit micro management. Second, we investigated the use of case-injected genetic algorithms to quickly and reliably generate high quality micro behaviors. Then we compactly encoded micro behaviors including influence maps, potential fields, and reactive control into fourteen parameters and used genetic algorithms to search for a complete micro bot, *ECSLBot*. We compare the performance of our ECSLBot with two state of the art bots, *UAlbertaBot* and *Nova*, on several skirmish scenarios in a popular real-time strategy game *StarCraft*. The results show that the ECSLBot tuned by genetic algorithms outperforms UAlbertaBot and Nova in kiting efficiency, target selection, and fleeing. In addition, the same approach works to create competitive micro behaviors in another game *SeaCraft*. Using parallelized genetic algorithms to evolve parameters in SeaCraft we are able to speed up the evolutionary process from twenty one hours to nine minutes. We believe this work provides evidence that genetic algorithms and our representation may be a viable approach to creating effective micro behaviors for winning skirmishes in real-time strategy games.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under grant number IIA-1301726 and by the Office of Naval Research grants N00014-12-1-0860 and N00014-12-C-0522.

TABLE OF CONTENTS

Abstract i
Acknowledgements ii
Table of Contents iii
List of Tables v
List of Figures vi

1 Introduction **1**
1.1 Structure of this Thesis 5

2 Background **8**
2.1 Real-Time Strategy Games 8
2.1.1 Decision Making Under Uncertainty 9
2.1.2 Spatial and Temporal Reasoning 10
2.1.3 Task Decomposition 11
2.2 Hill Climbers 13
2.3 Genetic Algorithms 14
2.4 Case Injected Genetic Algorithms 17
2.5 Parallel Genetic Algorithms 19
2.6 Related Work 20
2.7 Real-Time Strategy Environments 23
2.7.1 StarCraft and Bots 23
2.7.2 SeaCraft 25

3 Methodology **27**
3.1 Influence Maps 27
3.2 Potential Fields 29
3.3 Reactive Control 32
3.4 Fitness Evaluation 33
3.5 Bit Setting Optimizer and Random Flip Hill Climbers 36
3.6 Genetic Algorithm 37
3.7 Case Injected Genetic Algorithm 38
3.8 Parallel Genetic Algorithm 39

4 Phase One: Genetic Algorithms versus Hill Climbers **42**
4.1 Experiment Settings 42
4.2 Comparison in Quality and Reliability 44
4.3 Comparison in Robustness 47
4.4 Evolved Group Micro Behaviors 49
4.5 Conclusions 50

5 Phase Two: Case-Injected Genetic Algorithms **53**
5.1 Experiment Settings 53
5.2 Case Injection's Effect on Genetic Algorithms 55
5.3 Conclusions 60

6 Phase Three: ECSLBot versus the state of the art bots **62**
6.1 Experiment Settings 62
6.1.1 Training Scenarios 63
6.1.2 Testing Scenarios 64
6.1.3 Head-to-head Scenario 67
6.2 Evolved ECSLBot 67
6.3 ECSLBot versus the State of the Art Bots in Training Scenarios 71
6.4 ECSLBot versus the State of the Art Bots in Testing Scenarios 73
6.5 ECSLBot versus the State of the Art Bots in Head-to-head Scenario . 78
6.6 Conclusions 79

7 Phase Four: Parallel Genetic Algorithms **81**
7.1 Generalization and Transfer 82
7.2 Conclusions 84

8 Conclusion **86**
8.1 Contributions 88
8.2 Extensions and Future Work 89

Bibliography **92**

LIST OF TABLES

2.1 Nomenclature in Genetic Algorithms 16

3.1 Unit properties defined in StarCraft 27
3.2 Chromosome 34

4.1 Average fitnesses and standard deviations of 500 matches on three maps with different initialized enemy units' position. Dots on the left side of the map represent the friendly units, and dots on the middle of the map represent the enemy units. 48

6.1 Parameter values of best evolved individuals. 70
6.2 The performance of bots controlling 5 Vultures vs 25 Zealots units in scenario $Train_1$. 72
6.3 The performance of bots controlling 5 Vultures vs 6 Vultures in scenario $Train_2$. 72
6.4 The performance of bots controlling 5 Vultures vs opponent units on scenario $Test_1$ and $Test_2$. 74
6.5 The performance of bots controlling 10 Vultures vs ranged attack units on scenario $Test_3$ and $Teset_4$. 75
6.6 The performance of bots controlling 10 Vultures vs melee attack units on scenario $Test_5$ and $Test_6$. 76
6.7 The performance of bots controlling 10 Vultures vs mixed units on scenario $Test_7$ and $Test_8$. 77
6.8 Head-to-head scenario over 30 matches. 79

7.1 5 Vultures vs 25 Zealots over 30 matches in *StarCraft*. 83

LIST OF FIGURES

1.1 Typical RTS AI levels of abstraction. Inspired by a figure from [43]. . 2

2.1 A snapshot of a gameplay in StarCraft II. 9
2.2 StarCraft - Fog of War [3]. 10
2.3 Levels of abstraction and the correspondence to uncertainty, spatial and temporal reasoning described in [38]. 12
2.4 An example of a search space with a single hill. 13
2.5 An example of a search space with two hills. 14
2.6 Genetic algorithms in context. 14
2.7 Overview of a canonical genetic algorithm. 15
2.8 An individual in GAs. 16
2.9 One Point Crossover. 17
2.10 Bit-Wise Mutation. 17
2.11 Solving problems in sequence with CIGAR. 18
2.12 Different models of parallel genetic algorithms. 20
2.13 A snapshot of a game play in StarCraft. 24
2.14 A snapshot of a game play in *SeaCraft*. 26

3.1 An influence map based on two Tanks. 28
3.2 An IM representing the game world with enemy units and terrain. The light area on the bottom right represents enemy units. The light area surrounding the map represents a wall. 30
3.3 A typical PF Function. 31
3.4 Variables used to represent reactive control behaviors. The Vultures on the left side of map are friendly units. Two Vultures on the right are enemy units. 32
3.5 Structure of our Parallel GA. 40

4.1 Scenario 44
4.2 Average score of BSO, RFO, and GA over time. X-axis represents the evaluation times and Y-axis represents the average fitness evaluated by fitness function. 45
4.3 Best scores of BSO, RFO, and GA with 10 different random seeds. X-axis represents random seed and Y-axis shows the highest fitness found by each algorithm initialized with each random seed. 46
4.4 Average maximum and average fitness of GA running on two types of map. X-axis represents generation, and Y-axis represents fitness. . 49
4.5 A snapshot of one group positioning in StarCraft minimap. The dots on the map represent friendly units, and other part of the map was covered by fog of war. Single dot at the left of the map is Tank, and other dots are Marines. 51

5.1 Average maximum/average scores of GA and CIGAR over 10 runs on Intermediate scenario. The X-axis represents the generation and the Y-axis represents the fitness. 56
5.2 Average maximum/average scores of GA and CIGAR over 10 runs on Concentrated scenario. X-axis represents the generation and Y-axis represents the fitness. 57
5.3 Solution quality of each scenario. As more problems are solved, CIGAR produces better solutions than genetic algorithm. The X-axis represents 5 different scenarios. The Y-axis represents the highest fitness. The number on top of each bar of CIGAR shows the number of cases in case-base when CIGAR starts. 58
5.4 Number of generations to solutions found above 1100. As more problems are solved, CIGAR took less time compared to the GA. . . 59

6.1 Training scenarios. 64
6.2 Testing scenarios where only the initial position of units changed. . 65
6.3 Testing scenarios with ten friendly Vultures and different types of enemy units. 66
6.4 The scenario for head to head evaluation. 67
6.5 Average score of ECSLBot versus generations on scenario $Train_1$. X-axis represents time and Y-axis represents fitness by the fitness function F_m. 68
6.6 Average score of ECSLBot over generations in scenarios $Train_2$. X-axis represents time and Y-axis represents fitness by the fitness function F_r. 69
6.7 Learned optimal attacking route against 6 Vultures. 70
6.8 Learned kiting behaviors against Zealots and Vultures. The left side of the figure shows that our Vultures are moving backward and are pointed away from the enemy to kite enemy Zealots. The right side shows that our Vultures are facing the enemy Vultures with only one friendly Vulture moving backward to dodge. 71

7.1 Average fitness of bots over generations on 5 Vultures versus 30 Zealots scenario with parallel GA in *SeaCraft*. 82
7.2 Evolved kiting behavior in SeaCraft. 5 Vultures are moving toward or away from enemy Zealots. Enemy Zealots are surrounded by influence maps shown by dark squares. 83

CHAPTER 1
INTRODUCTION

Research on computational and artificial intelligence (CI and AI) on games has a long history [45]. Several automated game-playing programs are able to outperform world champions in classic games such as *Backgammon*, *Checkers* and *Chess* [46]. These efforts significantly promoted researches in search algorithms and machine learning techniques. More recently, Real-Time Strategy (RTS) games have become a new frontier of AI research due to their complex, realistic, and dynamic environments. RTS games also present a variety of challenges which distinguish them from traditional board games. In the context of AI development in RTS games, researchers usually use knowledge intensive techniques including scripting, finite state machines, and rule-based systems. These techniques however require significant domain knowledge to create and tune a competent AI player. In our research, we aim to create competitive game AI players by using evolutionary techniques to avoid tedious knowledge acquisition and tuning work.

In RTS games, the participants need to gather resources, train units, build structures, research technologies, and conduct simulated warfare in order to defeat their opponents. Players usually divide their decision making into two separate levels of tasks called macro and micro management, as shown in Figure 1.1. Macro is long term planning, like constructing buildings, conducting research, and scouting. Good macro management helps a player build a larger army or economy or both. On the other hand, micro is the ability to control a group of units in combat or other skirmish scenarios to minimize unit loss and maximize damage to opponent units. We decompose micro management into two parts: tactical and reactive control [38]. Tactical control addresses the overall positioning and movement of a

squad of units. Reactive control focuses on controlling a specific unit to move, fire, and flee in combat. This thesis investigates using heuristic search algorithms to find winning tactical and reactive control for skirmish scenarios. This is indicated by the dotted square in Figure 1.1. Common micro techniques in combat include concentrating fire on one target, withdrawing seriously damaged units from the front of the battle, and kiting[1] your units to take advantage of the enemy units' attack-distance limitation. We are interested in generating competitive micro as part of an RTS game player that outperforms an opponent with the same or greater number of enemy units. In the future, we plan to incorporate these results into the design of more complete RTS game players.

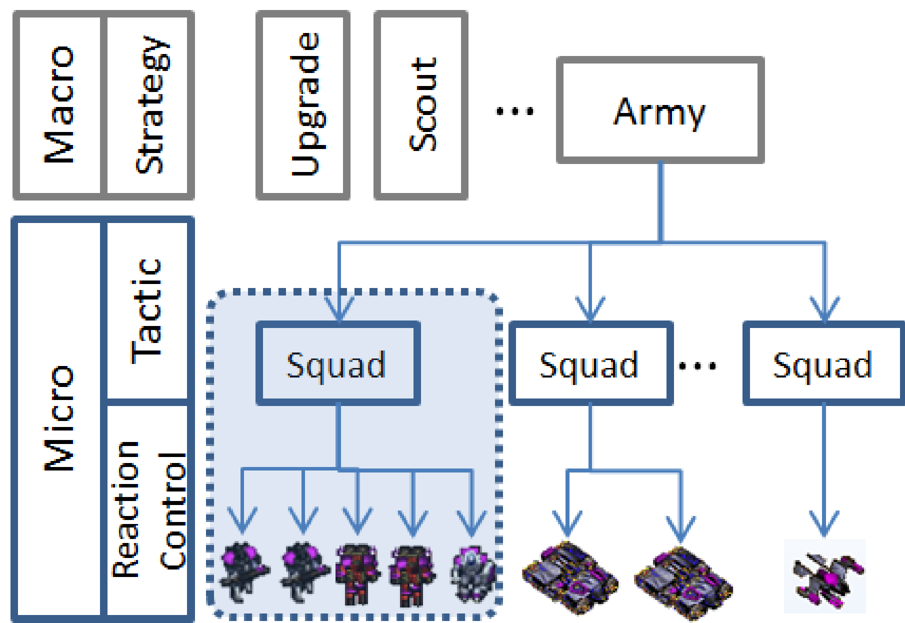


Figure 1.1: Typical RTS AI levels of abstraction. Inspired by a figure from [43].

Several challenges needed to be addressed in our research. First, what are the suitable approaches to generate high performance micro behaviors in RTS skirmish scenarios? Second, how do we represent micro behaviors in combat? Third, how well does our micro bot perform compare with other state of the art bots?

[1]Kiting refers to making your units pull back and shoot repeatedly similar to how you fly kites.

Fourth, do the produced micro behaviors generalize to scenarios not previously encountered? Finally, is our approach applicable to other RTS games? To investigate these issues, we compactly represent micro behaviors as a combination of two influence maps (IMs), two potential fields (PFs), and a set of reactive control variables. We explain IMs, PFs, and reactive controls in the Chapter 3. Our approach is to evolve (off-line) two sets of parameters for each unit type that we want to control. One set of parameters specifies behavior against melee units, the other set specifies behavior against ranged units. During a real-time skirmish, our micro bot switches between these two parameter sets, and their corresponding behaviors, based on whether our bot controlled unit's current target is a melee unit or a ranged unit. We apply this approach to Vultures, a fast but fragile Terran unit in our experiments. With the representation described above, we then use evolutionary algorithms to look for good combinations of these parameters that lead to winning micro behaviors.

The central claim of this thesis is that:

> *Genetic algorithms is a viable approach for generating effective micro behaviors for winning skirmishes in RTS games.*

The long term goal of our research is to create a complete human-level RTS game player and this research attacks one aspect of this problem: finding effective micro management for winning small combat scenarios. In our preliminary work, we compared the quality, reliability, and robustness of group movement behaviors produced by genetic algorithms (GAs) and two types of Hill Climbers (HCs). The results showed that our hill-climbers quickly find IMs and PFs that generate quality positioning and movement in our simulations, but they only find quality solutions fifty to seventy percent of the time. GAs on the other hand evolve high

quality solutions a hundred percent of the time, but take significantly longer [30]. Case-Injected Genetic AlgoRithms (CIGARs) combine GAs with case-based reasoning to learn to increase performance with experience. Since human players also learn to be better with practice and experience, we investigated using CIGARs to learn from experience to be quicker and still get high quality solutions (like the GA) for winning a sequence of skirmishes. The results show that CIGARs learn from prior experience to reliably find quality solutions on new scenarios in half the time taken by GAs.

Since the results indicate that the GA and CIGAR produced higher quality solutions more reliably and quickly, we settled on using GAs and CIGARs to search for effective micro parameters in the rest of this work. We created eleven skirmish scenarios in a popular RTS game *StarCraft: Brood War* and used GAs to search for winning micro behaviors in these scenarios [11]. We subsequently compared the performance of micro behaviors produced by our GAs with two state of the art StarCraft bots, UAlbertaBot [14] and Nova [48]. We chose these bots because their source code was available and they performed well in prior competitions. The results show that Nova performs well on kiting behavior against melee units, while UAlbertaBot does well against ranged attack units. Our ECSLBot tuned by GAs performs well both against melee units and ranged units in eleven scenarios.

Finally, we applied the same approach to another RTS game *SeaCraft* developed by our research group. We modeled the game play, unit properties in SeaCraft around StarCraft. Results show that we can successfully evolve equally effective micro behaviors in SeaCraft. Furthermore, we show that parameters that specify reactive control behaviors such as kiting, target selection, and fleeing evolved in SeaCraft are able to be transferred without change to StarCraft with very little loss

of unit micro performance in skirmishes. Since SeaCraft can simulate skirmishes much faster than StarCraft and can be easily used by a Parallel Genetic Algorithm (PGA), we can speed up micro parameter search by two orders of magnitude even on only two machines with sixteen cores.

To clarify the scope of our work, note that we do not focus on creating a complete RTS game AI player. Rather, we only focus on finding high performance micro management in combat for a game AI. In addition, the heuristic search algorithms including GA, CIGAR, and HCs are independent of the game AI and RTS game. The algorithms do not know how the game AI or RTS game work, they create micro behaviors and note the performance of the micro behaviors evaluated by the RTS game. Our work provides three main contributions towards finding effective micro behaviors in combat scenarios. First, we applied evolutionary algorithms including GAs and CIGARs to generate high performance micro management in RTS games. Second, we present a new representation of micro behaviors in combat based on IMs, PFs, and reactive controls. Third, we extend our GAs to be able to evaluate individuals in parallel based on Open MPI and apply our approach to another RTS game SeaCraft.

1.1 Structure of this Thesis

The next chapter provides background information on RTS games, the approaches used in our research, and related work. We start with an overview of RTS games and their challenges, including decision making under uncertainty, spatial and temporal reasoning, and task decomposition. Next we review the definitions of genetic algorithms, case-injected genetic algorithms, the two hill-climbers we used,

and parallel genetic algorithms. Finally, we provide a summary of academic research in games and RTS micro management.

Chapter 3 describes our representation. Spatial maneuvering is an important component of combat in RTS games. We applied a commonly used technique called influence maps to represent the spatial information of terrain and enemy units in a game. While good influence maps tell us where to go, good unit navigation tells our units how best to move there. We use potential fields to control a group of units navigating to particular locations on the map. Beside representation, this chapter also provides detailed specification of our heuristic search algorithms including two hill climbers, genetic algorithms, case-injected genetic algorithms, and parallel genetic algorithms.

Chapter 4 shows the group micro behaviors produced by our genetic algorithms and two hill climbers. We compare micro behaviors produced by genetic algorithms and two hill climbers with each other. Our initial results comparing genetic algorithms to hill climbers were published in the *Proceedings of the 2013 IEEE Congress on Evolutionary Computation* [30].

Chapter 5 shows the influence of case-injection on genetic algorithms. We apply CIGARs to speed up finding high quality solutions when solving similar problems. The results for using CIGARs to find effective group behaviors were published in the *Proceedings of the 2013 IEEE Conference on Computational Intelligence and Games* [31].

Chapter 6 details our evolved micro bot (ECSLBot) and the comparison between ECSLBot with two state of the art bots, UAlbertaBot and Nova. The results for comparing ECSLBot, UAlbertaBot, and Nova on a variety of scenarios were

published in the *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games* [29].

Chapter 7 describes our parallel genetic algorithms. We extend our genetic algorithm to be able to evaluate individuals in parallel based on Open MPI and apply our approach to another RTS game SeaCraft that enables easy GA parallelization. Finally, Chapter 8 discusses our conclusions and future work.

CHAPTER 2
BACKGROUND

This chapter first details gameplay and micromanagement in RTS games. We also give an overview of the terminology and an introduction to hill climbers, genetic algorithms, case-injected genetic algorithms, and parallel genetic algorithms. Finally, the last section of this chapter reviews work related to game AI and micro management in RTS games.

2.1 Real-Time Strategy Games

RTS games are a sub-genre of strategy video games where players need to gather resources, train units, build structures, research technologies, and conduct simulated warfare in order to defeat their opponents. Understanding each of these factors and their impact on decision making is critical for winning an RTS game. For example, Figure 2.1 shows a snapshot of gameplay in a popular RTS game *StarCraft II*. Compared to board games, players in RTS games do not take turns, but instead may perform as many actions as they can, while the game runs at a constant rate to simulate a continuous flow of time. Most RTS games are partially observable where players can only see the part of the map with a friendly unit or building nearby. Actions in RTS games are not always deterministic. Some actions have a probability of success. To measure the complexity in terms of game states, while Chess has around 10^{50} board states and Go has 10^{170} board states, a typical RTS game is estimated to have over $(10^{50})^{36000}$ different game states, more than the number of protons in the universe [38]. Therefore, traditional AI techniques used for playing board games, such as *MINIMAX* game tree search, cannot be directly

applied to RTS games [44].

Figure 2.1: A snapshot of a gameplay in StarCraft II.

RTS games provide an exciting opportunity for AI research, containing a variety of interesting and challenging problems within. The following sections describe three challenges presented by RTS games and how they are related to micro management.

2.1.1 Decision Making Under Uncertainty

In RTS games, the game worlds are usually covered by a *fog of war* which prevents players from accessing the complete information of the whole game world, except in locations containing friendly units or buildings. Figure 2.2 shows the three states of a fog of war. Locations near the friendly units are fully revealed as shown in the bottom right part of the map. The top of the map shows that the area currently has

Figure 2.2: StarCraft - Fog of War [3].

no friendly units nearby but has been visited before. The static terrain information is accessible in partially concealed area. Fully concealed locations are areas where the friendly units have never visited before, as shown in the left side of the map. Players have to send a friendly unit to the specific area to obtain terrain and enemy information. Due to the fog of war, players are able to deceive and mislead one another in a game. Such decision making under uncertainty is a significant research subject within the AI research community, especially in games like poker [5, 7].

