University of Nevada, Reno

COEVOLUTIONARY APPROACHES TO GENERATING ROBUST BUILD-ORDERS 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

Christopher Ballinger

Dr. Sushil Louis / Dissertation Advisor

December 2014

UNIVERSITY OF NEVADA RENO

THE GRADUATE SCHOOL

We recommend that the dissertation prepared
under our supervision by

CHRISTOPHER BALLINGER

entitled

COEVOLUTIONARY APPROACHES TO GENERATING ROBUST BUILD-ORDERS 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

Swatee Naik, Ph. D. – Committee Member

Larry Dailey, M. S. – Graduate School Representative

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

December 2014

ABSTRACT

We aim to find winning build-orders for Real-Time Strategy games. Real-Time Strategy games provide a variety of challenges, from short-term control to longer-term planning. We focus on a longer-term planning problem; which units to *build* and in what *order* to produce the units so a player successfully defeats the opponent. Plans which address unit construction scheduling problems in Real-Time Strategy games are called *build-orders*. A *robust* build-order defeats many opponents, while a *strong* build-order defeats opponents quickly. However, no single build-order defeats all other build-orders, and build-orders that defeat many opponents may still lose against a specific opponent. Other researchers have only investigated generating build-orders that defeat a specific opponent, rather than finding robust, strong build-orders. Additionally, previous research has not applied coevolutionary algorithms towards generating build-orders. In contrast, our research has three main contributions towards finding robust, strong build-orders. First, we apply a coevolutionary algorithm towards finding robust build-orders. Compared to exhaustive search, a genetic algorithm finds the strongest build-orders while a coevolutionary algorithm finds more robust build-orders. Second, we show that case-injection enables coevolution to learn from specific opponents while maintaining robustness. Build-orders produced with coevolution and case-injection learn to defeat or play like the injected build-orders. Third, we show that coevolved build-orders benefit from a representation which includes branches and loops. Coevolution will utilize multiple branches and loops to create build-orders that are stronger than build-orders without loops and branches. We believe this work provides evidence that coevolutionary algorithms may be a viable approach to creating robust, strong build-orders for Real-Time Strategy games.

ACKNOWLEDGEMENTS

This research is supported by ONR grant N000014-12-C-0522.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

We endeavour to find robust, strong build-orders for Real-Time Strategy games. Historically, computational intelligence and artificial intelligence research has benefited from advances made in board games such as *Backgammon*, *Checkers* (also known as *Draughts*), *Chess*, and *Go* [54]. Research using games typically focuses on developing *intelligent agents* that learn to play games. An intelligent agent is "...anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors" according to Russell and Norvig [52]. In the context of game research, the intelligent agent (colloquially called the *game AI*) acts as a player in the game with the goal of winning or defeating an opponent by addressing problems in the game. In recent years, researchers have turned their attention towards computer games [68, 69]. Computer games provide researchers with an environment which simulates simplified real-world problems. Compared to investigating real-world problems directly, computer games allow researchers to easily investigate new approaches to problems and quickly obtain results from the simulation. While different types of computer games offer interesting research problems, which we discuss in Chapter 2.5, Real-Time Strategy (RTS) games present particularly challenging problems [47].

1.1 Challenges of Real-Time Strategy Games

RTS games are simplified military simulators where players manage resources, expand infrastructure, gather intelligence on the opponent, and build an army comprised of different military units to destroy their opponent. Players do not

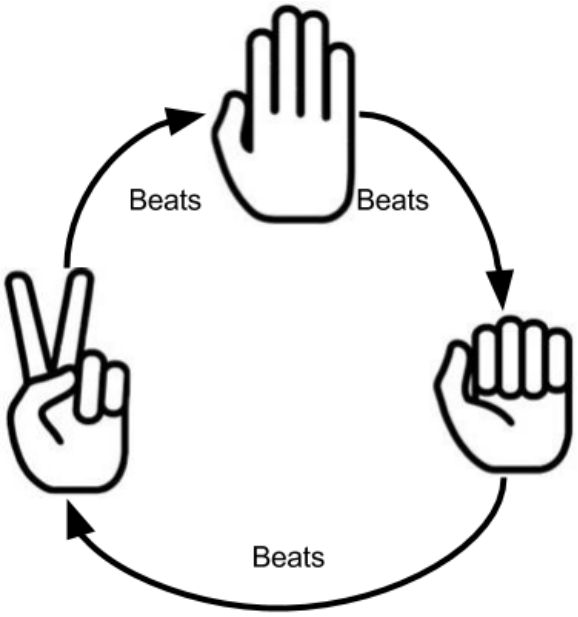


Figure 1.1: Intransitive relationships in rock-paper-scissors.

control every minor detail of their army. For instance, a player does not need to inform a tank which way to point the cannon or order the tank to fire each individual shot. Instead players focus on strategic planning and give higher-level commands (such as *Build a tank*, *Move to this position*, and *Attack the enemy army*) to their army. As a result, the player must handle two broad categories of problems: micromanagement and macromanagement [47]. Micromanagement deals with the short-term control problems of quickly interacting with units and organizing unit positions to maintain an advantage during combat. Macromanagement deals with the longer-term planning problems of gathering/spending resources, constructing new units, and expanding control over different territories. Our research investigates macromanagement; specifically, planning which units to *build* and in what *order* to produce the units so a player successfully defeats the opponent. In macromanagement terms, plans created to address construction scheduling problems in RTS games are called *build-orders*. Players must find build-orders that produce an

army strong enough to defeat their opponent's army. If a build-order slowly produces units, or produces units weak against the opponent's army, then the player will struggle to win. Before we discuss the details of RTS gameplay, which we cover in Chapter 2.1, it is important to understand why finding build-orders in RTS games is an interesting problem.

Players must address difficult problems to find winning build-orders. Intransitive relationships, which the game of *Rock-Paper-Scissors* exemplifies, makes RTS games particularly challenging to play. As Figure 1.1 shows, build-order A (Rock) defeats build-order B (Scissors), but build-order B defeats build-order C (Paper), which in turn defeats build-order A. As a result, no single dominating build-order defeats all other build-orders, which makes finding winning build-orders difficult. Additionally, the number of different unit options available to players creates a combinatorially explosive number of build-orders and possible game states. With many different build-orders available for players to explore, we're interested in finding *robust* build-orders which defeat many opponents. In addition to being robust, we want *strong* build-orders which defeat opponents quickly. Defeating opponents quickly gives the opponents less opportunity to find and exploit a weakness. However, just because a robust, strong build-order quickly defeats many opponents does not mean the build-order will defeat a particular opponent we are interested in. Players must constantly adapt their build-orders to defeat new opponents. All of the above problems make finding winning build-orders an interesting and challenging problem which can be addressed using different approaches.

In the past different approaches have been used to address problems in RTS games. In commercial games, developers typically use finite-state machines or rule-based systems to solve problems, while researchers have investigated a vari-

ety of other methods which we discuss in Chapter 2.5. However, these approaches focus on predicting an opponent's choices and adapting in real-time, rather than finding robust, strong build-orders ahead of time. Coevolution has been successfully applied to problems in RTS games, but not specifically towards generating robust build-orders. We believe that a coevolutionary approach would also be suitable for finding robust, strong build-orders.

1.2 Approach

A coevolutionary algorithm (CA) tests build-orders against other build-orders previously found by the CA. Specifically, build-orders are tested by a game AI which uses the build-orders to compete against opponents in an RTS game. Build-orders that enable the game AI to win often against the opponents share information with each other to produce a new set of build-orders with improved robustness and strength. Because the opponents used for testing are previously coevolved build-orders, as the CA finds build-orders that are more robust and strong, new build-orders must complete against more challenging opponents. This creates an arms race that drives the CA to find robust, strong build-orders. In contrast, genetic algorithms (GAs) and hill-climbers (HCs) do not bootstrap their own opponents. Instead, the GA and HC test build-orders against a set of unchanging opponents that a developer must predefine by hand. Our results indicate competing against predefined opponents enables the GA and HC to find strong build-orders that defeat the predefined opponents, but does not lead to finding robust build-orders. However, the build-orders that can be found by our approaches depends on how we represent build-orders.

Our research investigates two representations for build-orders: Build-Order Lists (BOL) and Build-Order Iterative Lists (BOIL). BOL represents build-orders as a sequence of build-actions that the game AI should perform. BOIL extends BOL by including a representation for branches and loops. We describe both representations in Chapter 3, and began our research using the BOL representation.

In our initial research, we compared the quality of build-orders produced by a GA, CA, and HC. The GA, CA, and HC must determine what order of build-actions will allow the game AI to defeat an opponent. However, in order to compare the quality of the build-orders generated by each approach, we needed an absolute measure of quality. Exhaustive search gives us an absolute quality measure, but we cannot exhaustively search all possible build-orders in a reasonable amount of time. Instead, we created three baseline build-orders by hand, limited ourselves to 5-action build-orders, and recorded the outcomes of each 5-action build-order competing against each baseline in an RTS game. Exhaustively searching the game outcomes allowed us to rank all possible 5-action build-orders by the number of wins, and allowed us to compare the rank of build-orders found by the GA, CA, and HC. We later extended the length our build-orders to 13-actions to match the length of our longest baseline build-order. Our results showed that with 5-action and 13-action build-orders, the GA found the highest scoring build-orders that defeated the baseline opponents, while the CA found build-orders that were more robust than the GA or HC build-orders.

Although the build-orders found by our CA defeated more opponents than build-orders found by the GA and HC, the build-orders may not defeat a specific opponent. Can we generate build-orders that are robust, yet also defeat a specific opponent? We investigated this issue by introducing case-injection into

coevolution. Case-injection places specific build-orders into coevolution's set of opponents. Our results showed case-injection encouraged build-orders to defeat the injected build-orders, while maintaining robustness. As an additional benefit, we can also use case-injection to influence build-orders to play like injected build-orders.

Our results thus far indicated our BOL representation enabled our GA and CA to find robust, strong build-orders. However, BOL only encodes a single and limited-length sequence of build-actions. We believe encoding multiple paths and repeating actions may allow our representation to encode stronger build-orders.

To this end, we expanded the BOL representation to BOIL. BOIL encodes loops and branches which enables build-orders to contain multiple possible sequences of build-actions. Build-actions that fall under a loop or branch are only performed when a predefined condition is met. Branches and loops enables build-orders to respond to different situations, and more compactly represent longer build-orders. Our results showed that the BOIL representation allowed the CA to take advantage of loops and branches and produced stronger build-orders. Our work provides evidence that coevolutionary algorithms are suitable for producing robust, strong build-orders for RTS games.

To make the scope of our work clear, note that we do not focus on creating a game AI. Rather, we only focus on finding build-orders *for* a game AI. There are many different ways a player or game AI can make decisions in an RTS game. Additionally, there are also many RTS games with different gameplay. However, the GA, CA, and HC we investigate are independent of the game AI and RTS game. The GA, CA, and HC do not know or care about how the game AI or RTS game operate, they simply create build-orders and note the performance of the build-

orders reported by the RTS game. Additionally, our work provides three main contributions towards finding robust, strong build-orders. First, to the best of our knowledge, our work is the first coevolutionary approach to generating robust build orders. Second, we show that case-injection enables coevolution to learn from specific opponents while maintaining robustness. Third, we show that co-evolved build-orders benefit from a representation which includes branches and loops. The next section in this chapter reviews the structure of the remainder of this thesis.

1.3 Structure of this Thesis

The next chapter provides background information on RTS games, our search methods, and related research . We first describe the rules and challenges in RTS games, and how RTS macromanagement problems relate to build-orders. Next we review the definitions of a genetic algorithm, coevolutionary algorithm, and hill-climber. Finally, we provide an overview of other research involving games. We cover different types of games used in research, ranging from board games to different types of computer games. Additionally, we cover various approaches used to address problems in games, including coevolution.

Chapter 3 describes the implementation of our GA, CA, HC, case-injection, and build-order representation. We also describe the implementation of two RTS games we created as research platforms: WaterCraft and Build-Order Simulation Software (BOSS). WaterCraft was initially described in our paper published in the *Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence* [4].

Chapter 4 shows the robustness of build-orders produced by our GA, CA, and

HC. In this section, we compare build-orders produced by the GA, CA, and HC to each other and to exhaustive search. Additionally, we also provide results on extending the length of build-orders. Our initial results comparing the GA and HC to exhaustive search were published in the *Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence*, while results comparing the CA to the GA, HC and exhaustive search were published in the *Proceedings of the 2013 Genetic and Evolutionary Computation Conference* [4, 3]. The results for increasing the length of build-orders were published in the *Proceedings of the 2013 IEEE Congress on Evolutionary Computation* [6].

Chapter 5 shows the influence of case-injection on coevolution. We analyse the effects of case-injection on a build-order's ability to play like injected build-orders, defeat injected build-orders, and remain robust. The results for using case-injection to defeat injected build-orders were published in the *Proceedings of the 2013 IEEE Conference on Computational Intelligence and Games*, while results for using case-injection to play like injected build-orders were published in the *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games* [5, 7].

Chapter 6 shows how the addition of branches and loops to our build-order representation influences the strength of coevolved build-orders. We compare the strength of build-orders that lack branches and loops to build-orders that use branches and loops. The results for representing branches and loops in our build-order representation are currently in submission to the *2015 IEEE Transactions on Computational Intelligence and Artificial Intelligence* [8]. Finally, Chapter 7 discusses our conclusions and possible future work.

CHAPTER 2
BACKGROUND

This chapter first details the gameplay and macromanagement problems in RTS games. We also give an overview of the terminology and definitions of GAs, CAs, and HCs. Finally, the last section of this chapter reviews work related to game AI and build-orders.

2.1 Real-Time Strategy Games

RTS games place players in the role of commanding an army. In order to win players must gather resources, build infrastructure, increase military power, expand control over the map, gather information on the opponent's concealed activities, and ultimately destroy their opponent's base while their opponent also attempts to do all of the above. Compared to board games, RTS games are more complex. In most board games, each player has their own turn, player actions take effect immediately with deterministic results, and the board state is fully observable. However, RTS games are more complex because there are no discrete player turns and players make moves simultaneously, actions are durative and are non-deterministic, and the board state is only partially observable. To quantify the increase in complexity, while *Chess* has 10^{50} board states and *Go* has 10^{170} board states, RTS games are estimated to have over $(10^{50})^{36000}$ board states to play an entire game to completion, more than the number of protons in the universe [47]. However, the difficulty and challenges players face varies depending on the RTS game being played.

Figure 2.1: StarCraft.

There are many commercially available RTS games, and Blizzard's *StarCraft: Brood War*, shown in Figure 2.1, is among the most successful. As soon the game begins, players are immediately faced with four longer-term macromanagement problems: resource allocation, temporal reasoning, force composition, and decision making under uncertainty. While gameplay varies for different RTS games, generally RTS games contain these four problems in some form. The following sections describe how these four problems are presented in RTS gameplay, relate to the real-world, and affect build-orders.

2.1.1 Resource Allocation

Players start with a limited number of units and resources, preventing players from immediately taking all the actions necessary to create a strong army. In StarCraft, players must instruct their initial units to gather two resources, *Vespene Gas* and *Minerals*. These two resources act as materials to construct new units and

buildings. Units allow a player to attack or defend against the opponent's army, while buildings allow players to produce new units or research technology that increases the unit's strengths. Different buildings and units require different quantities of minerals and gas, so players must determine how many units to dedicate to harvesting each resource. Players must decide how to allocate their limited resources between gathering resources faster, constructing more unit-producing buildings, and reinforcing their army with additional units.

However, dedicating resources towards one objective has trade-offs and opportunity costs for other objectives. By expanding your economy and gathering resources faster, your military power suffers in the short-term but can benefit in the long-term. In economics, this is a well studied problem called a production possibilities frontier [43]. A production possibilities frontier represents the production trade-offs for manufactured items. For example, in StarCraft if we spend more resources on producing marines, then we have fewer resources to spend on producing tanks, and visa-versa. As such, build-orders are directly related to resource allocation problems. A build-order determines what buildings and units to construct, and therefore determine what trade-offs should be made. In order to win, a build-order must make the right trade-offs at the right time in order expand the economy while maintaining a military force strong enough to defend the economy. Build-orders must also determine what units should be added to the military force.

2.1.2 Force Composition

There are many different units the player can build, and each unit has different costs, abilities, strengths, and weaknesses. In order to succeed, players must construct units that take advantage of weaknesses in the opponent's army. However, just as there are intransitive relationships between build-orders, there are intransitive relationships between individual units and no single unit can optimally destroy all other units. Players create armies comprised of different units to avoid having a single exploitable weakness, making their army harder to destroy. While players can overwhelm an opponent with shear numbers, players will conserve time and resources by finding the right mix of counter-units. Similar to resource allocation, build-orders directly influence force composition by determining what to build. However, while the effects of resource allocation are more focused on economical repercussions, force composition affects military and combat performance. In order to defeat the opponent, build-orders must select a mix of different units that will exploit the weaknesses in the opponent's force composition. Build-orders which do not exploit opponent weaknesses will defeat fewer enemy units, lose more allied units, and waste resources. Because different units have different strengths and weaknesses, players must identify which units their opponent plans to build, and construct effective counter-units. Determining what units the opponent has built or plans to build cannot be done easily due to the map only being partially observable to the player.

Figure 2.2: StarCraft - Fog of War.

2.1.3 Decision Making Under Uncertainty

A *fog-of-war* prevents players from observing opponent activity on the map, except in locations containing the player's units. Figure 2.2 shows the three states of the fog-of-war. Areas around the player's units are fully revealed, and allow the player to observe the enemy units and building in the revealed location. Locations where the player's units have not visited recently are partially concealed, and show the state of the map when the player last fully revealed that location. Player's must periodically return to partially concealed areas to receive updated information. Fully concealed locations are areas where the player's units have never visited, and reveals no information to the player. Players must dedicate some units to *scouting* the map for opponent activity and adjust their building schedule to counter the opponent's plan.

Decision making under uncertainty presents a challenging problem for finding build-orders. Without knowing the opponent's force composition, a player cannot

know what to build in order to perfectly counter the opponent's units. A player must rely on experience to determine what choices the opponent might make, and build units that cover most cases. To this end, build-orders that are robust and defeat many opponents help the player under uncertain conditions. In addition to determining *what* actions to take, the player must also consider *when* to take the actions.

2.1.4 Temporal Reasoning

Unlike board games, such as *Checkers*, where a player's movement or actions take immediate effect, movement and actions in RTS games take time into consideration. Players must plan around delays enforced by the RTS game to quickly build an army and execute actions. There are four time-influenced problems the player must plan for: movement, persistent actions, cooldown time, and preparation time.

When players tell their units to move to a new location, the units must travel some path to reach the destination. Usually there are obstacles obstructing the units' path, such as cliffs or boulders. In response to such obstacles, the units must take additional time to find a detour or destroy the obstacle blocking the path. Additionally, different units travel at different speeds, causing the units to arrive at the destination at different times. When a player plans to attack an opponent's base, the player must take their army's travel time into consideration. While the player may have the superior army when issuing the movement order, the opponent may produce additional units and gain the upper hand before the player's army reaches the destination. Build-orders must take travel time into considera-

tion, in order to assure the opponent's army will still be defeated when the player's army arrives.

Some actions available to players are persistent, meaning the action's effects last for a period of time. For example, the *Force Field* ability allows a player to place a temporary barrier to block an opponent's path. However, the temporary barrier only persists for 15 seconds, and the barrier will disappear after the time has expired. Not all abilities are persistent, such as the ability *Blink* which immediately teleports units a short distance and leaves no residual effect. Player using persistent abilities must determine when to use persistent abilities in order to maximum their effect. Meanwhile, opponents must determine what counter-actions to take during and after the persistent ability's duration.

Each ability has a *cooldown* time, which is the duration a player must wait before the ability can be used again. For example, the Blink ability has a cooldown time of 10 seconds, which means once the player uses Blink the player cannot use Blink again until at least 10 seconds have passed. However, waiting longer than 10 seconds to use Blink does not provide the player with additional uses of Blink or provide a reduced cooldown time. Player's must conservatively determine when to use an ability so that the ability will be available when needed most, while at the same time utilizing the ability to the fullest potential by using the ability often.

