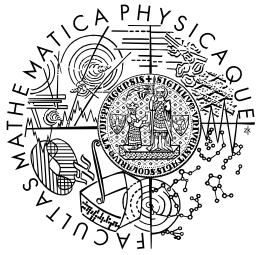

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Šťavík

Defending Choke Points in Star Craft: Brood War

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jakub Gemrot
Study programme:	Computer Science
Study branch:	Programming and Software systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Defending Choke Points in Star Craft: Brood War

Author: Petr Šťavík

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: Despite the effort in the field of artificial intelligence for real time strategy games, computer controlled agents (bots) still struggle even against average human players. One of the keys to success in such games is the ability to take advantage of various tactical points on the map, like chokepoints — narrow passages connecting open areas. With the use of genetic algorithms and SparCraft, a simplified simulator of StarCraft: Brood War, we present a method to generate advantageous unit layouts for defending chokepoints. Our experiments show that layouts produced using our method perform significantly better than random layouts, and are comparable in quality with layouts traditionally employed by human players. Our method may also be used to generate a database of advantageous unit layouts, which could be incorporated into an existing StarCraft: Brood War bot.

Keywords: artificial player, genetic algorithms, Star Craft: Brood War, defending choke points

Above all, thank you to my parents, family and several friends for the ceaseless support.

Thank you to Jakub Gemrot for his expertise and creative guidance.

Another thanks goes to Pavel Šmejkal for his valuable notes and advice.

Lastly, access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Contents

Introduction

In recent years, there have been several breakthroughs in artificial intelligence (AI) with computer controlled agents (bots) beating human world champions in games like Go (Silver et al., 2016) or no-limit Texas Hold'em Poker (Moravčík et al., 2017), which were few years ago thought of as unlikely for any bot to master. Despite this progress, there are still game genres that are far from being solved. Real time strategies (RTS) are one of them.

An RTS is a two-player video game with a simple win condition — destroy all enemy units and buildings. This is achieved by gathering resources for building own units and buildings, and advancing your own technology level. Some well-known games of this genre include Warcraft III[1], Age of Empires II[2], Dune II[3] and Starcraft: Brood War[4] (SC:BW).

The difficulty in developing a highly skilled bot for these RTS games (the RTS AI problem) results from several factors:

- Most RTS games are only partially observable and the player may only see parts of a map that contain his units — this effect is often referred to as *fog of war*.
- As the genre implies, such games are "real-time" and the game state changes (even though only slightly) around 24 times per second, which gives the player about 42ms of time to act before the next state change.
- Complexity of these games in terms of state space and in terms of number of actions available at each game cycle is enormous, orders of magnitudes higher than in the previously mentioned game Go. In a typical RTS game there are about 200 units on a 128x128 map. Considering only the unit positions we have about $16384^{200} \approx 10^{800}$ different states. In contrast it is estimated that the size of state space in Go is about 10^{170} (Ontanón et al., 2013).

Because of these difficulties, researchers try to divide the RTS AI problem into small sub-problems that are solvable by currently known techniques rather than solving the whole problem at once. Such sub-problems usually correspond to a single skill required for mastering an RTS game. Even though a few of these subproblems like micromanagement (Churchill and Buro, 2013), building placement (Barriga et al., 2014) or opponent matching Tong et al. (2011) were quite successfully tackled, bots that incorporate these results are still far from being competitive opponents for human players.

Using various geographical features on the map to gain an upper hand is yet another vital skill for mastering any RTS game. One such geographical feature present in nearly all RTS maps are chokepoints.

A chokepoint is a narrow passage on the map, like a defile or a bridge which, can greatly decrease the combat power of passing force. Such chokepoints can

[1] `https://us.blizzard.com/games/war3`
[2] `https://www.ageofempires.com/games/aoeii/`
[3] `http://duneii.com/`
[4] `https://us.blizzard.com/games/sc/`

Figure 1: Battle around chokepoint from the game Starcraft II by Blizzard Entertainment.

therefore allow a numerically inferior defending force to thwart otherwise certain defeat. A player may therefore decide to defend several important chokepoints on the map and focus more on gathering resources or upgrading his units — factors which will pay off in a longer game. But for these defenses to be successful, a player must properly arrange his units around the chokepoint before the clash.

In order to tackle this problem we need either an approximate simulator of some RTS game or a programming interface which directly communicates with the game itself. In the case of SC:BW, both options are available in the form of BWAPI (programming interface)[5] and SparCraft (simulator)[6]. Since the release of BWAPI, SC:BW became a standard test-bed for AI research in RTS games. Moreover, AI developers are further motivated by a few yearly tournaments like SSCAIT[7] or AIIDE StarCraft AI Competition[8]. For these reasons, we chose SC:BW as our platform too.

Our goal in this thesis is to propose and implement a generic method for generating initial unit layouts for defending chokepoints in SC:BW. To achieve this, we will leverage the power of genetic algorithms. In order to show the usefulness of our method, we compare the produced layouts against both random layouts and layouts traditionally used by human players.

The rest of this thesis is organized as follows: First, we point out several important aspects of RTS games, stressing the importance of chokepoints. Then we take a look at work related to our topic. Afterwards, we introduce our method for approaching this problem. A chapter explaining the implementation of our

[5] `https://github.com/bwapi/bwapi`

[6] `https://code.google.com/archive/p/sparcraft/`

[7] `http://sscaitournament.com/`

[8] `http://www.cs.mun.ca/~dchurchill/starcraftaicomp/`

approach follows. Next, we assess the performance of our method by performing a set of experiments. Finally, we conclude with a discussion on future work and possible practical usage of our result.

1. RTS background and chokepoints

In this chapter, we start by providing a general background to the genre of RTS required for fully understanding this thesis. Then, we will introduce a very common feature in RTS games — chokepoints.

1.1 RTS background

RTS video game sub-genre was born with the release of Dune II[1] in 1990. Since its origin, the genre slightly evolved and new sub-genres of RTS emerged, but core concepts still remain the same. As SC:BW belongs to the class of classical RTS games, which stay true to Dune II in almost every aspect, we will continue by describing this class of an RTS games only.

Similarly to Chess or Go, RTS games are played as independent *matches*. Matches, might be of various formats, but for the competitive scene, one versus one is the most common one.

To win a match, one must eliminate all the units and buildings of the opponent. In order to do this, the player must collect resources which he may spend to construct new buildings, train units (either military or civil), and research permanent bonuses (*upgrades*).

Usually there are about 2–4 types of resources like wood, ore or gold that are gathered with basic civil units (*workers*). The rate of gathering resources is often referred to as *economy*. Resources differ in their overall availability and usability. A common resource like wood might be used to produce only basic units and buildings, while a rarer resource like gold may be used for training strong military units and researching advanced upgrades. In SC:BW we have two resources — *minerals* and *gas*, with gas being the rarer one.

The types of units and buildings available to the player during the match depend on the *race* (or *faction*) he chooses to play as (in SC:BW, three races are available: Protoss, Zerg and Terran). Each race is usually capable of producing about 8–15 different types of units. Units are trained inside buildings and each unit has several attributes like *hit points* (how much damage it can take), movement speed or armor. Typical military units are also able to inflict damage and they may be classified according to the range from which they can deal it. We may have *melee units* like knights or swordsmen which can deal damage only from close range or *range units* like archers that may deal damage from distance. Apart from a basic attack, units may also have special abilities. Effects of these abilities may vary, but common examples include a temporary boost to a unit's attributes like higher movement speed or additional damage to enemies with spells like fireball. Using an ability require spending additional resource units may have — *energy points*, which regenerate slowly over time. However, there are exceptions to this rule. There are abilities which require spending hit points and abilities that do not require any resource at all — using these repeatedly is then always

[1] `http://duneii.com/`

limited by a short period of time, so called *cooldown*.