2.1.2 Spatial and Temporal Reasoning

Spatial reasoning problems involve static terrain and dynamic units in the RTS game world. Spatial reasoning is particularly essential for tactical reasoning in a battlefield for winning skirmishes in RTS games. Players need to decide where to

attack, where to defend, and how to assault in order to win a series of battles or even the whole game. For example, engaging an enemy force in front of a bottleneck will favor the friendly force. Micro operations like hit and run perform better in an open space than near a wall. These fundamentally difficult spatial decisions significantly affect RTS game playing strategies and tactics. In this research, we use a commonly used representational technique, influence maps, to represent terrain and enemy spatial information in a battlefield.

In canonical board games, actions or moves take effect immediately. While in RTS games, movement and actions take a specific amount of time to complete. For example, moving a unit from one location to another location may take 30 seconds. Therefore, players use timed attacks and retreats to gain advantage in battles. Decisions on temporal questions such as when to build military units, when to upgrade technologies, and when will an opponent attack are crucial in RTS games.

2.1.3 Task Decomposition

Players usually decompose the problem of playing an RTS game into a collection of smaller problems and solve the sub-problems independently. A common subdivision is:

- *Macro management* (or *Strategy*) corresponds to high-level and long-term decision making process. This is the top level of abstraction in RTS games. For example, adopting a specific build-order against a given opponent or upgrading a key technology against a specific race. Strategy decisions usually concern all the elements in the game.

- *Micro management* can be split into tactics and reactive control.
  - *Tactics*: considers a squad of units in a specific area in the game. It involves the positions and movements of a group of units.
  - *Reactive control*: consists of moving, targeting, firing, fleeing, kiting behaviors of individual units during battle. Reactive control focuses on a single unit.

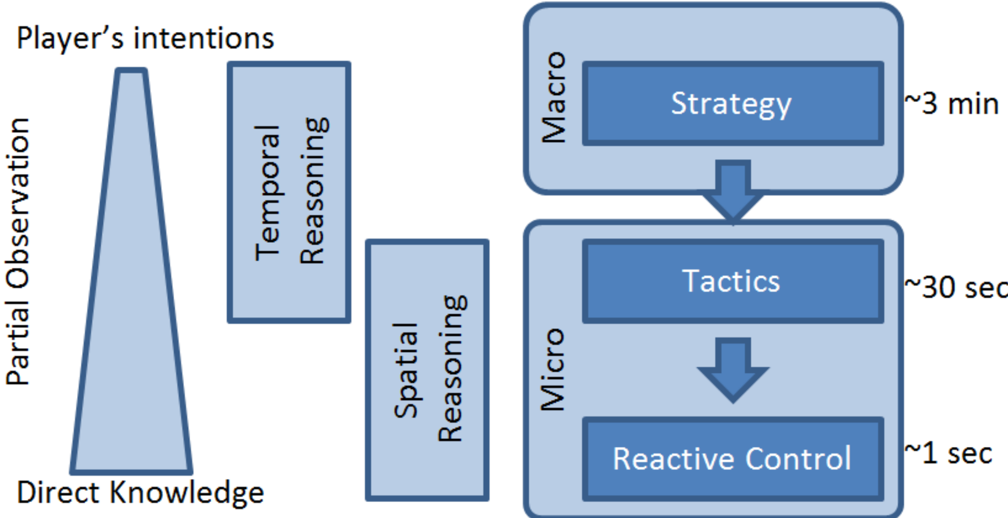


Figure 2.3: Levels of abstraction and the correspondence to uncertainty, spatial and temporal reasoning described in [38].

Figure 2.3 illustrates one common task decomposition and shows the levels of abstraction where strategy corresponds to long term decision making over several minutes, reactive control corresponds to short term decisions over a second, and tactics is in between. This research focuses on tactics and reactive control behaviors. We investigates heuristic search algorithms to find winning tactical and reactive control for skirmish scenarios in RTS games. Many different approaches have been applied to finding high performance micro management. In the following section, we provide an overview of the techniques we use (hill climbers, genetic

algorithms, case-injected genetic algorithms, and parallel genetic algorithms) as well as techniques used by other researchers.

2.2 Hill Climbers

Hill climbers are a local search optimization technique. A hill climber starts with a random initialized solution to a problem, then attempts to find a better solution by incrementally updating an element of the solution. The update will be kept if it produces a better solution. The process is repeated until no further improvements can be made. Hill climbers are good for finding a local optimum but are not guaranteed to find the global optimum. In problems with only a single hill in the search space, hill climbers are optimal. Figure 2.4 shows an example of a search space with only a single hill. However, in problems with multiple hills in the search space, hill climbing performs worse than other globe search algorithms like genetic algorithms and simulated annealing. Figure 2.5 shows an example of a search space with two hills.

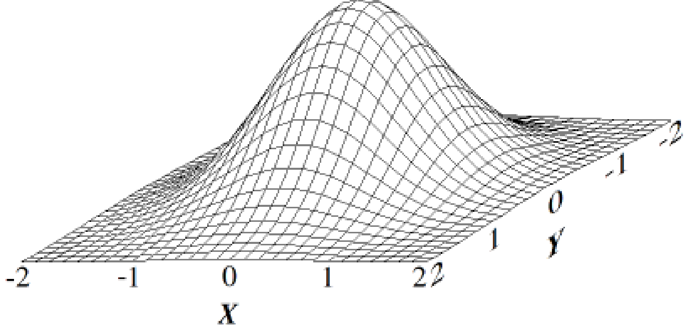


Figure 2.4: An example of a search space with a single hill.

The characteristic of being easily implemented makes hill climbers a popular first choice amongst optimizing algorithms. In this research, we compare two

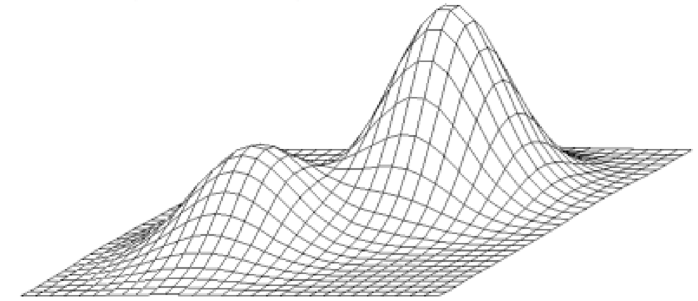


Figure 2.5: An example of a search space with two hills.

types of hill climbers against genetic algorithms in searching for competitive solutions to win skirmishes in RTS games.

2.3 Genetic Algorithms

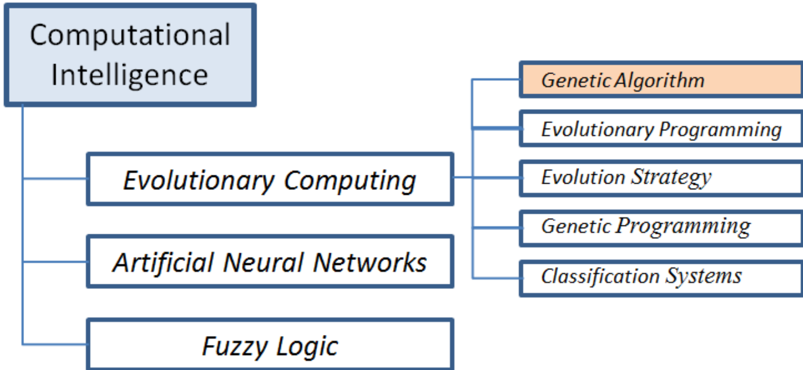


Figure 2.6: Genetic algorithms in context.

Genetic algorithms were first introduced by John Holland from his research on cellular automata at the University of Michigan in 1975 [24, 19]. Before discussing genetic algorithms in detail, we want to put these algorithms in context. In recent years, the fields of *Evolution Computing*, *Neural Networks*, and *Fuzzy Logic* are categorized together as techniques which are using numeric knowledge representation to solve complicated problems. The broader research domain is known as *Compu-*

tational Intelligence [8]. Figure 2.6 details the structure of computational intelligence and evolutionary algorithms.

A genetic algorithm is a heuristic search technique inspired by natural evolution, with inheritance, selection, crossover, and mutation. GAs are usually applied to generate useful solutions to optimization and search problems. Figure 2.7 shows a overview of a genetic algorithm.

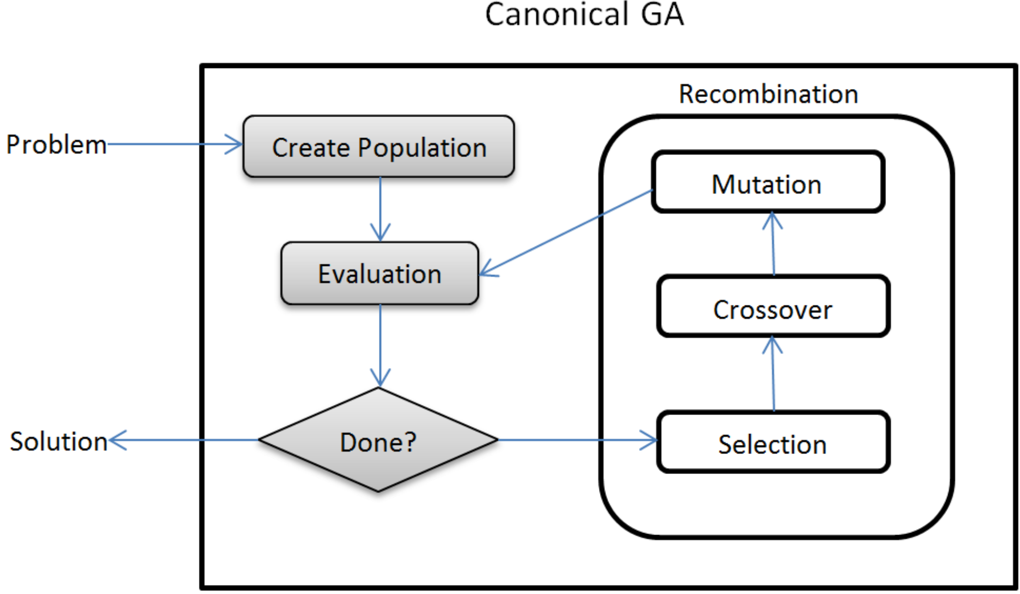


Figure 2.7: Overview of a canonical genetic algorithm.

GAs attempt to solve problems in an iterative process starting with a *population* of randomly initialized *individuals*. Each individual encodes a candidate solution to the problem in its *chromosome* that represents a set of parameters called the *genes*. Each gene is in turn encoded into a binary string. The values interpreted from the genes are called *alleles*, as shown in Figure 2.8. Table 2.1 lists the common terms and their description used in GAs [19].

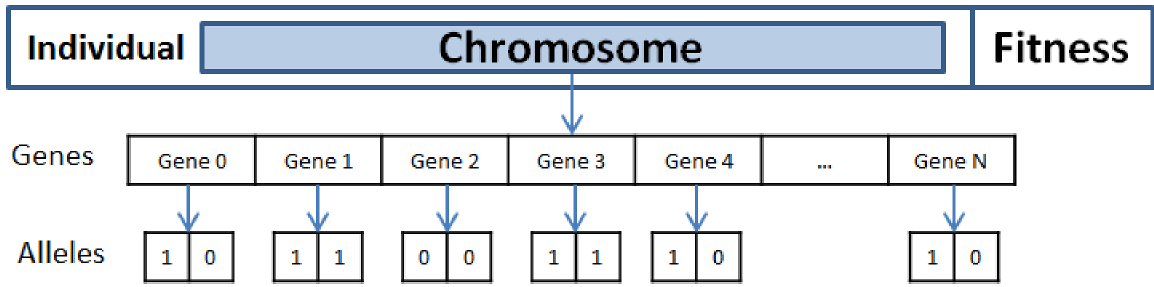


Figure 2.8: An individual in GAs.

Table 2.1: Nomenclature in Genetic Algorithms

Term	**Description**
Population	A set of individuals with their associated statistics.
Individual	A candidate solution includes a chromosome with an associated fitness.
Chromosome	One encoded string of parameters.
Gene	The encoded version of parameters of the problem to be solved.
Allele	The value which a gene can assume.
Fitness	A value indicating the quality of an individual as a solution to the problem.
Selection	Operation for selecting one individual from the population.
Crossover	Operation that exchange information of two selected parents to yield two new children.
Mutation	Operation that spontaneously changes one or more bits in a chromosome.

A fitness function is used to evaluate the fitness of each individual in the population. Once every individual in the population has a fitness, individuals are recombined and manipulated by the genetic operators of selection, crossover, and mutation, to evolve a new population. Selection chooses individuals with a probability proportional to the individual fitnesses. Crossover exchanges and recombines information between individuals in attempting to produce children with

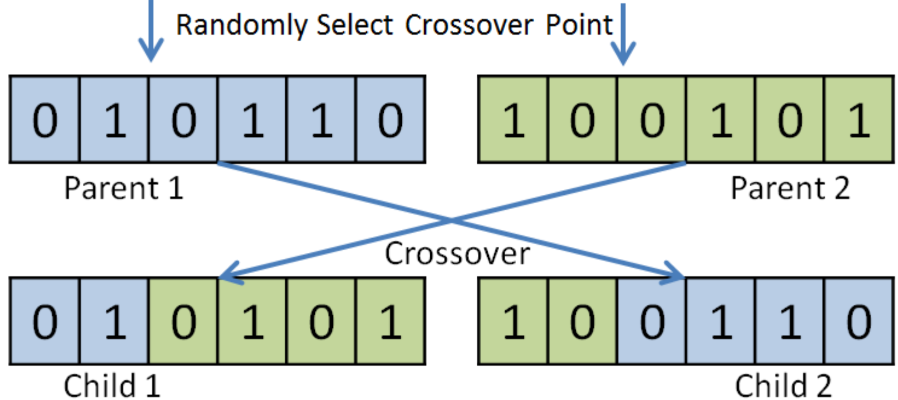


Figure 2.9: One Point Crossover.

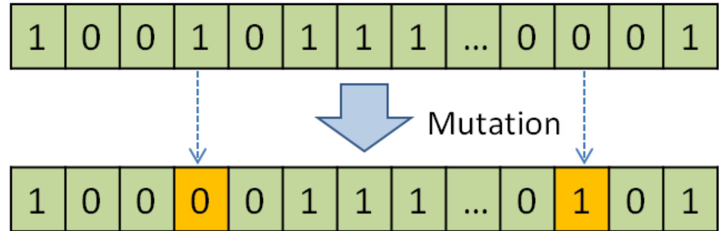


Figure 2.10: Bit-Wise Mutation.

higher fitnesses. Figure 2.9 shows an example of one point crossover. Mutation maintains genetic diversity from one generation to the next. Bit-wise mutation randomly flips a bit with a low probability, as shown in Figure 2.10. Using genetic algorithms in our research, we are able to evolve high performance micro behaviors for winning skirmishes in RTS games.

2.4 Case Injected Genetic Algorithms

CIGAR was first introduced by Louis and McDonnell. They borrowed ideas from case-based reasoning (CBR) in which experience from solving previous problems helps solve new similar problems [33]. This approach augmented GAs with a case-

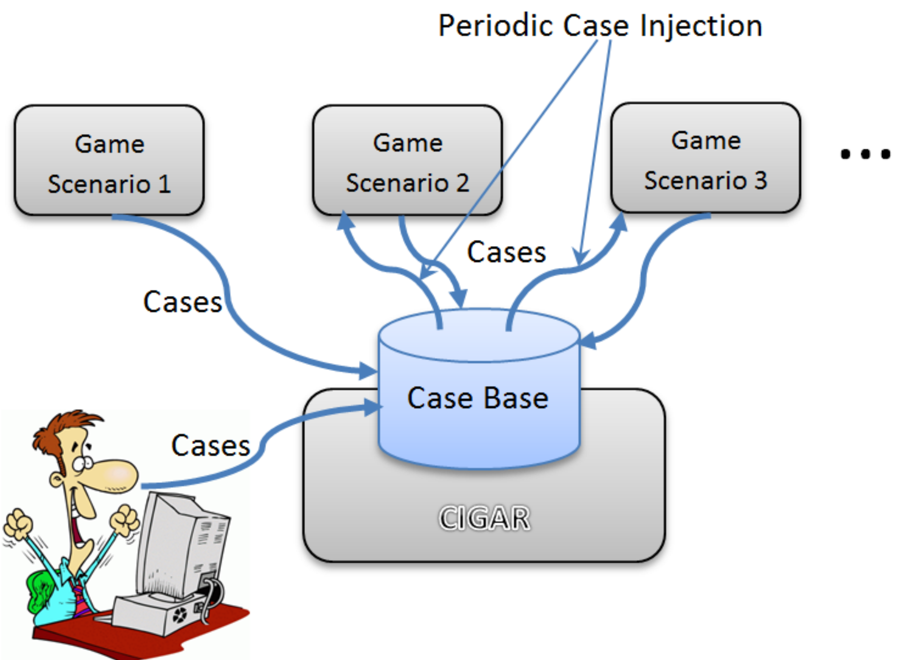


Figure 2.11: Solving problems in sequence with CIGAR.

based memory of past problem solving experience and was used to obtain better performance over time on sets of similar problems. Figure 2.11 shows how CIGAR solves a sequence of problems. An existing case-base is not necessary in case-injected GAs because the GA simply starts with a randomly initialized population to search the space. During such search, we save good chromosomes which are cases into the case-base for potential use in subsequent problem solving. Case-injection enable genetic algorithms to learn from experience. Louis and Li applied CIGAR for solving traveling salesman problems (TSPs) and showed performance improvement on similar TSPs [32]. Louis and Miles applied CIGAR in a strike force asset allocation game [34]. They used cases from both human's and system's game-playing experiences to bias CIGAR toward producing plans that contain previous important strategic elements.

In our research, we investigate using CIGARs to quickly generate high qual-

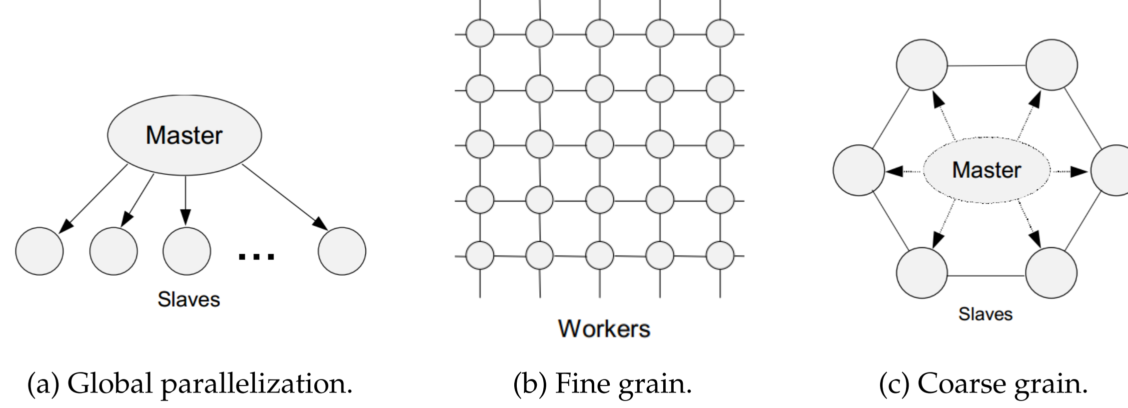
ity unit micro-management in real-time strategy game skirmishes. We consider a series of maps as a sequence of problems to be solved by CIGAR and expect that CIGAR will learn on problems early in the sequence to improve performance on problems later in this sequence.

2.5 Parallel Genetic Algorithms

One feasible way to run our GA faster is by parallelizing evaluation since most of the computational load comes from evaluation in the RTS game engine. Most parallel programs adopt the idea of a divide and conquer strategy to split a task into sub-tasks and solve sub-tasks simultaneously using multiple processors. This divide and conquer approach can be used in GAs too. There are three main types of Parallel GAs (PGAs):

1. Global single-population master-slave GAs.
2. Single-population fine-grained GAs.
3. Multiple-population coarse-grained GAs.

In a master-slave PGA there is a single population as in a canonical GA, but the evaluation of fitness is distributed among several processors, as shown in Figure 2.12a. Since selection and crossover operate on the entire population, this type of parallel GA is also called global single-population parallel GAs. Fine-grained parallel GAs are usually used in massively parallel computers environment and consist of one spatially-structured population. Selection and mating are restricted to a small neighborhood, but neighborhoods overlap permitting some interaction



(a) Global parallelization. (b) Fine grain. (c) Coarse grain.

Figure 2.12: Different models of parallel genetic algorithms.

among all the individuals as shown in Figure 2.12b. Multiple-population coarse-grained GAs are more complicated as they consist of several subpopulations which exchange individuals occasionally as shown in Figure 2.12c. In this research, we use a global single-population master-slave PGA to search for effective micro behaviors in RTS skirmishes.