In addition to the cooldown time, player's must also plan around the *preparation* time. Preparation time has the strongest influence on build-order performance. When player's select an action, a period of time may need to pass until the action's effects actually occur. Players most commonly encounter preparation time problems from constructing new units. When players choose to construct new units, the game enforces a delay between when the unit begins construction and when

the unit becomes available for use. The delay varies depending on the desired unit, and players expecting to have different units available at a specific time must plan around the delays. Each individual building typically produces new units sequentially, but constructing multiple buildings allows players to produce new units in parallel. Therefore, build-orders are strongly affected by the preparation time of units. In order to minimize the amount of time to produce an army, a build-order must schedule unit construction so that building a unit is not delayed by another unit's preparation time. If a build-order inefficiently schedules unit construction by creating many preparation time delays, the player's army will take longer to construct. At the same time, players can reduce preparation time conflicts by dedicating more resources towards constructing new buildings and fewer resources towards building units. Build-orders must determine what trade-offs to make between producing units sooner in the short-term and dedicating resources towards better production capabilities in the longer-term. There are many different approaches to finding build-orders that address the challenges in this section. In the following section, we provide an overview of the methods we use (a genetic algorithm, coevolutionary algorithm, and hill-climber) and methods used by other researchers.

2.2 Genetic Algorithms

Genetic Algorithms (GAs) were first described by John Holland in 1975 and were the subject of David Goldberg's seminal book [34, 28]. GAs find solutions to search and optimization problems by using the genetic processes of biological organisms as a model. As such, GAs also adopt terms from biology to help describe similar concepts. When a GA attempts to solve a problem, the potential solutions (also

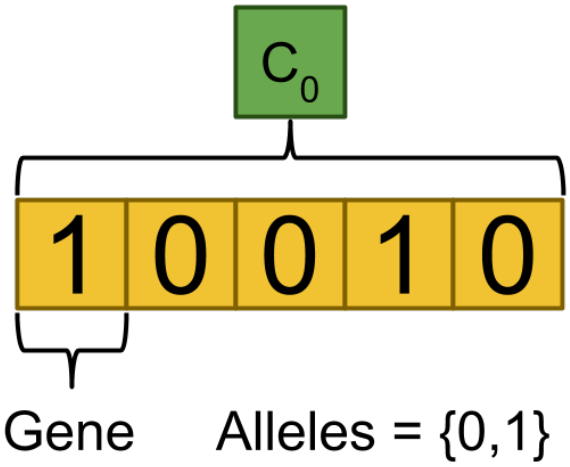


Figure 2.3: Components of a chromosome.

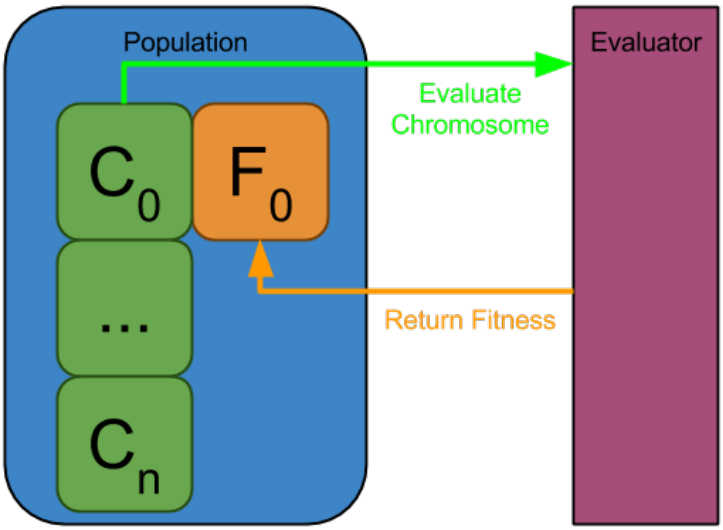


Figure 2.4: Evaluating a population of chromosomes.

called *individuals*) found by the GA are represented as *chromosomes*. Chromosomes aren't the actual solution to a problem, they are instructions that dictate how to create the solution. Figure 2.3 shows the break-down of an example chromosome C_0. A chromosome contains a sequence of *genes*, where each gene contains a symbol from a set of possible symbols called *alleles*.

GAs operate on a set of chromosomes called a *population*. Typically, the GA

initializes the population with randomly generated chromosomes. As Figure 2.4 shows, each chromosome in the population must be assigned a *fitness*, a measure of how well the build-order represented by the chromosome solves the given problem. The evaluation function or *evaluator* interprets the build-order from the chromosome and determines the chromosome's fitness.

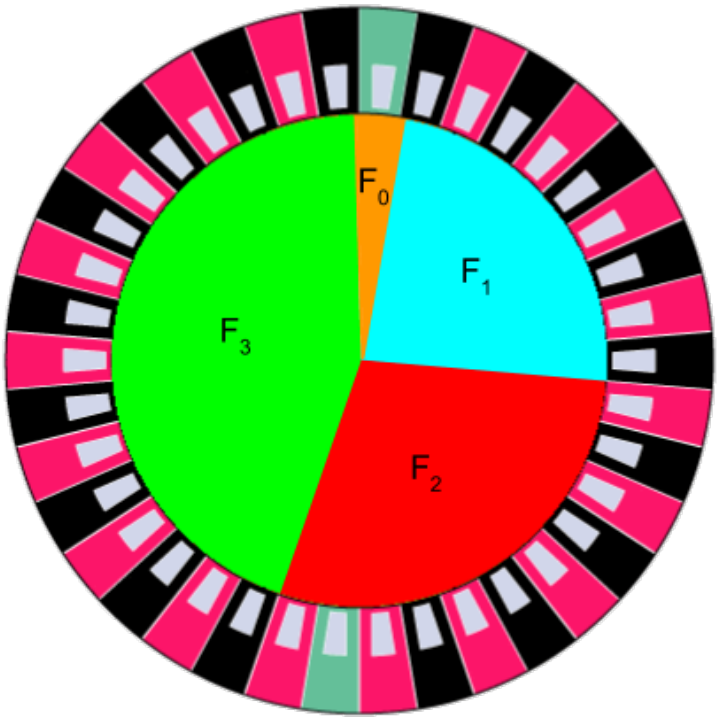


Figure 2.5: Roulette Wheel Selection.

Once every chromosome in the population has a fitness, we want the chromosomes to learn from each other. Pairs of chromosomes that will learn from each other are chosen through a *selection* step. There are different ways of choosing the pairs of chromosomes, but a commonly used method is *Roulette Wheel Selection* (RWS). RWS chooses chromosomes with a probability proportional to the chromosome fitnesses, as Figure 2.5 illustrates. Chromosomes with smaller fitnesses have a smaller chance of being selected, while chromosomes with larger fitnesses are more likely to be selected. When chromosomes have similar fitnesses, like F_1 and

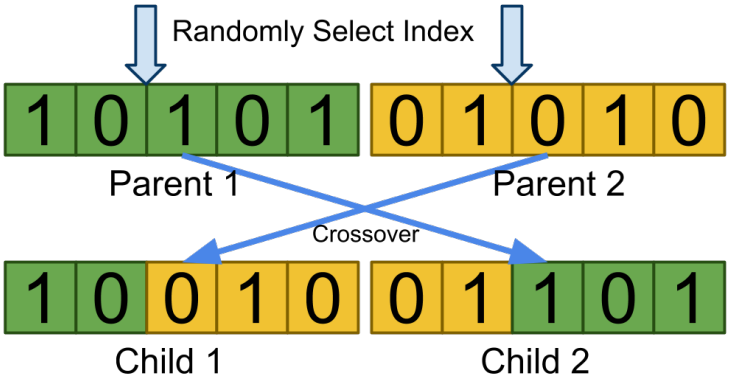


Figure 2.6: One-Point Crossover.

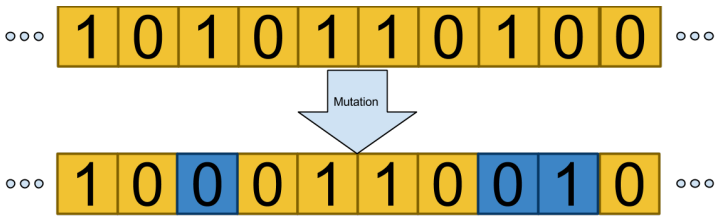


Figure 2.7: Bit-Wise Mutation.

F_2, on average the chromosomes will be chosen a similar number of times. Fitness proportional selection helps the GA learn by focusing on the most promising chromosomes. Chromosomes with a higher fitness solve the problem better, therefore the GA spreads the knowledge represented in higher fitness chromosomes more often.

Each pair of selected chromosomes are called *parents*, which share information between each other to create *children* chromosomes. Sharing information between chromosomes is called *crossover*. Crossover attempts to combine the best traits in each parent to produce better children. As with selection, there are different methods that can be used for crossover. Figure 2.6 shows a simple crossover method known as one-point crossover. One-point crossover selects a random gene index, splits both parents at the selected index, then *crosses the information over* between

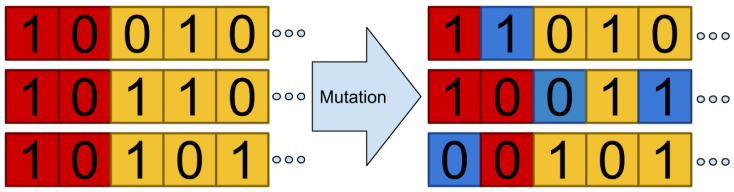


Figure 2.8: Mutation reintroducing genes extinct in all chromosomes.

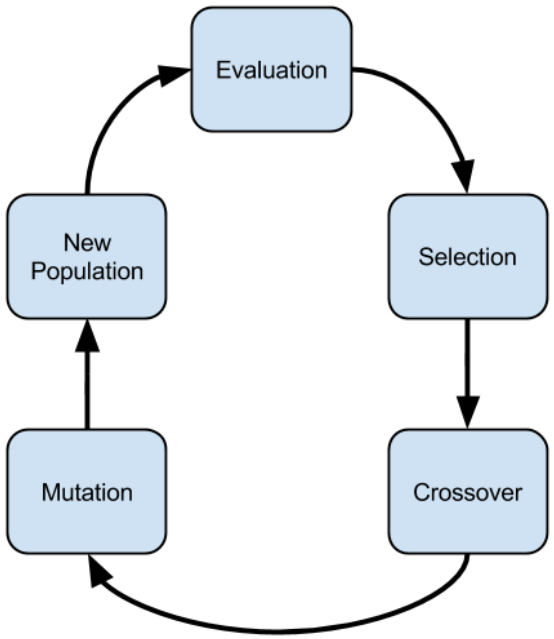


Figure 2.9: Genetic Algorithm generation cycle.

the two parents at the split-index. Crossover helps the GA explore new chromosomes, but may prematurely exclude other viable chromosomes. *Mutation* is one possible approach to prevent narrowing the solution space too quickly.

Bit-wise mutation has a small probability of changing the value of each gene in a chromosome, as shown in Figure 2.7. By changing the value of some genes, mutation helps the GA explore a broader range of the solution space that may not be found by crossover. Mutation is also useful for reintroducing genes that have been lost in the population. In the case shown in Figure 2.8, the two left-most genes

contain the same value in all chromosomes in the population. This creates a problem: no matter which chromosomes are selected for crossover, or what split-index is used, the two left-most genes will always have the same value in the children. The search space becomes limited to chromosomes which all contain the same two left-most genes. As illustrated by the two left-most genes changing color in Figure 2.8, mutation reintroduces genes into the population that otherwise could not be explored. The sequential steps of evaluation, selection, crossover, and mutation creates a new population of chromosomes. The new population replaces the old population, and evaluation, selection, crossover, and mutation are then sequentially repeated on the new population, as shown in Figure 2.9. Each iteration of this sequence is referred to as a *generation*. We therefore label each population the GA produces by the number of iterations taken to create the population. For example, a population created by iterating through the sequence 27 times is referred to as the 27*th generation*. The GA continues to iterate through the sequence until some condition is met, for example: reaching a maximum number of iterations or finding a chromosome that satisfies a minimum fitness.

While each generation generally leads the GA towards finding chromosomes with a higher fitness, there is no guarantee that children will have a higher fitness than the parents. Some GA's use *elitist* selection strategies that copy the highest fitness chromosomes from the parent population into the next generation. If crossover and mutation happen to produce mostly low-fitness children, then elitism preserves the parents that provide knowledge towards solving the problem.

Canonical GAs rely on an evaluation function to determine how well an individual solves a problem. Typically the evaluation function is static and determinis-

tic; at any point in time, given the same chromosome, the evaluation function will always return the same fitness. This means chromosomes are evaluated in isolation, and the performance of previous chromosomes has no bearing on the current chromosomes. However, there are problems that exist where determining a chromosome's fitness requires a comparison to previous chromosomes. Competitive games, such as *Chess*, are a straight-forward example of such a problem. Measuring a player's performance in a game cannot be done by only having one player explain their decision making process. The success or failure of the player's decision making process is entirely dependant on the opponent they play against. Instead, measuring performance requires observing the outcome of matches against different opponents.

2.3 Coevolutionary Algorithms

Coevolutionary Algorithms (CAs) are closely related to GAs, but with a few differences in how chromosomes are evaluated[9, 10, 2, 31]. Rather than using only a static evaluation function to evaluate chromosomes, chromosomes are compared to each other. In the setting of a competitive game, chromosomes are evaluated by playing a match against the other chromosomes in the population. The more opponents a chromosome defeats, the higher the chromosome's fitness. However, comparing all members of a population to each other cannot always be done in a reasonable amount of time. Instead, we evaluate a population against a subset of the population called a *teachset*[51]. The teachset contains several chromosomes that are diverse and different from each other in some way. For example, chromosomes in the teachset may be selected because they each defeat different sets of opponents, and therefore present different challenges. A diverse teachset allows

the CA to keep the number of comparisons to a minimum, while encouraging chromosomes to overcome different challenges and outperform all their predecessors. Because the teachset contains chromosomes from the population, a CA effectively bootstraps challenging opponents for chromosomes to compete against. As the population produces better chromosomes, the teachset will contain more challenging problems that new chromosomes must overcome. As a result, bootstrapping opponents creates an arms-race that drives the CA to find chromosomes that perform well against different opponents. We describe the specifics of our GA and CA implementation in Chapter 3, and in the next section we describe a bit-setting hill-climber.

2.4 Bit-Setting Hill-climber

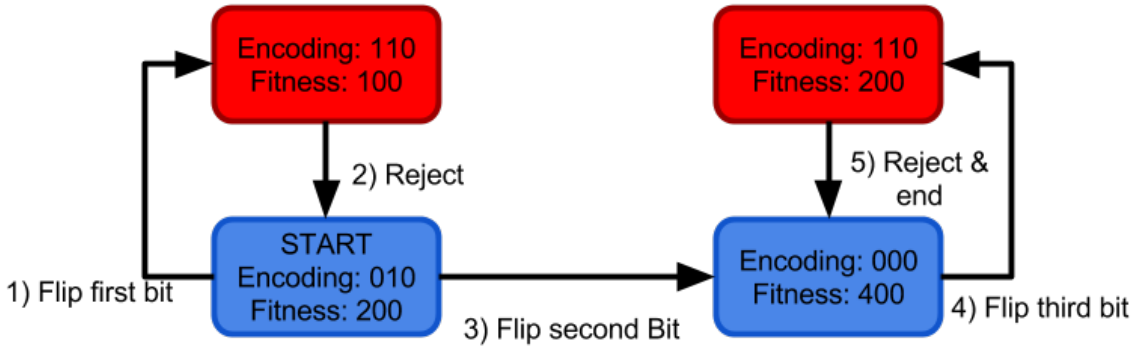


Figure 2.10: Example of a Bit-Setting Hill-climber.

Bit-Setting Hill-climbers (HCs) perform a local search around a chromosome, rather than having multiple chromosomes learn from each other [67]. A HC starts with a random chromosome, and makes changes to individual elements of the chromosome by flipping a bit, as shown in Figure 2.10. The HC flips one bit in the chromosome, and re-evaluates the result using an evaluation function. If changing

the bit's value leads to a increased fitness, then the change is kept, but if changing the value lead to a decreased fitness, the old value is restored. Each iteration changes the value of the chromosome one bit at a time, and incrementally produces better chromosomes that are farther away from the initial chromosome. While sequentially flipping bits allows the HC to improve upon the initial chromosome, HC's tend to get stuck on local optima since they only explore closely related chromosomes. HC's must run multiple times with different initial chromosomes in an attempt to find different local or global optima. In contrast, GAs and CAs are capable of finding chromosomes in a search space with multiple local or global optima. While our work uses a GA, CA, and HC to search for build-orders in RTS games, a large body of work exists that investigates different approaches to problems in games.

2.5 Related Work

Previous work in game AI research traditionally revolved around board games and card games, using a variety of different approaches. *Deep Blue*, a *Chess* playing AI capable of defeating human chess champions, used minimax search to test all possible board configurations 6-12 turns (or *ply*) in advance [17]. By assuming the opponent would always make an optimal move, Deep Blue could determine which moves to make that would leave the opponent in a weaker position in 6-12 ply. *Chinook*, a *Checkers* game AI, used a similar approach to search more than 19 ply in advance, and played competitively against human world champions [55]. Game AI for the card game *Poker* (specifically the *Texas Hold'em* variant) has been developed using rule-based expert systems, Monte-Carlo search, and Bayesian Networks [64, 61, 45]. *Othello*, *Backgammon*, and *Go* have also been used for game

AI research [13, 59, 12]. In addition to the approaches listed above, evolutionary approaches have been employed on these board games as well.

Chellapilla and Fogel created a *Checkers* game AI named *Blondie24* by using coevolution to find the weights of an Artificial Neural Network (ANN) [18, 19, 27]. The resulting game AI played competitively against humans on an online *Checkers* website *The Zone*, with the game AI winning most of the games played. The most challenging opponent the game AI defeated was ranked 98th out of 80,000 registered players, and was 24 ELO points away from the *Master* level [25]. Cowling et al. coevolved the weights of an ANN to create a challenging game AI for *The Virus Game* [23]. The best ANN produced by Cowling's approach defeated most opponents, including opponents never encountered during training. Davis and Kendall used coevolution to tune the parameters of an evaluation function for an *Awari* game AI [24]. The coevolved game AI played against the game AI provided in the commercial game *Awale*, and beat the commercial game AI on three of the four difficulty settings. Nitschke coevolved pursuer-players that cooperated with each other to capture evader-players in a pursuit-evasion game [46]. While in the past game AI research investigated board games, more recently game AI research has shifted towards computer games.

A variety of different computer games and approaches have been used in recent research [33, 32, 53]. Laird et al. developed game AI for the commercial FPS games *Quake II* and *Descent 3* using their Soar architecture and a knowledge base [62]. Spronck et al. created adaptive game AI for the Computer Role-Playing Game (CRPG) *Neverwinter Nights* with dynamic scripting [56]. Loiacono et al. used Q-learning to create successful overtaking/passing behaviours in the racing simulator TORCS [40]. Additionally, some of these games are so popular for game AI

research that researchers have formed annual competitions to compete their state-of-the-art game AIs against each other [35, 50, 39, 15]. Among all the different types of computer games, RTS games in particular are interesting research platforms.

Evolutionary approaches are useful for addressing many of the challenges in RTS games. Ponsen et al. showed that using an evolutionary algorithm to generate an RTS tactic knowledge-base improved strategies created by dynamic scripting [48]. Avery and Louis created RTS team strategies that enabled groups of entities to respond to opponent movements by coevolving influence maps [1]. Miles and Louis used a case-injected GA to produce strike force strategies in an RTS game [42]. In Miles' dissertation, Miles coevolved influence map trees that created a challenging RTS game AI [44]. Keaveney and Riordan used their own abstract RTS game to coevolve game AI that coordinated entity movement on multiple maps [36]. Their research showed that coevolving the game AI on only one map enabled the game AI to win on testing maps not used during training. However, game AI which coevolved on multiple maps performed better on the testing maps. Liu et al. used a case-injected GA to evolve influence maps and potential fields for micromanagement of entities during combat scenarios in the RTS game *StarCraft: Brood War* [38]. In addition to the above problems, researchers have investigated different RTS micromanagement and macromanagement problems, including some work into build-orders [66, 29, 58, 22].