Additional classification determines the ability to move around a map. We may have naval, ground or aerial units. Not all units from one class of this classification may attack units from the other one. For example, knight may attack only ground units, but archer may attack ground, naval or aerial ones.

Buildings are of various types in RTS games. There might be solely defensive structures like turrets which attack enemy units that reach them, structures for researching upgrades like armory or structures for training units like barracks. Player does not have all structures available at the beginning of the match but he must gradually build up to them. For example, training a strong, advanced unit like a knight may require building a castle. But to build a castle, one must build stables, and so on. This path of required buildings and therefore available units (and also upgrades) is referred to as *tech tree*. This structuring makes the game more ordered and predictable — when a player sees that his opponent is working on stables, he may prepare some aerial units that will be strong against knights.

1.1.1 RTS match

At the start of a match, each player is assigned with a few workers and basic buildings located at one out of several possible starting locations on a map (*spawn points*). The designated spawn point coupled with a player's buildings placed around it is referred to as the player's *main base*. Example of a main base from the advanced stages of a match may be seen in Figure 1.1. Each spawn point is close to some basic resources in order to enable first productions. However, these resources are limited in quantity and the player will, as the match progresses need to build new bases (*expansions*) close to some additional resources.

Figure 1.1: Example of a main base from the popular RTS game Age of Empires II from Ensemble Studios. On the picture, we can see a few workers gathering resources (food from the fields) and several buildings.

RTS match is commonly divided into three parts — *early game*, *mid game* and *late game*. Transitions between these stages are not marked by a fixed time point but rather by the technological progress of both players.

Each match starts in early game and each player starts with a certain *opening*. This opening refers to first set of structures and units the player chooses to produce. Openings determine the initial strategy of a player. Economy focused openings favor training additional workers at the expense of building structures for training military units. In the case of aggressive openings, the exact opposite happens. There are also openings that are between these two with player investing somewhat equally into economy as well as into military. In this stage, battles are rather rare but if they occur, they tend to decide the whole match since players do not have the economy to produce new units fast enough. Saving or killing only a single additional unit in these battles may therefore be the difference between loss and win. This may be typically achieved by careful control of individual units in battle, like moving wounded units out of range of enemy units. This ability to control individual units in battle is referred to as *micromanagement*.

As the match progresses, it transitions into *mid game*. In this stage players tend to be much more daring with a few expansions and considerable army at hand. At this point, it is very common for players to fully develop a single branch of tech tree and therefore having several advanced units as well. In mid game, larger battles start to occur and players try to exploit their opponents' weaknesses by for example attacking unguarded expansion.

The last stage is called *late game*. In this stage the differences in initial openings completely fade out. Players have several branches of tech trees fully developed, control massive armies and a few expansions that support strong economy. At this point, the map is a constant battlefield with various battles happening at the same time.

In any of these stages, various strategies or techniques are used to get ahead. For our work, *rushing* and *worker harass* are the most important ones.

Rushing is a type of extremely aggressive opening which tries to end the match in early game. The player starts with producing cheap basic military units as early as he can, omitting nearly any investments into economy. Once he has few of them ready, he starts sending them in waves at the opponent and hopes that his enemy failed to prepare a proper defense. However, if the opponent is able to repel these attacks, the heavy investment from the Attacker into this rush typically results in weaker economy and slower advancement in technology between early game and mid game.

Worker harass is a technique that tries to weak down the opponent's economy by killing his workers. Unlike rushing, it is frequently used throughout the whole game. With this technique, the player prepares a few basic units, favoring the ones with high mobility. With the units ready, he sends them towards the opponent's gathering locations with intention of killing as many workers as possible. If uncontested, these few units have the potential to cause significant damage, as workers are very weak.

1.2 Chokepoints

To better mimic real-life combat scenarios, maps in RTS games contain variety of impassable natural obstacles like mountains, forests or water areas, which may shape the map's terrain in interesting ways. One effect of this may be the creation of so called chokepoints (or choke points). A chokepoint is a real-life military term which is defined in Hanks (2011, Page 58) as follows:

> Choke point is a term used in military geography to denote a narrow passage, either on land or water, through which a military force is forced to pass ...

This description matches naturally created chokepoints like valleys, straits or defiles as well as artificially created ones like bridges or gates. As there are no naval units in SC:BW, we will only focus on land chokepoints from now on.

Chokepoints are a very common feature in RTS maps and a typical map contains several of them. This is achieved by dividing the map into polygons that are connected with each other only through narrow passages — chokepoints. Smaller polygons usually correspond to spawn points or possible expansions, while larger polygons offer more open ground for potential maneuvering during larger battles. An example of a typical map from the game Starcraft II may be seen in Figure 1.2.

This map layout also explains the need for passing these chokepoints. As each base is located at one distinct polygon, the need for passing the chokepoints is driven by the sole objectives of RTS games — claiming new expansions and destroying the opponent. If a player wants to utilize the main mean a typical faction provides — ground forces, for attacking his opponent or securing new expansion, he must almost inevitably move them through a few chokepoints. Typical RTS games offer also some other means for transporting ground units like air or naval vessels, however these units may not always be the option as these units are available only later with considerable progress in at least one branch of the tech tree. These observations will be important in the next subsection, where we explain the importance of chokepoints.

1.2.1 Importance of chokepoints

A player with good tactical thinking will immediately recognize chokepoints as opportunities for favorable conditions during combat. The tactical value of chokepoints may be summarized by the following description (continuing the definition of chokepoints from Hanks (2011, Page 58)):

> ... Choke points carry great military strategic importance, because the geography of such locations may be utilized by a smaller force to offset a disadvantage in size when facing a larger army.

To further explain the strategic importance, let us consider the following two combat situations.

In the first combat situation, we have a red player that is facing a battle on an open ground with his 3 rather strong knights against 10 weaker swordsmen. In one versus one combat, a knight is much stronger than a swordsman. However