2.6 Related Work

Typically, industry RTS AI developers create RTS AI not so much for beating opponents as to entertain and tutor users. Industry AI employs techniques such as finite state machines, rule based systems, and scripting [10, 40]. Some industry AIs may cheat. On the the other hand, academic RTS AI research focuses on using learning or reasoning techniques to win an RTS game and reach human competitive performance. Our research falls in the academic category.

Much work has been done in applying different techniques to build RTS AI players in academia. For example, some work has been done in our lab on co-

evolving robust build orders in WaterCraft [4]. Ontañón *et al.* used real-time case-based planning(CBP) in an RTS game Wargus [37]. Weber and Mateas presented a data mining approach to strategy prediction by learning from StarCraft replays [49]. Churchill *et al.* adopted an Alpha-Beta search approach from board games for RTS combat scenarios of up to eight versus eight units [15]. This paper focuses on the work related to using IMs and PFs for spatial reasoning and unit movement. Miles *et al.* applied IMs to evolve a LagoonCraft RTS game player [13]. Sweetser *et al.* developed a game agent designed with IMs and cellular automata, where the IM models the environment and helps an agent make decisions in their EmerGEnt game [47]. They built a flexible game agent that responds to natural phenomena and user actions while pursuing a goal. Bergsma *et al.* used IMs to generate adaptive AI for a turn based strategy game [6]. Su-Hyung *et al.* proposed a strategy generation method using IMs in the strategy game Conqueror. He applied evolutionary neural networks to evolve non-player characters' strategies based on the information provided by layered IMs [26]. Avery *et al.* worked on co-evolving team tactics using a set of IMs, guiding a group of friendly units to move and attack enemy units based on the opponent's position [2]. Their approach used one IM for each entity in the game to generate different unit movement. However, this method does not scale well to large numbers of units. For example, if we have two hundred entities, the population cap for StarCraft, we will need to recompute two hundred IMs every update. This could be a heavy load for our system. Preuss *et al.* used a flocking based and IM-based path finding algorithm to enhance group movement in the RTS game Glest [39, 16]. Raboin *et al.* presented a heuristic search technique for multi-agent pursuit-evasion games in partially observable space [41]. In this paper, we use an enemy units position IM combined with a terrain IM to gather spatial information and guide our units in producing

winning micro behaviors for RTS games.

Potential fields have also been applied to AI research in RTS games. Most of the prior work in PFs is related to unit movement for spatial navigation and collision avoidance [9]. This approach was first introduced by Khatib in 1986 while he worked on real time obstacle avoidance for mobile robots [28]. The technique was then widely used in avoiding obstacles and collisions, especially in multiple unit scenarios with flocking [36, 17, 42]. Hagelbäck *et al.* applied this technique to AI research within an RTS game [23]. They presented a Multi-Agent Potential Field based bot architecture in the RTS game ORTS [12] and incorporate PFs into their AI player at both tactical and unit reactive control level [22]. We have also done some prior work in PFss [30, 31] and use two PFs for group navigation in our work.

Reactive control, including individual unit movement and behavior, aims at maximizing damage output to enemy units and minimizing the damage to friendly units. Common micro techniques in combat include fire concentration, target selection, fleeing, and kiting. Uriarte *et al.* applied IMs for kiting, frequently used by human players, and incorporated kiting behavior into his StarCraft bot Nova [48]. Gunnerud *et al.* introduced a CBR/RL hybrid system for learning target selection in given situations during a battle [21]. Wender *et al.* evaluated the suitability of reinforcement learning algorithms to micro manage combat units in RTS games [50]. The results showed that their AI player was able to learn selected tasks like “Fight”, “Retreat”, and “Idle” during combat.

We scripted our reactive control behaviors with a list of unit features represented by six parameters. Each set of parameters influences reactive control behaviors including kiting, targeting, fleeing, and movement. As testbeds for our research, we next describe a popular commercial RTS platform, StarCraft, and a

purposely built RTS platform designed for AI research, SeaCraft.

2.7 Real-Time Strategy Environments

In our field, a suitable RTS research environment would be popular, open source, speed adjustable, and easily parallelizable. *StarCraft: Brood War* is one of the most successful commercial RTS games which is released in 1998 by *Blizzard Entertainment*[1]. StarCraft has become a new popular RTS research platform due to the StarCraft: Brood War Application Programming Interface (BWAPI) framework and the *AIIDE*[2] and *CIG*[3] StarCraft AI Competitions [11]. BWAPI provides an interface allowing our program to interact with StarCraft game data through code instead of keyboard and mouse. However, StarCraft was designed for human players and is difficult to run in parallel. Therefore, other RTS games such as *SeaCraft*, *Wargus*, and *ORTS* emerged and were designed specifically for scientific research. In our work, we run our experiments on the popular platform StarCraft that allows us to compare our work with our peers. On the other hand, we also use an open source RTS game SeaCraft we developed to speedup our search process by parallellizing evaluation, and then use the evolved solution in StarCraft.

2.7.1 StarCraft and Bots

StarCraft is one of the most well known RTS games with a huge player base and numerous professional competitions all over the world. Figure 2.13 shows a snap-

[1] http://www.blizzard.com
[2] http://www.StarCraftAICompetition.com
[3] http://cilab.sejong.ac.kr/sc_competition/

Figure 2.13: A snapshot of a game play in StarCraft.

shot of a game play in StarCraft. The game has three different races: Terran, Protoss, and Zerg. Thanks to the popularity of the StarCraft and recent StarCraft AI tournaments, many groups have been working on integrating cutting edge techniques into RTS AI players called bots which are capable of playing StarCraft. In our research, we apply heuristic search algorithms to generate effective micro behaviors and compare the micro performance of our ECSLBot with two other state of the art bots: UAlbertaBot and Nova. The bots we used in this paper are listed below:

- **UAlbertaBot**: Developed by D. Churchill from the University of Alberta. It is the champion of the AIIDE 2013 StarCraft competition.
- **Nova**: Developed by A. Uriate from Drexel University. Nova was ranked number 7 on the AIIDE 2013 StarCraft competition.
- **SCAI**: The default StarCraft AI. It was used as our baseline in evaluating the

micro performance of other bots.

- **ECSLBot**: Our bot that currently only does micro, using parameters generated by our approach.

The micro logic of the UAlbertaBot is handled by a MeleeManager and a RangedManager for all types of units rather than each specific unit type. This abstraction allows the bot to adapt the micro managers to different types of military units. However, the UAlbertaBot implementation ignores the difference between units. For example, both Vulture and Dragoon are range attackers and can "kite" or "hit and run" against melee units, but they should kite differently based on their unique weapon cool down times and target selection algorithms. In contrast, Nova uses IMs to control the navigation of multiple units and applied this idea to a kiting behavior.

2.7.2 SeaCraft

In our work using evolutionary computing algorithms, we need to run the game in order to evaluate competing micro. This makes evaluations computationally expensive. Thus, although StarCraft is a good platform for RTS AI research and we can compare our work with other researchers, one disadvantage of StarCraft is that it is difficult to parallellization. We therefore apply our approach to another playable RTS game that we developed, named SeaCraft, that makes it easier to parallellize evaluations for an evolutionary computing algorithm. SeaCraft was developed in our ECSL lab for evolutionary algorithms research in RTS games. SeaCraft uses the popular *OGRE* graphics engine and is implemented in C++ [27]. We modeled game play in SeaCraft around StarCraft to make comparisons and

transfer easier. Like in StarCraft, players can control several types of units, with the objective of destroying their opponent's force. SeaCraft runs in a Linux environment and allows researchers to turn off the graphics thread, which makes SeaCraft suitable for parallel evaluation. Figure 2.14 shows a snapshot of game play in SeaCraft.

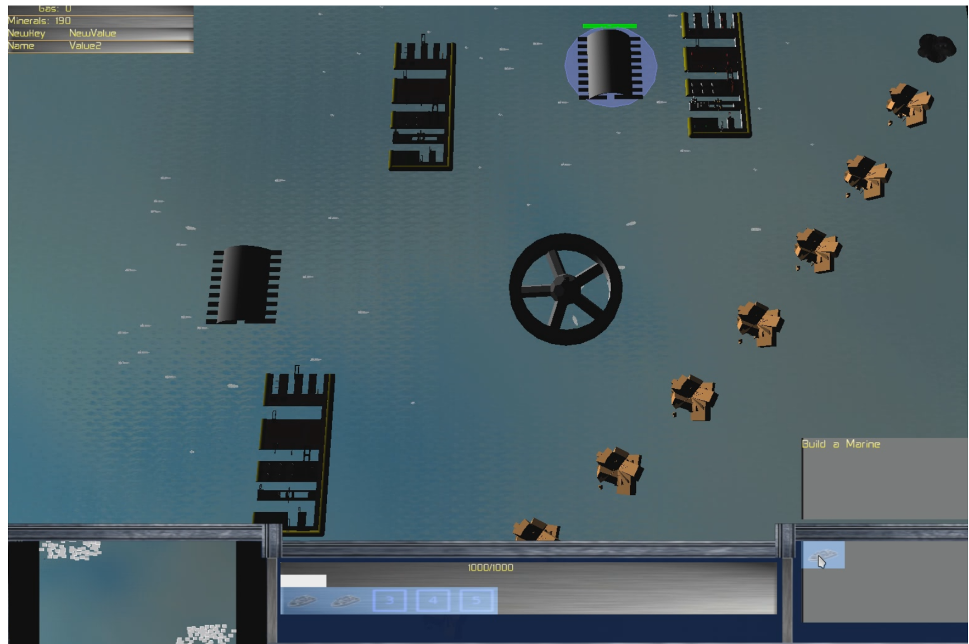


Figure 2.14: A snapshot of a game play in *SeaCraft*.

We applied different approaches for finding effective micro behaviors in both StarCraft and SeaCraft. The next chapter details the representation of micro behaviors in RTS games and the approaches we used in our experiments.

CHAPTER 3

METHODOLOGY

In our scenarios, ECSLBot attempts to defeat the opponent by eliminating enemy units while minimizing the loss of friendly units. A secondary objective is to do this as quickly as possible. Our first set of scenarios contain two StarCraft unit types: Vulture and Zealot. A Vulture is a Terran unit with a ranged attack weapon, low hit-points (easy to destroy), and fast movement. On the other hand, a Zealot is a Protoss unit with a melee weapon, high hit-points (hard to destroy), and slow movement. Table 3.1 shows the detailed parameters for Vultures and Zealots.

Table 3.1: Unit properties defined in StarCraft

Parameter	Vulture	Zealot	Purpose
Hit-points	80	160	Entity's health. Entity dies when Hit-points ≤ 0.
MaxSpeed	6.4	4.0	Maximum move speed of Entity.
Damage	20	8×2	Number of Hit-points that can be removed from the target's health by each hit.
Weapon	Ranged	Melee	The distance range within which an entity can fire upon target.
Cooldown	30	22	Time between weapons firing.
Destroy Score	150	200	Score gained by opponent when this unit has been killed.

3.1 Influence Maps

Spatial maneuvering is an important component of combat in RTS games. We applied a commonly used technique called Influence Maps (IMs) to represent terrain

and enemy units spatial information. IMs originated out of work on spatial reasoning within the game of *Go* and have been used widely in various video games [52]. An IM is a grid placed over a virtual world with values assigned to each square by an IM function [35]. Figure 3.1 shows an IM representing four units in a game, with the IMFunction being the number of Tanks within some radius. A Tank is a Terran unit with a ranged attack weapon, high hit-points, and slow movement. In case the two Tanks belongs to your opponent, and the radius of the circle was the weapon range of a tank, this IM could be used to find areas dangerous to friendly units.

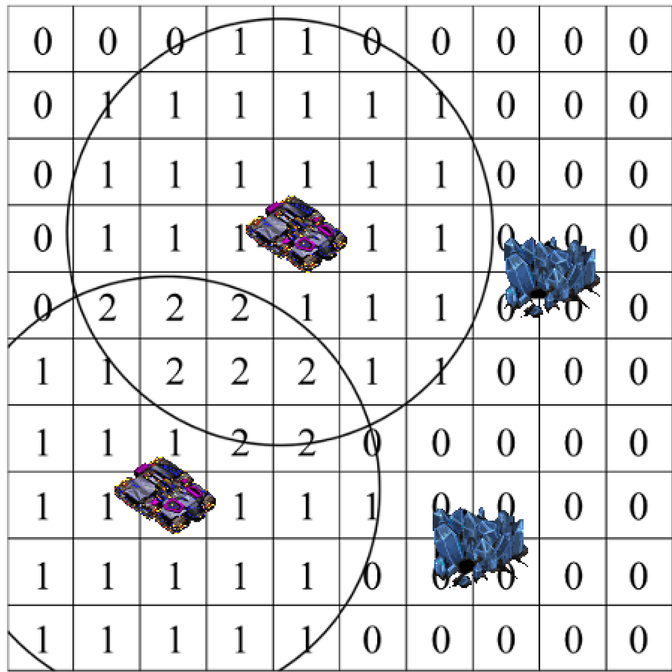


Figure 3.1: An influence map based on two Tanks.

Figure 3.2 shows an IM which represents a force of enemy units and the surrounding terrain in StarCraft. A unit IM is computed by all enemy units in the game. In our experiments, greater cell values indicate more influence by enemy units and more danger to friendly units. In addition to the position of enemy units, terrain is another critical factor for micro behaviors. For example, kiting en-

emy units near a wall is not a wise move. We then use another IM to represent terrain in the game world to assist micro management. We combine the two IMs and use this battlefield (or map) spatial information to guide our AI player's positioning and reactive control. Since computation time depends on the number of IM cells, we use a cell size of 32×32 pixels. The entire map consists of a 64×64 grid of such 32×32 pixel cells. Note that as enemy units move, the two IMs change. The sum IM therefore also changes and no matter what the actual StarCraft map we play on and where on that map opponent units are positioned, the sum IM indicates vulnerable positions to attack as well as positions not to attack. Algorithm 1 described below chooses a specific location to attack based on the current sum IM. We recompute all the IMs every second. In our research, we use heuristic search algorithms to find optimal IM parameters that help specify high quality micro behaviors of units in combat scenarios. Once learned, the parameters that define an IM generalize well to other game maps and is one reason IMs are a useful representation for spatial information.

Algorithm 1 Targeting and Positioning Algorithm

```
Initialize TerrainIM, EnemyUnitIM, SumIM;
Target = MinIMValueUnit on SumIM;
targetPos = Target.getPosition();
movePos = minSurroundingPos(targetPos);
squad.moveTo(movePos);
squad.moveAttack(Target);
```

3.2 Potential Fields

While good IMs tell us where to go, good unit navigation tells our units how best to move there. We use Potential Fields (PFs) in our research to control a group of units navigating to particular locations on the map. PFs are methods originat-

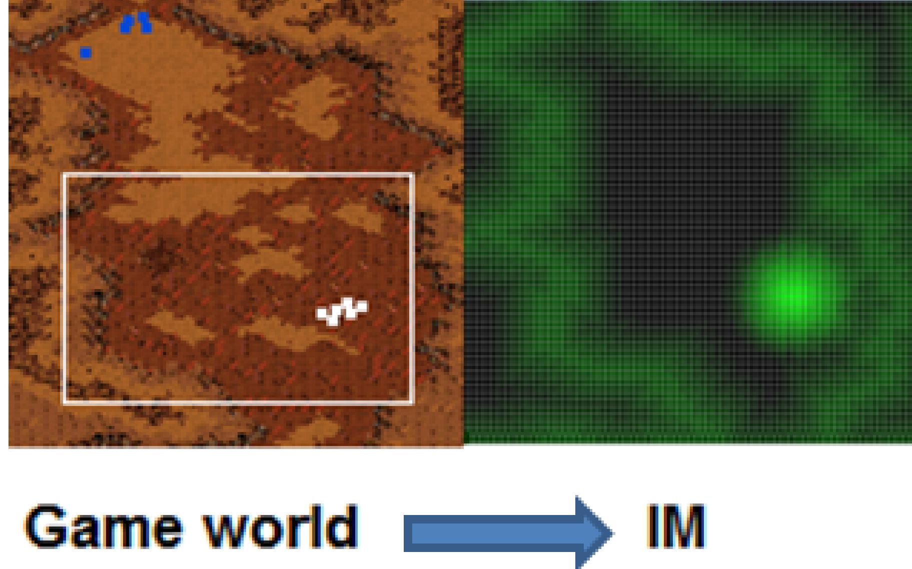


Figure 3.2: An IM representing the game world with enemy units and terrain. The light area on the bottom right represents enemy units. The light area surrounding the map represents a wall.

ing from maneuvering robots to avoid obstacles. A PF creates an attracting or a repelling field in a virtual space. Similar to magnetic charges, the sum of all the PFs determines a vector force with a strength and a direction at a given position in the virtual world. Equation 3.1 shows a typical PF function where $Force$ is the potential force on the unit, d is the distance from the source of the force to the unit. c is the coefficient and e is the exponent applied to distance and used to adjust the strength and direction of the vector force.

$$Force = cd^e \tag{3.1}$$

Figure 3.3 shows a typical potential function including both attraction and repulsion. The X-axis is distance between the destination and the entity, the Y-axis is the potential force. The negative part of the curve acts over a relatively short distance and represents repulsion. The positive part further away from the vertical axis represents the force of attraction.

Figure 3.3: A typical PF Function.

We use two PFs of the form described by Equation 3.1 to control the movement of units. Each PF calculates one force acting on a unit. The two potential forces in our game world are:

- **Attractor:** The attraction force is generated by the unit's destination - the unit is attracted to its destination. This force is inversely proportional to distance. A typical attractor looks like $Force = \frac{2500}{d^{2.1}}$. Here $c = 2500$ and $e = -2.1$ with respect to Equation 3.1.
- **Repulsor:** This keeps friendly units moving towards the destination from colliding with each other. It is usually stronger than the attractor force at short distances and weaker at long distances. A typical repulsor looks like $Force = \frac{32000}{d^{3.2}}$.

Each PF is determined by two parameters, a coefficient c and an exponent e. Therefore, we use four parameters to determine a unit's PFs:

$$PF = c_a d^{e_a} + c_r d^{e_r} \tag{3.2}$$

where c_a and e_a are parameters of the attractor force, c_r and e_r for the friend repulsor force. These parameters are then encoded into a binary string for our algorithms.

3.3 Reactive Control

Besides the group positioning and unit movement, reactive control behaviors must be represented in a way that our algorithms can process. In our research, we considered three reactive control behaviors: kiting, target selection, and fleeing frequently used in real games by good human players. Figure 3.4 shows the six variables used in our micro scripting logic and Table 3.2 explains the details and purpose of each variable.

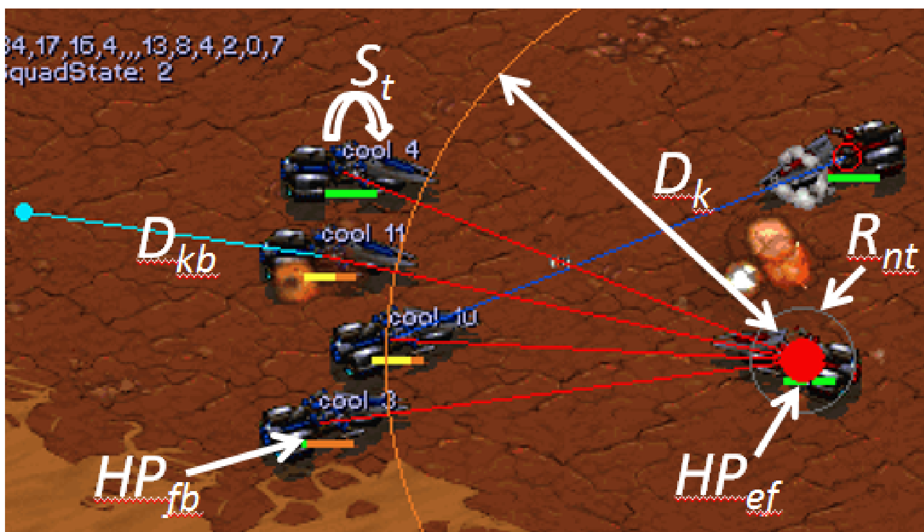


Figure 3.4: Variables used to represent reactive control behaviors. The Vultures on the left side of map are friendly units. Two Vultures on the right are enemy units.

- **Kiting**: Also known as "hit and run". This behavior is especially useful in combat where our units have a larger attack range than the enemy units. The variables used in kiting are S_t, D_k, D_{kb} as explained in Table 3.2.

- **Target Selection**: Concentrating fire on one target, or switching to a nearby enemy with low hit-points. The variables used in target selection are R_{nt}, HP_{ef}.
- **Flee**: Temporarily repositioning to the back of our forces away from the front line of battle when our units have low hit-points. HP_{fb} controls this "fleeing" behavior.