While only a small body of work focuses specifically on build-orders in RTS games, a few different approaches have been investigated. Kovarsky and Buro discussed the challenges of build-order optimization and modeled build-orders with Planning Domain Definition Language (PDDL) [37]. Cho et al. analyzed StarCraft replays to predict opponent strategy and changed their player's build-

order accordingly [20]. Weber and Mateas used case-base reasoning to select build-orders from a case-base according to the game state [65]. Churchill and Buro used a depth-first branch and bound algorithm to search for winning build-orders in real-time [22]. In contrast, our research uses a coevolutionary approach to finding robust, strong build-orders which learn from and defeat specific opponents. Our coevolutionary approach does not depend on a specific RTS game, so there are different RTS research environments we could use. We discuss our approaches and research environments in the following section.

CHAPTER 3
METHODOLOGY

This chapter details the approaches we used in our research. We describe the RTS environments we used, our representations for build-orders, and the implementation of our GA, CA, and HC. To begin our research, we first needed an RTS environment that allowed us to test the performance of build-orders.

3.1 Real-Time Strategy Environments

There are several platforms available for investigating problems in RTS games. The Brood-War API (BWAPI) allows developers to retrieve information about the game state and interact with units in StarCraft [16]. Stratagus provides a free, cross-platform RTS engine that has been used in several research projects [49]. WARGUS uses Stratagus as the back-end for game play, but uses the entity data from the commercial game WarCraft II [63]. Likewise, STARGUS uses Stratagus as the back-end for game play, but uses the entity data from StarCraft [57]. ORTS, another RTS engine, provides a complete programming environment for game AI research [14]. However, these projects are not designed with a specific problem-solving approach in mind, and may be difficult to use with our GA, CA, and HC. Instead of using an existing project, we designed two RTS environments that our GA, CA, and HC can easily and quickly interact with.

Figure 3.1: WaterCraft.

3.1.1 WaterCraft

Our initial research used an RTS game we called WaterCraft, shown in Figure 3.1. We developed WaterCraft specifically for researching evolutionary algorithms in RTS games. WaterCraft was programmed primarily with the Python scripting language, but the game physics were programmed using the C/C++ programming language for the sake of speeding up physics calculations. We implemented a game AI as a part of WaterCraft that communicated with WaterCraft directly, rather than through the Graphical User Interface (GUI) like humans must do. As a result, we did not need to implement our own GUI for the sake of the game AI. Instead, we used the readily available Python-OGRE graphics engine for the GUI [60]. We modeled the game play, unit strengths, unit weaknesses, and unit costs in WaterCraft around StarCraft II, which is known for having balanced unit properties [11]. While WaterCraft lacks some features provided in commercial games, we have implemented the core features of an RTS game. First, players can build several types of buildings and units, with the objective of destroying their oppo-

nent's base. Second, WaterCraft's GUI resembles and functions similar to GUIs in other RTS games. Third, human players have the option of playing against the game AI or another person over the network. While a player competes against an opponent, the player's increases their score by accomplishing different objectives. As such, we can benchmark a player's progress throughout a game by their score, and use score to identify winning build-orders.

Players typically determine how well they played overall by looking at their final score at the end of a game. In the context of our GA, CA, and HC, the final score acts as a build-order's fitness. Our fitness calculation in WaterCraft encourages build-orders to destroy enemy units and structures, as shown in Equation 3.1.

$$F_{ij} = SR_i + 2 \sum_{k \in UD_j} UC_k + 3 \sum_{k \in BD_j} BC_k \tag{3.1}$$

Where F_{ij} is the fitness of build-order i against build-order j, SR_i is the amount of resources spent by build-order i, UD_j is the set of units owned by build-order j that were destroyed, UC_k is the cost to build unit k, BD_j is the set of buildings owned by build-order j that were destroyed, and BC_k is the cost to construct building k. Using Equation 3.1, the higher a build-order's fitness, the better the build-order utilizes resources Our fitness equation enables us to identify winning build-orders, but we cannot determine a build-order's fitness with the build-order alone.

For a build-order to receive a fitness, the build-order must compete against an opponent in an RTS game. However, build-orders only give instructions on which units to build and when to build the units, not how to use or control the units. In order to evaluate a build-order in WaterCraft, there needs to be a game AI which uses a given build-order to compete against an opponent.

WaterCraft receives build-orders as a sequence of build-actions that Water-

Craft's game AI uses to defeat an opponent. The game AI sequentially issues each build-action in the build-order. Before issuing each build-action, the game AI automatically issues a build-action for any missing dependencies first, such as a building required to produce the unit for the given build-action. Automatically inserting missing dependencies allows us to use a smaller build-order representation, which we discuss in Section 3.2. Inserting missing dependencies also prevents a build-order from deadlocking on an invalid build-action, but still requires the GA, CA, and HC to determine the overall sequence of build-actions. The game AI will attempt to issue build-actions as quickly as possible. When the game AI fails to execute the current action because of a lack of resources or pending prerequisite being built, the game AI waits a short duration and reattempts to execute the action until the action succeeds. Using the same build-order against different opponents will yield a different outcome. As a result, a build-order must compete against each opponent in order to obtain a score for each outcome. However, having a build-order compete against multiple opponents can be time consuming.

Evaluating the performance of a build-order requires simulating an entire game in WaterCraft against an opponent. Additionally, GAs and CAs need to perform thousands of evaluations. To minimize the amount of time taken to evaluate a build-order, we turned off the OGRE graphics engine when a GA, CA or HC used WaterCraft to evaluate a build-order. By disabling the graphics we greatly reduced the amount of time required to run a simulation, without changing the outcome. However, an RTS game has different computationally expensive components, such as physics, that are unnecessary for evaluating build-orders. Unnecessary computations increased the amount of time required to evaluate build-orders, but can not be disabled. In order to further reduce our evaluation time, we created another RTS environment called *Build-Order Simulation Software* (BOSS).

3.1.2 Build-Order Simulation Software

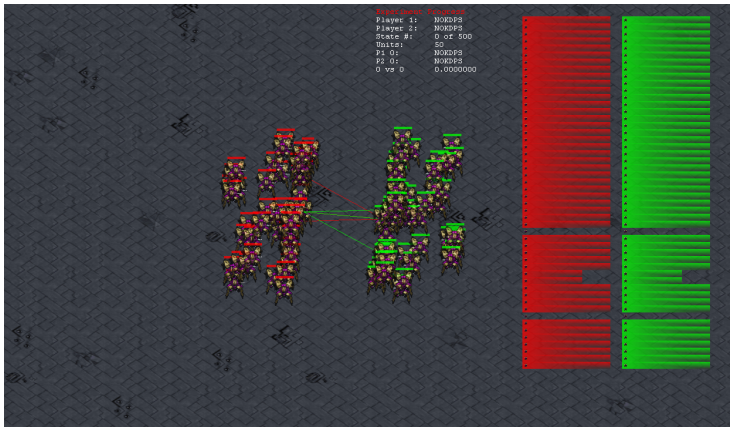


Figure 3.2: SparCraft.

We created BOSS specifically for investigating build-orders. Rather than simulate a full RTS game like WaterCraft, BOSS only simulates build-orders. Like WaterCraft, BOSS receives build-orders as a sequence of build-actions, and needs a game AI that uses the build-order to play the game. When BOSS receives a build-order, BOSS' game AI steps through the sequence of build-actions, and BOSS reports the time at which each unit becomes available to the player. Available resources are estimated by summing the average resources-per-second of all available resource gathering units (called Space Construction Vehicles or *SCVs*) over time. When calculating available resources, BOSS takes into consideration when new SCVs are produced and when SCVs become available/unavailable due to constructing a new building. While BOSS' game AI optimally assigns a build-action to the building that will complete the build-action soonest, the game AI will not change the sequence of build-actions in the build-order. We determined the fitness of a build-order by how well the build-order enabled BOSS' game AI to defeat an opponent in combat. However, BOSS only simulates when units are produced from a build-order, and has no combat. To evaluate a build-orders fitness, we

needed another program that estimated the outcome of combat.

We pair BOSS with Churchill's StarCraft combat simulator *SparCraft*, shown in Figure 3.2, to test performance of our build-orders against opponents [21]. Like BOSS, SparCraft was modelled around StarCraft: Brood War and was created to perform only one RTS task, combat. This allows SparCraft to quickly simulate the results of combat and speeds up our evaluation time, without interfering with BOSS. When the game AI declares an intent to attack at a time-frame, BOSS reports all units available for both players at that time-frame and sends the units to SparCraft. SparCraft simulates the outcome of combat between each game AI's group of available units, and scores each game AI based on the number of surviving units. Build-orders which destroy all of the opponent units while protecting allied units are considered the highest scoring build-orders, just as a player would expect in a full RTS game. WaterCraft, BOSS, and SparCraft are all easy to use with our GA, CA, and HC, but there are some significant differences.

While WaterCraft was modelled around StarCraft II, BOSS and SparCraft were modelled around StarCraft: Brood War. Being modelled around different games means that the same build-order will perform differently in WaterCraft than in BOSS. By limiting BOSS and SparCraft to only build-order simulation and combat simulation rather than fully simulating an RTS game, we drastically reduced the run time of our GA, CA, and HC. However, because we ignore or abstract some elements of RTS games, BOSS and SparCraft do not perfectly model StarCraft: Brood War. Despite being different, we are confident that BOSS paired with SparCraft are close enough to an RTS game that our results are applicable to other RTS games, which we show in Chapter 6. To keep our games similar, WaterCraft and BOSS both use a game scenario containing the same units and starting conditions.

3.1.3 Game Scenario

In our first investigation, which we discuss in detail in Chapter 4.1, we want to exhaustively search the outcome of build-orders in an RTS game. To make exhaustive search feasible, our research works with only four types of units and the five types of buildings needed for their production, shown in Table 3.1. Marines are

Table 3.1: Available unit types and prerequisites

Unit Type	Prerequisites
SCV	Command Center
Marine	Barracks
Firebat	Barracks, Refinery, Academy
Vulture	Barracks, Refinery, Factory

quick and cheap to build, with Barracks being the only prerequisite required beforehand, but with low offensive capabilities. Firebats are stronger than Marines, but require a Refinery and Academy in addition to the Barracks, and cost more to produce. Vultures are the strongest unit and require a Refinery, Factory, and Barracks to produce, but cost the most and take the longest to build. SCVs are produced from Command Centers, and are used for gathering resources and building new structures, but have little offensive or defensive value. When building a structure, the SCV must move to the build location of the structure, and becomes unavailable until the structure is complete. When gathering resources, the SCV must move to the location of the resource, gather a small portion, and deliver the resource to a Command Center. Once the SCV delivers the resource to the Command Center, the resource is added to a bank that players use to pay for additional units and structures. When we initialize a game, each player starts with five SCVs

and a Command Center. We place the SCVs and Command Center near sources of additional resources to decrease the amount of time spent gathering and delivering resources. In order to measure how well a build-order worked in WaterCraft and BOSS under this scenario, we tested build-orders against three hand-coded build-orders.

3.1.4 Baseline Opponents

We created three hand-tuned *baseline* build-orders that provided opponents for the GA and HC to train against, and allowed us to test the robustness of build-orders produced by our GA, CA, and HC. The baselines we created each provided a different challenge for build-orders to overcome. The first baseline (Baseline Small) attacks quickly with a small force. Baseline Small builds three additional SCVs for gathering minerals, followed by constructing five Marines, then attacks the player. This build-order challenges the player by quickly building enough units to attack before the player builds much. The second baseline (Baseline Large) does the same, but builds ten Marines instead of five. While much slower than the Baseline Small build-order, the Baseline Large build-order produces a much harder force to overcome if left uninterrupted. The third baseline (Baseline Medium) builds two SCVs for gathering minerals, three SCVs for gathering gas, five Vultures, then attacks. Baseline Medium also takes a long time to complete, but focuses on building fewer units that are individually stronger. These three baselines pose diverse problems, and our GA, CA, and HC must search for build-orders which overcome all three challenges. In order to allow WaterCraft's and BOSS' game AI to easily use these baseline build-orders to compete in an RTS game, we use the same build-order representations for our GA, CA, HC and baseline build-orders.

3.2 Build-Order Representation

Since GAs and CAs prefer binary representations, we represented build-orders as a sequence of 0's and 1's, a bit-string [28]. A bit-string representation makes our GA, CA, and HC easier for us to implement. Additionally, our GA, CA, and HC can operate on any bit-string representation for any problem, without modifying our GA, CA, and HC. Our research used two bit-string representations for build-orders: Build-Order Lists (BOL) and Build-Order Iterative Lists (BOIL).

3.2.1 Build-Order List

Table 3.2: BOL action encodings

Bit Sequence	Command	Prerequisites
000-001	Build SCV (Gather Minerals)	None
010	Build Marine	Barracks
011-100	Build Firebat	Barracks, Refinery, Academy
101	Build Vulture	Barracks, Refinery, Factory
110	Build SCV (Gather Gas)	Refinery
111	Attack	N/A

BOL represents a build-order as a sequence of actions. Every three bits of the binary string encodes an action, as shown in Table 3.2. Each *build*-action instructs the game AI to build the corresponding unit, and any missing prerequisites. The attack-action instructs the game AI to send all constructed units to move towards the opponent's Command Center and attack the enemy units. BOL assumes that

the game AI will automatically detect any missing prerequisites and insert them before any given action, as shown in Figure 3.3. Encoding buildings and units

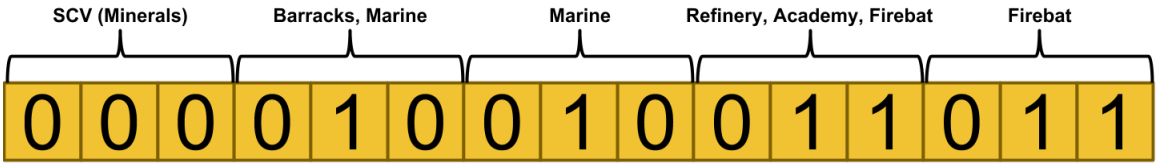


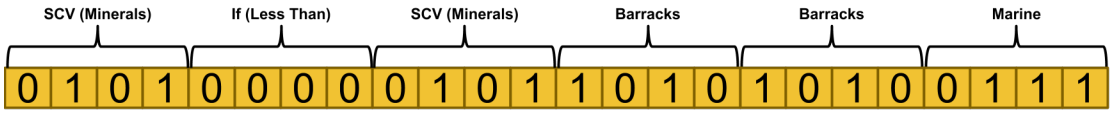
Figure 3.3: BOL encoding example.

would require a representation with 4-bits per action, unlike encoding only the units which requires 3-bits per action.

However, there are more combinations of 3-bit sequences available than build-commands. In order to assure all possible bit strings represent a valid build-orders, some build-actions are represented by more than one 3-bit sequence. This may bias our GA, CA, and HC towards preferring actions with multiple encodings, however the bias should not matter in the long run. Build-orders which perform poorly due to over represented actions will not survive for long in the GA, CA, and HC. While BOL enabled us to represent a build-order in our initial research, BOL can only represent a single, limited length sequence of actions. We later extended our BOL representation to Build-Order Iterative List (BOIL), which allowed us to represent a build-order with multiple arbitrary length sequences of actions.

3.2.2 Build-Order Iterative List

BOIL extends the BOL representation by including branches and loop, as shown in Table 3.3. As with BOL, build-orders are a sequence of actions encoded as a binary string. However, if a branch (IF) or loop (WHILE) is encoded, then the next three



(a) BOIL encoding.

SCV (Minerals)
If (Less Than)
    SCV (Minerals)
    Barracks
    Barracks
Marine

(b) BOIL hierarchy.

Figure 3.4: Two-Condition BOIL encoding example.

encoded actions are treated as if they are in the scope of the branch or loop. For example, the BOIL shown in Figure 3.4(a) would translate into a list of actions the hierarchy shown in Figure 3.4(b).

If the condition for a branch is false, then all the actions in the branch's scope are skipped. Otherwise, when the condition for a branch is true all the actions in the branch's scope are sequentially queued for execution. Once the actions from the branch's scope complete, the next action after the branch's scope is considered. A loop behaves the same as a branch, with one exception. When the actions from the loop's scope are completed, instead of progressing to the next action after the loop's scope, the loop condition is checked again and repeats the actions in the loop until the condition is false. There are two conditions that branches and loops can test: That the number of completed units is greater than or equal to seven, and that the number of completed units is less than seven. We chose to limit ourselves to a scope length of three and completed unit check of seven because these were the largest valid values we could use while always allowing two branches of execution in our shortest BOIL representation (5-actions). We compare four BOIL configurations in Chapter 6, using Table 3.3: BOIL with no conditions, loops or branches,

Table 3.3: BOIL action encodings

No Conditions	One Condition	Two Conditions	Action	Prerequisites
N/A	0000	0000	IF(Condition1)	N/A
N/A	0001-0010	0001	WHILE(Condition1)	N/A
N/A	N/A	0010	IF(Condition2)	N/A
N/A	N/A	0011-0100	WHILE(Condition2)	N/A
0000	0011	0101	Build SCV (Gather Minerals)	None
0001-0010	0100	0110	Build SCV (Gather Gas)	None
0011-0100	0101-0110	0111	Build Marine	Barracks
0101	0111	1000	Build Firebat	Barracks, Refinery, Academy
0110-0111	1000	1001	Build Vulture	Barracks, Refinery, Factory
1000-1001	1001-1010	1010	Build Barracks	None
1010	1011	1011-1100	Build Factory	Barracks
1011-1100	1100	1101	Build Refinery	None
1101-1110	1101-1110	1110	Build Academy	Barracks
1111	1111	1111	Attack	N/A

BOIL with only the greater-than condition, BOIL with only the less-than condition, and BOIL with both conditions. We also remove the automatic dependency assumption, requiring the GA and CA to determine and encode the prerequisites as they are needed. Removing automatic dependencies makes finding valid build-orders more difficult, since there can be many invalid build-orders. For example, a build-order which attempts to build a Marine before building a Barracks is invalid, since the Marine cannot be produced without the Barracks. However, explicitly encoding buildings allows the evolutionary methods to determine when and how many unit production facilities to build, allowing quicker production of combat units. This means BOIL must encode build-actions for all possible units and building, requiring a larger representation. While BOL requires 3-bits per build-action,

BOIL requires 4-bits per build-action. This larger representation makes exhaustive search infeasible, but the addition of branches and loops enabled us to represent stronger build-orders. In order to find winning build-orders using the BOL and BOIL representation, we investigate three approaches: a GA, a CA, and a HC. The next section describes the implementation of our GA.

3.3 Genetic Algorithm

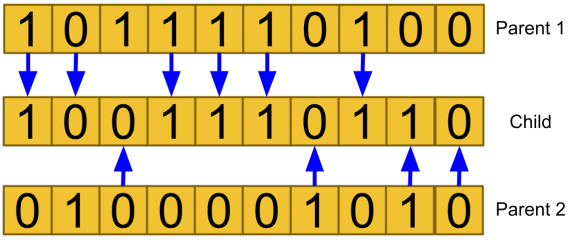


Figure 3.5: Uniform Crossover.

As we described in Chapter 2.2, a GA tests build-orders in an RTS game, and shares information between build-orders that show promise. GAs accomplish this by repeating four main steps: selection, crossover, mutation, and elitist selection. Our GA uses roulette wheel selection, so the probability of a chromosome being selected for crossover is proportional to the chromosome's fitness. RWS enables our GA to focus on spreading the information contained in the most promising chromosomes. Once pairs of parent chromosomes have been selected by RWS, the selected parents exchange information with each other through crossover. Our GA uses *uniform crossover* to produce new children chromosomes from the parent chromosomes. For each gene in a child chromosome, uniform crossover takes the index of the child chromosome's gene, randomly selects one of the two parents, and copies the parent's gene at the same index to the child, as shown in Figure 3.5.