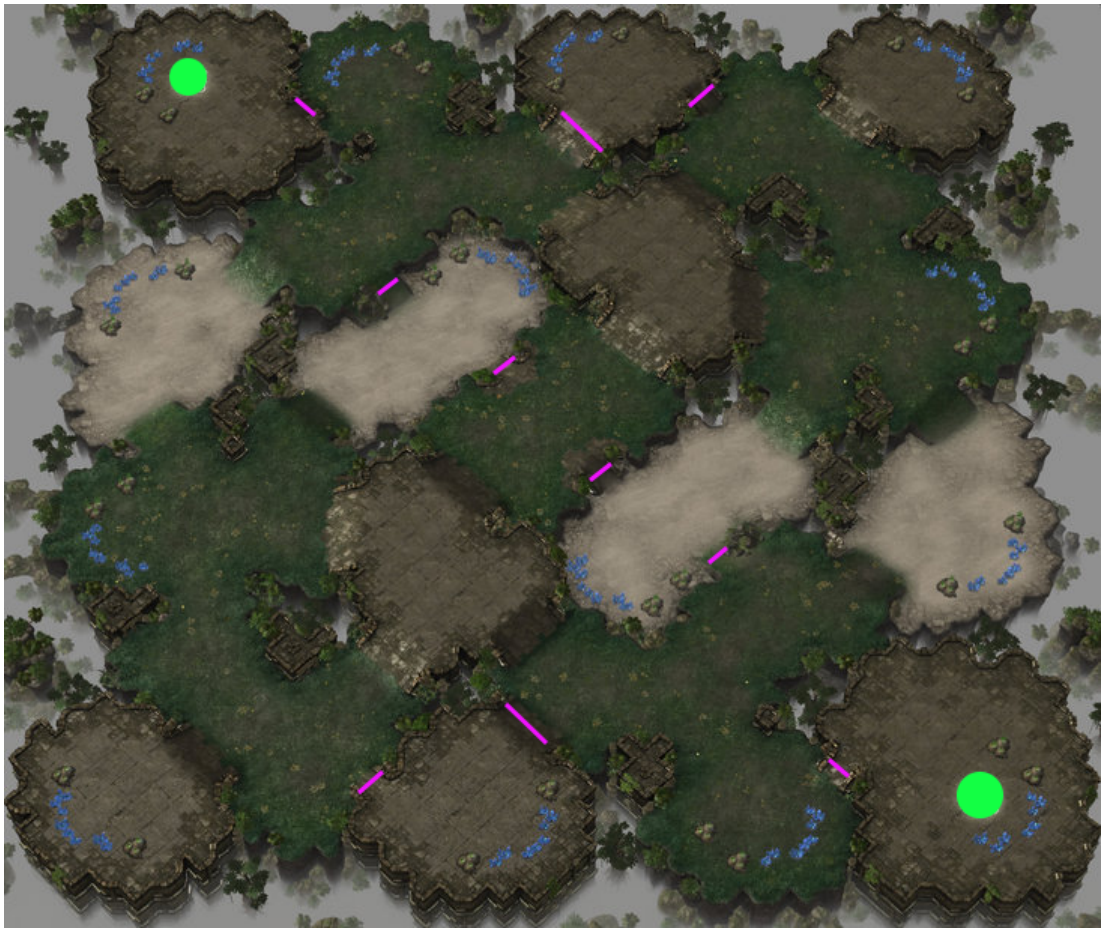

Figure 1.2: An example of a map from the game Starcraft II by Blizzard Entertainment. The green circles mark spawn points, magenta lines mark some of the chokepoints.

in this case, the numeric superiority of swordsmen will get in role. They will surround the knights and defeat them. This situation is depicted in Figure 1.3.

In the second combat situation, the players control the same forces — a red player with 3 knights and a blue player with 10 swordsmen. However, in this case, the blue player tries to utilize a chokepoint. He enclosed one entrance to the chokepoint with his force and now awaits the opponent's forces. The narrowness of the chokepoint allows the swordsmen to enter only in the rows of two, while the stationary knights create a concave formation (or simply *concave*) which allows all 3 of them to attack at the same time. In this situation, we can see that the red player finds himself at a considerable disadvantage — only two of his swordsmen may attack the knights at a time and the other 8 of them are blocked by their fighting allies. In this case, the knights have no problem with defeating the swordsmen. This situation is depicted in Figure 1.4.

This property makes chokepoints especially useful for more economy focused players in early game, where the differences in numbers are the most prominent and they provide one of the few reliable ways of defending rushes. If a player expects rush, he may focus on defense of chokepoints leading to his main base and by blocking them in time (either with units or buildings), he may prevent rushes from succeeding.

Figure 1.3: A combat situation on an open ground with 3 knights controlled by the red player and 10 swordsmen controlled by the blue player. This image was taken from game Warcraft III by Blizzard Entertainment.

Nevertheless, their correct utilization is helpful throughout the rest of the match as well. In mid game or late game their usability shifts a bit where they provide overall more favorable position during larger battles. Players are very well aware of that and each player tries to outmaneuver his opponent and lure him into the disadvantageous conditions for combat. In addition, chokepoints may also be used for easy defense against ground worker harass. Player will again block chokepoints leading to his bases, giving the opponent's attempts for worker harass little chance of succeeding.

Figure 1.4: A combat situation with chokepoint utilization employed by the red player with his 3 knights against the blue player's 10 swordsmen. This image was taken from game Warcraft III by Blizzard Entertainment.

2. Related work

As presented in the example under Section 1.2.1, the outcomes of battles around chokepoints greatly depend on the positioning of units. The importance of units' formations, however, applies to battles anywhere on the map.

One possible approach to designing formations of units is based on a technique called "flocking", with which the units move together as a flock of birds forming a formation. This approach was addressed in Danielsiek et al. (2008), where the authors combined the flocking behavior with pathfinding based on influence maps. This approach was then tested against movement without flocking and showed improvement in every conducted experiment. The formations produced with general flocking behavior are, however, not very suitable for advantageous formations around chokepoints. The flocking algorithm requires, that each individual participating in the flock reacts only to its surrounding neighborhood and in the case of chokepoint defense, we could use more flexibility.

A different approach that does not address unit formations before the battle but rather produces emergent ones based on a current situation was presented in Young et al. (2012, Section Micro Management) with their SCAIL bot. In this work, each unit had two parameters, namely "confidence" and "caution", with values changing depending on the number of nearby allies or enemies. For example, a unit will become much more confident if there are no enemy units that may attack it. These parameters allowed units to occasionally produce interesting concave formations, traditionally used by human players. As with the flocking, this approach is not that suitable for defending chokepoints, as we would rather produce the favorable formation before combat.

In the case of base assaults, the outcome of a battle may not only depend on the positioning of units but also on the positioning of structures. This problem has been tackled in Barriga et al. (2014). Authors gathered 57 lost base assaults from human as well as bot replays and with the use of genetic algorithms, evolved the placement of buildings in the bases. Depending on the parameters of the genetic algorithm, they were able to turn between one and two thirds of the lost assaults into wins.

As genetic algorithms are the method of our choosing as well, it is worth mentioning their other uses in RTS games. Apart from building placement optimization, they've been used in various ways. Liu et al. (2014) presented ECSLbot with evolved unit behavior based on artificial potential fields, influence maps and reactive control parameters. Different approach with evolution of parameters describing behavior of a whole bot was presented in Fernández-Ares et al. (2011). Other typical usages include procedural content generation of maps like in Togelius et al. (2010), opponent matching as in Tong et al. (2011) or difficulty adjustment as in Olesen et al. (2008).

2.1 RTS and chokepoints

Despite their importance presented in section 1.2.1, tactical map locations like chokepoints do not receive that much attention.

One of the first works that enabled bots in RTS games to better reason about

map terrain features was presented in Perkins (2010) who provided an algorithm for region decomposition and chokepoint detection. The algorithm first indentifies obstacle polygons by a simple flood-fill of the map. The edges of these polygons now serve as seeds for computation of a Voronoi diagram. This diagram is then pruned, and the remaining vertices may be used to identify regions as well as chokepoints. The presented algorithm was then implemented and evaluated against standard set of maps from SC:BW. Its accuracy was then compared with chokepoints identified by human participants. Although the algorithm sometimes produces false positive and false negative results, it generally works very well. This algorithm is implemented in BWTA (Brood War Terrain Analyzer) library[1] and, as it is written against BWAPI, it may be used within any SC:BW bot. In our work, this library will serve as a tool for initial chokepoint identification.

Apart from detection, one problem concerning defense of chokepoints has been already addressed — blocking them with buildings (also referred to as *wall-in*).

In Čertický (2013), a declarative way to deal with this problem was presented. The wall-in building placement problem was formulated as a Constraint Satisfaction Problem (CSP) with individual instances written as ASP (Answer Set Programming) logic programs. Instances may then be solved by any available ASP solver. However, solving the encoding presented in this work may take up to 200ms, which limits online usability of this approach during a match because of the computational constraints for bots introduced by some tournaments (rules for AIIDE SC:BW competition[2] disallow the bots to exceed frame time of 55ms more than 200 times).

This issue was addressed in Richoux et al. (2014) where the same problem was tackled but with a slightly different approach. The problem was again formulated as CSP but this time, with a different set of constraints and different method of solving. Apart from simply satisfying all the constraints, their solver allows optimization over 3 different criteria — number of buildings, number of gaps between the buildings or required technology level of buildings. Their experiments were conducted under the constraint of 20ms in order to allow online usability. By giving their solver 5 attempts (which may be performed in different frames) to find a valid wall-in, it was able to find in one more than 90 percent times. Moreover, their optimization runs over global time out of 150ms were able to greatly decrease the number of imperfect wall-ins, number of needed buildings or required technological level for used buildings.