We encoded a candidate solution into a 60-bit string. The detailed representation of IMs, PFs, and reactive control parameters are shown in Table 3.2. Note that the sum IM is derived by summing the enemy unit IM and terrain IM so it does not need to be encoded. When the game engine receives a candidate solution, it decodes the binary string into corresponding parameters according to Table 3.2 and directs friendly units to move to a location specified by Algorithm 1 and then attack enemy units. The fitness of this candidate solution at the end of each match is then computed and sent back to our GA. Algorithm 1 shows that we first find the lowest value IM cell that contains an enemy unit, call this $Cell_t$. This cell denotes the enemy that the IM indicates is most separated from the rest of the enemies. The algorithm then finds the IM cell with the lowest value that is also nearest $Cell_t$. We call this new IM cell, $Cell_{nt}$. The algorithm then chooses $Cell_{nt}$ as the location to first move to and from which to then launch the attack. In case $Cell_{nt}$ is far from the $Cell_t$, ECSLBot may perform less well resulting in a lower fitness. The IM parameters that result in poor performance should be quickly eliminated by the GA.

3.4 Fitness Evaluation

We evolve the behaviors for fighting against melee units and ranged attack units separately. Our first fitness evaluation maximizes the damage to enemy units, min-

Table 3.2: Chromosome

	Variable	Bits	Description
IM	W_U	5	Enemy unit weight in IMs.
	R_U	4	Enemy unit influence range in IMs.
	W_T	5	Terrain weight in IMs.
	R_T	4	Terrain influence range in IMs.
PF	c_a	6	Attractor coefficient.
	c_f	6	Repulsor coefficient.
	e_a	4	Attractor exponent.
	e_f	4	Repulsor exponent.
Reactive Control	S_t	4	The stand still time after each firing. Used for kiting.
	D_k	5	The distance from the target that our unit start to kite.
	R_{nt}	4	The radius around current target. Other enemy units within this range will be considered to be a new target.
	D_{kb}	3	The distance for our unit to move backward during kiting.
	HP_{ef}	3	The hit-points of nearby enemy units, under which target will be assigned.
	HP_{fb}	3	The hit-points of our units, under which unit will flee.
	Total	60	

imizes the damage to friendly units, and minimizes the game duration in scenarios with ranged attack enemy units. In this case, a unit remaining at the end of game will contribute 100 to its own side. The fitness of an individual will be determined by the difference between the number of friendly units and the number of enemy units at the end of each game. For example, suppose three friendly Vultures and one enemy Vulture remain at the end of the game, the score will be $(3-1)*100 = 200$ as shown in the first term of Equation 3.3. Negative fitnesses when the number of enemy units is greater than the number of friendly units were not allowed to reproduce and typically stopped appearing after three generations. The detailed

evaluation function to compute fitness against ranged units (F_r) is:

$$F_r = (N_F - N_E) \times S_u + (1 - \frac{T}{MaxT}) \times S_t \quad (3.3)$$

where N_F represents how many friendly units remained, N_E is the number of enemy units remaining. S_u is the score for saving a unit (100) as defined above. The second term of the evaluation function computes the impact of game time on score. T is the time spent on the whole game, the longer a game lasts, the lower $1 - \frac{T}{MaxT}$ becomes. S_t in the function is the weight of time score which was set to 100. Maximum game time is 2500 frames, approximately one and a half minutes at normal game speed. We took game time into our evaluation because "timing" is an important factor in RTS games. Suppose combat lasts one minute. This might be enough time for the opponent to relocate backup troops from other places to support the ongoing skirmish thus increasing the chances of our player losing the battle. Therefore, combat duration becomes a crucial factor that we want to take into consideration in our evaluation function.

In scenarios against melee attack units, good players can be expected to maximize damage and destroy all opponents by kiting well. To reflect this bias towards damage we increase the weight given to destroying an enemy unit and reduce the weight for losing a friendly unit. Therefore, we want to see how many enemy units can be eliminated by friendly units during 2500 frames, we add 200 to the score for destroying an enemy unit while losing a friendly unit will subtract 150, therefore, the second melee specific fitness function (F_m) is:

$$F_m = N_E \times DS_{ET} - N_F \times DS_{FT} \quad (3.4)$$

where N_F represents how many enemy units were killed, N_E is the number of friendly units being killed. DS_{ET} and DS_{FT} are the destroy scores for the types

of unit being killed as defined in StarCraft. We apply this fitness function in experiments dealing with scenarios where we want to evaluate how fast our bots can eliminate melee attack enemy units. Although this equation does not specifically deal with time, we have 25 Zealots in this scenario and the faster you can destroy a Zealot, the more Zealots can be destroyed. This implicitly drives evolution towards parameters that lead to faster elimination of enemy units.

Summarizing, ECSLBot learns how to fight melee units in $Train_1$ shown in Figure 6.1a using fitness function F_m. ECSLBot learns how to fight against ranged units in the $Train_2$ shown in Figure 6.1b using fitness function F_r. After training and learning to handle both melee and ranged units, ECSLBot simply switches between the two sets of learned parameters, P_m and P_r, according to the current target enemy unit type (melee or ranged).

3.5 Bit Setting Optimizer and Random Flip Hill Climbers

We use the Bit-Setting Optimization (BSO) hill climber to search for a locally optimal solution by sequentially flipping each bit and keeping the better fitness solution [51]. Algorithm 2 shows the pseudo code of our BSO hill climber. BSO is defined over a Hamming space where points in the space are represented by binary strings. The performance of BSO depends on a random initial point, and it searches a local hill since they only explore closely related points in the search space. We run BSO multiple times with different initial points in an attempt to find a higher local optima or even the global optima.

The bit setting hill-climber searches only a relatively small subset of the whole search space to find local optima, and it depends heavily on the initial starting

Algorithm 2 Bit Setting Optimization Hill Climber

```
currentNode = startNode;
while number of evaluation ≤ Max do
  index = 0
  while index < LEN(currentNode) do
    nextNode = FLIP(currentNode, index++)
    if EVAL(nextNode) > EVAL(currentNode) then
      currentNode = nextNode
    end if
  end while
end while
```

point. However, Random Flip Optimization (RFO), a different hill-climber could search a different and larger space from the same initial points. Algorithm 3 shows the pseudo code for our random flip hill-climber which starts from the same set of ten initial points as our BSO.

Algorithm 3 Random Flip Optimization Hill-climber

```
currentNode = startNode;
while number of evaluation ≤ Max do
  index = RANDOM(0, LEN(currentNode))
  nextNode = FLIP(currentNode, index)
  if EVAL(nextNode) > EVAL(currentNode) then
    currentNode = nextNode
  end if
end while
```

3.6 Genetic Algorithm

We used a CHC based GA in our experiments [25, 18]. CHC selects the N best individuals from the combined parent and offspring populations ($2N$) to create the next generation after recombination. Early experiments indicated that our CHC GA worked significantly better than the canonical GA on our problem.

Algorithm 4 CHC Genetic Algorithm

```
initial population
eval(population)
while (current ≤ maxGeneration) do
    if generate offspring then
        selection(population)
        crossover(population)
        mutation(population)
    end if
    eval(offspring)
    tmpPopulation = rank (population, offspring)
    offspring = top half of tmpPopulation
end while
```

Following prior experiments in our lab, we set the population size to 20 and ran the GA for 30 generations. The probability of crossover was 88% and we used CHC selection. We also used bit-mutation with 1% chance of each individual bit flipping in value. The default SCAI was used to control the opponent force in our evaluations. Standard roulette wheel selection was used to select chromosomes for crossover. CHC being strongly elitist, helps to keep valuable information from being lost if our GA produces low fitness children. These operator choices and GA parameter values were empirically determined to work well.

3.7 Case Injected Genetic Algorithm

The CIGAR used in this paper operates on the basis of hamming distance for solution similarity [33]. Therefore, our solution similarity distance metric is computed by the following formula:

$$D(A, B) = \sum_{i=0}^{l-1} (A_i \oplus B_i) \tag{3.5}$$

where l is the chromosome length, $\oplus$ represents the *exclusive or* operator (*XOR*), and A_i represents the ith bit of solution A. Algorithm 5 shows the pseudo code for our CIGAR.

Algorithm 5 Case-Injected Genetic Algorithm

```
t = 0;
Initialize P(t);
while (current ≤ maxGeneration) do
  if (t MOD injectPeriod) == 0 then
    InjectFromCaseBase(P(t), CaseBase);
  end if
  Select P(t+1) from P(t);
  Crossover P(t+1);
  Mutate P(t+1);
  t = t + 1;
  if NewBest(P(t)) then
    CacheNewBestIndividual(P(t), Cache);
  end if
end while
SaveCacheToCaseBase(Cache, CaseBase).
```

We use a "closest to best" injection strategy to choose individuals from the case-base to be injected into CIGAR's population. We replace the four (10% of the population size) worst individuals with the individuals retrieved by our injection strategy. We chose the injection interval to be $log_2(N)$ where N is the population size. Therefore, we inject four "closest to best" cases every $log_2(40) \approx 6$ generations and replace the four worst individuals. We configured the population, selection, crossover and mutation in the CIGAR to be the same as the GA.

3.8 Parallel Genetic Algorithm

Parallel Genetic Algorithms (PGAs) are extensions of canonical GAs. The well-known advantage of PGAs is their ability to speed up the evaluation process. We

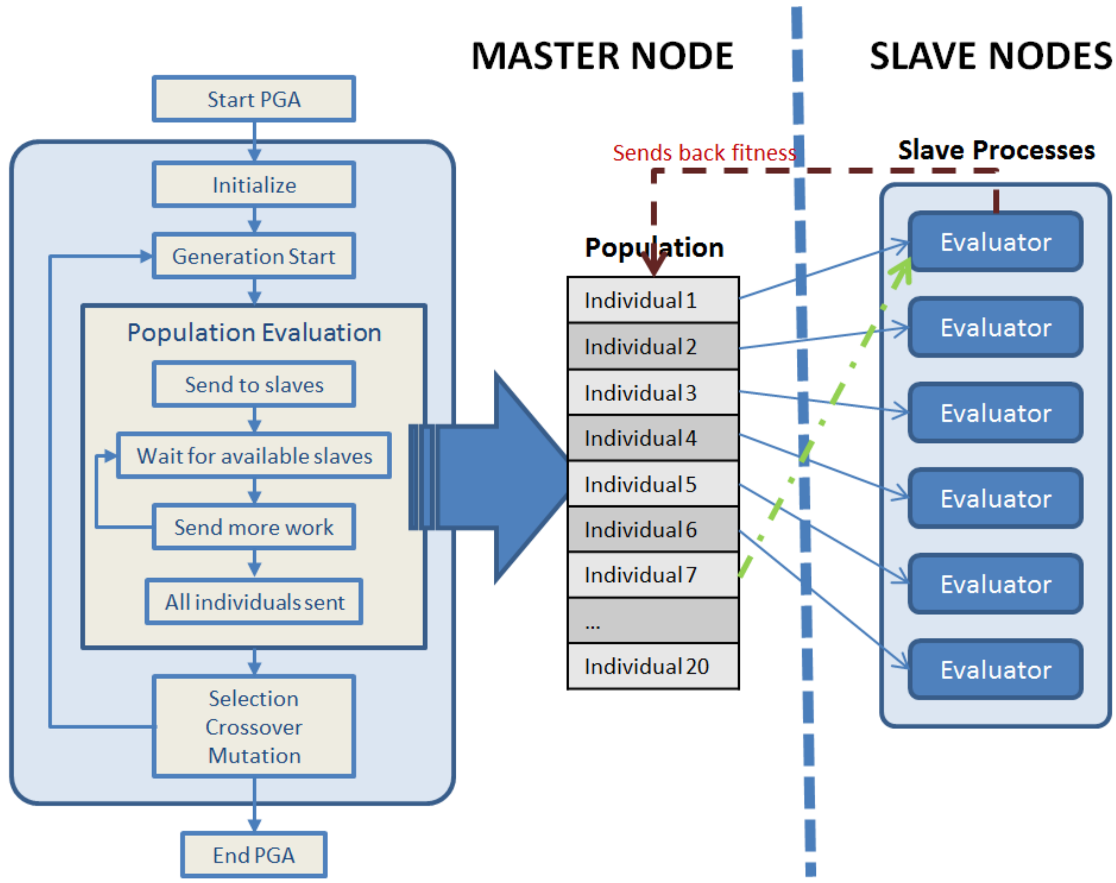


Figure 3.5: Structure of our Parallel GA.

implemented our PGA as a single population master-slave PGA. In our PGA, there is only a single panmictic population that exists on the master node, like the canonical GA. However, unlike the canonical GA, all individuals in the population are distributed to slave nodes and evaluate in parallel. Since the game evaluation is the computationally expensive part of the GA, parallel evaluation on multiple slave nodes through *SeaCraft* can linearly speedup the entire evolutionary process. The evaluation of the population is distributed on a first come first served basis. Individuals are sent to any unoccupied slave node from the master node. We use Open-MPI as our inter-processor communication backbone [20]. Figure 3.5 shows the structure and data flow of our PGA implementation.

We detailed the representation of micro behaviors in RTS games based on influence maps, potential fields, and reactive control in the first part of this chapter. We then described the heuristic search algorithms we used in our experiments including hill climbers, genetic algorithms, case-injected genetic algorithms, and parallel genetic algorithms. In the next chapter, we will discuss our results on the first set of experiments: comparing heuristic search algorithms including GAs and two type of hill climbers on finding effective micro behaviors in a StarCraft skirmish scenario.

CHAPTER 4
PHASE ONE: GENETIC ALGORITHMS VERSUS HILL CLIMBERS

The first goal of our research is finding a suitable search algorithm for searching effective solutions in our RTS combat scenarios. Therefore, we compare the quality, reliability, and robustness of solutions produced by genetic algorithms, bit-setting optimizer hill climber, and random flip hill climber. To simplify the comparison, we limited ourselves to evolve only influence maps and potential fields parameters for effective group positioning and movement in this set of experiments. We use StarCraft's built-in AI as our opponent baseline against which to make our comparisons. We represent group behaviors in combat as a combination of influence maps and potential fields parameters. We then are able to compare genetic algorithms performance versus much faster hill climbers. We also compare the performance of our micro bot running in different scenarios to investigate the robustness of the solutions produced by genetic algorithms and hill climbers. The following section details our results for comparing the quality, reliability, and robustness of solutions produced by genetic algorithms, bit-setting optimization hill climbers, and random flip hill climbers.

4.1 Experiment Settings

We used StarCraft's game engine to evaluate our evolving solutions in this set of experiments. In order to increase the difficulty and unpredictability of the game play, the behavior of the game engine was set to be non-deterministic for each game. In this case, some randomness is added by the game engine thus affecting the probability of hitting the target and the amount of damage done. This

randomness is restricted to a small range so that results are not heavily affected. These non-deterministic settings are used in ladder games and professional tournaments as well. This does not impact some scenarios such as Vultures against Zealots too much, because theoretically Vultures can "kite" Zealots to death without losing even one hit-point. But the randomness may have an amplified effect on other scenarios. For example, 5 Vultures fighting with 5 Vultures may end up with up to a 3 units difference in fitness at the end. To mitigate the influence of this non-determinism, individual fitness is computed from the average scores in 5 games. Furthermore, our results are collected from averaged scores over ten runs, each with a different random seed. The definition of game speed in BWAPI is the wait time between two consecutive frames. A number of 0 in game speed indicates that frames are executed immediately with no delay. Early experiments showed that the speed of game play affects outcomes as well. Therefore, instead of using the fastest game speed possible: 0, we set our games to a slower speed of waiting 10 milliseconds between any two frames to reduce the effect of the randomness [1].

We created a customized StarCraft map using *StarEdit*, a free tool provided by Blizzard Entertainment to build custom scenarios. The scenario contains eight Marines and one Tank on each side, as shown in Figure 4.1. A Marine has low hit-points and a short attack range but can be built fast and cheaply. A Tank is stronger than a Marine, with high hit-points and attack range but it costs more to produce. The goal of this scenario is to eliminate opponent units while minimizing the loss of friendly units as well as minimizing the game duration. In case both sides have the same number and types of units left at the end of a game, we will get a higher score for a shorter game. We used the fitness evaluation function F_r as shown in Equation 3.3 to bias the search toward the goal of this scenario.

[1] Human players play StarCraft at game speed 42 in terms of BWAPI game speed.

Figure 4.1: Scenario

4.2 Comparison in Quality and Reliability

We first compare the micro performance of solutions produced by GAs, BSO, and RFO in terms of quality and reliability. Figure 4.3 shows that the BSO HC could find good solutions 5 out of 10 times. The average score of BSO shown in Figure 4.2 climbed fast in the first 250 evaluations, and slowed down in the rest of evaluations. This tells us that the BSO could find local optima quickly, but had difficulty finding high quality, globally optimal solutions. The final average score for BSO was only 887.0. This was the lowest average score among the three tested algorithms. The RFO HC works slightly better than the BSO. Similar to BSO, RFO climbed fast in the first 250 evaluations and then slowed down for the rest. However, the RFO found high quality solutions 7 out of 10 times with the same starting points as BSO. The RFO is more reliable than BSO based on average score as shown

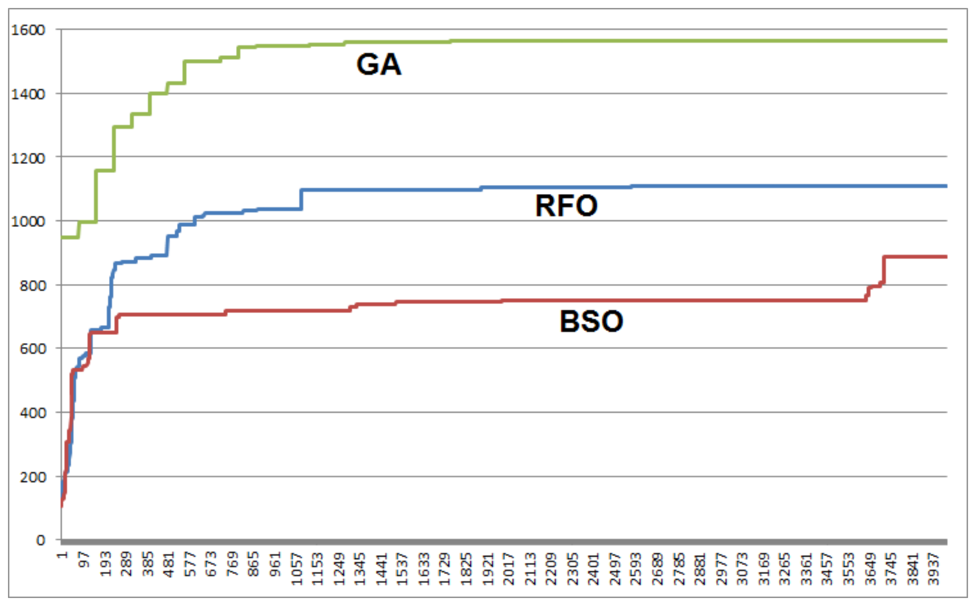


Figure 4.2: Average score of BSO, RFO, and GA over time. X-axis represents the evaluation times and Y-axis represents the average fitness evaluated by fitness function.

in Figure 4.2. Final average score was 1106.6 which was better than BSO. The best score found by RFO was 1567 and the corresponding tactic ended up with no own unit being destroyed. Furthermore the RFO was 5 seconds faster than the BSO in ending the skirmish.