Uniform crossover has a 95% chance of occurring on each selected pairs of parent chromosomes. If crossover does not occur on a pair of chromosomes, the parents are copied to the child population instead. Uniform crossover enables our GA to search for children chromosomes which incorporate the features that make each parent work well. Once crossover has produced a child population, we mutate some genes with a low probability. We use bit-wise mutation on all the chromosomes in the child population. For each bit in every chromosome, bit-wise mutation has a .1% probability that the bit will change value. In order to prevent information relevant to winning from being lost due to many unfit children being produced, we use an elitist selection strategy called *CHC selection* which evaluates the child population [26]. Once all the children have been evaluated, the CHC selection merges the parent and child populations, sorts the chromosomes from highest fitness to lowest fitness, and discards the bottom 50%. This enables CHC selection to determine how many parents should be preserved. If many child chromosomes are less fit than the parent chromosomes, then many parent chromosomes are copied into the next generation. But if many child chromosomes are more fit than the parent chromosomes, then only a few parent chromosomes are copied into the next generation. In order to find robust build-orders, our GA uses three methods to compute chromosome fitnesses: a teachset, shared fitness, and scaled fitness.

3.3.1 Teachset

In order to evaluate the fitness of build-orders, the GA must test the build-orders against multiple opponents. The GA teachset is our hand-tuned baseline build-orders described in Section 3.1.4. Chromosomes in a population compete against

the members of the teachset, and receive a fitness against each opponent. While fitness measures the performance of a chromosome against an individual opponent, for the purposes of selection and crossover we are interested in the performance of a chromosome against all opponents. Additionally, we want to encourage chromosomes to win against many opponents, as well as winning against opponents that other chromosomes cannot defeat. To find chromosomes that meet these goals, we compute a *shared fitness* for every chromosome in a population.

3.3.2 Shared Fitness

Shared fitness rewards chromosomes that contribute new information, in addition to the number of opponents defeated [51]. Usually, only chromosomes that defeat many opponents have a high fitness. However, shared fitness rewards chromosomes for defeating teachset members few other chromosomes defeat. Chromosomes that defeat teachset members few others can, contain new and important innovations for winning. Giving these unique chromosomes a higher fitness allows the GA's population to contain different niches of chromosomes, which address the different challenges presented in the teachset. We calculated fitness sharing as shown in Equation 3.2. Where f_i^{shared} is the shared fitness of chromosome i, D_i is the set of teachset members chromosome i defeated, j a teachset member in D_i, L_j is the number of times j lost against all chromosomes, and F_{ij} is the fitness of chromosome i against teachset member j.

$$f_i^{shared} = \sum_{j \in D_i} \frac{1}{L_j} F_{ij} \tag{3.2}$$

L_j is the total number of build-orders that baseline j lost to in the current population. We calculate L_j using Equation 3.3, where P is the set of all chromosomes in

a population, and i is an element of P.

$$L_j = \sum_{i \in P} GameResult(j, i) \tag{3.3}$$

$$GameResult(j, i) = \left\{ \begin{array}{ll} 0, & F_{ji} >= F_{ij} \\ 1, & F_{ji} < F_{ij} \end{array} \right\} \tag{3.4}$$

GameResult is a function that returns 1 if baseline j lost against build-order i, and returns 0 if baseline j won against build-order i, as show by Equation 3.4. Since Equation 3.2 only takes the sum of baselines that were defeated j_l can never be 0, since if a baseline j was never defeated j would not be in set D_i. This fitness sharing formula rewards build-orders which defeat many opponents, as well as chromosomes that contribute important new information for winning against previously undefeated opponents. We also multiply the shared fitness by the fitness the chromosome received against each defeated opponent. This allows us to identify chromosomes that not only win, but perform significantly better against the opponent. While shared fitness identifies which chromosomes are the most promising, we also want Roulette Wheel Selection to select diverse pairs of chromosomes for crossover. We use fitness scaling in order to enable more diverse selections while still giving preference to the higher-fitness chromosomes.

3.3.3 Fitness Scaling

Fitness scaling adjusts the shared fitness of all chromosomes in a population. In some cases, a population may contain a chromosome with a fitness that dwarfs all other chromosome fitnesses. Large outlier fitnesses will be over-exploited by RWS, and limits the GA's exploration of the search space. Conversely, if all chromosomes in a population have similar fitnesses with only minor differences, RWS

will select chromosomes at random, rather than focusing on the more promising solutions. Fitness scaling helps alleviate both these problems by evening out the selection pressure, as shown in Figure 3.6. Selection pressure is evened out by scal-

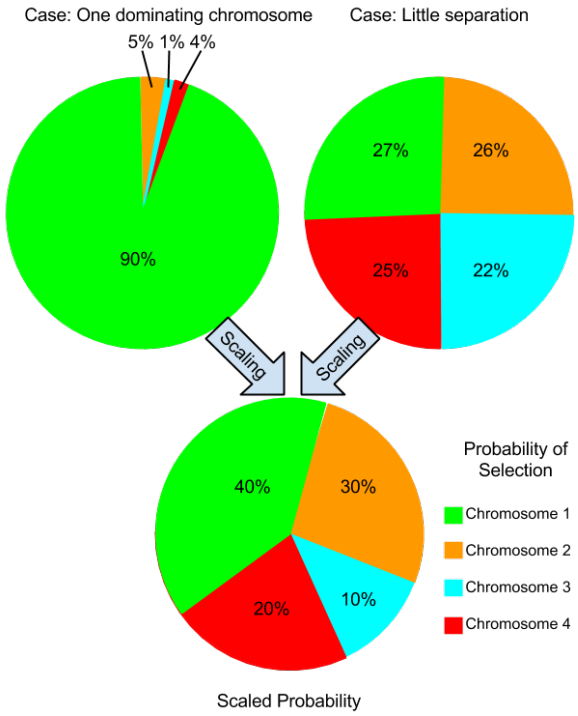


Figure 3.6: Fitness Scaling on two population fitness distributions.

ing the fitness of all chromosomes in a population, such that the highest-fitness chromosome maintains a relation relative to the average fitness of the population. For example, our GA and CA uses a fitness scaling constant of 1.5, meaning that chromosome fitnesses are evenly adjusted so that the highest fitness chromosome will be selected 1.5 times on average. This prevents a chromosome with an exceptionally high fitness from over influencing selection, while at the same time biasing selection towards the highest fitness chromosome if the entire population has similar fitnesses. These methods enabled our GA to find build-orders that defeated our baselines. However, in order for the GA to search for build-orders, the baselines must be provided for the GA to train against. Additionally, we also wanted to find

build-orders which are robust against many opponents, and not just the baselines. To search for robust build-orders, we investigated a coevolutionary approach.

3.4 Coevolutionary Algorithm

A CA is closely related to a GA, as we described in Chapter 2.3. In order for a GA to work, we must provide hand-coded opponents for the GA to test build-orders against. However, a CA creates its own opponents, and enables us to find winning build-orders without providing hand-coded opponents beforehand. This enables the CA to test build-orders against a wider variety of opponents, and produce more robust build-orders. Our CA uses all the same methods and parameters as our GA, but changes the opponents contained in the teachset. Rather than always containing the same three baseline build-orders, our CA's teachset contains eight build-orders from previous generations, and changes every generation. We limited our teachset eight chromosomes, as eight evaluations for every chromosome in a population was typically the highest number of evaluations we could do in a reasonable amount of time. Chromosomes in the CA's teachset change every generation, and are selected from two sources: four chromosomes from shared sampling and four chromosomes from the Hall-of-Fame (HoF).

3.4.1 Shared Sampling

Algorithm 1: Shared Sampling

unsampled = current population
sampled = empty list
while size(sampled) ¡ samples wanted **do**
  for all $s \in unsampled$ **do**
    calculate s_i
  end for
  best = $s \in unsampled$ with highest s_i
  unsampled.remove(best)
  sampled.append(best)
  for all $j \in best.defeated$ **do**
    j_b+ = 1
  end for
end while

$$s_i = \sum_{j \in D_i} \frac{1}{1 + j_b} F_{ji} \tag{3.5}$$

Shared sampling is a method that selects opponents that offer diverse challenges. To enable chromosomes in a population to improve over previous generations, normally the chromosomes would have to compete against all their predecessors. In most cases, evaluating a population of build-orders against all build-orders the previous generation requires too much time. In order to reduce the number evaluations required, shared sampling selects a diverse set of chromosomes which represent the different challenges presented by the previous generation. Shared sam-

pling works by increasing a chromosomes *sample fitness* for each teachset member the chromosome defeats and adding the chromosome with the highest sample fitness to the next population's teachset. However, shared sampling increases a chromosome's shared fitness by a smaller amount for defeating teachset members already defeated by chromosomes previously selected by shared sampling, as shown in Algorithm 1. This encourages the teachset to contain build-orders which defeat different sets of opponents, and offer a variety of different challenges. The sampling fitness is given by Equation 3.5 where s_i is the sample fitness, D_i is the set of teachset members chromosome i defeated, j_b is the number of time j has been defeated by chromosomes selected by shared sampling, and F_{ji} is the fitness of teachset member j against chromosome i. Sample fitness is recalculated each time a chromosome is sampled, so that chromosomes that defeat teachset members not defeated by the currently sampled chromosomes are given higher preference. This allows us to maintain a diverse teachset, while keeping the number of opponents needed to a minimum. However, shared sampling only selects opponents from the previous generation, and there may be chromosomes from more distance generations that would also pose a challenge. Our CA maintains a HoF in order to preserve older chromosomes which may not longer exist in the previous population.

3.4.2 Hall-of-Fame

The HoF is a list of chromosomes that performed well in previous generations. At the end of each generation, the chromosome with the highest shared fitness joins the HoF. Build-orders that are successful in early generations, may fail in later generations and be forgotten by the CA. Because of the intransitive relation-

ships between build-orders, the forgotten build-orders may be successful again in future generations. Preserving build-orders in a HoF prevents new build-orders from forgetting how to defeat build-orders from previous generations. After each generation, our CA selects four chromosomes from the HoF and adds the chromosomes to the teachset. Our CA selects the chromosomes from the HoF at random, as Rosin and Belew found that updating the fitness of chromosomes in the HoF and selecting the highest fitness chromosomes did not provide a great enough benefit to justify the extra computational expense [51].

Creating the teachset from shared sampling and HoF enables the CA to bootstrap challenging opponents and find robust build-orders. However, build-orders found by the CA are not guaranteed to defeat specific build-orders we are interested in, such as build-orders used by a human player. In order to defeat a specific opponent, the CA would need to learn from the experience of other build-orders, or learn from the specific opponent directly. At the same time, we want the CA to continue to produce robust build-orders. One approach that enables a CA to learn from the experience of other build-orders is case-injection.

3.4.3 Case-Injection

In the past, case-injection has enabled GAs to learn from previous solutions [41, 42]. Case-injection enables a GA to learn from the experience of other solutions, and incorporate that knowledge to create better solutions. The solutions (or *cases*) are created from a source external from the GA, such as hand-coded solutions or solutions generated from other search methods. By introducing (or *injecting*) these cases into a GA, the GA quickly learns new solutions that work well. In the con-

text of our research, case-injection takes build-orders produced from outside the CA and injecting the outside build-orders into the CA. Cases can be injected into either the CA's population or teachset. Injecting into the teachset encourages the CA build-orders to defeated the injected cases, while injection into the population encourages the CA build-orders to play like the injected cases. We use four different case injection methods with our CA: case injection into only teachset, case injection into only the population, case injection into both the teachset and population, and no case injection. Our no case-injected method enables us to compare the influence of the other case injection methods on the coevolutionary population. Cases that are injected into a CA are stored in a training case-base, and the same training case-base acts as the source of the injected cases for all case injection methods. The training case-base contains five build-orders from our previous work that were either hand-tuned or evolved. Some of the selected build-orders were robust and defeated many possible opponents, while the other selected build-orders were specialized and defeated only a few of the robust build-orders we had previously evolved. We also maintain 5 different testing build-orders from previous works. We test chromosomes in every population against all chromosomes the testing cases in order to measure the influence of case-injection on the population. Note that the testing cases are separate from the training case-base, and are never seen during training for any of our methods. In our initial research, we investigated the effects of teachset injection on a CA.

Teachset case-injection randomly selects two cases from the training case-base every generation and injects them into the teachset. The two injected cases replace the last chromosome selected from shared selection and one randomly selected HoF chromosome, keeping the teachset size to eight. This influences the CA to prefer build-orders that defeat the injected cases, while the remainder of the teachset

keeps the build-orders robust and prevents them from overspecializing for defeating the injected cases. In addition to learning to defeat specific build-orders, we also want to use the case-base to teach build-orders to play like injected cases. To this end, we also investigated the effects of case-injection into the CA's population.

Population case injection randomly selects two cases from the case-base every five generations, which replace the two lowest shared fitness chromosomes in the population. This influences the population to play like the injected cases, while the teachset continues to encourage the build-orders to remain robust. We limit ourselves to injecting only two cases to prevent the CA from being overly biased towards the injected cases, and narrowing the search space explored by the CA too quickly. Opponents in the teachset change as coevolution finds new build-orders, which means the effectiveness of an injected solution changes depending on when we inject the case. We do periodic case injection every five generations to allow the injected cases a chance to demonstrate their effectiveness against different solutions in the teachset. In addition to investigating the effects of case-injection into the teachset and population separately, we also investigate case-injection into the teachset and population simultaneously.

When we inject cases into both the population and teachset, we use both of the above methods without any modifications, since they do not interfere with each other. We selected the frequency and number of injections based on what worked experimentally well, but future work may look at finding optimal values. Our later work focus' on our GA and CA, which share information between build-orders, at the cost of evaluating multiple build-orders. In order to achieve quicker results, our preliminary work also investigates a hill-climber, a local-search method that requires fewer evaluations than a GA or CA to produce winning build-orders.

3.5 Hill-climber

Algorithm 2: Bit Setting Optimization Hill-climber

```
chromosome = initialize()
select first bit
evaluate(chromosome)
while not end of chromosome do
    flip current bit
    evaluate(chromosome)
    if fitness decreased then
        flip current bit back
    end if
    select next bit
end while
```

We use the bit-setting optimization HC shown in Algorithm 2, which attempts to find an effective solution by sequentially flipping each bit and keeping the value with the highest fitness. We determine the fitness by playing a chromosome against all three baselines and taking the sum of the differences in scores, as shown in Equation 3.6.

$$f_i = \sum_{j \in B} F_{ij} - F_{ji} \tag{3.6}$$

Where f_i is the fitness of chromosome i, j is a baseline in the set of all baselines B, F_{ij} is the fitness chromosome i received against baseline j, and F_{ji} is the fitness baseline j received against chromosome i. This allows the hill-climber to perform a local search by testing the build-orders most closely related to the initial build-order, and incrementally increasing the distance from the initial build-order

towards build-orders with a higher fitness.

HC performance depends on the initial seed, so we initialize this HC with thirty-two different seeds: a chromosome set to all 0's, a chromosome set to all 1's, and thirty randomly generated chromosomes. Our GA, CA, and HC are able to find winning build-orders. However, without an absolution measure of quality, we cannot tell how well the build-orders found by our GA, CA, and HC performed compared to each other or other possible build-orders which were not found. In order to address this limitation, we investigated performing the largest exhaustive search we possible could.

3.6 Exhaustive Search

Exhaustive search evaluates all 2^N possible build-orders against all three of our baselines, where N is the bit string length of the chromosome. For our research, we limit N to the number of bits required to encode 5 build actions(15-bits for BOL and 20-bits for BOIL), because we could not exhaustively search beyond 5 actions in a reasonable amount of time. Exhaustive search enables us to rank all possible N-bit build-orders, and compare the effectiveness of build-orders against the baselines found by our GA, CA, and HC. The next chapter discusses our results comparing build-orders from our GA, CA, HC, and exhaustive search.

CHAPTER 4

PHASE ONE: BUILD-ORDER ROBUSTNESS

The first goal of our research is finding robust build-orders, which enable a game AI to defeat multiple opponents. Robust build-orders are desirable because the game AI will encounter different opponents that must be defeated, and robustness helps the game AI win under uncertain conditions. We use a GA, CA, and HC to generate build-orders, however, in order to compare the build-orders found by the GA, CA, and HC we need an absolute measure of quality, such as exhaustive search. Exhaustively searching the outcome of all possible build-orders competing against each other would be infeasible. In order to make exhaustive search possible, we limit ourselves to exhaustively searching the outcome of all 5-action build-orders against our three hand-tuned baselines. Additionally, we use our BOL representation, which enables us to compactly represent build-orders, since BOL does not require explicitly encoded prerequisites. The next section describe our results for exhaustively searching 5-action BOLs, and comparing BOLs generated by our GA, CA, and HC.

4.1 5-action BOL

Our first step was to exhaustively search the outcome of all 2^{15} $(32,768)$ 5-action (15-bit) BOLs against our three hand-tuned baselines. Exhaustive search shows that 80% of the available BOLs end up losing to all three baselines, as shown in Figure 4.1. This may not be very surprising since the baselines have the advantage of performing more than the five actions encodable in our chromosome representation. Despite this advantage, 19.9% of BOLs find and exploit a weakness in at least

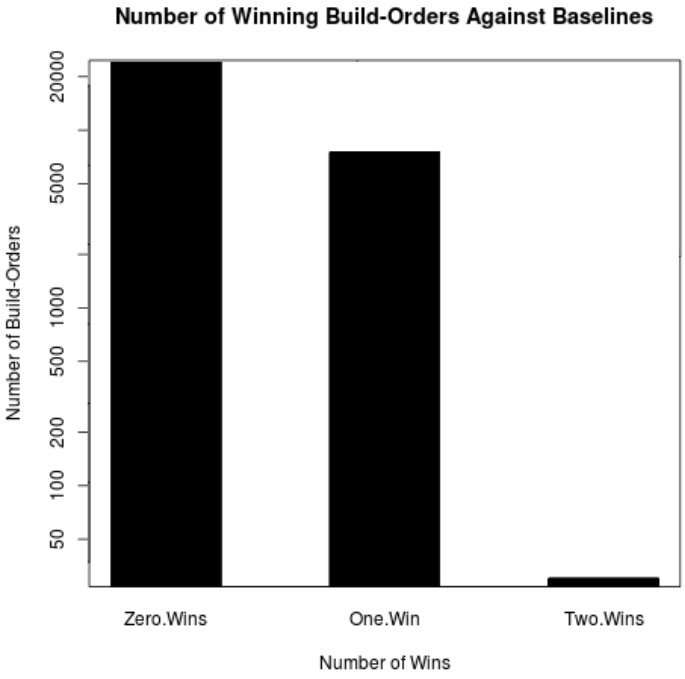


Figure 4.1: Win frequency of all 5-action BOLs against three baselines.