Both of the previous approaches, however, did not address the issue of unit placement around chokepoints in any way. To the best of our knowledge nor did any other work.

[1] `https://code.google.com/archive/p/bwta/`

[2] `http://www.cs.mun.ca/~dchurchill/starcraftaicomp/`

3. Problem analysis

With properties presented in Section 1.2.1, it is apparent that bots may greatly benefit from correct utilization of chokepoints. However, current state-of-the-art bots usually do not treat them as the valuable tactic terrain features they are. In this chapter, we will present our approach for facilitating their utilization.

3.1 Choosing the problem

When we think about chokepoint utilization there are three areas that stand out the most: building wall-in, micromanagement of units during battle and placement of units before battle.

Although the problem of walling chokepoints with building is interesting, it has already been addressed in Čertický (2013) or in Richoux et al. (2014). Therefore, we would rather choose one of the other two problems. Even though the general case of micromanaging units anywhere on the map has been already addressed in several previous works like in Churchill and Buro (2013) or Liu et al. (2014), we believe that the special case of unit control in chokepoints would be worth tackling on its own. During a chokepoint defense, the player wants to, for example, move his melee units to the front while having range units in the back or he wants to prioritize highest threats — as he would want anywhere on the map, but there are some exceptions to unit control. For example moving the wounded unit that blocks the chokepoint out of the battle may not necessarily be the correct play.

The problem of initial unit placement around chokepoints is interesting as well. As presented in Section1.2.1, the outcome of a battle around a chokepoint may sometimes depend much more on the initial positioning of units rather than anything else. From these two options of unit micromanagement and initial unit placement, we chose the latter one.

In this work, we will therefore tackle the problem of initial positioning of units around chokepoints before battle. Let's consider an AI player P_d that tries to utilize a chokepoint CP with his units U_d. He wants to prepare a good formation, in which he will defend the chokepoint. We will call this player the *defender*. Now, let us assume a player P_a that tries to assault this chokepoint with his force U_a. We will call this player the *attacker*. Given the instance $I = (CP, U_d, P_d, U_a, P_a)$, our goal is to find a layout L^*, that means position p for each $u \in U_d$, such that the outcome of the battle for player P_d will be as favorable as possible, according to some objective function $f : \mathcal{F} \mapsto \mathbb{R}$, where $\mathcal{F}$ is a set of feasible solutions for our instance I.

It is worth pointing out, that we could certainly optimize the layout for U_d against multiple combinations of forces $U = \{U_{a_1}, U_{a_2}, \ldots, U_{a_m}\}$, and this approach may seem best at first glance. We would obtain a result that was optimized against the whole set U and the resulting layout L^* should provide quite a robust result. However, by optimizing against only one unit composition U_{a_1}, we will obtain a much more specific result and although the final layout will probably not be good against multiple enemy compositions, it should perform really well specifically against U_{a_1}. We can then optimize the layout U_d against each of the

composition $U_{a_2}, \ldots, U_{a_m}$ and this way achieve much more specific, and therefore better results for each pair (U_d, U_{a_i}), $i = 1, \ldots, m$ of the unit combinations.

3.2 Choosing the method

With the problem formulated, we may now consider possible approaches for solving it.

The first option is to choose a case based approach, where we would pick a feasible solution based on some set of predefined static rules. This approach would certainly, to some extent, work — we could use a set of rules that would tell melee units to, for example, block the chokepoint and another set of rules that would emplace range units behind them. However, this approach is highly inflexible, requires a lot of domain expert knowledge and is exhaustive to write. Therefore, it would be better to resort to other techniques.

Another set of options to consider are search methods, using which we could search over the set of feasible solutions. The initial state would be some random unit positioning around a chokepoint and its successors would be states with units moved in all directions (we would also of course need a black box that would tell the algorithm that it reached its goal state). Given that units may typically move in 8–16 directions, uninformed brute force search techniques are not even worth considering. As a next try, we may consider some informed searches. As we do not care about the path to some goal state, algorithms like A* do not make that much sense. Instead, we may try to use local search techniques. Variants of simple hill climbing would certainly work, but they tend to get stuck in local optima. Instead, we may try to use Genetic Algorithms (GAs), which were successfully used in similar problem of building placement in Barriga et al. (2014).

One potential disadvantage with choosing genetic algorithms is that they take considerable amount of time. This factor rules out the possibility for solving the problem on-line during a match with this method and we must resort to off-line solving. However, this does not limit usability of genetic algorithms in this manner as we could generate a database of favorable unit placements and then use it in an actual match on-line.

3.3 Background on Genetic algorithms

Genetic algorithms were invented in 1960s by John Holland (Holland, 1992) as one of many evolutionary computation methods whose idea is to "evolve" a set of initial candidate solutions toward better ones by mimicking the processes of biological evolution — candidate solutions are subjected to the process of natural selection and natural genetic variations like mutation.

The simplest form of a Genetic algorithm (GA) works like this: The algorithm starts with an initial population of N *chromosomes*. Chromosomes are made up of *genes* and each chromosome encodes one feasible solution to our problem. Now, a *fitness function* is used to assign viability value (*fitness*) to each candidate solution. Then, a new population of N chromosomes is created by repeatedly performing the following steps. First, *selection* selects two candidate solutions — *parents*, which will serve as basis for the creation of two *offspring*. Now, the

Algorithm Genetic algorithm

Input: an instance I of optimization problem
Output: a feasible solution

```
1: function GENETIC-ALGORITHM(I)
2:     P ← empty population
3:     INITIALIZE(P)
4:     loop
5:         for all chromsomes C in P do
6:             ASSIGNFITNESS(C)
7:         if termination condition is met then
8:             return best chromsome from P
9:         P_N ← empty population
10:        while |P_N| ≠ |P| do
11:            C_1, C_2 ← SELECTWO(P)
12:            if crossover probability then
13:                CROSSOVER(C_1, C_2)
14:            for all genes g_1 in C_1, genes g_2 in C_2 do
15:                if mutation probability then
16:                    MUTATE(g_1)
17:                if mutation probability then
18:                    MUTATE(g_2)
19:        P ← P_N
```

Figure 3.1: Pseudocode for the basic variant of Genetic algorithm.

crossover operation has a chance to exchange some genes between the offspring and *mutation* has a chance to change some genes. The two resulting offspring are inserted into the new population. This process is repeated until the new population has N. The algorithm typically terminates after a given number of iterations or after reaching a plateau. Pseudocode for this basic variant of a GA may be seen in Algorithm 3.1.

All the presented parameters and methods are highly problem specific and picking the right mutation rate or selection type usually requires several trials. However, this paragraph describes some common values for the above parameters as well as some typical methods for performing the aforementioned operations:

- There are many ways how to encode feasible solutions. We may for example use binary or string encoding. The chosen method then affects behavior of crossover and mutation.
- The selection operator selects chromosomes for mating based on their fitness. Several approaches for choosing the parents exist:
  - Tournament selection — in order to select a parent, two random chromosomes are selected from the population (uniformly). The one with lower fitness is discarded. The same goes for selection of second parent.
  - Roulette wheel selection — a parent is selected with probability proportional to its fitness. Higher fitness means a higher chance of selec-

tion. Same goes for the second parent.

- The probability of crossover typically ranges from 0.7-0.95. Again, there are several variants of crossover:
  - One point crossover — given two chromosomes, we select a position in their encoding. The parts of the encoding past this position are exchanged between the two chromosomes.
  - Two point crossover — instead of choosing a single position like in one point crossover, we select two. Chromosomes exchange parts of their encoding between these two positions.
- Chance of mutation is typically very low, about 0.01. The meaning of mutation depends on the chosen encoding. Given a binary encoding of some chromosome, the mutation may, for example, flip some bits of this chromosome.