We also applied GAs to compare the quality and reliability against HCs. Figure 4.2 shows the average of maximum scores in each generation. The GA converged quickly to 1500 during the first 900 evaluations, and then increased fitness slowly during the remaining evaluations. Compared to HCs, the GA always (a hundred percent of the time) found good solutions. Also the average of the best scores converged to 1566, and the best score from the GA is 1567. This indicates that every run of the GA found high quality, near-optimal solutions. Furthermore, skirmishes fought by the best solution from the GA end 5 seconds faster than those fought by the best solution produced by the RFO and 10 seconds faster than those produced by the BSO. This indicates that the skirmish finishes in a short time, and

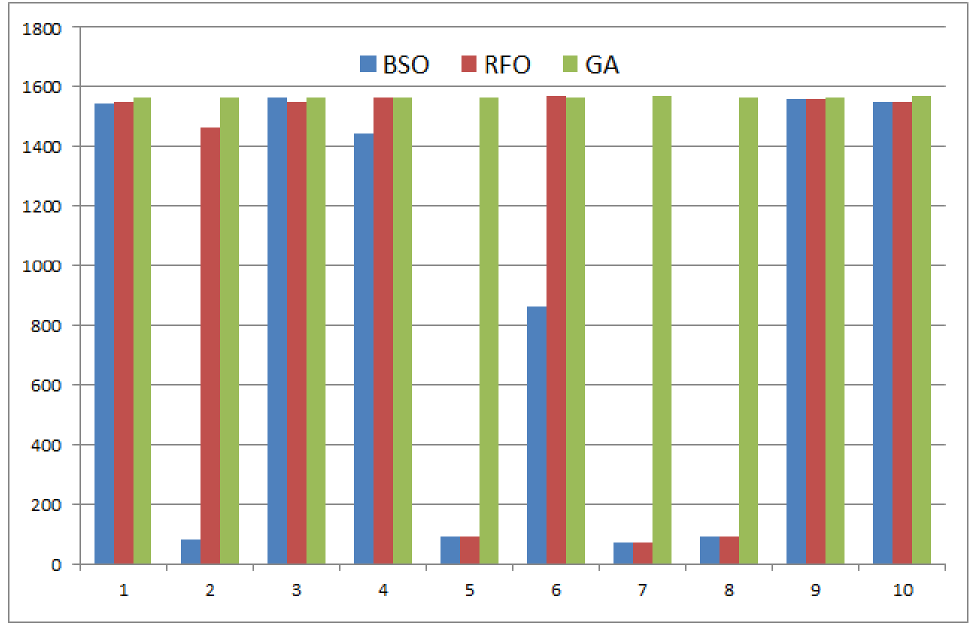


Figure 4.3: Best scores of BSO, RFO, and GA with 10 different random seeds. X-axis represents random seed and Y-axis shows the highest fitness found by each algorithm initialized with each random seed.

reduces uncertainties from opponent reactions, and increases the safety of our own units.

The above results show that both BSO and RFO can find local optima quickly against the default SCAI, but they are not guaranteed to find good solutions every time. They find good solutions between 50% to 70% of the time starting with ten different random seeds. Compared to HCs, the GA always find good combinations of IMs and PFs parameters and produce higher quality solutions compared to the HCs. However, GAs take much longer to converge. Good solutions find good unit attack positions, produce smooth unit movement, avoid unit collisions, synchronize attacking, and complete the skirmishes quickly.

4.3 Comparison in Robustness

We were also interested in the robustness of the solutions found by GAs and the two HCs from the point of view of the enemy's initial positioning. We wanted to know how our optimal solutions applied in different environments and were also curious how enemy initial position impacted our fitness scores. We designed three types of different custom maps in StarCraft in which the enemy were initially well dispersed, well concentrated, or in an intermediate position. The intermediate map was used for all prior results above. Table 4.1 specifies these initial enemy positions. These three types of scenarios can usually be found in human player matches. Dispersed units have less concentrating fire power but more map control and information gain. However, concentrated units have less map control but are harder to destroy.

We applied the solutions obtained from our initial intermediate scattered map to the two other maps. Each map was tested 500 times to get the average scores and their standard deviations. Table 4.1 shows these test results. The optimum was obtained from a GA running in intermediate initial position and had the highest fitness of 1567. We tested this solution 500 times and on average obtained a fitness 1380.728 on the intermediate map. The standard deviation over all 500 tests on this map was 198.9 indicating the average error is within 2 Marines. We tested the same solution in a map with dispersed enemy units initial positions. The average fitness of 500 tests on this map was 1423.364. This is a higher score on a map never seen by the GA. The reason for the higher average score is that enemy units are dispersed and can be eliminated one by one with very little damage. This tells us the optima we get from intermediate scattered position could work well or even better with more scattered enemy units. This also showed how group position-

Table 4.1: Average fitnesses and standard deviations of 500 matches on three maps with different initialized enemy units' position. Dots on the left side of the map represent the friendly units, and dots on the middle of the map represent the enemy units.

Enemy Initial Position	Description	Fitness
	Intermediate enemy position initialized, maximum distance from 2 units is 6 IM cells, which is also the default map for all the HCs and GAs experiments.	1380.7 σ = 198.9
	Dispersed enemy position initialized, maximum distance from 2 units is 11 IM cells. New scenario added to test robustness of the solution of previous map.	1423.4 σ = 62.6
	Concentrated enemy position initialized, maximum distance from 2 units is 3 IM cells. New scenario added to test robustness.	181.8 σ = 346.1

ing is important in combat. The standard deviation of the tests on this map was low at 62.6. This matches our intuition that dispersed units are easily destroyed one by one quickly without much damage to the opponent, and our group with concentrated units has more concentrated fire power to damage the enemy. On the other hand, the average fitness of the tests on a map with concentrated units is low at 181.838. This means the matches were even on this map and only one or two units survived on average in 500 tests. The Marines and Tank were not able to synchronize their movement to confront enemy units at the same time, while concentrated enemy units could maximize their damage by firing at the same time. The standard deviation is 346.1, and is the highest on the three types of tests.

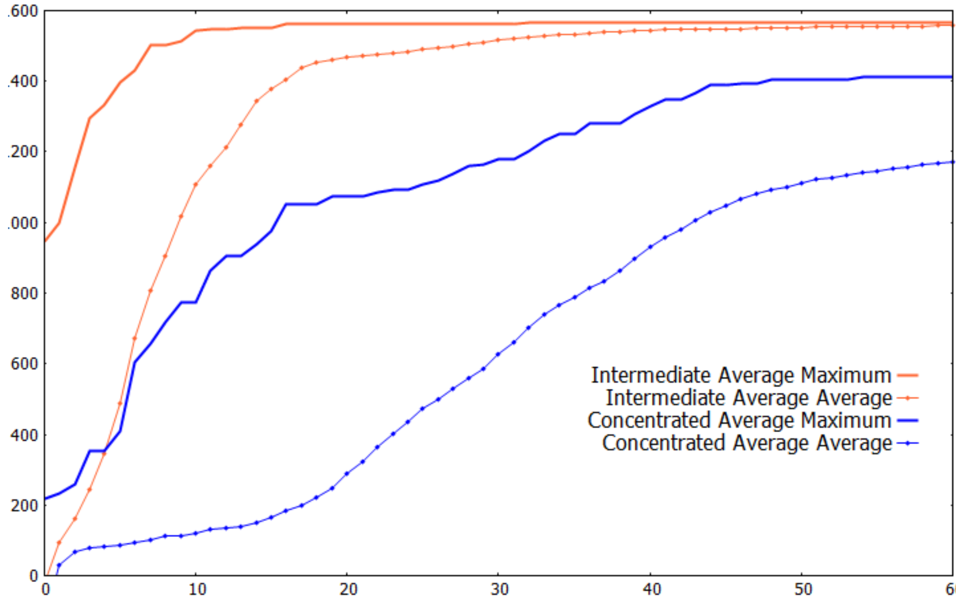


Figure 4.4: Average maximum and average fitness of GA running on two types of map. X-axis represents generation, and Y-axis represents fitness.

For comparison, Figure 4.4 shows the performance of the same GA running on the intermediate scattered map and concentrated map. The graphs show that the enemy group with more scattered units is easier to be eliminated and faster for the GA to find quality solutions. It converged in the 20_{th} generation. However, the enemy group with more concentrated units got lower fitness because the enemy could damage opponent units more. The convergence rate is also slower than scattered units because the quality solution is harder to find.

4.4 Evolved Group Micro Behaviors

From the point of view of IMs and PFs we can use IMs for guiding our units' positioning and move smoothly to attack the opponents units using PFs to guide our movement. Figure 4.5 shows an example of generated IMs positioning. The units could take advantage of this positioning to concentrate their fire and maximize

their damage to the opponent. The group positioning and movement that evolves first learns to ensure that single units stay away from enemy unit controlled territory or to move outside of the map. If the enemy repulsor force is too small, units might move into enemy territory and be destroyed. On the other hand, if the force is too large, it will push the units to the border of the map and lead to avoiding the enemy altogether. Second, the parameters for the IMs were learned to guide our unit's positioning. The IM calculated the enemy's weak spots from the current position of enemy units and generates attraction points to guide our units in preparing for the skirmish. Different IM parameters lead to different locations, if the IM range of Marine and the IM range of Tank are small, the locations might be inside the enemy units' attack range. If they are too large, the units may spend more time on the way and result in longer games, and low S_t. The enemy replusor and friend attractor were learned last. This affects detailed unit movement. Good combinations of attractors and replusors allow the group to move and attack smoothly and effectively. Units move to the right locations quickly and destroy enemy units faster. At the same time our units have more opportunity to survive. Therefore, our evaluation function is biased towards short movement, more enemy units eliminated, more own units survival, and shorter game duration.

4.5 Conclusions

This chapter compared GAs with two HCs to generate group positioning and unit movement based on influence maps and potential fields for beating opponents in a typical skirmish scenario in RTS games. We used StarCraft's built-in AI as our opponent baseline against which to make our comparisons. We represented

Figure 4.5: A snapshot of one group positioning in StarCraft minimap. The dots on the map represent friendly units, and other part of the map was covered by fog of war. Single dot at the left of the map is Tank, and other dots are Marines.

group behaviors in a combat as a combination of IMs and PFs parameters and restricted our search space to 2^{48}. We were able to compare GA performance versus much faster BSO and RFO hill climbers. Results show that both BSO and RFO HCs can find local optima quickly against the baseline, but they are not guaranteed to find good solutions every time. They find good solutions between 50% to 70% of the time starting with ten different random seeds. Compared to HCs, the GA always find good combinations of IMs and PFs parameters and produce higher quality solutions compared to the hill-climbers. However, GAs take much longer to converge. Good solutions find good units attack positions, produce smooth unit movement, avoid unit collisions, synchronize attacking, and complete the skirmishes quickly. We also compared the performance of GAs running in different scenarios for testing robustness of the solutions found by GAs and HCs. The result shows that we could apply the solutions found in one scenario to more dispersed enemy units' position with high fitness. However, these solutions do not do well against more concentrated enemy positions.

The results in this chapter indicate that our hill-climbers were quick but unreliable while the genetic algorithm was slow but reliably found quality solutions a hundred percent of the time. Therefore, we are interested in techniques which can reliably and quickly find high quality solutions in skirmish scenarios in RTS games. The next chapter will investigate case-injected genetic algorithms which are designed to learn from experience to increase problem solving performance on similar problems.

CHAPTER 5

PHASE TWO: CASE-INJECTED GENETIC ALGORITHMS

In the previous chapter, we investigated applying different heuristic search algorithms for generating effective micro behaviors in our skirmish scenarios in an RTS game StarCraft. We compared the quality, reliability, and robustness of solutions produced by GAs, BSO, and RFO HCs. The results showed that HCs can find serviable IM and PF parameters that can occasionally defeat the default SCAI, but they are not guaranteed to find good solutions every time. Compared to HCs, the GAs always find good combinations of IMs and PFs parameters and produce higher quality solutions, but take much longer to converge. Therefore, we are interested in techniques that can reliably and quickly find high quality solutions. In this chapter, we investigate applying case injected genetic algorithms to learn from "experiences" generated from previous problems and use this information to bias our search and speed up the process of finding high quality solutions. We defined five similar scenarios with different difficulty levels that are similar to scenarios in typical human player matches. We applied CIGARs to these five sequential scenarios to assess how "experience", stored as cases in a case-base, affects performance compared to a GA. GAs with exactly the same parameters as used by CIGAR thus served as a baseline.

5.1 Experiment Settings

To evaluate the performance of genetic algorithm with case injection and without case injection, we designed five different but similar scenarios. The difference between scenarios is the initial positions of enemy units'. Our five scenarios are:

- **Intermediate Position:** Enemy units located in the middle of the map. The maximum distance between any two enemy units is six IM cells.
- **Dispersed Position:** The maximum distance between any two enemy units is eleven IM cells. Enemy units in this scenario have the most scattered positions and the weakest concentrated fire power of all the five scenarios.
- **Concentrated Position:** Enemy units located in the middle of the map, and the maximum distance between any two enemy units is three IM cells. Enemy units in this scenario have the most concentrated fire power.
- **Corner Position:** Enemy units located at the northeast corner of the map, concentrated as much as in the previous scenario. Enemy units in this scenario have the strongest concentrated fire power as well as the best defensive positions.
- **Split Position:** Enemy units located in the middle of the map and split into two groups. Enemy units in this scenario have stronger concentrated fire power than Dispersed and Intermediate positions but weaker concentrated fire power than Concentrated and Corner positions.

These five types of scenario can usually be found in human player matches. Dispersed units have less concentrated fire power but more map control and information gain. On the other hand, concentrated units have less map control but are harder to destroy. Since our experiments do not have fog of war, the advantage of dispersed enemy units do not apply in this case resulting in making the more concentrated enemy units harder to destroy. Therefore, the first two scenarios and the last one are relatively easy for GAs to find high quality solutions to; scenario three and four are harder.

According to the evaluation function F_r defined in Equation 3.3 and the five scenarios we introduced above, the theoretic maximum score for eliminating enemy forces is 1500 and maximum time score (corresponding to minimal time) is 100, therefore, the maximum evaluation score or fitness is 1600. Note that the first two digits in an evaluation score represent the unit elimination score, and the last two digits represent time score. For example, if the final score ends up at 1357 we can infer the following:

- The score being positive means our AI player defeated the built-in AI.
- 1300 represents the unit elimination score and compared to the maximum of 1500, our AI player lost 200 which indicates that two Marines were killed by the enemy during the game.
- The last two digits being 57 represents $(1 - \frac{57}{100}) \times 2500 = 1075$ frames spent during the entire game which is approximately 38.4 seconds converted to standard game speed.

5.2 Case Injection's Effect on Genetic Algorithms

In our experiments, we tested GAs and CIGARs running with ten different random seeds. Each such test took 14 hours to run the $40 \times 60 \times 5 = 12,000$ evaluations, where 40 is population size, 60 is the number of generations to run, and 5 represents the five scenarios. Scenarios are tested sequentially in the following order. Intermediate, Dispersed, Concentrated, Corner, and Split. CIGAR extracts the best individual in each generation and stores this individual into the case-base; duplicates are discarded. When running on a problem, suitable cases chosen according

to the “closest to the best” strategy from the case-base are injected into CIGAR’s population. Each case may contain useful information about the new search space and be a partial solution to the current problem and thus bias the genetic algorithm to take advantage of “good” genes found by previous search attempts. The number of cases in the case-base usually increases with the number of problems solved.

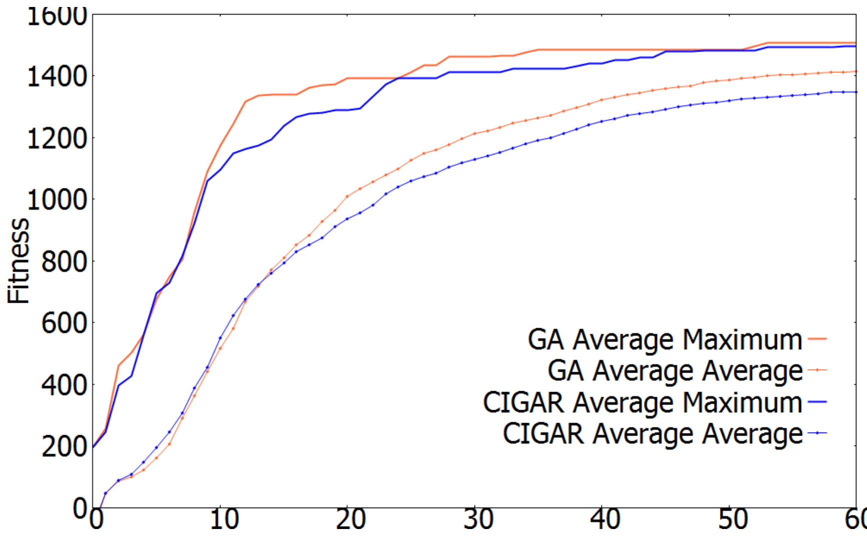


Figure 5.1: Average maximum/average scores of GA and CIGAR over 10 runs on Intermediate scenario. The X-axis represents the generation and the Y-axis represents the fitness.

The performance of GAs and CIGARs running in the Intermediate scenarios as shown in Figure 5.1 tell us that their performance is similar. This is because the Intermediate scenario is the first problem to be attempted and CIGAR has no cases in its case-base when running on this problem. They are not exactly the same because there is some randomness in the game evaluation as explained earlier. On the other hand, the results from the Concentrated scenario show the difference in performance (see Figure 5.2). CIGAR has solved two problems (Intermediate and Dispersed scenarios) before this scenario, and 12.2 cases on average (over the ten runs) exist in the case-base. We can see that CIGAR outperformed the GA both

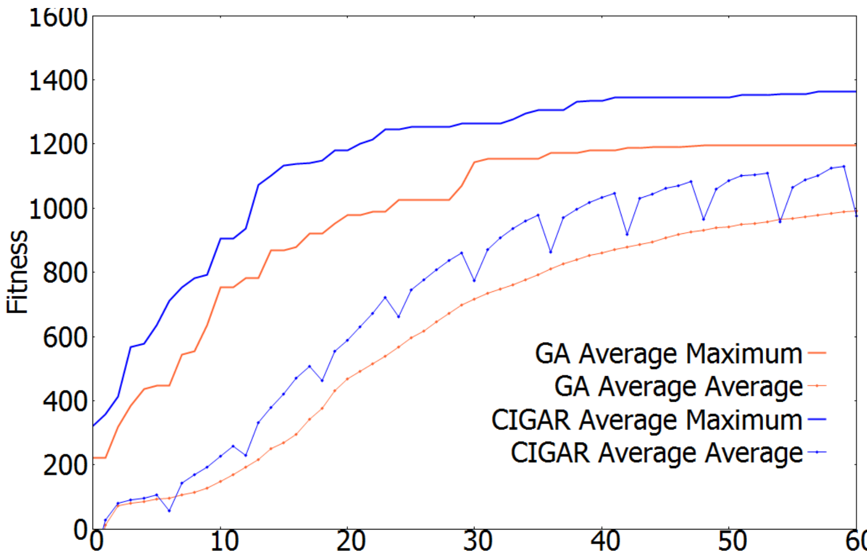


Figure 5.2: Average maximum/average scores of GA and CIGAR over 10 runs on Concentrated scenario. X-axis represents the generation and Y-axis represents the fitness.

in quality and speed in the Concentrated scenario when CIGAR's case-base contains cases from previous scenarios. The curve for CIGAR's average fitness in the Concentrated scenario dropped a little every 6 generations because four cases from the case-base were injected into the population. The new cases may only contain partial solutions with lower fitness in the population, which cause the average fitness to drop. However, average fitness rises again quickly after the drop. This shows the cases injected into the population may have introduced useful information leading to better performance.

Figure 5.3 compares the quality of solutions found by the GA and CIGAR in all five scenarios. The number on top of each bar for CIGAR shows the average number of cases when CIGAR starts in the corresponding scenarios. The case-base is empty on the Intermediate (first) scenario, and increases to 27.1 on the last scenario. Therefore, CIGAR's "experience" generated from solving problems increases scenario by scenario. Both GA and CIGAR reliably found quality solutions

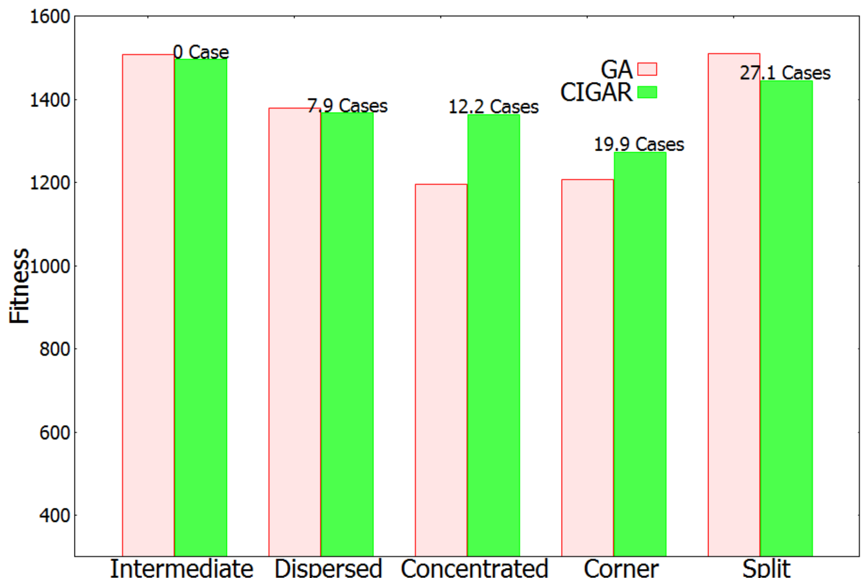


Figure 5.3: Solution quality of each scenario. As more problems are solved, CIGAR produces better solutions than genetic algorithm. The X-axis represents 5 different scenarios. The Y-axis represents the highest fitness. The number on top of each bar of CIGAR shows the number of cases in case-base when CIGAR starts.

above fitness 1200 a hundred percent of the time. The first two scenarios and the last one show that the GA and CIGAR found similar quality solutions. The reason behind this is that the Intermediate, Dispersed and Split scenarios are relatively easy to solve because scattered enemy units are easily destroyed one-by-one quickly without much damage to the opponent. Therefore, both GA and CIGAR performed very well. On the other hand, the Concentrated and Corner scenarios show the difference in performance between the GA and CIGAR. Concentrated enemy units have stronger fire power than scattered units which leads to high quality solutions being harder to find by GAs in the search space. In this case, we believe CIGAR had an advantage and case-injection biased search to more quickly find solutions with higher fitness.