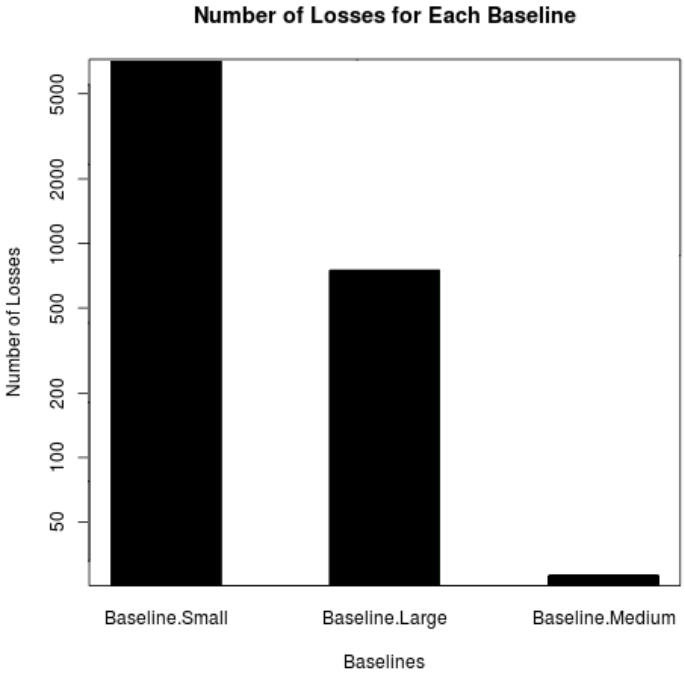


Figure 4.2: Number of losses for each baseline against all 5-action BOLs.

one of the baselines. Only .1% of BOLs manage to beat two baselines, but there are none that beat all three. Breaking down these results further, we can see from Figure 4.2 the difference in difficulty that each baseline provides against all possible build-orders. Baseline Small provides the easiest build-order to overcome, Baseline Large is harder, and Baseline Medium rarely loses. The average scores also reflect these difficulties, as shown by Figure 4.3. Most of the baseline losses can be

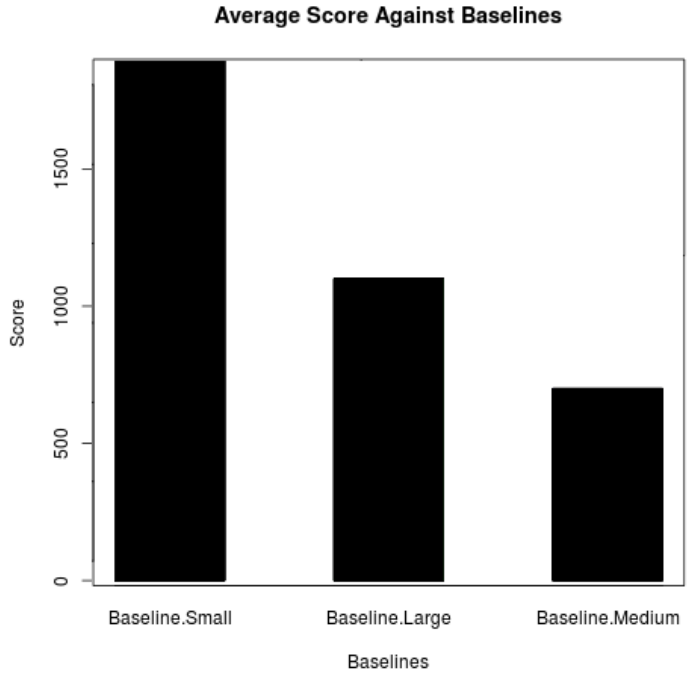


Figure 4.3: Avg. score of all 5-action BOLs against three baselines.

attributed to BOLs that are tuned solely for beating individual baselines. Though rare, exhaustive search clearly shows that there exist BOLs within our search space that can beat two opponents. However, there are no 5-action BOLs that beat all three baselines. As we can see from these exhaustive results, this problem provides a search space where BOLs that beat multiple baselines are few and may be difficult to find. In order to search for some of these winning build-orders, we first investigated using a HC to quickly produce BOLs.

Our HC ran thirty two-times with different starting BOLs (or seeds), and only two seeds lead to an optimal BOL that could beat both Baseline Large and Baseline Medium. Fifteen more seeds were able to lead the HC to find BOLs within the top 20 BOLs for defeating only Baseline Small. The remaining 17 seeds lead to BOLs that lost against all baselines. However, the overall average score of all thirty-two HC BOLs against the three baselines was still better than exhaustive search, as shown in Figure 4.4. While the HC was able to quickly produce winning build-

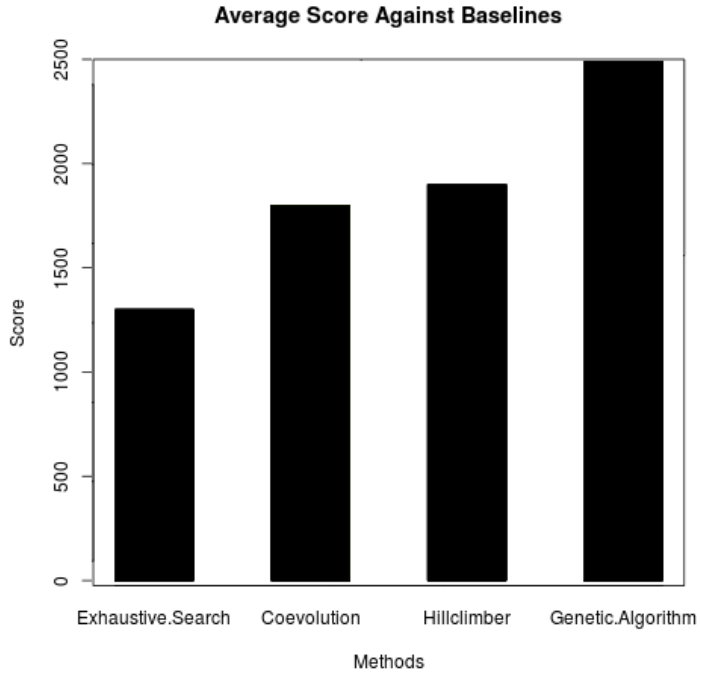


Figure 4.4: Avg. score of 5-action BOLs generated by each approach.

orders, the HC does not reliably find the optimal build-orders. Our next step was to investigate if a GA could produce winning building-orders more reliably than the HC.

The GA used a population size of 50 and iterated for a maximum of 100 generations. We ran the GA a total of ten times, with each new run starting with a

random population of chromosomes. Our results showed that the GA's population contained at least one of the best BOLs as early as generation 20. By playing against all three baselines, the GA's final population always contained the same three optimal BOLs. Two of the BOLs defeat a pair of baselines; Baselines Small and Large, and Baselines Large and Medium. The third BOL found was the optimal BOL for defeating only Baseline Large. Because of our fitness sharing, the best chromosome of each generation would cycle between three BOLs. As BOLs that could defeat Baseline Small and Baseline Medium start to take over the population, BOLs that can defeat Baseline Large start to die out but are given more shared fitness weight. Eventually the shared fitness crosses a threshold where BOLs that only defeat Baseline Large are given so much weight they briefly have the highest shared fitness. As BOLs that defeat Baseline Large start to make a comeback in the population, the weight given to those BOLs becomes lower. Eventually the chromosomes with the highest shared fitness becomes BOLs that can defeat Baseline Small and Baseline Large (although the BOL scores less against Baseline Large than a BOL that defeats only Baseline Large). Then as these BOLs start to take over the population, the cycle restarts. By generation 100, the population consists entirely of these three BOLs. These results indicate that our GA reliably finds high-quality BOLs to defeat the baselines. However, in order to find high-quality BOLs, we had to hand-code the baselines and let the GA use the build-orders during training. In order to produce high-quality BOLs without the use of hand-coded opponents, we must investigate a new approach. As such, we expanded our investigation to include a CA.

We ran coevolution with a population size of 50 for 100 generations, in order to match the population size and number of generations performed by our GA. Because coevolution does not train against specific opponents, we measured the

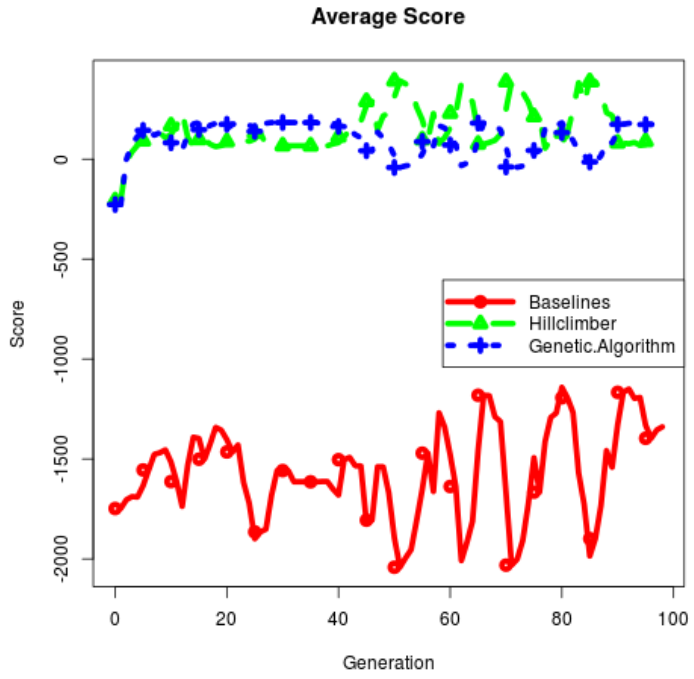


Figure 4.5: Avg. score of CA population against different opponents.

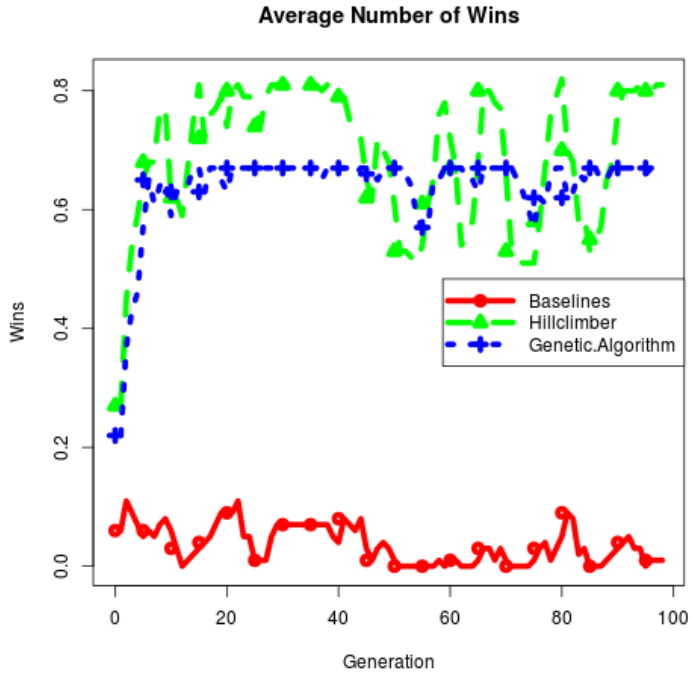


Figure 4.6: Avg. win rate of CA population against different opponents.

progress coevolution makes by playing all members of the population at each generation against three sets of opponents: the three best BOLs produced by the GA, the 32 BOLs produced by the HC, and the same three hand-coded baselines used to evaluate the GA and HC BOLs.

Figure 4.5 shows that coevolution very quickly moves to increase the average score of the population, but not by very much. However, this slight increase in average score has a huge affect on the number of wins the BOLs achieve, as shown by Figure 4.6. During the first ten generations, the increase in score leads to an increased number of wins against the GA and HC, but a decreased number of wins against the baselines. In later generations, an increase in score correlates to an increase in wins for the GA and baselines, but a decrease in wins for the HC. While Figure 4.5 and Figure 4.6 shows that our coevolved BOLs are not as good against the baselines as the GA or HC results, both figures seem to indicate that an increase/decrease in performance against the GA and HC BOLs correlates to an increase/decrease in performance against the baselines. Finally, Figure 4.4 shows how well on average the CA, GA, HC, and exhaustive search performed against the all three baselines. While the CA was usually able to do better than the average of exhaustive search against the baselines, the CA does not perform as well as the GA or HC BOLs, which were tuned using the baselines.

Analysis of the populations at generations with high average scores showed that the favored BOL was to build mostly Vultures followed by a one or two Firebats. One of the BOLs found by the GA was the opposite of this, preferring to build mostly Firebats followed by the Vultures. These two BOLs cost the same to construct, however the BOL found by coevolution provides a stronger defense in exchange for taking longer to complete. Populations at generations with low

average scores had similar BOLs, but issued an attack command as the final action. Attacking with multiple Vultures and Firebats can inflict heavy losses, before the attacking units are destroyed by the defending units. However, against opponents that are faster to attack or build up an equally strong defense, such as with our baselines, GA produced BOLs and HC produced BOLs, the attacking force is wiped out to quickly to benefit.

4.1.1 Conclusion

This section compared a hillclimber, genetic algorithm, coevolutionary algorithm, and exhaustive search for generating build-orders to defeat opponents in real-time strategy games. We used three different hand-coded build-orders as baselines upon which to make our comparisons. In order to make exhaustive search possible, we restricted our search space to 5-action BOLs. Once we performed exhaustive search, we generated build-orders using a GA, CA, and HC.

Our results show that the HC quickly generates BOLs that defeat our baseline opponents, but does not find the best BOLs reliably. The HC finds the best BOLs only about 6% of the time starting with different random seeds. The GA, on the other hand, reliably (100% of the time) finds the best possible BOLs, but takes longer than the HC to find them. Coevolution finds BOLs that defeat or tie against the GA and HC BOLs approximately 80% of the time, showing that the coevolved BOLs defeat BOLs previously shown to be capable of defeating our challenging baselines. When compared directly against those same baselines, our coevolved BOLs increased their performance over time, and defeated or tied the baselines 20% of the time without being trained against the baselines beforehand. Clearly,

this shows our game exhibits the rock-paper-scissors balance. While our results show coevolved 5-action BOLs can beat other challenging 5-action opponents, we do not yet know how well these BOLs would rank in an exhaustive list of all 2^{15} BOLs played against all 2^{15} BOLs.

These results help specify trade-offs to be made when choosing between GAs, CAs, and HCs in the kind of problem spaces found in RTS games. The results also agree well with our understanding of GA, CA, and hill-climbing theory. Our results indicate that if you are interested in a quick satisfying solution but not overly worried about optimality, a HC will probably work best. On the other hand, if you are interested in high quality build-orders you should probably use a GA. Using a GA against a set of hand-tuned opponents produces BOLs that defeat those opponents. Lastly, if you do not want to hand-code training opponents, a CA can bootstrap opponents and generate robust build-orders. While CA produced BOLs that were robust against other 5-action BOLs, the CA BOLs were not robust against our hand-coded baselines. However, all three baselines used more than five actions, and had a large advantage against the 5-action BOLs. The quality of BOLs found by our GA and CA would be different if the BOLs were more evenly matched against the baselines. As such, our next approach experiment extended the length of the BOLs generated by the GA and CA to 13-actions (39-bits), the length of our longest baseline.

4.2 13-action BOL

We extended our chromosome bit-length from 15-bits (5-actions) to 39-bits (13-actions) and ran our GA and CA eight times with a population size of 50 for 50

generations. Because 13-actions is to large to exhaustively search, we measure our progress by taking our best BOL produced by the GA, our best three CA BOLs, our best three BOLs in the CA teachset, our three baselines, ten randomly generated BOLs and having them all compete against each other. BOLs were selected from the CA based on how many different opponents the BOLs could beat in the final generation. The GA converged to single BOL which could defeat all three baselines. The BOL found by the GA built two SCVs, several Firebats, another SCV, several more Firebats, two Vultures then attacked. This BOL defeats all three baselines by destroying the opponent's Command Center. Interestingly the BOL plans to build the third SCV seconds after losing SCVs in an attack from Baseline Large. The CA found three BOLs that were particularly effective. One BOL found by the CA built two SCVs, followed by mostly Vultures and a few Firebats, then attacked. This BOL provides a strong defense against weaker opponents, while building up a large army for a powerful attack. A different BOL found by the CA used two attack actions, one attack after the first five units were built and a second attack after the next five units were built. This BOL disrupts an opponent that plans to build a more powerful army, and allows the second attack force to destroy the Command Center. The final BOL focused on only defense by building Firebats and one Vulture as the final action, with no additional SCVs. This BOL maximizes the score based only on resources spent, so that it can outscore other defensive opponents that never attack. This BOL also defends against very strong and slow opponents, but leaves itself vulnerable to opponents that attack early. We ran our CA eight times, and each time similar BOLs would start to appear between generation 25 and generation 40.

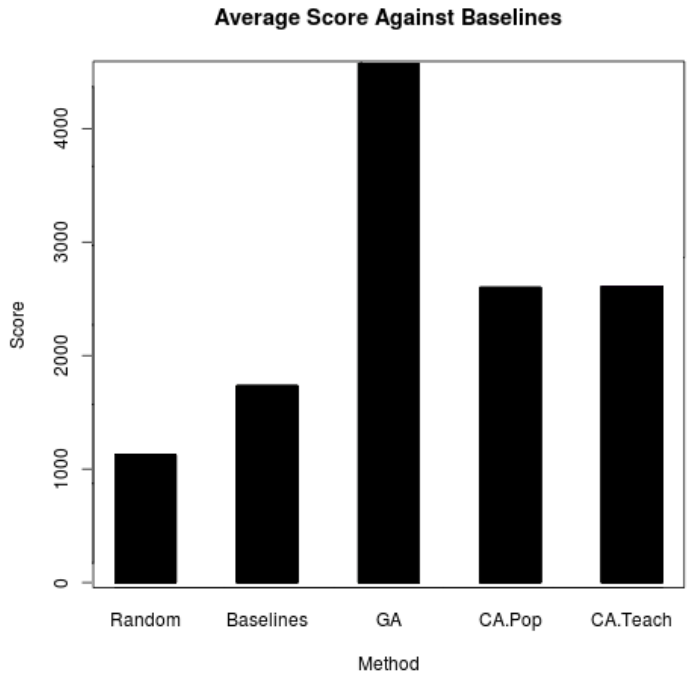


Figure 4.7: Avg. score of 13-action BOLs against three baselines.

Figure 4.7 shows us the average score the BOLs from our GA, CA, and random creation achieved against our three baselines. We can see that the average score for the random BOLs is only 1000 and is clearly the lowest average among all the BOLs. Not surprisingly, the best GA BOL performs better against the baselines than any other approach. The best GA BOL performs much better against the baselines since the GA used only the baselines to train, producing a BOL optimized for defeating the baselines. However, this means the best GA BOL over specialized, and does not perform as well against BOLs produced by the other approaches. Table 4.1 breaks down the results to show the average score of BOLs from each approach got when they competed against each other, and highlights the highest score against approach. From Table 4.1 we can see that although the best GA BOL gets the highest score against the baselines, the best CA BOLs have

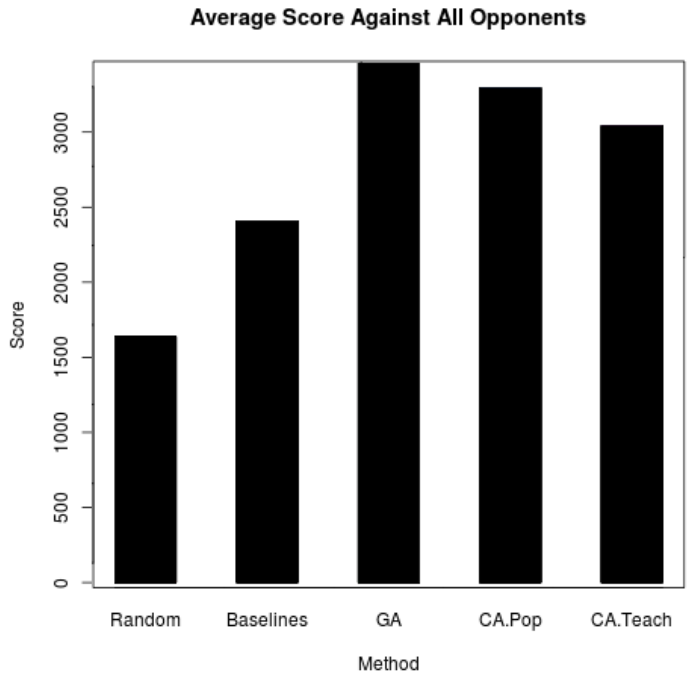


Figure 4.8: Avg. score of 13-action BOLs against each other.

Table 4.1: Avg. score of 13-action BOLs against opponents.

Player \ Opponent	Baselines	Best GA	Best CA	Best Teachset	Random
Baselines	1733.33	2341.66	1975.0	1775.0	2935.0
Best GA	4591.66	2875.0	2175.0	2833.33	3573.33
Best CA	2600.0	3925.0	2830.55	3322.22	3775.0
Best Teachset	2611.11	3533.33	2355.55	2877.77	3379.99
Random	1124.16	2017.5	1456.66	1498.33	1851.0

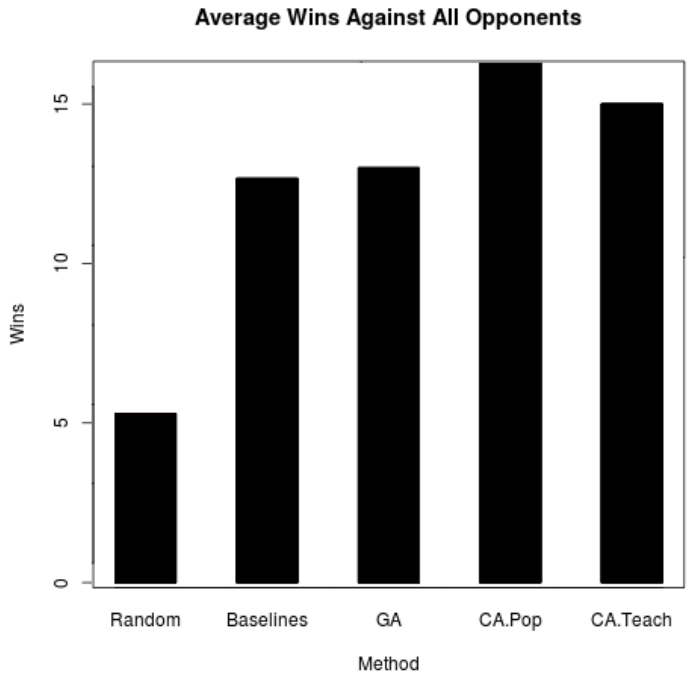


Figure 4.9: Avg. number of wins of 13-actions BOLs against each other.