There are also many other advanced techniques that are frequently applied to genetic algorithms. Here are the ones that are relevant to our work:

- Elitism — this technique ensures that the GA does not throw away the best solutions found. Instead of creating an entirely new generation in each iteration, we will copy several chromosomes with the highest fitness into the new generation. Typical rate ranges from 0.05 to 0.15.
- Plague — this technique tries to prevent the GA from getting stuck in a local optimum by adding some entirely new candidate solutions to the generation. This may be done for example every tenth or twentieth iteration and the rate may range from 0.25 to 0.75.

3.4 Defending chokepoints with GAs

With the method chosen, we may now proceed by applying GAs on the presented problem.

Let us consider an instance $I = (CP, U_d, P_d, U_a, P_a)$, where $U_d = (u_1, ..., u_n)$. Each gene will represent one position of a unit from U_d. Each chromosome therefore represents one layout for U_d. The order of genes inside chromosomes will be maintained throughout the whole run of the algorithm. This means that first gene of any chromosome will always correspond to the position of unit u_d and so on. This way, we can relate each gene to a specific unit. The mutation operator will move the associated unit in some direction and the crossover operator will exchange some positions among chromosomes.

So far, we did not address the issue of choosing the objective function, and, therefore, the fitness function for evaluation of individual chromosomes. The problem is, that there is no easy way how to determine layout quality simply by looking at the positions. In order to evaluate the layout, we will need to simulate the actual assault of U_a controlled by P_a on the chokepoint that P_d defends with U_d. Given the result of the simulated battle, we can now use a scoring function that would score it and this score would then be used as the fitness of our chromosome.

Let us consider a chromosome C with a layout L_d and let us consider a tuple $X = (U_d, P_d, L_d, U_a, P_a, L_a)$ — an instance of a battle and the outcome of this simulated battle — $\text{sim}(X)$. Now, let us consider unit sets U'_d and U'_a from the $\text{sim}(X)$ that represent the original units U_d for defender and U_a for attacker with their HP's altered after the simulation (some units died, others were injured). In order to score the $\text{sim}(X)$, we can then for example use the following function, already used in Barriga et al. (2014), where it worked well:

$$\text{resourceCost}(\text{sim}(X)) = \sum_{u \in U'_d} \frac{\text{hpleft}(u)}{\text{maxHp}(u)} (\text{mineralPrice}(u) + 1.5\text{gasPrice}(u)) \tag{3.1}$$

$$- \sum_{u \in U'_a} \frac{\text{hpleft}(u)}{\text{maxHp}(u)} (\text{mineralPrice}(u) + 1.5\text{gasPrice}(u)), \tag{3.2}$$

where the reason for the multiplication of the unit's gas price is that gas is a rarer resource in SC:BW. Now, the fitness of our chromosome C is the value of the resourceCost formula

$$\text{fitness}(C) = \text{resourceCost}(\text{sim}(X)).$$

There remains one more issue not addressed yet. What should the layout of attacker's unit set U_a be? For the defender's units U_d, we will evolve the layout and each chromosome may, therefore, start with a completely random one. We may for example try to choose a random layout for U_a as well, which will be fixed throughout the whole evolution. However, as we have already argued, the outcome of any battle may greatly depend on the layouts of both unit sets. Choosing a random layout for U_a could therefore possibly be an issue. However, depending on the results, we could try a few other approaches that could possibly solve it:

- RoundRobin GA — instead of having a single random layout for U_d fixed throughout the whole run, we would have m of them $L = \{L_1, L_2, \ldots, L_m\}$. Then, during the evaluation phase of chromosome C with layout L_d we would simply evaluate our U_d with L_d against U_a with each L_i, where $i = 1, \ldots, m$. The resulting fitness of chromosome C could then be the sum of values of these simulations divided by the number of performed simulations.

- CoEvolution — this approach is based on the evolution of a layout for U_a throughout the run of the algorithm. We will start with a generation of random layouts, G_d, for U_d and a generation of random layouts, G_a, for U_a. First, we will pick a random chromosome from G_a, say L_a. We will then optimize the layout for U_d against U_a with L_a for several iterations. Now, we will pick the best chromosome from G_d, say L_d and evolve the layout for U_a against U_d with L_d for several iterations. After these iterations, we will not pick a random chromosome from G_a as in the first step, but as we've evolved the G_a for some time, we can pick the best one. This process now repeats until the termination. The limit on the number of iterations (or other parameters) may vary, as our main goal is to find a good layout for U_d, we may give the defender, for example, 20 iterations and then give the attacker only 5.

Our system is depicted in Figure 3.2.

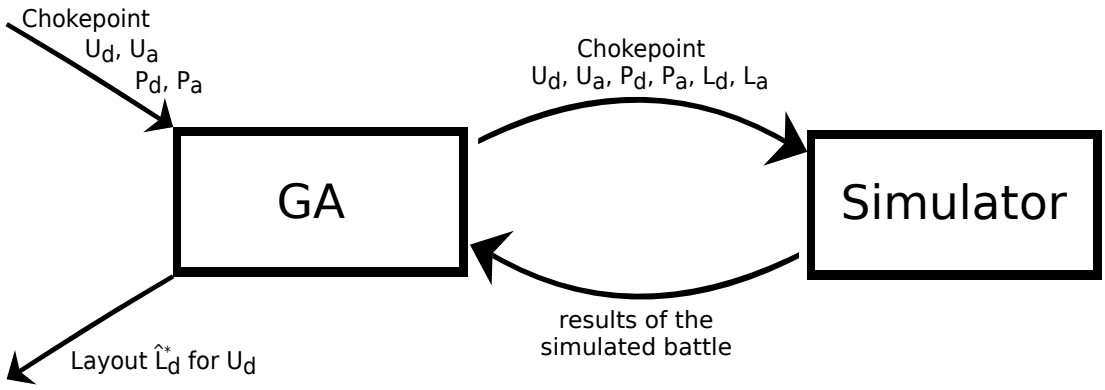

Figure 3.2: Diagram showing the basic pipeline of our system.

4. Implementation

With the problem formulated and method chosen, we may now proceed to the implementation part of our work. In this chapter, we will first choose one of the available RTS domains and settle on the method of combat simulation. Then, we will describe two core modules of our system — SparCraft and FormationsEvolver.

4.1 Choosing RTS domain

As of now, the standard domain for RTS AI research is SC:BW[1]. Its popularity rose with the release of programming interface BWAPI[2] which enabled any enthusiast or researcher to write their own bot. It is very well documented and several guides enable anyone to jump right in and code. Furthermore, there are a few other useful libraries available. We have a simplified simulator of the game itself, SparCraft[3] or a terrain analyzer, BWTA[4]. Developers are further motivated by a few tournaments. There is the annual AIIDE SC:BW AI competition[5] or SSCAI tournament[6], both with a few dozens of submissions every year. Moreover, despite being released in 1998, the game still has a relatively large fan base. As such, there exist many strategy tutorials and guides that may help anyone understand common strategies or tactics. In addition, SC:BW is one of the more demanding RTS games. The game is very complex overall and its mastering requires a good grasp on various skills like high level decision making or opponent prediction — skills mostly lacking in current bots. All of these factors combined make SC:BW a very interesting and attractive challenge for RTS AI research.

Prior to the release of BWAPI, there were also two other frequently used domains — ORTS[7] and wargus[8] which we briefly considered. Wargus is a clone of the game Warcraft II[9]. ORTS is an open engine for RTS games. However, the lack of community, tutorials and overall documentation has discouraged us from trying them and with SC:BW at hand, there was no reason to.