We wanted techniques for finding high quality solutions quickly and reliably for winning a skirmish in an RTS game. Thus, we measured the number of gener-

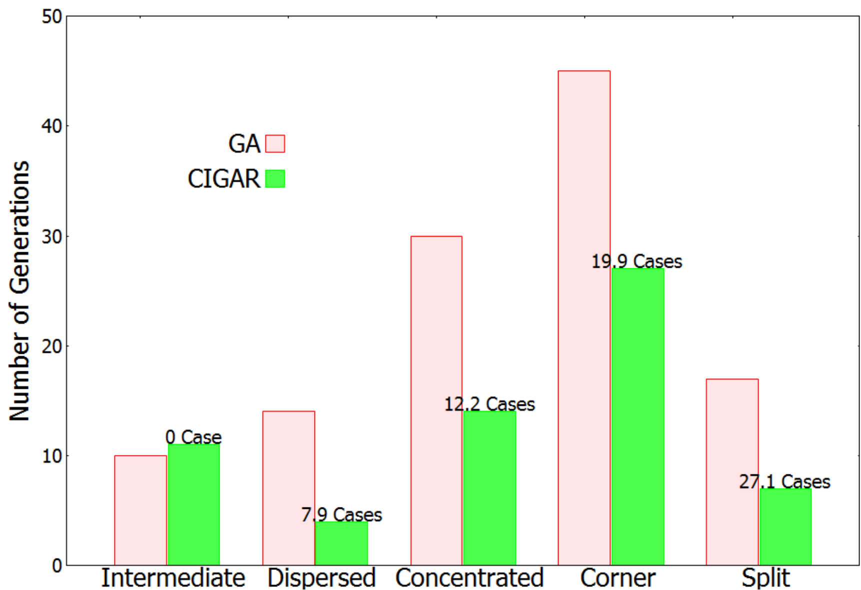


Figure 5.4: Number of generations to solutions found above 1100. As more problems are solved, CIGAR took less time compared to the GA.

ations our GA and CIGAR took to produce quality solutions with fitness above a threshold quality of 1100. Figure 5.4 shows the number of generations needed to find quality solutions above 1100 from our GA and CIGAR. The GA ran on each scenario without any bias from injected cases and so GA performance can be used to indicate a level of difficulty for each scenario. A low number of generations indicates that the GA finds quality solutions easily. A high number of generations indicates the GA has a hard time finding quality solutions. So we can think of the GA performance in Figure 5.4 as showing us the difficulty levels of our five scenarios. In order from easy to hard, the scenarios are: Intermediate, Dispersed, Split, Concentrated, and Corner. The first scenario's result shows CIGAR found quality solutions 1 generation on average slower than our GA because CIGAR's case-base is empty. However, after CIGAR runs on the first scenario, on average 7.9 cases are stored into the case-base. CIGAR only takes 4 generations to find solutions with fitness greater than 1100 on this second scenario (Dispersed), while our GA takes 10 generations. Having found quality solutions for another scenario, the number

of cases in our case-base increased again. CIGAR finds quality solutions for the third scenario in 14 generations compared to the GA which takes 30 generations. CIGAR found quality solutions 21 generations faster for the fourth scenario (the most difficult for the GA), and 10 generations faster for the last scenario.

5.3 Conclusions

This chapter focuses on applying case injected genetic algorithms to generate group positioning and unit movement in order to win skirmish scenarios in RTS game. The results in Chapter 4 showed that hill-climbers can find serviable IM and PF parameters that can occasionally defeat the default AI, but they are not guaranteed to find good solutions every time. Compared to hill-climbers, the genetic algorithms always find good combinations of IMs and PFs parameters and produce higher quality solutions, but take much longer to converge. Therefore, we are interested in techniques that can reliably and quickly find high quality solutions. In this chapter, we investigated applying case injected genetic algorithms to learn from "experiences" generated from previous problems and use this information to bias our search and speed up the process of finding high quality solutions. We defined five similar scenarios with different difficulty levels that are similar to scenarios in typical human player matches. We applied CIGARs to these five sequential scenarios to assess how "experience", stored as cases in a case-base, affects performance compared to a GA. GAs with exactly the same parameters as used by CIGAR thus served as a baseline.

The results show that CIGARs performed similar to our GAs in the first scenario when the case-base is empty. In scenarios with more scattered enemy units,

including scenarios one, two, and five, which are relatively easy problems, both GAs and CIGARs found high quality solutions. However, CIGARs find high quality solutions up to twice as fast as GAs. Finally, in scenarios two and three, with more concentrated enemy units, CIGARs not only find higher quality solutions than GAs, but also doubled the speed of finding a quality solution above 1100. This indicates that CIGARs are a suitable technique to apply across similar problems in RTS games. In addition, these "experiences" generated from solved problems provided valuable information and could help to speed up solving other similar problems.

After the results showed that GAs and CIGARs are effective on producing high quality micro behaviors, we extend our representation to cover not only IMs and PFs but also reactive controls in the next chapter. We will compare the micro performance of our evolved ECSLBot against two state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in StarCraft.

CHAPTER 6
PHASE THREE: ECSLBOT VERSUS THE STATE OF THE ART BOTS

Since the results in Chapter 4 and 5 indicate that the GAs and CIGARs produced higher quality solutions more reliably and faster, we settled on using GAs and CIGARs to search for effective micro parameters in the rest of this work. We extended our representation to cover not only group tactics and movement but also reactive control behaviors including kiting, target selection, and fleeing. We test the micro performances of our evolved ECSLBot against two state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in StarCraft. We designed two training scenarios and eight testing scenarios in which bots need to control a number of Vultures against different types of enemies, to evaluate micro performance.

6.1 Experiment Settings

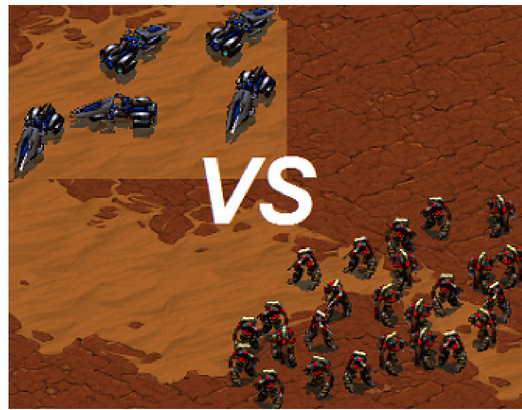
We created a series of customized StarCraft maps as our *training* scenarios and *testing* scenarios to evolve and evaluate our ECSLBot. As explained earlier, we want ECSLBot to learn to control a specific type of Terran unit (Vulture) to fight against different types of enemy units in these predefined scenarios. More specifically, we evolve Vulture kiting behavior against melee enemy units, and target selection and fleeing against ranged enemy units. Melee and ranged units are the two broad types of units in RTS games like StarCraft. After learning how to play against different types of enemies in two training scenarios, we test the generalizability of learned behaviors on eight previously unseen testing scenarios. Finally, we compare our ECSLBot, Nova, and UAlbertaBot against each other using Vultures.

Players in RTS games are usually not able to access complete state information because of the "fog of war" as described in Section 2.7.1. However, the "fog of war" influences longer-term planning for distant units, while we are focused on short-term planning for units in close proximity. Furthermore, good human players will start a skirmish only when they already have enemy and terrain information through scouting. Therefore, we allow all bots to access complete state information for all scenarios in this research. In another word, there is no "fog of war" in our scenarios.

6.1.1 Training Scenarios

Human players usually use different micro behaviors against melee units than they use against ranged units. Therefore, we evolve our bot against exemplars of these two broad types of enemy units, separately. In this set of experiments, ECSLBot learns how to control Vultures to defeat melee enemy units on a scenario containing 5 friendly Vultures and 25 opponent Zealots (a Protoss melee unit) as shown in Figure 6.1a. We call this training scenario, $Train_1$ and the parameters evolved on this scenario, P_m. The game runs to evaluate fitness and evaluation ends after 2500 frames. The goal of this scenario is eliminating as many enemy Zealots as possible. Kiting efficiency is important in this type of combat and will be evolved by our GAs.

Our second scenario was created for fighting against ranged attack units. We call this scenario, $Train_2$, and the parameters evolved here, P_r. Figure 6.1b shows that our ECSLBot controls 5 friendly Vultures positioned at the top left to fight against 6 enemy Vultures positioned at the bottom right. Positioning and target



(a) $Train_1$: A bot controlling 5 Vultures at top left fighting against 25 Zealots at the bottom right in 2500 frames. The score will be higher with more Zealots killed.

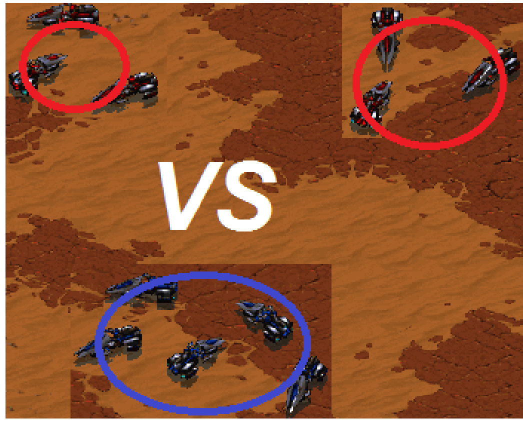
(b) $Train_2$: A bot controlling 5 Vultures at top left fighting against 6 Vultures at the bottom right. The score will be higher with more Vultures killed and shorter of the game duration.

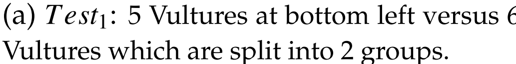
Figure 6.1: Training scenarios.

selection become key contributors in this scenario. $Train_2$ also runs for 2500 frames or until one side is eliminated.

6.1.2 Testing Scenarios

We evolve micro behaviors for fighting against melee and ranged enemy units in the previous two training scenarios. However, we are interested in evaluating the generalizability of our evolved behaviors on scenarios never encountered before. Therefore, we created eight new testing scenarios with more units, mixed types of opponent units, and different terrain. We expect ECSLBot evolved on two simple training scenarios to perform similarly in other unseen situations because our evolved parameters represent a range of behaviors that are relatively position and terrain independent and simply switching between parameter sets takes into account the two broad opponent types in RTS games. We first test our evolved ECSLBot on the two test scenarios shown in Figure 6.2. Here, friendly Vultures





(a) $Test_1$: 5 Vultures at bottom left versus 6 Vultures which are split into 2 groups.

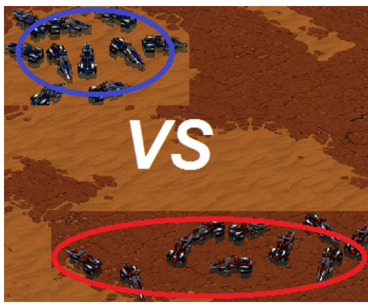
(b) $Test_2$: 5 Vultures at bottom left versus 25 Zealots which are split into 2 groups.

Figure 6.2: Testing scenarios where only the initial position of units changed.

spawn at the bottom left of the map instead of top left. Enemy units spawn at the top of the map and are split into two groups. These two scenarios ($Test_1$ and $Test_2$) test whether ECSLBot is able to adapt when the initial positions of friendly units and enemy units change.

Next, we considered four scenarios where we change the number of units controlled by our bots. We evolved ECSLBot for controlling five Vultures against enemy units. We now investigate how ECSLBot performs when controlling more than five Vultures. Figure 6.3 shows four scenarios ($Test_3$, $Test_4$, $Test_5$, and $Test_6$) where our ECSLBot controls ten Vultures instead of five, to fight against differing numbers of enemy units. Furthermore, we consider more complex terrain. $Test_6$ contains an untraversable block in the middle of the map as shown in Figure 6.3d. Results show that ECSLBot learned to avoid map boundaries while kiting during training, and this generalized to the case of the never before encountered untraversable block in the center of $Test_6$. ECSLBot adapts to the block in the center and kites well avoiding the block.

Finally, in the last two scenarios we evaluate generalizability against mixed op-



(a) $Test_3$: 10 Vultures at top left versus 10 Vultures at the bottom right of the map.

(b) $Test_4$: 10 Vultures at top left versus 12 Vultures at the bottom right of the map.

(c) $Test_5$: 10 Vultures at top left versus 40 Zealots which are spread into 3 groups at the bottom of the map.

(d) $Test_6$: 10 Vultures at top left versus 40 Zealots at bottom right with unwalkable area in the middle of the map.

(e) $Test_7$: 10 Vultures at top left versus 8 Zealots and 8 Vultures at the bottom right of the map.

(f) $Test_8$: 10 Vultures at top left versus 18 Zerglings and 10 Hydralisks at the bottom right of the map.

Figure 6.3: Testing scenarios with ten friendly Vultures and different types of enemy units.

ponent types. ECSLBot simply switches between P_m and P_r, the two evolved parameter sets according to whether the current target unit is melee or ranged respectively. Figure 6.3e shows the testing scenario, $Test_7$, where ECSLBot controls ten Vultures and fights against eight Zealots and eight Vultures. Since we were particularly interested in the performance of ECSLBot fighting against mixed opponent units, we also created the last scenario, $Test_8$, shown in Figure 6.3f. ECSLBot controls ten Vultures to fight against eighteen *Zerglings* and ten *Hydralisks* which are melee and ranged unit types respectively, from a different StarCraft Race (Zerg). These units have different melee damage and different weapons range when compared to Zealots and Vultures.

6.1.3 Head-to-head Scenario

After we compared our ECSLBot, UAlbertaBot, and Nova on a variety of testing scenarios where our bots control different number of Vultures against different types of enemy units controlled by the default SCAI, we were also interested in how they perform when competing against each other with identical units. A new scenario was designed for this comparison where all three bots control five Vultures against each other.

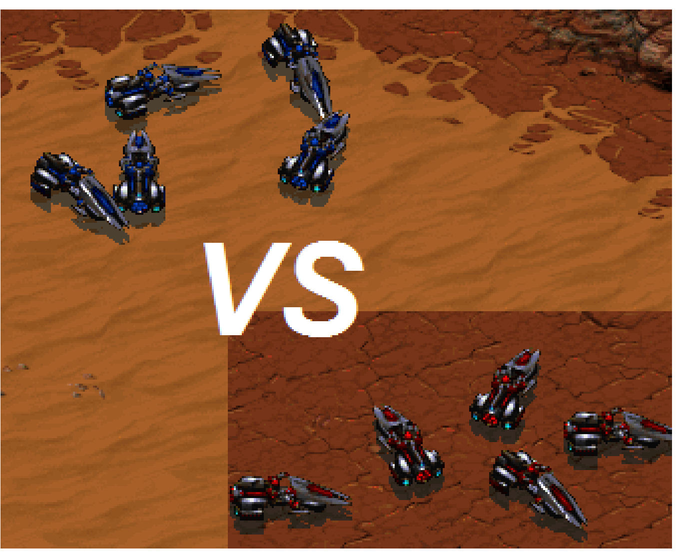


Figure 6.4: The scenario for head to head evaluation.

6.2 Evolved ECSLBot

Scenario $Train_1$ as shown in Figure 6.1a evaluates the efficiency of kiting behavior against melee attack units. F_m as defined in Equation 3.4 is used as our evaluation function in this scenario. Figure 6.5 shows the average scores of ECSLBots running on this kiting scenario. We can see that the maximum fitness in the initial population is as high as 3217, which means our bot eliminated 16 Zealots within

2500 frames. However, the average of maximum fitness increases slowly to 3660 at generation 30, which is 18 Zealots. This results tell us that our GAs can quickly find a kiting behavior to perform "hit and run" against melee attack units while trading off damage output. Our ECSLBot trades off well between kiting for safety and kiting for damage to enemy.

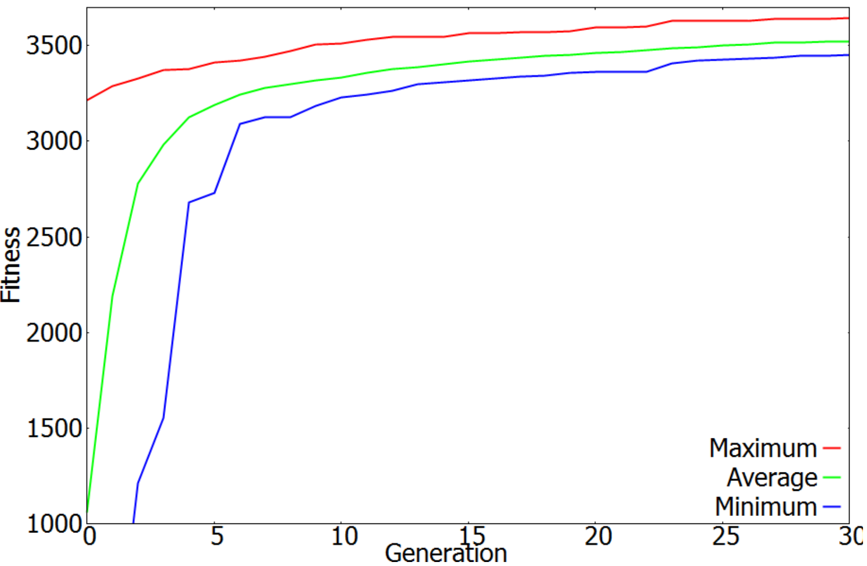


Figure 6.5: Average score of ECSLBot versus generations on scenario $Train_1$. X-axis represents time and Y-axis represents fitness by the fitness function F_m.

Besides performance against melee attack units, we are also interested in performance against ranged attack units. In this case, positioning and target selection become more important than kiting because the additional movement from kiting behavior will waste enemy damage output while avoiding enemy attack. We used our GAs to search for effective micro behaviors using the same representation as in the previous scenario. However, we changed our fitness evaluation function to F_r as shown in Equation 3.3 to maximize killing of enemy units, minimize the loss of friendly units, and minimize combat duration. Figure 6.6 shows the average score of the evolving ECSLBots in scenario $Train_2$. The average maximum fitness found by GAs is 336, which means 3 friendly units remained at the end of the game and

all enemy units were eliminated. Considering that the Vulture is a vulnerable unit and easily dies, 3 Vultures saved after a skirmish is, we believe, good performance.

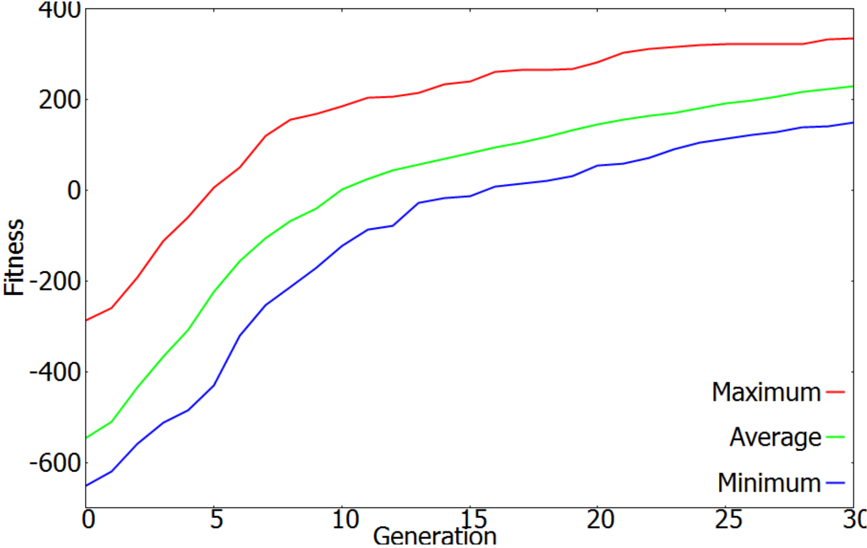


Figure 6.6: Average score of ECSLBot over generations in scenarios $Train_2$. X-axis represents time and Y-axis represents fitness by the fitness function F_r.

We are interested in the differences in evolved parameters for the two training scenarios - against melee attack units and ranged attack units. Table 6.8 lists the details of optimal solutions in different scenarios. Videos of all learned micro behaviors can be seen online [1]. We would like to highlight two findings in these results. The first concerns the learned optimal attack route in the scenario against six Vultures as shown in Figure 6.7. The IM parameters evolved by the GA and our control Algorithms 1, lead to a gathering location at the left side of the map to move toward before the battle. Our ECSLBot then commands the five Vultures to follow this route to attack enemy units. The result is that only three of the enemy units are triggered in the fight against our five Vultures at the beginning of the fight. This group positioning helped ECSLBot minimize the damage taken from enemy units while maximizing damage output from outnumbered friendly units.