Table 4.2: Avg. wins of 13-action BOLs against opponents.

Player \ Opponent	Baselines	Best GA	Best CA	Best Teachset	Random
Baselines	1.0	0.33	1.33	1.0	9.0
Best GA	3.0	1.0	0.0	1.0	8.0
Best CA	2.0	1.0	1.33	2.0	10.0
Best Teachset	2.0	0.66	1.0	1.33	10.0
Random	0.30	0.4	0.0	0.0	4.5

a higher average score against BOLs from every other approach. As expected, we also see that the random BOLs average score is the lowest against all the BOLs. Although the best CA BOLs perform better overall, the extremely large score of the best GA BOL against the baselines skews the total average as seen in Figure 4.8, which shows the average score of the BOLs from each approach against each other. If we account for this outlier, the results are more in-line with what we would expect, with the best CA BOLs having the highest average, followed by the best GA BOL and the baselines.

Setting aside how well these BOLs can outscore their opponents, we can clearly see that the best CA BOLs defeat more opponents on average in Figure 4.9, which shows the average number of wins the BOLs from each approach got against each other. At first it may seem that the random BOLs do fairly well, but by breaking these results down into Table 4.2, we get a different picture. From this table we see that the majority of the random BOL wins are against other random BOLs, only a couple of random BOLs win against one of the baselines or the best GA BOL. As expected, the best GA BOL wins against the baselines more than BOLs from any other approach, in fact the best GA BOL always beats all three baselines. However, the CA BOLs get the highest average number of wins against BOLs from every other approach, and still manages to beat two baselines on average despite never training against any of the baselines.

While simply outscoring your opponent still counts as winning, we want to find BOLs capable of destroying an opponent's Command Center whenever possible. Figure 4.10 shows us on average how many Command Centers the BOLs from each approach destroy when competing against each other. We can clearly see that the random BOLs can not destroy any Command Centers, while the BOLs

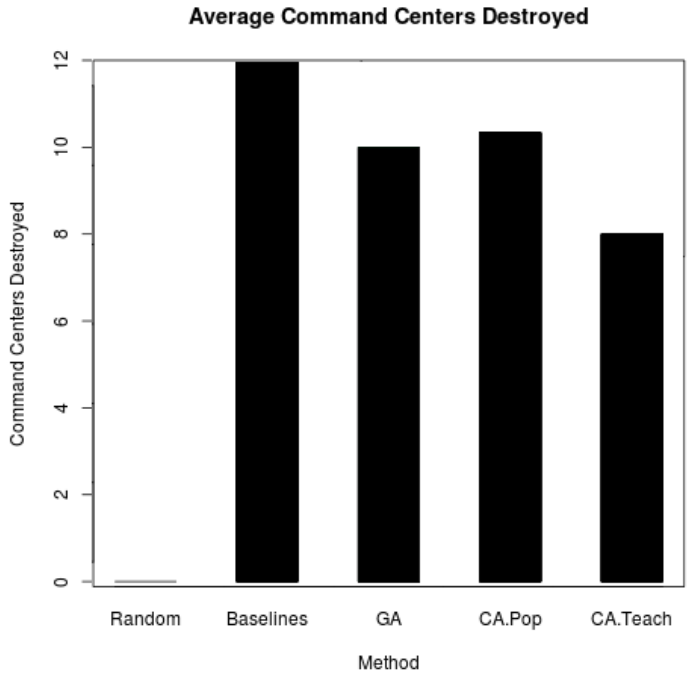


Figure 4.10: Avg. number of Command Centers destroyed by 13-action BOLs against each other.

Table 4.3: Avg. Command Center kills of 13-action BOLs against opponents.

Player \ Opponent	Baselines	Best GA	Best CA	Best Teachset	Random
Baselines	0.66	0.33	1.33	1.0	8.66
Best GA	3.0	0.0	0.0	1.0	6.0
Best CA	1.33	0.66	0.33	1.33	6.66
Best Teachset	1.33	0.33	0.0	0.33	6.0
Random	0.0	0.0	0.0	0.0	0.0

from other approaches are very similar to each other. In addition to being unable to destroy the opponent's Command Center, the random BOLs nearly all lose by having their Command Center destroyed, as shown in Table 4.3, These results show that winning BOLs are not trivial to find, and simply defending your own Command Center proves difficult. We can also see from Table 4.3 that, once again, the best GA BOL manages to destroy the Command Center of all three baselines. Table 4.3 and Figure 4.10 seem to indicate that on average the three baselines destroy more Command Centers than the CA BOLs. However, bear in mind that while all three baselines eventually attack, one of the three CA BOLs never attacks and therefore can never destroy a Command Center. So despite being having only two attacking BOLs, on average the best CA BOLs destroys more Command Centers than the three attacking baselines against three of the approaches, and only does slightly worse than the baselines against random BOLs. This shows that our best CA attacking BOLs are very effective at winning.

Finally, I took on the role of the human player and competed against our three baselines, GA BOL and three best CA BOLs. I am an experienced RTS game player who has played several different RTS games. I have has shown myself capable of defeating the hardest AI settings and other moderately experienced human players in these RTS games, such as Gold and Platinum ranked players in StarCraft II. Initially, I was restricted to only taking the same actions the game AI was capable of and limiting myself to taking 13 actions, the same number of actions encoded by our BOLs. Under these restrictions, I found that the best CA BOLs were the hardest to overcome, never winning a game against the CA BOLs. We then removed the restriction on the number of actions, and allowed myself to take as many actions as I wanted. Eventually I learned several build-orders to defeat the CA BOLs, but took at least 25 actions to win against the CA BOL's 13 actions. We then removed

the other restriction, and I made moves the game AI was incapable of. This allowed the me to use strategies not seen by the opponent BOLs, and made winning much easier. For example, our baselines, GA BOL and CA BOLs never send the SCVs to attack their opponent's Command Center, so the BOLs found by the CA and GA never prepare for this contingency. If I sent my SCVs along with the rest of my attack force to damage the opponent's Command Center, the opponent could easily be defeated. Even when using unrestricted strategies, I noted that the best CA BOLs still took the longest to defeat.

4.2.1 Conclusions

This section evaluates the performance of real-time strategy game BOLs produced by coevolution and compares them to the performance of three hand-coded baselines and the BOLs found by a genetic algorithm. We also had a human player (me) play against these BOLs to gauge their difficulty. In the previous section, we limited ourselves to 15-bit BOLs, in order to make exhaustive search possible. However, 5-action BOLs are at a large disadvantage against the larger baselines. To even the playing field for the GA and CA generated BOLs, we increased the bit length to 39-bits (13-actions) and tested how BOLs produced by genetic algorithms and coevolution do against the baselines and each other. Our results show that while the best BOL produced by the genetic algorithm scores the highest against the three baselines, the best BOLs produced by coevolution could defeat two of the baselines and the genetic algorithm's BOL. That is, coevolution produced more robust BOLs. The coevolutionary results were more robust as they maintained a higher average score and destroyed their opponent's Command Center more often than the genetic algorithm BOL. A human player that fought against these BOLs found

that the coevolutionary results were the most difficult to overcome. However, by using a strategy not encodable in our representation, the human player could, as expected, easily beat the best coevolutionary BOLs.

These results indicate that while genetic algorithms perform better against specific opponents, coevolution produces more robust BOLs. However, BOLs generated by coevolution are not guaranteed to defeat specific build-orders we are interested in. Additionally, we were later able to create new build-orders to defeat the CA BOLs. In order for our CA to adapt to specific opponents, we require another approach that enables a CA to learn from specific opponents. To address this problem, we investigated case-injection into coevolution.

CHAPTER 5
PHASE TWO: CASE-INJECTION

In Chapter 4, we showed that a CA produces robust BOLs, but may not defeat specific opponents we are interested in. In order to enable our CA to learn from specific opponents, this chapter investigates case-injection. There are two ways we can introduce specific build-orders into a CA: case-injection into the CA's teachset, or case-injection into the CA's population. Because our primary goal for case-injection is enabling the CA to learn to defeat specific opponents, we first investigate teachset injection.

5.1 Teachset Injection

In the previous chapter, I acted as the human player and competed against the BOLs produced by a GA and CA. We recorded my actions as I competed against the CA BOL, and found two significantly different BOLs that could win. The first winning BOL I used, which we call the Easy Human (EH) BOL, was to build two Marines, attack, and repeat six times. This BOL slowly chipped away at the CA BOL's Command Center, while also destroying some of the CA BOL's SCVs early on and slowing down how quickly structures were built. The second winning BOL I used, which we call the Hard Human (HH) BOL, was to build nine SCVs, then build seven Firebats and seven Vultures in parallel. This BOL builds a strong defense and waits to destroy the CA BOL's attack force. Once I destroyed the CA BOL's attack force, I build a few more units and destroys the CA BOL's Command Center. Both human BOLs required 75-bits to encode, compared to the 39-bits the CA BOL used. We then reinitialized the CA population to random chromosomes

and injected the two human BOL into the CA's teachset. We also removed the baselines from our GA, and instead ran the GA against only the human BOLs. We ran our GA and CA ten times, and used the average score of the entire population at each generation for our results.

Our results show that the EH BOL was trivial to beat, and often could be beaten by random chromosomes, as shown by Figure 5.1. On the other hand, the HH BOL proved to be overwhelmingly difficult, and was never defeated in the initial generation. This balance issue caused a problem for the GA. Although improving

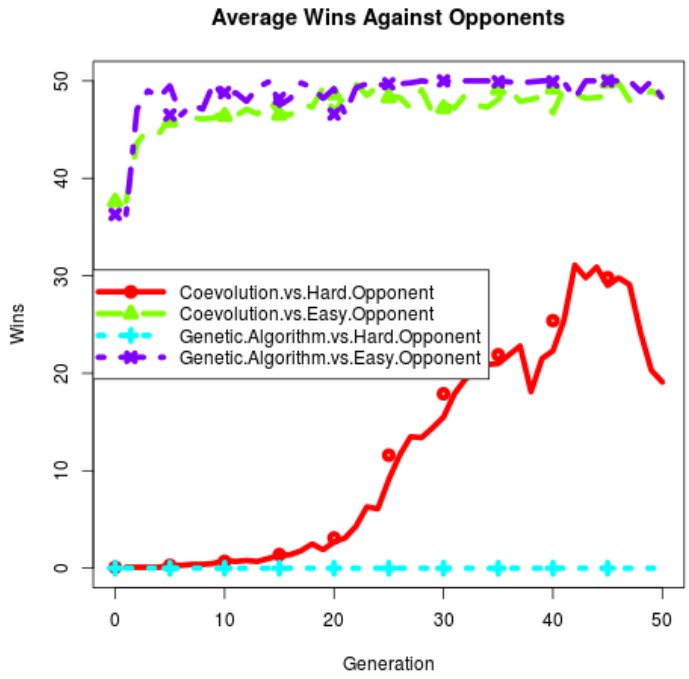


Figure 5.1: Avg. number of wins of 13-action BOLs against human build-orders.

the score against the EH BOL also slightly improved the score against the HH BOL, as shown in Figure 5.2, the improvement did not lead to successful BOLs against the HH BOL. As a result, the GA produced BOLs that were overspecialized to defeat the EH BOL. The BOLs found by the GA quickly build two SCVs and a

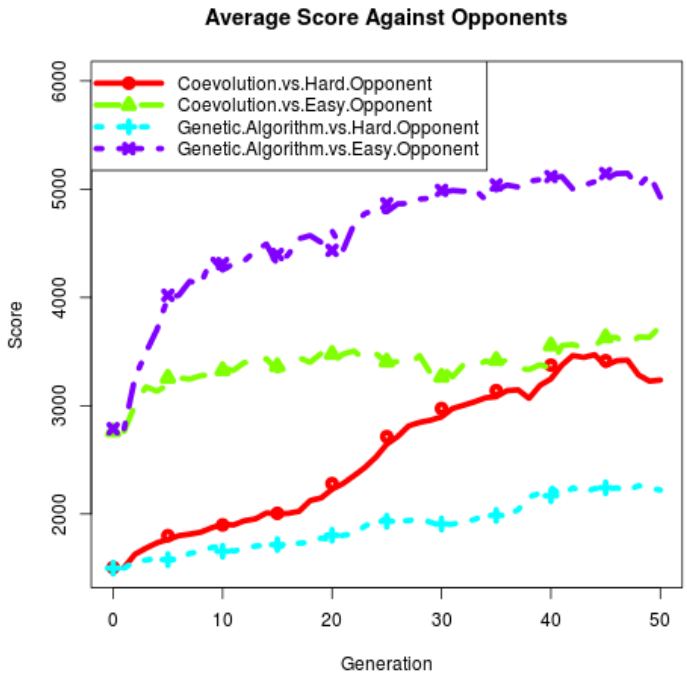


Figure 5.2: Avg. score of 13-action BOL build-orders against human build-orders.

couple Firebats to ward off the Marines while minimizing casualties, followed by a mix of Firebats and Vultures used to attack the opponent's base. The GA finds the best BOL to defeat the EH BOL after about thirty generations.

However, our CA was able to quickly find solutions to defeat both human BOLs, as shown in Figure 5.1. The CA typically found at least one BOL to defeat the HH BOL by generation ten. We also see that the average score against the EH BOL peaks almost immediately, while the GA continues to improve and overspecialize over all fifty generations. While the CA also immediately finds BOLs to defeat the EH BOL, the other opponents in the teachset force the CA to explore BOLs that defeat different opponents and prevent the CA from overspecializing at the beginning. This leads to enough diverse BOLs that work well in general that the CA finds solutions that also work against the HH BOL. These BOLs build two

SCVs, followed by Vultures, then attack. This BOL loses a few more units to the EH BOL than the GA does, but enables the BOL to build a strong enough defense to defend against the HH BOL. As Figure 5.2 shows, this compromise lowers the score against the EH BOL significantly compared to the BOLs found by the GA but dramatically increases the score against the HH BOL.

5.1.1 Conclusions

In this section, we want to find robust BOLs that also defeat specific opponents. In the previous chapter, we compared BOLs found by a genetic algorithm trained against three baselines to BOLs found by a coevolutionary algorithm. We then had a human player (me) compete against the BOLs produced by the genetic algorithm and coevolutionary algorithm. The human player found that the BOLs produced by the coevolutionary algorithm were the most challenging to defeat.

This section expands upon the previous chapter by introducing case-injection to our coevolutionary algorithm's teachset, and comparing BOLs found by the coevolutionary algorithm to the BOLs found by a genetic algorithm. We had our human player compete several times against the most challenging BOL found by coevolution in the previous chapter. As the human player competed, we recorded their actions to a bit-string, allowing us to replicate their actions and outcome against the opponent. We recorded two different successful BOLs the human player used against coevolution's BOL. We then used the human player's BOLs to evaluate BOLs using a genetic algorithm, and injected the same two player BOLs into coevolution's teachset.

Our results showed that when an opponent poses a significant challenge, the

genetic algorithm will not find BOLs to defeat the opponent, and will instead over specialize against the easier opponent. This was unexpected, since genetic algorithms have been shown to produce good BOL against the opponents used in training. However, when there are multiple opponents with a large difference in difficulty, the genetic algorithm may converge too quickly on BOLs for the easy opponents, which does not lead to solutions for the harder opponents. On the other hand, coevolution finds BOLs to defeat both opponents after ten generations. Coevolution finds BOLs to beat the challenging opponent because coevolution uses a diverse teachset that gradually increases in difficulty. This prevents coevolution from converging to quickly, and allows coevolution to move towards BOLs more capable of defeating the challenging opponent, even if no BOLs currently beat that opponent. We also show that coevolution increases the population's average score against the challenging opponent much faster than the genetic algorithm does. These results are interesting compared to our previous studies, where our genetic algorithm produced better BOLs than coevolution for defeating specific opponents, despite the genetic algorithm only competing against the same three opponents while coevolution competed against eight opponents that changed over time. This shows that while genetic algorithms can produce good BOLs to defeat specific opponents, genetic algorithms can also be mislead if only a few opponents are used for evaluation, or if the opponents have a large gap in difficulty.

These results indicate that teachset case-injection helps coevolution to quickly find BOLs that defeat specific opponents, while maintaining robustness against other opponents. However, learning to defeat specific opponents is not the only way a CA can learn from our stored cases. Learning to play like a specific opponent will allow the CA to quickly find winning BOLs and adapt the specific case into

new BOLs, but may also affect the robustness of the generated build-orders. In order to see if a CA can learn to play like a specific case, and if learning to play like a specific case affects BOL robustness, we expanded our investigation to case-injection into a CA's population.

5.2 Population Injection

While the previous section only tested teachset injection, this section tests four case-injection scenarios: no case-injection, case-injection into only the teachset, case-injection into only the population, and case-injection into both the teachset and population. When we do case injection into the teachset, every generation we select two random BOLs from our case-base and put them in the teachset. With case injection into the population, every five generations two random BOLs are selected from our case-base to replace the two lowest shared fitness chromosomes in the population. There are five cases we select for injection into the CA, which were all hand-coded produced by our CA and GA in the previous chapters. We ran each of our four injection scenarios ten times with a population size of 50 for 50 generations with a chromosome length of 39-bits (13-actions). We measure the robustness of the BOLs produced by comparing them to five test cases that are never seen during training. These test cases are different from the cases used for case injection, but were also produced from hand-coding or coevolution in our previous studys and were either robust or defeated our best evolved BOLs. Measuring the robustness of every chromosome in every population would take excessively long, so we limit ourselves to testing the *best* chromosome (the chromosome with the highest shared fitness) in the population for every generation. We measure robustness by examining the average score and number of wins against all five testing cases. We

also measure the Hamming Distance of the best chromosome to each of the five injected cases [30]. We compare the all bits of two chromosomes, and for each bit-position if the bit-value in one chromosome does not match the bit-value in the other chromosome, we increase the Hamming Distance by one. This means that the higher the Hamming Distance between two chromosomes, the less similar they are to each other. Conversely, the lower the Hamming Distance between between two chromosomes, the more similar they are to each other. This tells us how much case injection has influenced the population to play like the injected cases. Finally, our results are calculated by taking the average of the best chromosome for each generation across all ten runs.

Our results show that the influence of case injection on a population varies based on the cases and injection methods. We limit ourselves to showing the similarity to only three of the five injected cases because they exemplified the results of the remaining two. Our figures show that without case injection, that the best chromosomes tends to become less similar to Training Case 1 (Fig. 5.3), more similar to Training Case 2 (Fig. 5.4), and have little change in Hamming Distance to Training Case 3 (Fig. 5.5). Injecting cases into only the teachset has little to no effect on influencing the best chromosome to play like the injected cases. All three figures show that the best chromosomes produced from teachset injection have almost the same Hamming Distance as chromosomes produced without case injection.