4.1.1 Units in SC:BW

With the domain chosen, we may now present units from SC:BW that will be important to our work. These units are divided by their races (in SC:BW, there are three races — human-like Terran, insectoid Zerg and psionic Protoss):

Protoss

Zealot very robust basic melee unit

[1] `https://us.blizzard.com/games/sc/`
[2] `https://github.com/bwapi/bwapi`
[3] `https://code.google.com/archive/p/sparcraft/`
[4] `https://code.google.com/archive/p/bwta/`
[5] `http://www.cs.mun.ca/~dchurchill/starcraftaicomp/`
[6] `http://sscaitournament.com/`
[7] `https://skatgame.net/mburo/orts/`
[8] `http://wargus.sourceforge.net/index.shtml`
[9] `http://us.blizzard.com/games/legacy/`

Dragoon versatile fast moving range unit

High Templar slow, but very powerful spell-casting unit; its ability — **Psionic Storm** deals high area damage (AoE), but costs almost half of the templar's energy points

Terran

Marine basic range unit that, when researched, may use the **Stimpack** ability, which temporarily boosts the marine's movement and attack speed but requires HP's to be casted

Firebat low range unit that deals *splash damage* (extra damage dealt to units that are not primary targets of the attack) in a small radius; it can also use the Stimpack ability

Medic supportive spell-casting unit with one key spell — **Heal**, which may be used to heal wounded friendly biological unit; medics are very often used in combination with marines and firebats

Siege Tank mechanical unit with two modes of attack — normal mode and siege mode; while in siege mode, the tank is incapable of moving, but its range of attack greatly increases and it also deals heavy splash damage

Vulture extremely fast, but fragile unit; due to its mobility, it is often used for worker harass.

Science Vessel aerial spell-casting unit; its **EMP shockwave** ability may be used to drain energy points from spell-casting units by shooting a fast projectile towards a selected area

Zerg

Zergling fast basic melee unit with relatively low HP but high damage, which makes them a favorite units for rushes or worker harass

Hydralisk versatile range unit

Mutalisk fast aerial unit that can attack both ground and air units; it has a special attack that may bounce up to two additional targets

Lurker special unit with two modes — normal and burrowed; while in normal state the unit can only move and cast the **Burrow** ability, which will change its mode to burrowed; while in burrowed state, the unit cannot move, but it may attack with a special range attack — line of spines that moves to its target while also damaging every enemy in its path

4.1.2 Combat simulation in SC:BW

As already mentioned in Section 3.4, we will need to repeatedly simulate combat in order to determine the fitness of individual chromosomes. Let us now consider our options in SC:BW.

Our first option is to use actual SC:BW. The problem is that BWAPI disallows us from manipulating its local instances directly. This way, we would have to, for

each chromosome evaluation, construct an entirely new SC:BW scenario, perform the combat and then gather results. In addition, BWAPI imposes a certain time limit on the maximum in-game speed. As this would be too slow for our needs, we must resort to some approximations and use a simplified model of SC:BW.

For that, we can use the SparCraft simulator. Now, whenever we need to evaluate two sets of forces against each other, we will simply construct a new SparCraft simulation, execute it, and obtain the results. We still have to keep in mind, that SparCraft is a simplified approximation of the game itself and as such, the results of the simulated battle will not exactly match the results in an actual game. However, we believe that simulations in SparCraft will not affect key properties of the resulting layout and that it will still be relevant.

4.2 SparCraft

SparCraft[10] is an open source combat simulator of the game SC:BW developed by David Churchill. SparCraft may be used in two main ways. First, it may be incorporated into a SC:BW bot to allow on-line combat simulations. Churchill himself uses this simulator in his UAlbertaBot[11], where it serves as a tool for fast decisions about the chances of army in combat — when allied units encounter enemy ones, the bot first simulates combat between them. Based on its results, the bot either orders the units to attack or to retreat. Secondly, it may be used to create standalone off-line simulations. The off-line simulations are further enhanced by an option for visualization which makes it really easy to reason about the SparCraft's behavior.

SparCraft models many aspects of the game precisely. Units have the same amount of HP's, they deal the same damage, move at the same speed and so on. However, few rules are relaxed and some things are not yet implemented. Units may move in only 4 directions (in SC:BW, they can move in 16 directions), the units do not accelerate while moving like in SC:BW, there is no fog of war, several of the advanced units are not implemented and there are no collisions between units.

4.2.1 SparCraft simulation

In order to create a new SparCraft simulation, 7 things are necessary — a map M, two lists of units U_1, U_2, their layouts L_1, L_2 respectively and specification of two AI players P_1, P_2 that control the units U_1 and U_2 respectively.

The map M is just a binary two dimensional array, 0 denotes impassable and 1 denotes passable map field. The two lists of units are just identifiers of units from SC:BW, like Protoss_Zealot or Terran_Marine. As SparCraft does not support all units, these lists must contain only supported ones. The layouts are a lists of initial positions on battlefield for each unit. The units must be placed on some passable map fields. The AI players are again identifiers for the AI players (from now on, we will refer to them simply as *players*) implemented in SparCraft.

With new simulation constructed we may now proceed to its execution. The execution of a each frame consists of three main steps:

[10] `https://code.google.com/archive/p/sparcraft/`

[11] `https://github.com/davechurchill/ualbertabot`

1. *generation phase*: based on the current state of the simulation, all possible moves are generated for both of the players; these moves have the form of: unit X may attack unit Y, or unit Z may move in upward direction and so on

2. *selection phase*: players will now, from the generated moves, pick the moves they want to actually perform with their units based on their policy

3. *execution phase*: lastly, all the moves are performed — units are moved, their properties like HP's or energy points are altered if they get hit or casted a spell and so on

We will refer to this three step frame execution as a *frame loop*.

The simulation ends when a certain time limit is met or when one player looses all of his units.

4.2.2 Architecture

The main functionality of SparCraft is implemented in 5 classes — Game, GameState, Unit, Action and Player.

Game: each instance of this class represents one simulation. It holds a current state of the simulation — instance of class GameState and players that play this simulations — instances of class Player. The main method of this class — *playNextTurn()* performs execution of one frame and as such it consecutively calls appropriate methods that perform each of the phases of frame loop. The method for generation and execution is provided by the GameState class while the method for selection is provided by the Player class.

GameState: this class represents state of the simulation. The most important fields in this class are two lists of units, one for each player. The units are instances of class Unit. The GameState provides a variety of methods. However, the most important ones are *generateMoves()* and *makeMoves()*. The generateMoves method performs the first step of the frame loop. As already explained, this method generates all possible moves for a player it received as an argument — it iterates through every unit of the player and for each such unit checks if it can move, which unit it can attack and so on, based on this information, it generates instances of class Action, one for every possible action unit can make. This list of all possible actions is the returned from this method. The second method, makeMoves(), performs the third step of the frame loop. As an argument it receives a list of actions selected by a player and then performs each action. If a unit X decided to attack unit Y, this method will perform all the necessary actions — it subtracts HP's of the unit Y (also, removes this unit from the game if it died) and sets attack cooldown for the unit X.

Unit: each instance of this class represents one unit on the battlefield. It has again many member variables like current hit points, position, id.

Action: instances of this class represent actions unit can perform. It holds a type of the action, which is just an enumeration constant like MOVE, ATTACK or WAIT. This class also holds a placeholders for the possible actions. For example, it holds a position, which will be filled for a MOVE action or it holds a unit target which will be filled for an ATTACK action.

Player: instances of this class represent players of the simulation. SparCraft has several players implemented which mirror some commonly used reasoning of units in battle. It provides some scripted players like ATTACKCLOSEST, or ATTACKWEAKEST. The ATTACKCLOSEST player controls his units such that each unit tries to attack its closest enemy in range. Units controlled by ATTACKWEAKEST try to attack weakest enemy unit in range. In addition, SparCraft provides more advanced search based players. For example, we have a player that implements Alpha-Beta search algorithm for its decision. Each player has an identifier and provides method *getMoves()* which maps to the second step of the frame loop. This method receives a list of all possible actions for this player. He iterates through them and returns the selected ones. This single method implements all of the player's behavior.