[1] http://www.cse.unr.edu/~simingl/publications.html

Table 6.1: Parameter values of best evolved individuals.

Scenario	IM				PF				Reactive Control					
$Train_1$, 25 Zealots	3	9	15	8	43	55	6	2	1	5	6	7	7	0
$Train_2$, 6 Vultures	16	13	20	10	50	26	13	4	12	9	1	7	6	7

Although we describe a behavior that is specific to this scenario, the IM parameters tend to guide our units to such vulnerable positions with respect to enemy forces located anywhere on any map. As performance in never-before-seen testing scenarios shows, using an IM helps generalizability of learned behaviors. This is detailed in Section 6.4 where our forces move to a location that the IM indicates is a location where enemy forces are less concentrated.

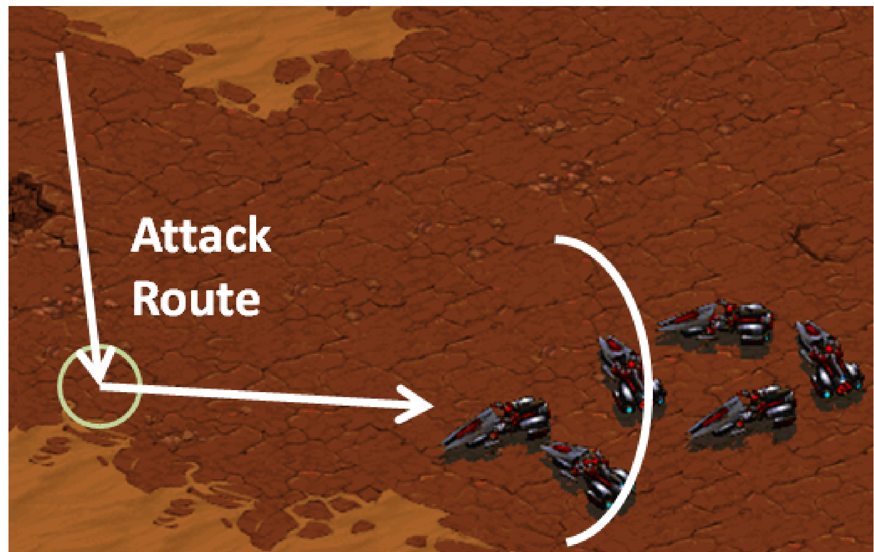


Figure 6.7: Learned optimal attacking route against 6 Vultures.

The second interesting finding is that different micro behaviors are learned by ECSLBot in different scenarios. Figure 6.8 shows that our ECSLBot kited heavily against Zealots as shown on the left side, but seldom move backward against ranged attack units as shown on the right side. The values of our parameters reflect this behavior. Table 6.1 shows the parameter values found by our GAs in the two

training scenarios. We can see that S_t (the first parameter in the reactive control section) is 1 frame in the scenario against melee attack units, which means a Vulture starts to move backward right after every shot. On the other hand, S_t is much bigger (12 frames) against ranged attack units. This is because our units will gain more benefit after each weapon firing by standing still and firing again as soon as possible rather than moving backward immediately against ranged attack units.

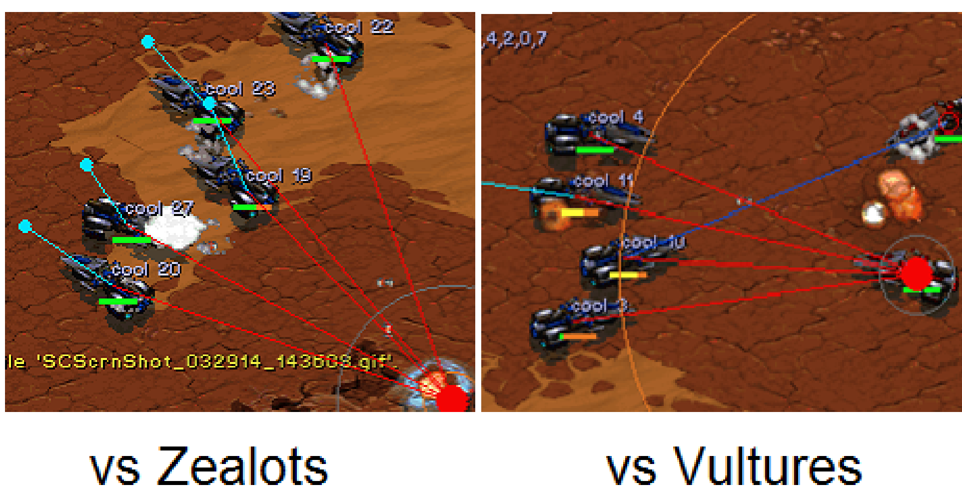


Figure 6.8: Learned kiting behaviors against Zealots and Vultures. The left side of the figure shows that our Vultures are moving backward and are pointed away from the enemy to kite enemy Zealots. The right side shows that our Vultures are facing the enemy Vultures with only one friendly Vulture moving backward to dodge.

6.3 ECSLBot versus the State of the Art Bots in Training Scenarios

Next, we investigate the performance differences among our ECSLBot (the best bot evolved by GAs), UAlbertaBot, and Nova in the two training scenarios. We used UAlbertaBot and Nova to control the same number of Vultures (5) against 25 Zealots in training scenario $Train_1$. Table 6.2 shows the results for all three bots versus the baseline SCAI over 30 runs in the training scenario $Train_1$. We can see

Table 6.2: The performance of bots controlling 5 Vultures vs 25 Zealots units in scenario $Train_1$.

	Win	Draw	Lose	Avg Killed / σ	Avg Left / σ
UAlbertaBot	0	0	30	3.33 / 2.1	0 / 0
Nova	30	0	0	20.03 / 2.37	4.7 / 0.4
ECSLBot	30	0	0	20.2 / 2.57	4.8 / 0.4

Table 6.3: The performance of bots controlling 5 Vultures vs 6 Vultures in scenario $Train_2$.

	Win	Draw	Lose	Avg Killed / σ	Avg Left / σ
UAlbertaBot	0	0	30	2.67 / 0.54	0 / 0
Nova	2	0	28	3.13 / 1.45	0.13 / 0.56
ECSLBot	18	0	12	5.2 / 1.35	1.8 / 1.7

that the UAlbertaBot performed poorly against melee attack units. This seems to be mainly because UAlbertaBot uses the same logic for all its units and the logic is optimized only for Protoss units. It eliminated only 3.33 Zealots on average in each game, while losing all of its Vultures. Note that UAlbertaBot was designed to play as Protoss. On the other hand, Nova's performance is good. Nova killed 20.03 Zealots and lost only 0.3 Vultures on average per run. This is because Nova has hard coded and tuned logic specifically for Vultures and is optimized to control Vulture kiting behavior against melee attack units. We then tested ECSLBot on scenario $Train_1$. The results show that ECSLBot got the higher score on average over 30 runs. 20.2 Zealots being killed in one match on average, while losing only 0.2 Vultures. Visually, ECSLBot and Nova seem to have very similar kiting behavior and performance and statistically, the difference in performance is not significant at $P = 0.795$ using the t-test.

Table 6.3 shows the results from all of our three bots tested in scenario $Train_2$.

All the bots run 30 times against the default SCAI. This time, both UAlbertaBot and Nova perform poorly. UAlbertaBot loses all 30 games against 6 Vultures, killing 2.67 enemy Vultures on average in each game, while losing all of its units. Nova performed slightly better than UAlbertaBot with 2 wins and 28 losses out of 30 runs. However, ECSLBot outperformed both the others with a 60% win rate. 5.2 enemy Vultures were eliminated and 1.8 friendly Vultures survived on average in each run. This is statistically significantly different at $P = 6.04 \times 10^{-7}$ using the t-test on the number of Zealot killed by ECSLBot and Nova. This result indicates that in scenarios against ranged attack units, certain behaviors like kiting are not as effective versus melee attack units. Positioning and target selection become more important than kiting in such scenarios. UAlbertaBot and Nova did not optimize micro behaviors in all scenarios and performed poorly in these cases. Note however, that ECSLBot needs about 21 hours to evolve either P_m or P_r.

6.4 ECSLBot versus the State of the Art Bots in Testing Scenarios

Since we evolved micro behaviors for ECSLBot in two training scenarios, we investigate how the corresponding parameter sets perform in scenarios never encountered before. Therefore, we compared our evolved ECSLBot to Nova and UAlbertaBot on eight testing scenarios as shown in Figure 6.2 and Figure 6.3. Table 6.4, 6.5, 6.6, and 6.7 shows the results of the three bots playing against SCAI on all eight scenarios. The table provides standard deviations as well.

The first two testing scenarios $Test_1$ and $Test_2$ as shown in Figure 6.2 are somewhat similar to the two training scenarios $Train_1$ and $Train_2$ with the only change being the units' positions. The purpose of these two scenarios is to evaluate how

Table 6.4: The performance of bots controlling 5 Vultures vs opponent units on scenario $Test_1$ and $Test_2$.

	$Test_1$: 5 Vultures vs 6 Vultures (split)				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	29	0	1	5.97 / 0.18	2.0 / 0.8
Nova	11	2	17	5.03 / 1.05	0.73 / 1.06
ECSLBot	27	0	3	5.87 / 0.43	2.87 / 1.28
	$Test_2$: 5 Vultures vs 25 Zealots (split)				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	0	0	30	2.63 / 1.64	0 / 0
Nova	30	0	0	17.4 / 2.36	3.77 / 0.84
ECSLBot	30	0	0	19.8 / 1.51	3.93 / 0.85

well ECSLBot's parameters work when the positions of both friendly units and enemy units are changed. The results in Table 6.4 show that ECSLBot and Nova are still good at kiting Zealots in these new scenarios. 17.4 Zealots were killed by Nova and 19.8 Zealots were killed by ECSLBot during 2500 frames on $Test_2$. This testing scenario performance is similar to performance on the training scenarios. UAlbertaBot only killed 2.63 Zealots and lost all 5 Vultures on $Test_2$. However, UAlbertaBot performs better when destroying split enemy Vultures on $Test_2$. 5.97 Vultures were killed and 2.0 Vultures survived on average over thirty runs. ECSLBot performs similar to UAlbertaBot on $Test_2$. Nova performs badly on fighting against ranged attack units which is similar to the results on training scenarios. Eleven out of thirty matches are lost against six split Vultures.

In order to test the micro performance of ECSLBot for controlling different numbers of Vultures, other than five we trained on, we conducted experiments where bots control ten Vultures to fight against different types of enemy units including ranged units, melee units, mixed units, and different terrain. The results

Table 6.5: The performance of bots controlling 10 Vultures vs ranged attack units on scenario $Test_3$ and $Teset_4$.

	$Test_3$: 10 Vultures vs 10 Vultures				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	23	0	7	9.43 / 1.15	3.97 / 2.71
Nova	30	0	0	10 / 0	6.93 / 1.1
ECSLBot	30	0	0	10 / 0	8.1 / 0.83
	$Test_4$: 10 Vultures vs 12 Vultures				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	11	0	19	10 / 2.18	1.13 / 1.58
Nova	25	0	5	11.6 / 1.05	3.06 / 1.9
ECSLBot	29	0	1	11.97 / 0.18	6.4 / 1.74

on scenarios $Test_3$ and $Test_4$ as shown in Table 6.5 indicate that all three bots do well in destroying enemy units. However, ECSLBot saved 8.1 and 6.4 out of 10 friendly Vultures in two scenarios. Nova saved 6.93 and 3.06 Vultures and UAlbertaBot saved only 3.97 and 1.13 Vultures. The difference in performance on saved units between ECSLBot and Nova is statistically significant at $P = 5.35 \times 10^{-5}$ on $Test_3$. These results show that ECSLBot outperformed Nova and UAlbertaBot both on training and testing scenarios against ranged opponents. The ECSLBot seems to find a good balance in the trade off between concentrating fire and kiting.

On scenarios $Test_5$ and $Test_6$, Nova outperformed the other two bots on scenarios with and without obstacle as shown in Table 6.6. Nova destroyed on average 37.6 and 32.6 Zealots in the two scenarios. ECSLBot performs close to Nova and destroyed 34.87 and 30.8 Zealots. In scenario $Test_5$, the difference in performance between Nova and ECSLBot is statistically significant at $P = 2.43 \times 10^{-8}$. The difference between UAlbertaBot and the other two bots is statistically significant. This results show that in terms of kiting efficiency against melee units, ECSLBot per-

Table 6.6: The performance of bots controlling 10 Vultures vs melee attack units on scenario $Test_5$ and $Test_6$.

	$Test_5$: 10 Vultures vs 40 Zealots (split)				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	1	0	29	12.7 / 2.69	0.03 / 0.18
Nova	30	0	0	37.6 / 1.8	9.4 / 0.7
ECSLBot	30	0	0	34.87 / 1.36	8.8 / 0.65
	$Test_6$: 10 Vultures vs 40 Zealots (obstacle)				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	3	0	27	17.6 / 0.18	2.0 / 0.8
Nova	30	0	0	32.6 / 2.54	7.6 / 0.99
ECSLBot	30	0	0	30.8 / 1.42	7.5 / 0.8

forms as well as Nova and better than UAlbertaBot on both training and testing scenarios. Moreover, ECSLBot is able to use the same set of parameters to perform well despite an untraversable obstacle in the center of the map by using the terrain influence map learned during training.

Since we test our bots on scenarios against melee units and ranged attack units independently, we next investigated how our bots fight against enemy units containing both melee and ranged units. We test our bots on scenario $Test_7$ where the enemy is composed of a mix of eight Zealots and eight Vultures as shown in Figure 6.3. $Test_8$ looked at a completely different set of units from the Zerg race in StarCraft. Eighteen Zerglings and ten Hydralisks, Zerg melee and ranged units, never encountered by ECSLBot during training made up the opponents in scenario $Test_8$. Results on these two scenarios are displayed in Table 6.7 and show that Nova performs as well as ECSLBot. Both eliminate all enemy units and more than six friendly Vultures survive. The differences in performance of saved units between ECSLBot and Nova on both $Test_7$ and $Test_8$ are not statistically signif-

Table 6.7: The performance of bots controlling 10 Vultures vs mixed units on scenario $Test_7$ and $Test_8$.

	$Test_7$: 10 Vultures vs 8 Zealots & 8 Vultures				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	6	0	24	8.9 / 1.57	0.53 / 1.18
Nova	30	0	0	16 / 0	6.43 / 1.87
ECSLBot	30	0	0	16 / 0	6.5 / 1.5
	$Test_8$: 10 Vultures vs 18 Zerglings & 10 Hydralisks				
	Win	Draw	Lose	Average Killed / σ	Average Left / σ
UAlbertaBot	5	0	25	24.5 / 2.3	0.67 / 1.57
Nova	30	0	0	28 / 0	6.67 / 1.66
ECSLBot	30	0	0	28 / 0	6.33 / 1.37

icant. However, the difference between UAlbertaBot and the other two bots is statistically significant. ECSLBot performs well in these mixed opponent type scenarios by switching between P_m and P_r depending on target type. For example, if the current target is a Zergling which is a melee unit, ECSLBot uses P_m, the parameters evolved in the $Train_1$ melee scenario for the IM, PF, and reactive control. As soon as the target changes to an enemy Vulture, *ECSLBot* will use ranged reactive control parameters that refer to the ranged attack IM and PF (P_r). This mechanism performs well even when fighting against enemy units not seen during training by simply comparing the weapon attack ranges between the target unit and the friendly unit to determine ranged or melee and thus which parameter set to use.[2]

[2]In RTS games, weapons ranges and damage information is known and publicly available

6.5 ECSLBot versus the State of the Art Bots in Head-to-head Scenario

We have compared the performance of three bots playing against SCAI on two training and eight test scenarios and the results show that ECSLBot works well on all scenarios while Nova and UAlbertaBot perform well on some and perform badly on others. However, what are the results when they play against each other? To answer this question, we set up our last set of experiments with a Head-to-head scenario. Each bot plays against the other two bots thirty times with identical units. Since we evolved parameters for Vultures which are a ranged unit, we used Vultures as the unit type in this scenario. ECSLBot thus used P_r for control against UAlbertaBot and Nova. The result shows that ECSLBot beats Nova but is defeated by UAlbertaBot. The replays show that ECSLBot's positioning micro is driven by training against by SCAI and does not generalize well to other Bots. Thus although ECSLBot's representation and control algorithms evolve to generalize over opponent positions, terrain, and opponent types, they are specifically evolved to beat SCAI.

Therefore, we evolved another set of parameters directly against UAlbertaBot and applied ECSLBot with this set of parameters against UAlbertaBot and Nova. Table 6.8 shows the detailed results among all the bots. We can see UAlbertaBot wins 24 matches, draws 5, and loses 1 against Nova. After examining game replays for these games, we found that Nova's micro kites against any type of opponent units. However, as our experiments with the scenario $Train_2$ showed, kiting too much against the same ranged attack units actually decreased micro performance. UAlbertaBot on the other hand, disabled kiting when fighting against the equal

Table 6.8: Head-to-head scenario over 30 matches.

	Win	Draw	Lose	Units Remaining
UAlbertaBot vs Nova	24	5	1	2.33
ECSLBot vs Nova	30	0	0	3.37
ECSLBot vs UAlbertaBot	17	1	12	0.30

weapon range units and defeated Nova easily. Similarly, ECSLBot defeated Nova on all 30 games without a loss or draw. The average number of units surviving was 3.37 which is higher than UAlbertaBot's 2.33. The final comparison was between ECSLBot versus UAlbertaBot. The results show that ECSLBot wins 17 matches, draws 1 match, and loses 12 matches out of 30. ECSLBot performed quite well on this scenario against the other bots.

Although our approach can evolve good micro against specific opponents while generalizing over maps and unit types, for the longer term, we are investigating co-evolutionary approaches to evolving micro that is effective against a variety of opponents. Co-evolutionary approaches have been shown to work well in board games and other video games and provide a promising computational intelligence approach to robust behavior evolution.

6.6 Conclusions

This chapter focuses on generating effective micro management: group positioning, unit movement, kiting, target selection, and fleeing in order to win skirmishes in real time strategy games. We compactly represented micro behaviors as a combination of influence maps, potential fields, and reactive control parameters in a 60 length bit-string.

The results in Chapter 4 showed that genetic algorithms perform better than two hill climbers. In this chapter, we test the micro performances of our evolved ECSLBot against two state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in StarCraft. We designed eight testing scenarios in which bots need to control a number of Vultures against different types of enemies, to evaluate micro performance. The results show that ECSLBot performs well by switching parameter values depending on the currently targeted unit. Simple parameter switching can be done in real-time and ECSLBot thus achieves good micro performance. The results also indicate that Nova is highly effective at kiting against melee attack units but performs poorly against ranged attack units. UAlbertaBot, the AIIDE 2013 champion, performs poorly against melee attack units but is excellent against ranged attack units in our scenarios. Compared to the UAlbertaBot, we generate unit specific micro behaviors instead of a common logic for all units. With the right parameters, our ECSLBot beats both UAlbertaBot and Nova.

CHAPTER 7

PHASE FOUR: PARALLEL GENETIC ALGORITHMS

The results in the previous chapter show that we successfully generated high performance micro behaviors for our ECSLBot which could compete with two state of the art bots in our skirmish scenarios. However, one run of our algorithm takes twenty one hours to find high quality solutions. We are interested in techniques to further decreasing the run time of our GAs. One feasible way to run our GA faster is parallelizing evaluation since most of the computational load comes from evaluation in StarCraft. Therefore, we applied the same approach used in StarCraft to the new RTS game SeaCraft, which runs in parallel. We used the same GA configuration in SeaCraft as we did in StarCraft. The only improvement we made to the GA is distributing the evaluation of our individuals to multiple processes through Open MPI. We setup a skirmish scenario similar to 5 Vultures versus 25 Zealot kiting scenario we used in StarCraft. The objective of this scenario is to kill Zealots using only 5 Vultures during 2500 frames. Our experiment environment was two workstations with a *Intel Xeon E5-1620* CPU. The operating system was *Ubuntu* 12.04, with *Open MPI* 1.5.4. Running on 16 cores in total.

The results shown in Figure 7.1 are promising and similar to the results we get in StarCraft. Our bot found a micro behavior that killed 15.57 Zealots in the first generation. After a few generations, our bot learned to kill 19.70 Zealots on average in each SeaCraft match. This means our bot evolved effective kiting behaviors based on the properties of both our units and opponent units SeaCraft. Figure 7.2 shows a snapshot of learned kiting behavior in SeaCraft. Kiting videos of our bot in SeaCraft can be found online [1].