When we inject chromosomes into only the population or into both the population and teachset, the results vary a bit more. Fig. 5.3 shows that after we inject our first chromosomes into the population at generation 5, over time the best chromosomes tend to become more similar to Training Case 1 than the chromosomes

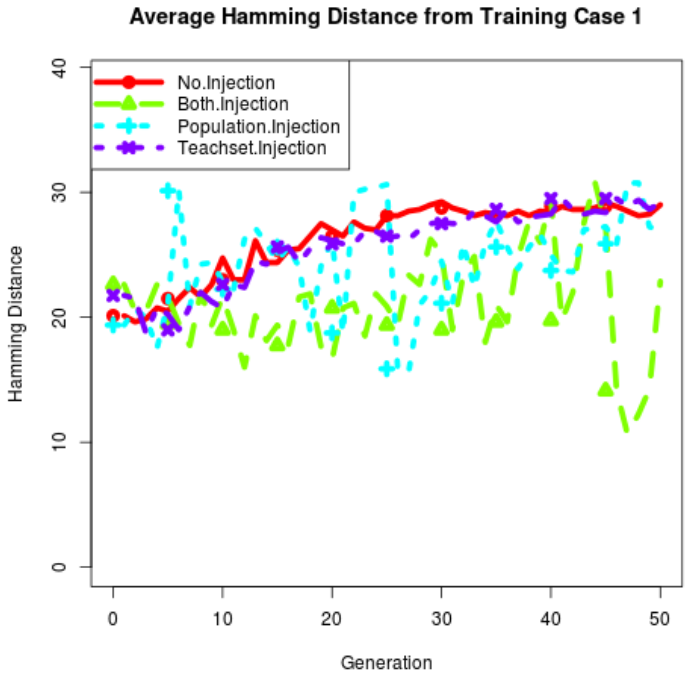


Figure 5.3: Avg. Hamm. Dist. of 13-action BOLs to Case #1.

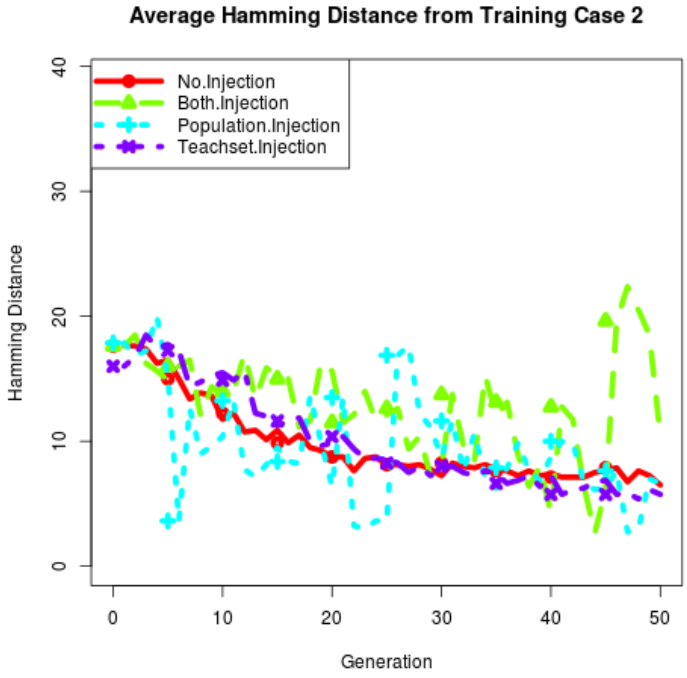


Figure 5.4: Avg. Hamm. Dist. of 13-action BOLs to Case #2.

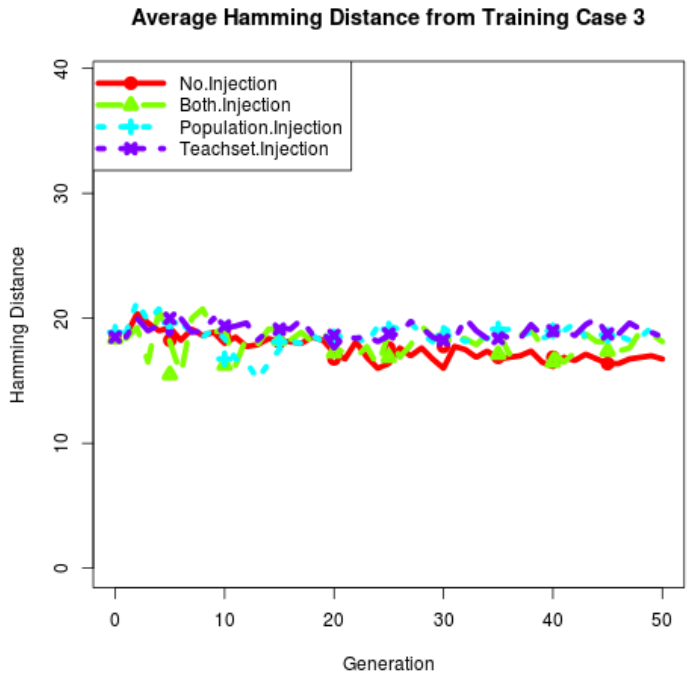


Figure 5.5: Avg. Hamm. Dist. of 13-action BOLs to Case #3.

produced without case injection. On the other hand, Fig. 5.4 shows the opposite effect. While the best cases produced without case injection tend to become more similar to Training Case 2, the best cases produced from population injection tend to become less similar over time. However, when we look at Training Case 1 and Training Case 2, we see that with a Hamming Distance of 32 (out of a maximum of 39) from each other these two cases are dissimilar. As one of these cases begins to influence the population to play in a similar manner, then naturally that means the population begins to play less like the other case. We see happening in Fig. 5.3 and Fig. 5.4 , since the results in each almost perfectly mirror each other. Training Case 1 builds two SCVs, ten Marine, then attacks, while Training Case 2 builds ten Vultures instead of Marines. The coevolved solutions that are similar to Training Case 1 replaces three of the Marines with an additional two SCVs at the beginning and a Firebat towards the end, allowing for an earlier attack that is suc-

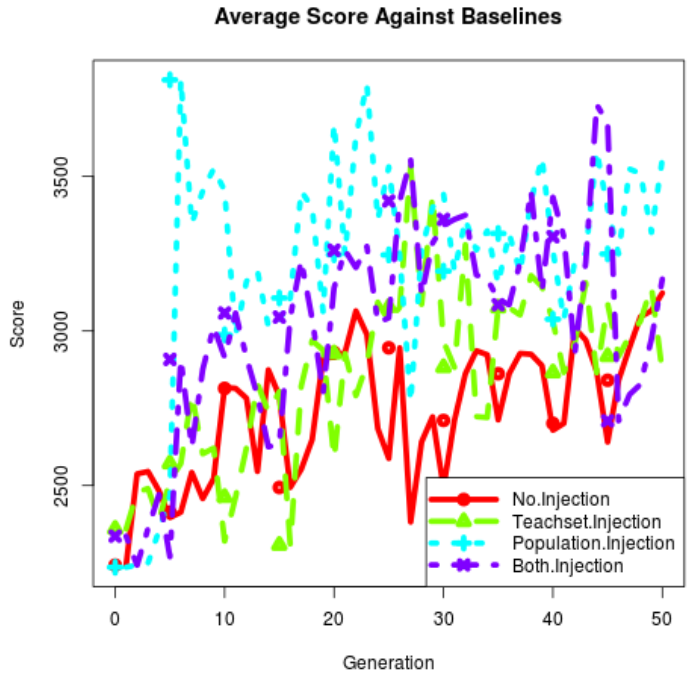


Figure 5.6: Avg. score of 13-action BOLs vs all Testing Cases.

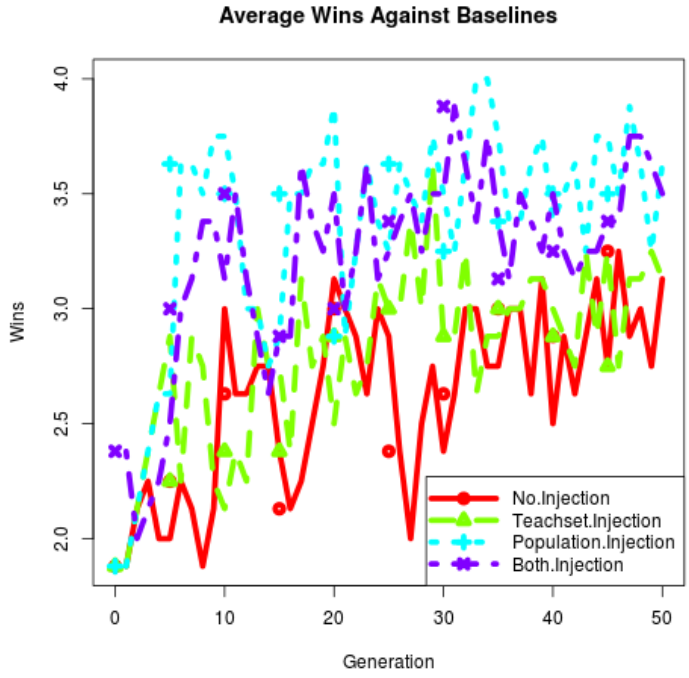


Figure 5.7: Avg. number of wins of 13-action BOLs vs all Testing Cases.

cessful against opponents that produce units slowly. Coevolved solutions similar to Training Case 2 replaced one SCV and the attack command with a Firebat and additional Vulture, allowing for a stronger defense force that takes longer to produce. Finally, while Training Case 1 influences the best cases to play in a similar manner, Fig. 5.5 shows this does not affect the best cases' similarity to Training Case 3, which has a Hamming Distance of 19 to Training Cases 1 and 2.

While our CA manages to learn new BOLs from injecting cases into the population, we also want our coevolved BOLs to defeat a wide variety of opponents. We test the robustness of the BOLs the CA produces by playing them against five chromosomes we had previously hand-tuned or coevolved and that were never seen during training. Our results show that while case injection into the population increases the score against unknown opponents in early generations, BOLs produced in the long term score only slightly higher than our other injection methods, as shown in Fig. 5.6. Fig. 5.7 also shows that over time our coevolved BOLs win against more of the opponents. While Fig. 5.3 shows us that population injection influences our best chromosomes to play like some of the injected cases, Fig. 5.6 and Fig. 5.7 show us that our best BOLs continue to be at least as robust as coevolution without case injection.

5.2.1 Conclusions

In this section we want to find robust, winning BOLs for Real-Time Strategy games. We also want these BOLs to incorporate knowledge from winning BOLs other players have used. We believe we can accomplish these goals by using case injection with a coevolutionary algorithm. Case injection takes BOLs contained in

a case-base, and injects them into our coevolutionary algorithm's population or teachset. There are four case injection methods that we examine: case injection into only the population, case injection into only the teachset, case injection into both the population and teachset, and no case injection. We used each of these methods ten times with a population size of 50 for 50 generations, and took the average across all ten runs for the best chromosome at each generation. When we do case injection into the teachset, every generation we select two random BOLs from our case-base and put them in the teachset. With case injection into the population, every five generations two random BOLs are selected from our case-base to replace the two lowest shared fitness chromosomes in the population. There are five winning BOLs contained in our case-base that were produced from hand-tuning or coevolution.

Our results show that while injecting into only the teachset does not affect the similarity to the injected cases, injecting cases into the population had different effects. We measured the similarity by calculating the Hamming Distance from chromosomes in the population to each of the five cases in our case-base [30]. Some injected cases affected the population more than others, influencing the chromosomes to play like some injected cases while also influencing to play unlike other injected cases. For other cases, there seems to be no change in Hamming Distance to the population. This shows that a coevolutionary population can be influenced to play like injected cases, but some injected cases may have more influence than others. However, we do not want chromosomes that learn to play like someone else, to also inherit their flaws and lose robustness.

We tested the robustness of our results by competing the best chromosomes against five chromosomes never seen during training. These testing chromosomes

are different from the chromosomes used for case injection, but were also produced from hand-tuning and coevolution. Our results showed that although injecting cases into the population influenced the best chromosomes in the population to play like the injected cases, these new chromosomes did not lose robustness, and defeated at least as many opponents as BOLs produced from coevolution without case injection.

These results indicate case injection into a coevolutionary population will influence coevolution to quickly produce similar winning BOLs, while maintaining robustness. This informs our research into adapting new BOLs from previously encountered opponents. Our results thus far have indicated that a CA is a viable approach for generating robust BOLs. We are also interested in finding build-orders which are strong, and defeat opponents quickly. While our CA produced winning build-orders with our BOL representation, BOL only encodes a single, arbitrary length sequence of actions. We believe that a representation that encodes multiple sequences would enable a CA to produce stronger build-orders. As such, we extended our representation from BOL to BOIL.

CHAPTER 6

PHASE THREE: BUILD-ORDER STRENGTH

We believe that adding branches and loops to our BOL representation will enable our evolutionary methods to produce build-orders with better performance. However, investigating this new representation requires a large amount of evaluations that WaterCraft could not complete in a reasonable amount of time. As a result, instead of using WaterCraft for this investigation, we changed our RTS environment to BOSS and SparCraft. In order to test BOSS and SparCraft, we performed exhaustive search on the outcome 5-action BOLs against our three hand-coded baselines in BOSS. As Figure 6.1 shows our results are qualatatively the same, Baseline

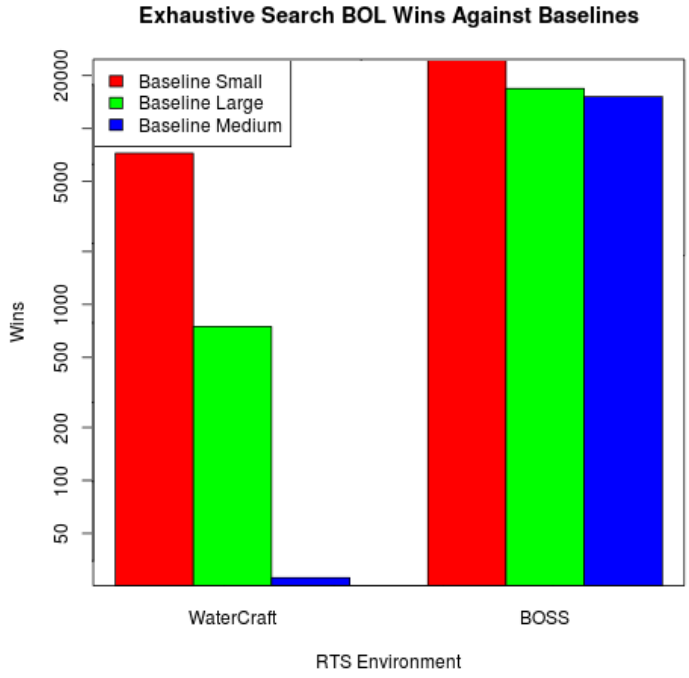


Figure 6.1: Comparing 5-action BOL outcomes in WaterCraft and BOSS.

Small is the easiest to defeat while Baseline Medium is the most difficult to defeat. However, Baseline Large and Baseline Medium are significantly easier to defeat

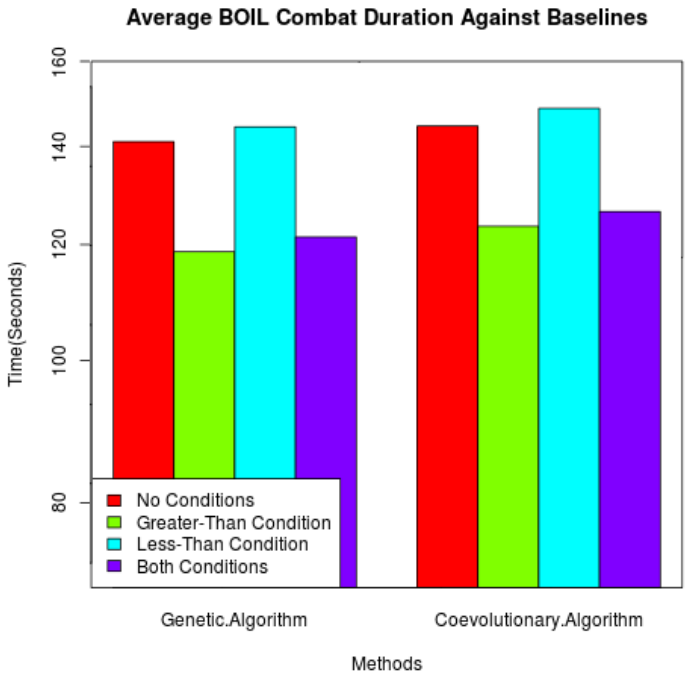


Figure 6.2: Avg. combat duration of 13-action BOILs.

in BOSS than in WaterCraft. Due to the different gameplay and unit properties, the baselines pose less of a challenge in BOSS than in WaterCraft, because BOSS and WaterCraft are functionally different games. Despite the differences, our approaches should work in BOSS equally as they would in WaterCraft or any other RTS game. As such, we use BOSS and SparCraft to continue our research into our BOIL representation.

BOIL removes the automatic dependency resolution for units, requiring the CA and GA to determine which prerequisites are needed and when. Without automatic dependency resolution, finding valid build-orders becomes more difficult, but also allows the evolutionary methods to determine how much unit production infrastructure to build, allowing quicker production of combat units. We use BOSS to compare GA and CA results to four BOIL configurations: BOIL with no encoded

branches/loops (acts as our baseline), BOIL with one encoded condition (less than seven available units), BOIL with one encoded condition (greater than or equal to seven available units), and BOIL with both conditions. Build-orders are given a maximum of seven minutes to construct units. After seven minutes the available units for both players are forced to attack, though players can choose to attack and begin combat earlier in the game. We run our GA and CA for all cases 50 times, with a population of 50 for 100 generations and take the average of the results in each generation. We restrict BOIL to 13-build actions, to keep consistent with our previous research.

Our results in Figure 6.2 show that for all BOIL representations, the CA finds build-orders that have close to the same performance as the GA build-orders. We attribute this result to the baselines being easier in BOSS than in WaterCraft. Due to the collision detection limitations in SparCraft, SCVs and Marines are more effective in SparCraft than in WaterCraft. Build-orders produced by the CA and GA are biased towards building many SCVs and Marines, and waiting for the opponent to attack. The CA bootstraps opponents more difficult than the baselines, leading the CA to find build-orders that defeat the baselines as quickly as the GA build-orders do.

Build-orders produced with BOIL's less-than condition perform the worst, taking 10 seconds longer to defeat all opponents in combat. The GA and CA use the less-than condition in the beginning of the build-order to build four SCVs and two Barracks. However, since players start the game with 5 SCVs, players negate the condition after finishing the construction of two units. The GA and CA organize build actions in the loop such that the 7th total unit finishes constructing after the condition for the 2nd iteration of the loop evaluates as *true*, allowing the loop to

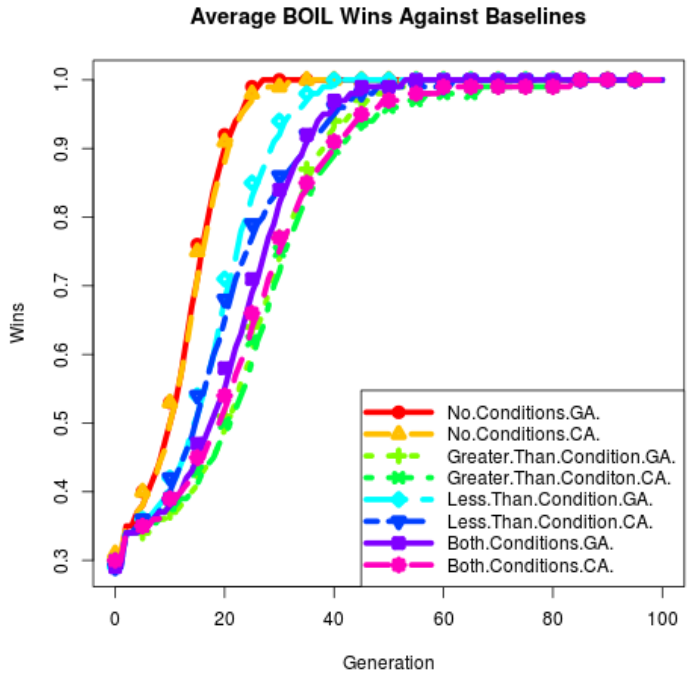


Figure 6.3: Avg. wins of 13-action BOILs from different approaches.