4.2.3 Our modifications

As already mentioned, the original version of SparCraft does not model some aspects of SC:BW correctly and some does not implement at all. In the previous use cases the simplifications of SparCraft either did not matter that much or were even necessary. Let's consider the already mentioned usage in UAlbertaBot. With the collision system properly implemented, it might not be possible to use SparCraft the way UAlbertabot does simply because the simulation could take too long. However, in our case, collisions are extremely important and without them, our algorithm can not possibly produce good results. That is because the situation when enemy unit gets stuck in chokepoint, which would be the preferred case, would be indistinguishable from the situation when they do not.

We have therefore decided to modify the original SparCraft for the purpose of this work. Our modified version has several new features. First, we have implemented full support for collisions. Furthermore, we have also added or refined a few unit-related features. The most notable feature is an infrastructure for adding units' abilities into SparCraft. This infrastructure was then used to implement several distinct abilities. There were two main reasons for addition of this infrastructure. Firstly, we wanted to try at least some of the advanced units from SC:BW. Secondly, the lack of abilities has been already mentioned in a few works that also used SparCraft, like in (Barriga et al., 2014, Section Conslusions and Future Work) or in (Alburg et al., 2014, Section 6.8).

Collisions and Pathfinding

Before we can give a basic idea of our solution, we need to explain how did MOVE actions work in the original SparCraft and how did they affect each phase of the frame loop. In the original SparCraft, each MOVE action was so called micro action — the units did not have any notion of path, but rather, they moved in a small steps toward their goals. This basic idea was followed throughout the whole frame loop in the following way:

1. generation phase: for each unit, this phase considered all 4 directions; given some direction X for unit Y, simple test was performed: if the unit moved in direction Y does step on some map tile with obstacle the MOVE action in direction X was not generated, otherwise, it was; (we can therefore see,

that the the original SparCraft had a very basic collision-awareness, but only with obstacles on the map)

2. selection phase: in this phase, player observed the generated MOVE actions; if his policy dictated to choose MOVE action in some direction, he simply chose it (for example, ATTACKCLOSEST player will select such MOVE action which will move the unit closer to the closest enemy unit)

3. execution phase: in this phase, the MOVE actions were simply executed no matter the collisions — unit was moved in a direction it chose to and its position was altered

Into this 3 phase system, we had to insert a way for handling collisions. To do this, we have inserted a module between the selection and execution phase. This module — *CollisionResolver* takes all MOVE actions that should be performed in the execution phase, detects all the ones that collide and repairs them. The correction of the colliding MOVE actions is done in two phases. First, collisions between moving and standing units are repaired. Then, correction of collisions between moving units follows. The correction may alter the MOVE action in three different ways. It may be shortened (unit can still move in the direction it wanted to, but not as much as it wanted) or it can be replaced by one of two actions — WAITONEFRAME or STUCK. The action type for replacement is determined by the number of consecutive collisions this unit had. Currently, the limit on consecutive collisions is set to 2, but can be easily changed. So, the action is replaced by WAITONEFRAME if the number of consecutive collisions is less than or equal to 2. Otherwise, it is repaired to STUCK action.

However, adding this module is of course not enough. We also need to adjust each phase accordingly. Our solution is based on the addition of a new macro action — MOVETOPOSITION and movement by path. Generation phase now always generates a macro action MOVETOPOSITION for each unit. Player may then choose this action for his units. When an execution phase sees the use of MOVETOPOSITION action, it calls a method to our additional module *PathFinder*. This module provides one method for finding a shortest path. It uses an A* algorithm and it is, to match the new functionality of collisions, collision-aware — this way, we have also secured handling of collision of units and obstacles. When finished, the pathfinder returns a path which is now set to the unit that used the MOVETOPOSITION action. When a generation phase sees that a unit has some path preselected and the unit did not got stuck in the last frame, it will also select a micro MOVE action from the path. This micro action now again allows the units to actually move in some direction.

It is worth pointing out that we have of course considered other approaches as well. One idea would be to place the collision resolving part somewhere else, maybe between generation and selection phase. This way, the player would receive only those move actions that he could perform. However, this approach would not work, because, we can not possibly predict actions the players would make. Collisions are products of selecting move actions in the selection phase and the first information we have about all the collisions that can occur is after the selection phase.

New unit features

The ability infrastructure is built upon three entity types — Abilities, Effects and Projectiles.

The first entity — Ability, represents an ability from SC:BW like stimpack or psionic storm. Each ability has several properties like cooldown or casting time and is tied with one unit. In order to allow the use of abilities, we have added a new action type — CASTABILITY. By adding this new action type, we have again, as with collisions, needed to modify each phase of the frame loop accordingly. When the execution phase sees some unit with supported ability it as with other moves, generates the action CASTABILITY if the unit can cast it (e.g. has enough energy points). Again, player may based on his policy react to this CASTABILITY action and select it for execution. Now, when the execution phase sees this ability, it calls the ability's core method, *beCasted()*. This method, apart from e.g. subtraction of unit's energy points, may produce either a Projectile or an Effect. For example, the casting of emp shockwave ability produces projectile while casting of stimpack produces effect. Neither effects nor projectiles affect any of the three phases, instead, they are handled at the beggining of each frame, before the generation phase. Each projectile has again several properties like speed or target position. It provides a method *nextAction()* that typically moves the projectile towards its destination. When the projectile reaches its destination, it can e.g. produce an Effect. For example, the projectile for EMP Shockwave ability would produce a blast that would drain energy from units in its area. The last entity — Effect represents product of either an projectile or casted ability. It provides method *affectUnit()* for affecting a single a unit.

This infrastructure was then used for implementation of the following features: stimpack ability, high templar's storm ability, medic's heal ability, siege tank's ability to change between states, science vessel's emp shockwave ability, lurkers burrow ability and also its very unique attack.

In addition to these changes, we have also modified attacks of a few units so they better match SC:BW. We have implemented firebat's splash attack, siege tank's splash attack (while it is in siege mode) and mutalisk's bouncing attack. These features affect only the execution phase. Whenever the execution phase sees an attack from a unit with a unique attack, it simply forwards the control to a special method that is able to handle these attacks and select appropriate targets.

4.2.4 Implications of our modifications

We have already seen, that there are multiple AIs implemented in SparCraft that we could potentially use for our evaluation. However, with our changes explained, mainly the collision system, it is now apparent that we have actually no players in SparCraft available, because these player do not know how to use our newly added action types like MOVETOPOSITION or CASTABILITY. We could do without the support for new abilities and units — we would simply evolve layouts for unit types the original SparCraft supported. However, as already mentioned, collisions are crucial to our work. We have therefore needed to modify some so we can perform the simulations correctly.