[1] http://www.cse.unr.edu/~simingl/publications.html

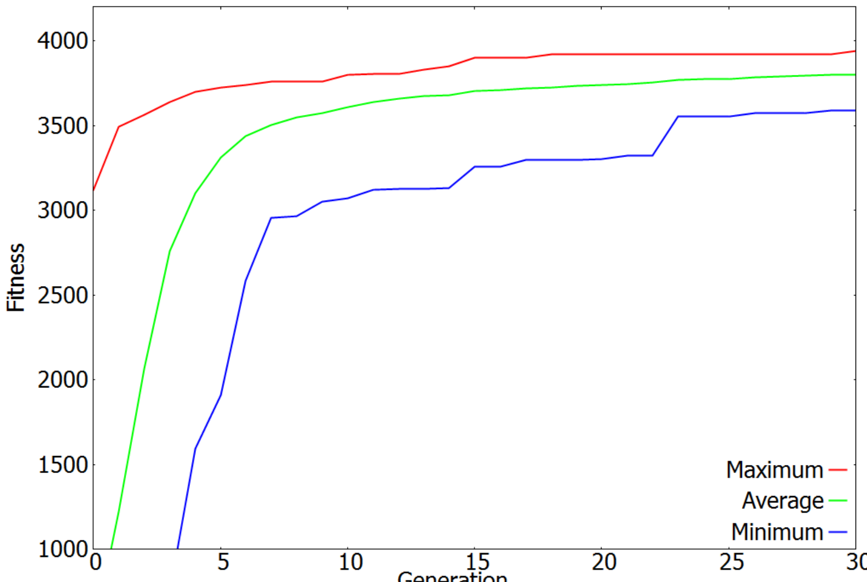


Figure 7.1: Average fitness of bots over generations on 5 Vultures versus 30 Zealots scenario with parallel GA in *SeaCraft*.

7.1 Generalization and Transfer

Previous experiments show that our approach of generating effective micro behaviors in StarCraft can also be applied to another similar RTS game, SeaCraft with similar results. This indicates our approach and representation are not specific to any particular RTS game. Since micro behaviors like kiting and target selection are useful in both games, we are interested in whether the high performance micro behaviors evolved in a fast platform like SeaCraft can be used in the slow but popular platform StarCraft. We adjusted our unit properties, physics, and combat in SeaCraft to be like in StarCraft for comparison. Due to the closed source nature of StarCraft, we have yet to reverse engineer and tune SeaCraft to be **identical** to StarCraft in movement physics, opponent AI, and combat mechanisms. However, we wanted to see if results in SeaCraft transferred to StarCraft, even with a quick initial tuning attempt. We therefore ran our parallel GA in SeaCraft, copied the best parameters found to ECSLBot in StarCraft and tested our ECSLBot on the 5

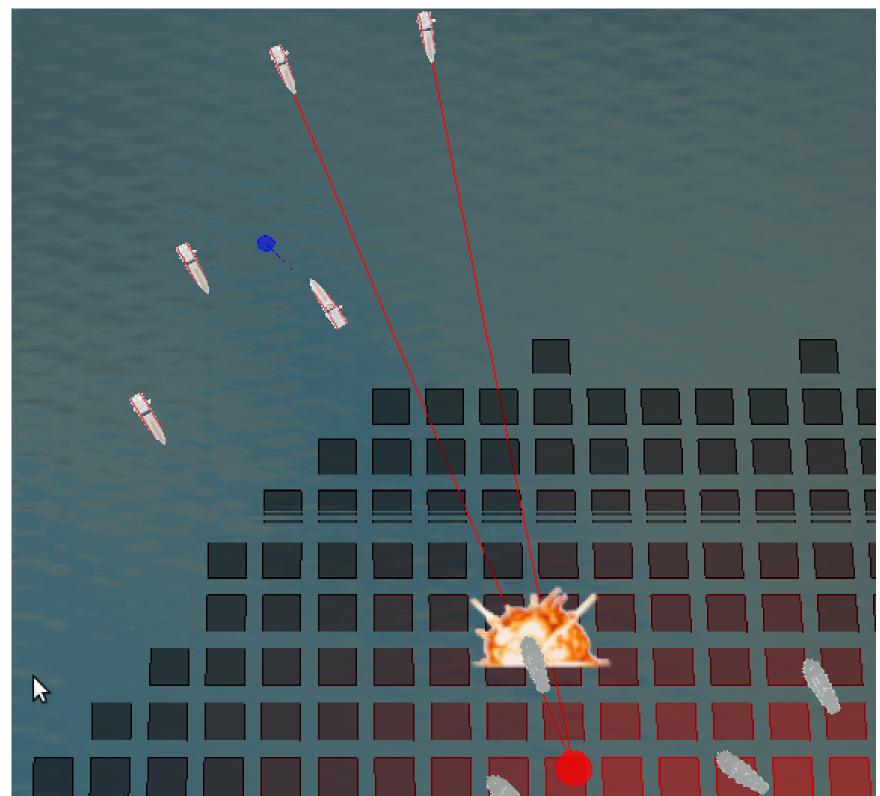


Figure 7.2: Evolved kiting behavior in SeaCraft. 5 Vultures are moving toward or away from enemy Zealots. Enemy Zealots are surrounded by influence maps shown by dark squares.

Table 7.1: 5 Vultures vs 25 Zealots over 30 matches in *StarCraft*.

	Avg Score	Avg Killed	Avg Lost
ECSLBot(StarCraft) vs SCAI	3566.67	17.83	0.20
ECSLBot(SeaCraft) vs SCAI	1386.67	7.13	0.27
ECSLBot(Combined) vs SCAI	3043.54	15.48	0.35

Vultures versus 25 Zealots scenario.

Table 7.1 shows the results of ECSLBot using micro parameters evolved from SeaCraft on the 5 Vultures versus 25 Zealots over 30 matches. The first row shows that ECSLBot with parameters evolved in StarCraft kills 17.83 Zealots within 2500 frames. The second row shows the results of copying all 14 SeaCraft evolved

parameters to ECSLBot running in StarCraft. This performs poorly in kiting the Zealots. Only 7.13 Zealots are killed on average by this bot. This implied that we cannot yet directly use the parameters evolved in SeaCraft in StarCraft. However, we were interested in whether the reactive control parameters, which are somewhat independent of the exact movement physics, evolved in SeaCraft could be used in StarCraft. We extracted IMs and PFs parameters from the best individual evolved in StarCraft and extracted reactive control parameters from the best individual evolved in SeaCraft and measured its micro performance. The last row in Table 7.1 shows that the bot with the combined parameters performed fairly well in StarCraft. Our bot killed 15.48 Zealots, only 2 Zealots less than ECSLBot evolved in StarCraft and Nova. This implies that the reactive control parameters are transferable between SeaCraft and StarCraft and we believe that this is because behaviors like kiting, target selection, and fleeing, deriving from unit properties (not movement physics) are very similar in both games. For example, how far away from enemy should our unit start to kite. When should our unit switch target? On the other hand, IMs depend on the map and distribution of units, while PFs are sensitive to physics differences. Therefore, IM and PF parameters evolved in SeaCraft worked poorly in StarCraft.

7.2 Conclusions

This chapter investigate extending our genetic algorithm to be able to evaluate individuals in parallel based on Open MPI on anther RTS game SeaCraft that enables easy GA parallellization. The results show that we can evolve high performance hit and run behaviors in SeaCraft similar to StarCraft but in 8.77 minutes instead of the 21 hours that the process took in StarCraft. Furthermore, we show that pa-

rameters that specify reactive control behaviors such as kiting evolved in SeaCraft are able to be transferred without change to StarCraft with very little loss of unit performance in similar skirmish scenarios. In the next chapter, we discuss our conclusions and contributions towards advancing RTS game AI research.

CHAPTER 8
CONCLUSION

Our research investigates generating effective micro management: group positioning, unit movement, kiting, target selection, and fleeing in order to win skirmishes in real time strategy games. We compactly represented micro behaviors as a combination of influence maps, potential fields, and reactive control parameters in a 60 length bit-string. First, we compared the performances of different heuristic search algorithms including bit-setting optimizer hill climber, random flip hill climber, and genetic algorithms on finding high quality micro behaviors. We used StarCraft's built-in AI as our opponent baseline against which to make our comparisons. Results show that both bit-setting optimizer and random flip hill climbers can find local optima quickly against the baseline, but they are not guaranteed to find good solutions every time. They find good solutions between fifty to seventy percent of the time starting with ten different random seeds. Compared to hill climbers, the genetic algorithms always find good combinations of influence maps and potential fields parameters and produce higher quality solutions compared to the hill climbers. However, genetic algorithms take much longer to converge.

We are therefore interested in techniques that reliably and quickly find high quality solutions. We then investigated applying case injected genetic algorithms to learn from "experiences" generated from previous problems and use this information to bias our search and speed up the process of finding high quality solutions. We defined five similar scenarios with different difficulty levels that are similar to scenarios in typical human player matches. We applied case-injected genetic algorithms to these five scenarios in sequence to assess how experience, stored as cases in a case-base affects performance compared to a genetic algorithm. Genetic

algorithms with exactly the same parameters as used by case-injected genetic algorithms thus served as a baseline. The results show that case-injected genetic algorithms performed similar to our genetic algorithms in the first scenario when the case-base is empty. In scenarios with more scattered enemy units, both genetic algorithms and case-injected genetic algorithms found high quality solutions. However, case-injected genetic algorithms find high quality solutions up to twice as fast as genetic algorithms. Finally, in scenarios with more concentrated enemy units, case-injected genetic algorithms not only find higher quality solutions than genetic algorithms, but also doubled the speed of finding quality solutions. This indicates that case-injected genetic algorithms are a suitable technique to apply across similar problems in RTS games.

After showing that evolutionary algorithms perform better than two hill climbers, in the rest of this work we settled on using genetic algorithms to search for effective micro parameters. We test the micro performances of our evolved ECSLBot against two state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in StarCraft. We designed two training scenarios and eight testing scenarios in which bots need to control a number of Vultures against different types of enemies, to evaluate micro performance. The results show that our genetic algorithm quickly evolves good micro for handling melee attack units and ranged attack units. ECSLBot performs well by switching parameter values depending on the currently targeted unit. Simple parameter switching can be done in real-time and ECSLBot thus achieves good micro performance. The results also indicate that Nova is highly effective at kiting against melee attack units but performs poorly against ranged attack units. UAlbertaBot, the AIIDE 2013 champion, performs poorly against melee attack units but is excellent against ranged attack units. Compared to the UAlbertaBot, we generate unit specific micro behaviors instead

of a common logic for all units. With the right parameters, our ECSLBot beats both UAlbertaBot and Nova. Our representation leads to good generalization over different numbers of units, different initial positions, and different terrain obstacles. However, evolving against a specific AI (SCAI) means that ECSLBot performs well against SCAI, but not as well against other AIs.

Finally, we extended our genetic algorithm to be able to evaluate individuals in parallel based on Open MPI and apply our approach to anther real-time strategy game SeaCraft that enables easy GA parallellization. The results show that we can evolve high performance hit and run behaviors in SeaCraft similar to StarCraft but in 8.77 minutes instead of the 21 hours that the process took in StarCraft. Furthermore, we show that parameters that specify reactive control behaviors such as kiting evolved in SeaCraft are able to be transferred without change to StarCraft with very little loss of unit performance in similar skirmish scenarios.

8.1 Contributions

Our work has contributed to computational and artificial intelligence research in three ways. First, we applied genetic algorithms and case-injected genetic algorithms towards finding effective micro behaviors in RTS games. While genetic algorithms and case-injected genetic algorithms have previously been used to address problems in real-time strategy games, to the best of our knowledge genetic algorithms and case-injected genetic algorithms have not been applied specifically towards generating effective micro behaviors. Our results showed that genetic algorithms and case-injected genetic algorithms are promising approaches for generating high performance micro behaviors.

Second, we introduced a new approach to represent group micro behaviors in real-time strategy skirmishes through influence maps, potential fields, and reactive controls. We combined a unit influence map and a terrain influence map to represent battlefield spatial information and guide our AI player's positioning and reactive control. We use potential fields to control a group of units navigating to particular locations on the map. With this representation of the problem domain, we are able to apply genetic algorithms, case-injected genetic algorithms, and hill climbers to search high performance micro management for winning skirmishes in real-time strategy games.

Third, we extended our genetic algorithm to be able to evaluate individuals in parallel based on Open MPI and apply our approach to another real-time strategy game SeaCraft that enables easy genetic algorithm parallellization. The results show that we can evolve high performance hit and run behaviors in SeaCraft similar to StarCraft but in 8.77 minutes instead of the 21 hours that the process took in StarCraft. Furthermore, the results indicated that parameters that specify reactive control behaviors such as kiting evolved in SeaCraft are able to be transferred without change to StarCraft with very little loss of unit performance in similar skirmish scenarios.

8.2 Extensions and Future Work

There are many directions for future research that would benefit from our current work. First, we are interested in techniques which can further speed up finding high quality solution for skirmish in real-time strategy games. Some methods like a more sophisticated case injection or an expert system may be added to our sys-

tem in the future to increase solution finding performance. Our parallel genetic algorithms could also benefit from a faster simulator for StarCraft to reduce the evolving process.

Second, instead of evolving solutions based on a static baseline such as the built-in StarCraft AI or UAlbertaBot, we could apply co-evolutionary techniques to produce both sides of the game AI. Genetic algorithms tend to be good at finding the global solution for a specific opponent. However, the global solution might be overfit to the trained opponent and performs poor against a new opponent. Co-evolutionary algorithms are able to generate more diverse and robust micro behaviors which are competitive against more opponents.

Third, we are also interested in applying genetic algorithms and case-injected genetic algorithms to more complicated scenarios where we consider mixed unit types instead of a single type of unit, more complex terrain. In addition, we want to integrate the usage of unit abilities (abilities are different from weapons) like the Terran Ghosts EMP pulse, into our micro behaviors.

There are a broad variety of applications for our simulation gaming and evolutionary computing based techniques. On interesting avenue for future research lies in using genetic algorithm based techniques for resource management in more realistic simulations. Such simulation “games” loosely based on *SimCity*, can be used for education and for research into resource management of critical resources [1]. With good simulations modeling a city, state, or country’s resources, ecology, and economics, we can investigate applying genetic optimization algorithms to efficiently explore trade-offs and manage water, renewable energy, non-renewable energy, economic drivers, and other factor to gauge short and long term environmental impact. Furthermore, we can visualize such systems in 3D virtual environ-

ments and use such interactive visualizations (serious games) for educating the lay public. What-if simulation scenario results that can be easily interpreted by decision makers have the potential to lead to better decisions by policy makers and managers.

BIBLIOGRAPHY

[1] Paul C Adams. Teaching and learning with simcity 2000. *Journal of Geography*, 97(2):47–55, 1998.

[2] Phillipa Avery and Sushil Louis. Coevolving influence maps for spatial team tactics in a RTS game. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 783–790, New York, NY, USA, 2010. ACM.

[3] Christopher Ballinger. *Coevolutionary approaches to generating robust build-orders for real-time strategy games*. PhD thesis, University of Nevada, Reno, 2014.

[4] Christopher Ballinger and Sushil Louis. Learning robust build-orders from previous opponents with coevolution. In *Conference on Computational Intelligence in Games (CIG)*. IEEE, 2014.

[5] Luigi Barone and Lyndon While. Evolving adaptive play for simplified poker. In *International Conference on Evolutionary Computation Proceedings*, pages 108–113. IEEE, 1998.

[6] Maurice Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE))*, 2008.

[7] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Poker as a testbed for ai research. In *Advances in Artificial Intelligence*, pages 228–238. Springer, 1998.

[8] Piero P Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft computing*, 1(1):6–18, 1997.

[9] Johann Borenstein and Yoram Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[10] Mat Buckland and Mark Collins. *AI techniques for game programming*. Premier press, 2002.

[11] Michael Buro. Real-time strategy games: A new AI research challenge. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1534–1535, 2003.

[12] Michael Buro. ORTS - A free software RTS game engine. *Accessed March*, 20:2007, 2007.

[13] Ryan Leigh Christopher Miles, Juan Quiroz and Sushil J Louis. Co-evolving influence map tree based strategy game players. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 88 –95, april 2007.

[14] David Churchill and Michael Buro. Build order optimization in StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 10/2011 2011.

[15] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for rts game combat scenarios. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2012.

[16] Holger Danielsiek, R Stuer, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent moving of groups in real-time strategy games. In *IEEE Symposium On Computational Intelligence and Games (CIG)*, pages 71–78. IEEE, 2008.

[17] Magnus Egerstedt and Xiaoming Hu. Formation constrained multi-agent control. *IEEE Transactions on Robotics and Automation*, 17(6):947–951, 2001.

[18] Larry J Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging. *Foundations of Genetic Algorithms 1991 (FOGA 1)*, 1:265, 2014.

[19] David E Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, 1989.

[20] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–9. IEEE, 2006.

[21] Martin Johansen Gunnerud. A CBR/RL system for learning micromanagement in real-time strategy games, 2009.

[22] Johan Hagelbäck and Stefan J Johansson. The rise of potential fields in real time strategy bots. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2008.

[23] Johan Hagelbäck and Stefan J. Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2*, AAMAS '08, pages 631–638, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[24] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press, 1975.

[25] John H Holland. Adaptation in natural and artificial systems, university of michigan press. *Ann Arbor, MI*, 1(97):5, 1975.

[26] Su-Hyung Jang and Sung-Bae Cho. Evolving neural NPCs with layered influence map in the real-time simulation game Conqueror. In *IEEE Symposium On Computational Intelligence and Games (CIG)*, pages 385 –388, dec. 2008.

[27] Gregory Junker. *Pro OGRE 3D programming*. Apress, 2006.

[28] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986.

[29] Siming Liu, S.J. Louis, and C. Ballinger. Evolving effective micro behaviors in rts game. In *Computational Intelligence and Games (CIG)*, pages 1–8, Aug 2014.

[30] Siming Liu, Sushil J Louis, and Monica Nicolescu. Comparing heuristic search methods for finding effective group behaviors in RTS game. In *Congress on Evolutionary Computation (CEC)*, pages 1371–1378. IEEE, 2013.

[31] Siming Liu, Sushil J Louis, and Monica Nicolescu. Using CIGAR for finding effective group behaviors in RTS game. In *Conference on Computational Intelligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[32] Sushil J Louis and Gong Li. Case injected genetic algorithms for traveling salesman problems. *Information Sciences*, 122(2):201–225, 2000.

[33] Sushil J Louis and John McDonnell. Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 8(4):316–328, 2004.

[34] Sushil J Louis and Christopher Miles. Playing to learn: Case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation*, 9(6):669–681, 2005.

[35] Chris Miles and Sushil J Louis. Co-evolving real-time strategy game playing influence map trees with genetic algorithms. In *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*, pages 0–999, 2006.

[36] Reza Olfati-Saber, J Alex Fax, and Richard M Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, 2007.

[37] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Learning from demonstration and case-based planning for real-time strategy games. In *Soft Computing Applications in Industry*, pages 293–310. Springer, 2008.

[38] Santiago Ontañon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, Mike Preuss, et al. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):1–19, 2013.

[39] Mike Preuss, Nicola Beume, Holger Danielsiek, Tobias Hein, Boris Naujoks, Nico Piatkowski, Raphael Stuer, Andreas Thom, and Simon Wessing. Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):82–98, 2010.

[40] Steve Rabin. *AI Game Programming Wisdom 4*, volume 4. Cengage Learning Trade, 2014.

[41] Eric Raboin, Ugur Kuter, Dana Nau, and SK Gupta. Adversarial planning for multi-agent pursuit-evasion games in partially observable euclidean space. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[42] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM Siggraph Computer Graphics*, 21(4):25–34, 1987.

[43] Keith D Rogers and Andrew A Skabar. A micromanagement task allocation system for real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):67–77, 2014.

[44] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[45] Jonathan Schaeffer. *One jump ahead: challenging human supremacy in checkers*. Springer Science & Business Media, 1997.

[46] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29, 2001.

[47] Penelope Sweetser and Janet Wiles. Combining influence maps and cellular automata for reactive game agents. *Intelligent Data Engineering and Automated Learning (IDEAL)*, pages 209–215, 2005.

[48] Alberto Uriarte and Santiago Ontañón. Kiting in RTS games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[49] Ben George Weber and Michael Mateas. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 140–147. IEEE, 2009.

[50] Stefan Wender and Ian Watson. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar. In *Conference on Computational Intelligence and Games (CIG)*, pages 402–408. IEEE, 2012.

[51] Stewart W Wilson and SW Wilson Ga-easy. Ga-easy doe not imply steepest-ascent optimizable. 1991.

[52] Albert L Zobrist. A model of visual organization for the game of go. In *Proceedings of the spring joint computer conference*, pages 103–112. ACM, 1969.