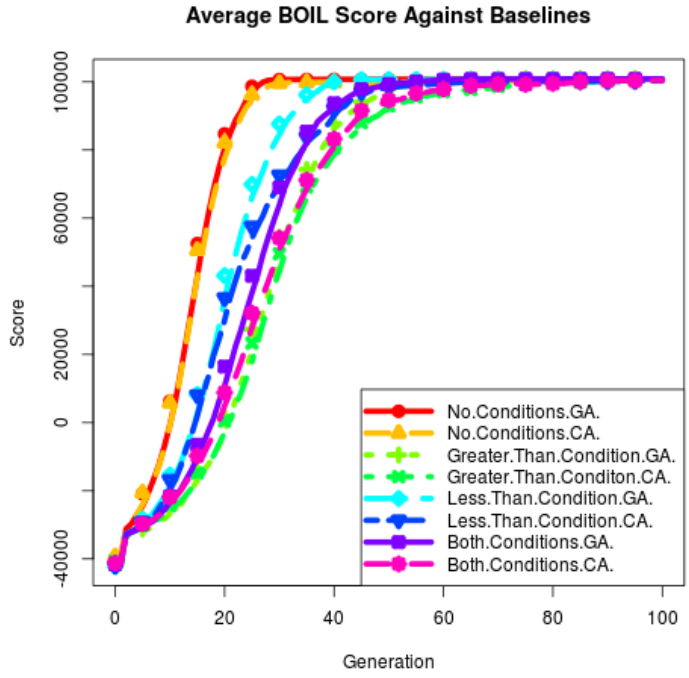


Figure 6.4: Avg. score of 13-action BOILs from different approaches.

repeat once. After finishing the loop with BOIL's less-than condition, only a finite number of build actions remain. The remaining build actions make use of the Barracks previously built by constructing Marines and SCVs in parallel with the remaining build actions.

With the greater-than condition, the GA and CA first builds a Barracks and SCVs outside the loop, then use the greater-than condition at the end of the build-order to build as many Marines and SCVs as possible. BOIL that utilizes the greater-than condition or both conditions wins combat the quickest, winning in combat 20 seconds faster than BOIL with no conditions. With both conditions, BOIL utilizes the less-than condition to build multiple Barracks and SCVs in a loop, constructs an additional Barracks and SCVs once outside the loop, then uses the greater-than condition to infinitely loop through build actions to quickly produce Marines and SCVs. BOIL only utilizes the branch/IF instruction inside of loops to extend the length of the loop. Unlike with BOL, interpreting the behaviour of BOIL build-orders is difficult without observing the build-order during a game. Nesting loops and branches makes deciphering which units are built in which order hard for humans.

Figure 6.3 and Figure 6.4 shows us that while all the BOIL methods learn to quickly defeat the baselines, no condition BOIL learns the fastest. BOIL with conditions takes longer to tune than without conditions, and the less-than condition takes longer to defeat all opponents. However, Figure 6.5 shows by incorporating both conditions, BOIL outperforms the single-condition BOILs, as well as no condition BOIL. Two-condition BOIL takes over 50 generations to defeat opponents as quickly as no condition BOIL, but ultimately finds build-orders that defeat opponents faster. Figure 6.5 also shows on average, random 13-action BOIL build-

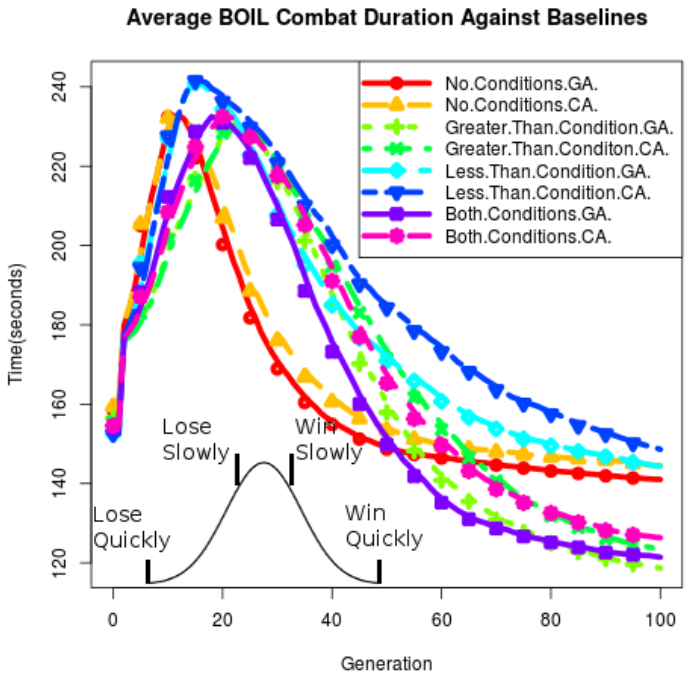


Figure 6.5: Avg. combat duration of 13-action BOILs from different approaches.

orders only win 30% of the time against three baselines, while Figure 6.1 shows all three baselines lose to 50% of all possible 5-action BOL build-orders. Despite BOIL encoding more actions than BOL, the build-orders are more difficult for BOIL build-orders to overcome. Finding winning build-orders without automatic prerequisite detection poses a greater challenge for BOIL build-orders. These results show the trade-offs made when representing branches and loops in evolutionary methods. Our BOIL build-order representation benefits from multiple conditions even when one of the conditions is not advantageous individually. Evolutionary methods take advantage of the branches and loops to produce build-orders that win 20 seconds faster than build-orders without branches and loops. However, the added complexity of branches and loops take evolutionary methods longer to optimize.

6.1 Conclusions

In this chapter, we investigated a new representation that enabled our CA to produce stronger build-orders which defeated opponents quicker. While our previous BOL representation enabled the CA to find robust build-orders, BOL can only represent a single, limited length sequence of actions. In order to create stronger build-orders, we needed a new representation that can represent multiple sequences. To this end, we extended our BOL representation by including branches and loops with different conditionals, and called the extended representation Build-Order Iterative Lists (BOIL). We investigated four different BOIL configurations: BOIL with no encoded branches/loops (acts as our baseline), BOIL with one encoded condition (less than seven available units), BOIL with one encoded condition (greater than or equal to seven available units), and BOIL with both conditions. However, in order to evaluate four BOIL configures, we needed an RTS environment that could produce results quicker. As such, we developed BOSS specifically for simulating build-orders, and paired BOSS with SparCraft to evaluate the results of combat.

Our results show that evolutionary methods benefit from branches and loops with multiple conditions to choose from, allowing stronger build-orders that win faster. Additionally, evolutionary methods show a benefit from using multiple conditions, even when one of the conditions show no advantages individually. A BOIL representation that encodes two conditions enables the CA to produce a build-order which wins against opponents faster than build-orders using other BOIL configurations. However, it takes a CA more time to tune the branches and loops in order to produce a winning BOIL, compared to a BOIL with no branches and loops. These results inform our research into coevolving stronger

build-orders. Combined with our results from the previous chapters, our research has taken a small step towards finding strong, robust build-orders. In the next chapter, we discuss our conclusions and contributions towards advancing game AI research.

CHAPTER 7
CONCLUSION

Our research investigates a coevolutionary approach to producing robust, strong build-orders for Real-Time Strategy (RTS) games. Finding build-orders is a challenging scheduling problem that players must master in order to defeat opponents in an RTS game. However, in order to search for robust build-orders, we required an RTS environment that allowed us to test build-orders in a game. As such, we developed two RTS systems, WaterCraft and Build-Order Software Simulation (BOSS) to act as testbeds for our research. In addition to an RTS environment, we also required a build-order representation that our search methods could utilize, so we initially represented build-order as a sequence of actions we call Build-Order Lists (BOLs). In order to begin our research into generating robust build-orders, we compared BOLs produced by three different approaches: a genetic algorithm (GA), a coevolutionary algorithm (CA), and a bit-setting hill-climber (HC).

To compare the quality of build-orders found by our CA, GA, and HC, we required an absolute measure of quality, such as exhaustive search. However, exhaustive search is computationally expensive, so we limited ourselves to exhaustively searching the outcomes of 5-action build-orders against three hand-coded baselines. Our research showed that compared to exhaustive search against our three baseline opponents, a GA always finds the highest scoring build-orders that defeat the baselines, but not the most robust build-orders. On the other hand, a CA found build-orders that are robust and defeat many opponents, but the build-orders did not always defeat our specific baselines. These results inform us that a CA is a promising approach for generating robust build-orders, but has a limitation on defeating specific opponents. In order to influence a CA towards defeating

specific opponents, we needed a method that enabled the CA to learn from specific opponents. As such, we investigated case-injection into a CA.

Case-injection introduces specific build-orders into a CA, which enables a CA to incorporate knowledge from the injected cases. In order to enable our CA to learn from specific opponents, we investigated two case-injection approaches: case-injection into a CA's teachset and case-injection into a CA's population. Additionally, we compared the build-orders produced by a case-injected CA to a GA which competed against only the cases used for injection. Our results showed that contrary to previous results, a GA can be misled when a large difficulty gap exists between training opponents. Instead of learning to defeat both opponents, the GA overspecialized for the easier opponent, while ignoring the difficult opponent. On the other hand, build-orders generated by a CA remained robust and eventually found build-orders that defeated both opponents. We also showed that case-injection into the CA population produced build-orders similar to some of the injected cases, without negatively impacting build-order robustness. These results indicate that a CA is suitable for learning from specific opponents, while maintaining robust build-orders. However, while the BOLs generated by the CA are robust, BOLs only encode a single sequence of build-actions. In addition to being robust, we also want build-orders that are strong and defeat opponents quickly. In order to find stronger build-orders, we investigated a new representation which enables a build-order to contain multiple sequences of actions.

To this end, we extended our BOL representation by introducing branches and loops, creating a new representation we called Build-Order Iterative List (BOIL). We used a CA to produce BOILs with four different configurations: BOIL with no encoded branches/loops (acts as our baseline), BOIL with one encoded condition

(less than seven available units), BOIL with one encoded condition (greater than or equal to seven available units), and BOIL with both conditions. Our results showed a CA takes advantage of multiple loops and conditions in order and produced build-orders stronger than build-orders lacking branches and loops. However, generating strong build-orders with branches and loops required more time than generating build-orders without branches and loops. These results indicate that CAs are a promising approach for generating robust, strong build-orders, and our work has contributed towards advancing computational and artificial intelligence in RTS games.

7.1 Contributions

Our work has contributed to computational and artificial intelligence research in three ways. First, we applied a CA towards finding robust build-orders in an RTS game. While CAs have previously been used to address problems in RTS games, to the best of our knowledge CAs have not been applied specifically towards generating robust build-orders. Our results showed that a CA is a promising approach for generating robust build-orders, which may indicate a CA would also be suitable for solving similar real-world problems.

Second, we showed that case-injection enabled our CA to learn from specific opponents, while maintaining robustness. Injecting specific build-orders into a coevolutionary teachset enabled the CA to generate build-orders that defeated the specific cases we injected. Additionally, we also injected cases into coevolution's population, which enabled the CA to generate build-orders which played like the injected cases. These results indicate that a CA can learn from specific cases, while

continuing to produce robust build-orders. Being able to quickly learn from specific opponents is an important feature in an environment with no single dominating strategy. Defeating many opponents is meaningless if the build-order cannot defeat the opponent currently being faced. Learning quickly from opponents enables build-orders to adapt and stay relevant against new opponent innovations.

Third, our results showed that CAs benefit from a representation which includes branches and loops. Branches and loops enabled our representation to create build-orders with multiple sequences of build-actions. Our results indicated that branches and loops enables the CA to find stronger build-orders, but required additional time to optimize the branches and loops. These results showed the trade-offs that we must make with our build-order representation, whether we want to quickly produce robust build-orders, or slowly produce stronger build-orders. Showing that a CA can utilize branches and loops to improve build-order performance demonstrates that a CA is a promising approach to generating complex rules and behaviours for build-orders. While our research has taken a small step towards advancing game AI, additional extensions and problems remain for future investigations.

7.2 Extensions and Future Work

Our work would benefit from different directions of additional research. For the sake of a compact representation and exhaustive search, our work only investigated build-order representations which included four out of over forty units available in StarCraft. Including more of these StarCraft units would enable more diverse build-orders, and creates more intransitive relationships between build-

orders. It would be interesting to see how a CA copes with the additional complexity, and how many different niches of build-orders the CA population might contain. Increasing the complexity of the coevolved build-orders may also make case-injection even more important.

While selecting cases at random and our rate of injecting cases worked well in practice, a CA may benefit more from different injection parameters. Identifying which cases to inject in order to create stronger build-orders could help a CA learn faster and create more diverse build-orders. Additionally, determining the frequency at which to inject cases, and whether to inject into the CA's population or teachset could also help. Our case-injection work also investigated injecting cases that were modeled after a human player's behaviour. However, we only modeled a human's behaviour using the BOL representation.

Modeling human behaviour in a representation with branches and loops, such as BOIL, presents additional challenges. We may not be able to perfectly model all of a specific human's behaviour in BOIL, and there may be several different BOILs that closely approximate a specific human's behaviour. Determining how to model a specific human's behaviour in BOIL and selecting from multiple BOILs that approximate a specific human's behaviour could help a CA learn faster from stronger build-orders. However, human's make decisions by considering a wide variety of conditions. For example, rather than consider how many total units are built, a player may consider how many units of each type are built. In order to model this behaviour and create more diverse build-orders, we would need to encode many more conditions for branches and loops to consider. Additionally, in order to more accurately model human behavior's, the number of actions that a branch or loop contain may need to be larger, or vary on a case to case basis.

In addition to macromangement and build-order planning, human behaviour is also defined by micromanagement decisions, such as unit positioning and selecting an attack target. To this end, future work could investigate build-orders which enable multiple game AI's to win, and later expand to coevolving micromanagement behaviours along with the build-orders. Finally, our current and future work could be used to create a system to perpetually learn from humans and coevolve new build-orders. As humans compete against a game AI, we could automatically model the human's build-order and inject it into coevolution as necessary. The next time the human plays the game, the human will have to compete against a newly coevolved build-order that learned from the previous encounter with the human.

BIBLIOGRAPHY

[1] 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*, pages 783–790. ACM, 2010.

[2] Robert Axelrod. The evolution of strategies in the iterated prisoners dilemma. *The dynamics of norms*, pages 1–16, 1987.

[3] Christopher Ballinger and Sushil Louis. Comparing coevolution, genetic algorithms, and hill-climbers for finding real-time strategy game plans. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 47–48. ACM, 2013.

[4] Christopher Ballinger and Sushil Louis. Comparing heuristic search methods for finding effective real-time strategy game plans. In *Computational Intelligence for Security and Defense Applications (CISDA), 2013 IEEE Symposium on*, pages 16–22. IEEE, 2013.

[5] Christopher Ballinger and Sushil Louis. Finding robust strategies to defeat specific opponents using case-injected coevolution. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[6] Christopher Ballinger and Sushil Louis. Robustness of coevolved strategies in a real-time strategy game. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1379–1386. IEEE, 2013.

[7] Christopher Ballinger and Sushil Louis. Learning robust build-orders from previous opponents with coevolution. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[8] Christopher Ballinger, Sushil Louis, and Siming Liu. Coevolving robust build-order iterative lists for real-time strategy games (submitted for consideration). *Computational Intelligence and AI in Games, 2015 IEEE Transactions on*, 2015.

[9] Nils Aall Barricelli. Numerical testing of evolution theories. *Acta Biotheoretica*, 16(1-2):69–98, 1962.

[10] Nils Aall Barricelli. Numerical testing of evolution theories. *Acta Biotheoretica*, 16(3-4):99–126, 1963.

[11] Blizzard Entertainment. StarCraft, March 1998.

[12] Bruno Bouzy and Guillaume Chaslot. Monte-carlo go reinforcement learning experiments. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 187–194. IEEE, 2006.

[13] Michael Buro. The othello match of the year: Takeshi murakami vs logistello. *ICCA Journal*, 20(3):189–193, 1997.

[14] Michael Buro. Orts - a free software rts game engine, 2005.

[15] Michael Buro and David Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106, 2012.

[16] BWAPI Team. BWAPI: An api for interacting with StarCraft: Broodwar.

[17] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[18] Kumar Chellapilla and David B Fogel. Evolving neural networks to play checkers without relying on expert knowledge. *Neural Networks, IEEE Transactions on*, 10(6):1382–1391, 1999.

[19] Kumar Chellapilla and David B Fogel. Evolving an expert checkers playing program without using human expertise. *Evolutionary Computation, IEEE Transactions on*, 5(4):422–428, 2001.

[20] Ho-Chul Cho, Kyung-Joong Kim, and Sung-Bae Cho. Replay-based strategy prediction and build order adaptation for starcraft ai bots. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–7. IEEE, 2013.

[21] David Churchill. Sparcraft, 2013.

[22] David Churchill and Michael Buro. Build order optimization in starcraft. In *AIIDE*, 2011.

[23] Peter I Cowling, Munir Hussain Naveed, and M Alamgir Hossain. A coevolutionary model for the virus game. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 45–51. IEEE, 2006.

[24] James Edward Davis and Graham Kendall. An investigation, using coevolution, to evolve an awari player. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1408–1413. IEEE, 2002.

[25] Arpad E Elo. *The rating of chessplayers, past and present*, volume 3. Batsford London, 1978.

[26] Larry J Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundations of genetic algorithms*, pages 265–283, 1990.

[27] David B Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 2001.

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

[29] Johan Hagelback. Potential-field based navigation in starcraft. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 388–393. IEEE, 2012.

[30] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[31] W Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1):228–234, 1990.

[32] Stephen Hladky and Vadim Bulitko. An evaluation of models for predicting opponent positions in first-person shooter video games. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 39–46. IEEE, 2008.

[33] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *AIIDE*, pages 63–68, 2005.

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

[35] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):55–67, 2012.

[36] David Keaveney and Colm O'Riordan. Evolving robust strategies for an abstract real-time strategy game. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 371–378. IEEE, 2009.

[37] Alex Kovarsky and Michael Buro. A first look at build-order optimization in real-time strategy games. In *Proceedings of the GameOn Conference*, pages 18–22, 2006.

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

[39] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A Pelta, Martin V Butz, Thies D Lonneker, Luigi Cardamone, Diego Perez, Yago Sáez, et al. The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131–147, 2010.

[40] Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, and Luigi Cardamone. Learning to overtake in torcs using simple reinforcement learning. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

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

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

[43] Nicholas Gregory Mankiw. *Principles of Economics*. Harcourt College Publishers, second edition, 1998.

[44] Christopher Miles. *Co-Evolving Real Time Strategy Game Players*. PhD thesis, University of Nevada, 2007.

[45] Ann E Nicholson, Kevin B Korb, and Darren Boulton. Using bayesian decision networks to play texas holdem poker. *International Computer Games Association (ICGA) Journal (Unveröffentlichter Entwurf). Monash University, Victoria, Australien*, 2006.

[46] Geoff Nitschke. Co-evolution of cooperation in a pursuit evasion game. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 2, pages 2037–2042. IEEE, 2003.

[47] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research

and competition in starcraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4), 2013.

[48] Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David W Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75, 2006.

[49] Marc JV Ponsen, Stephen Lee-Urban, Héctor Muñoz-Avila, David W Aha, and Matthew Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. *Reasoning, Representation, and Learning in Computer Games*, page 78, 2005.

[50] Philipp Rohlfshagen and Simon M Lucas. Ms pac-man versus ghost team cec 2011 competition. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 70–77. IEEE, 2011.

[51] Christopher D Rosin and Richard K Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[52] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[53] Spyridon Samothrakis, David Robles, and Simon Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(2):142–154, 2011.

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

[55] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996.

[56] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

[57] Stargus Team. Stargus, 2009.

[58] Gabriel Synnaeve and Pierre Bessiere. Special tactics: A bayesian approach to tactical decision-making. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 409–416. IEEE, 2012.

[59] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[60] Torus Knot Software Ltd. Ogre - open source 3d graphics engine, February 2005.

[61] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *Advances in Machine Learning*, pages 367–381. Springer, 2009.

[62] Michael van Lent and John Laird. Developing an artificial intelligence engine. *Ann Arbor*, 1001:48109–2110, 1998.

[63] Wargus Team. Wargus, 2011.

[64] Ian Watson and Jonathan Rubin. Casper: A case-based poker-bot. In *AI 2008: Advances in Artificial Intelligence*, pages 594–600. Springer, 2008.

[65] Ben George Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2009)*, 2009.

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

[67] Stewart W Wilson. Ga-easy does not imply steepest-ascent optimizable. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 85–89. Morgan Kaufman, 1991.

[68] Geogios N Yannakakis. Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, pages 285–292. ACM, 2012.

[69] Sule Yildirim and Sindre Berg Stene. A survey on the need and use of ai in game agents. In *Proceedings of the 2008 Spring simulation multiconference*, pages 124–131. Society for Computer Simulation International, 2008.