We will need at least two players to support the chromosomes evaluation. We

will need one to represent the defender, we will call him *Defender_AI* and one that will represent the attacker, we will call him *Attacker_AI*. At the core, both of these players behave the same — they behave according to NOKDPS player that was already in the original SparCraft, but this time changed for collision support. The NOKDPS player comes out of a very simple ATTACKCLOSEST player, which controls his units such that they try to attack their closest enemy. NOKDPS improves this behavior by two features. First, the units do not attack units that were already assigned with lethal damage in current frame. Let's consider the selection phase, where player selects actions for two of his units — unit X and unit Y, both of which can attack some enemy unit Z with very low hit points, such that only one attack from either X or Y would kill it. Now, player during the selection of move for unit X selects the attack on Z. By selecting this move, he ensured the kill on unit Z. Now, selection of action for unit Y. This unit can also attack unit Z and some players could dictate the unit Y to attack Z as it is for example closest enemy to Y. But, NOKDPS player considers the fact that unit Y will be guaranteed to die this frame. Based on this knowledge, he chooses some different move for Y. This No OverKill behavior (first three letters from the player's name) is quite common as it increases the damage output. The last three letters of NOKDPS stand for Damage Per Second and it denotes the behavior of target selection for attacks – second feature that is improved over the ATTACKCLOSEST player where units select their target for attacks based on their distance. The damage per second selection is based on selecting the target with highest damage per frame, which is just unit's damage in one hit divided by its attack cooldown. This way, units prioritize highest threats in their attack range. In addition, this selection is further improved by another change. The damage per frame value of target unit Z is further divided by the remaining hit points of unit Z. This way, the units also try to prioritize enemy units with low hit points. The unit target selection is therefore based on the following formula 4.1 — units select such unit u in their range that has the highest dpsHPvalue.

$$\text{dpsHPvalue}(u) = \frac{(\text{damagePerHit}(u)/\text{attackCooldown}(u))}{\text{hpRemaining}(u)} \tag{4.1}$$

This behavior of NOKDPS player (with change to support collisions) is then depicted by behavior diagram on the Figure 4.1.

Our Attacker_AI will play according to the NOKDPS player, Defender_AI will with two exceptions. Firstly, as it will control the defender's units, we need to add one additional condition that will keep the units stationary until the opponent assaults the chokepoint. The modified version for defender does not move his units in anyway until one of the following conditions is satisfied — either there is an enemy unit in range of the defender's units or one of the allied units received any damage. When the condition is satisfied, the defender plays exactly as the attacker. Secondly, we have also implemented a simple script that allows the defender to control one more complicated unit — medics. This script controls medics in such way, that they try to reach the nearest friendly injured unit. The reason why the Attacker_AI does not have this behavior implemented is that we would need a more sophisticated script in order to prevent possible unwanted blocks in the chokepoint that would degrade the attacker's performance.

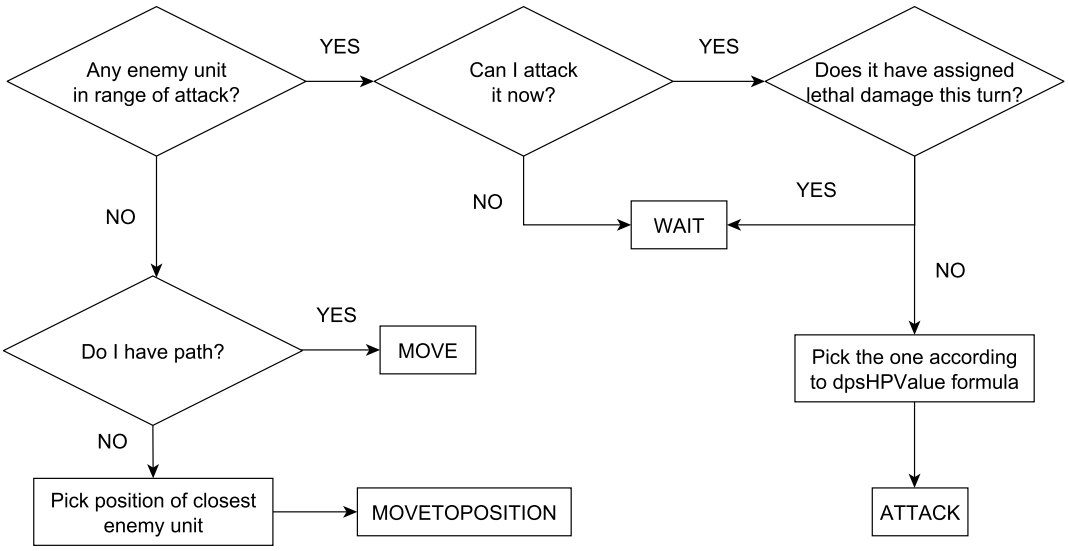

Figure 4.1: NOKDPS (No overkill, damage per second) behavior diagram

4.2.5 Implications of using SparCraft

We can now see that by using SparCraft in our work, we have imposed several restrictions to our algorithm. As we have only our two relatively simple players in SparCraft available — Defender_AI and Attacker_AI, we will always evolve layout for unit combination U_d given that the units are controlled by Defender_AI against unit combination U_a given that the units are controlled by Attacker_AI. In addition, we can use only units supported by SparCraft.

However, these restrictions should not diminish our result. The SparCraft may be further improved — new players may be easily added and new units may be added as well.

4.3 FormationsEvolver

This part will cover the second core module of our system, which is responsible for the actual finding of a favorable layout with the use of Genetic Algorithms.

4.3.1 Evolving single layout

This sub-system follows the basic system diagram presented in Figure 3.2. It requires 5 inputs — a chokepoint CP, players P_d and P_a and their unit sets U_d and U_a respectively. It expects each of these inputs to be placed in a separate file.

The file with chokepoint is divided into two sections, the first section holds some basic information about the map area with chokepoint. The second section contains a two dimensional array that describes this map area. Each map field [x][y] corresponds to one 8x8 tile from an actual SC:BW map. For example, value 0 means impassable map field and 1 means passable one. These input files were created by one of our supportive sub-systems — ChokePointParser in

combination with BWTA library [12] for initial chokepoint detection.

Players P_d and P_a are identifiers of SparCraft players. As explained in the section with SparCraft (4.2.5), in our case, these will be always fixed to our Defender_AI and Attacker_AI. In order to allow future use of other players, these identifiers for player are also provided in their own input file. The files for unit sets have on each line one type of unit with its count, where the type is identifier for SparCraft, for example Protoss_Zealot 3.

In addition to these 5 input files, this sub-system also expects a file that is used to configure most of the parameters of genetic algorithm. This file contains configurations to the number of iterations, rates of genetic operators (chances for mutation or crossover), their types (One point or Two point crossover), type of genetic algorithm (whether to use RoundRobin or CoEvolution) and a few other — whether to use Plague or Elitism. It is worth pointing out, that with rare exceptions specific to the types of algorithms we use, all of the remaining settings of the framework for generating layouts are configurable by these input files. This allows really easy reuse without the need for recompilation.

When the input is processed, the control is transferred to modules with the genetic algorithm itself. The algorithm may vary in several aspects depending on the configurations, however, the following basic steps from the genetic algorithm presented in 3.1 are independent of other configurations: First, initial generation of chromosomes is created, where each chromosome starts with a random layout for U_d. Now, the GA continues in iterations, each *GAiteration* consists of two main steps — *evaluation phase* and *new generation phase*. In evaluation phase, each chromosome is assigned with fitness based on one or more simulations in SparCraft. The new generation phase creates a new population by repeatedly selecting two chromosomes from the old generation and applying genetic operators like mutation or crossover.

The GAiterations are repeated until the termination condition is met. The final layout is then outputted into a file with the description of this experiment.

4.3.2 Architecture

Architecture of the FormationsEvolver is overall highly modular and follows the basic principles of Objected Oriented Programming. As such, this system is very easily extensible by a new types of e.g. genetic operators, or genetic algorithms. The core parts of its architecture may be seen on Figure 4.2. We will now cover each part of this system in a greater detail.

GA_LayoutFinder: Each instance of this class represents one run of our genetic algorithm. It provides only one method — *findBestLayout()*. Calling it will based on the current configuration of genetic algorithm first construct an instance of class derived from class GA_Base. Now, the rest of this method represents only a thin layer over the main method of GA_Base, which it repeatedly calls, until a termination condition of the GA is satisfied. When this condition is met, the method returns the best layout returned by the instance of GA_Base.

GA_Base: This class represents the main logic of a genetic algorithm. As of now, we provide two implementations of this abstract class that correspond to two main types of algorithms we have used in our work. That is CoEvolution and

[12] `https://code.google.com/archive/p/bwta/`

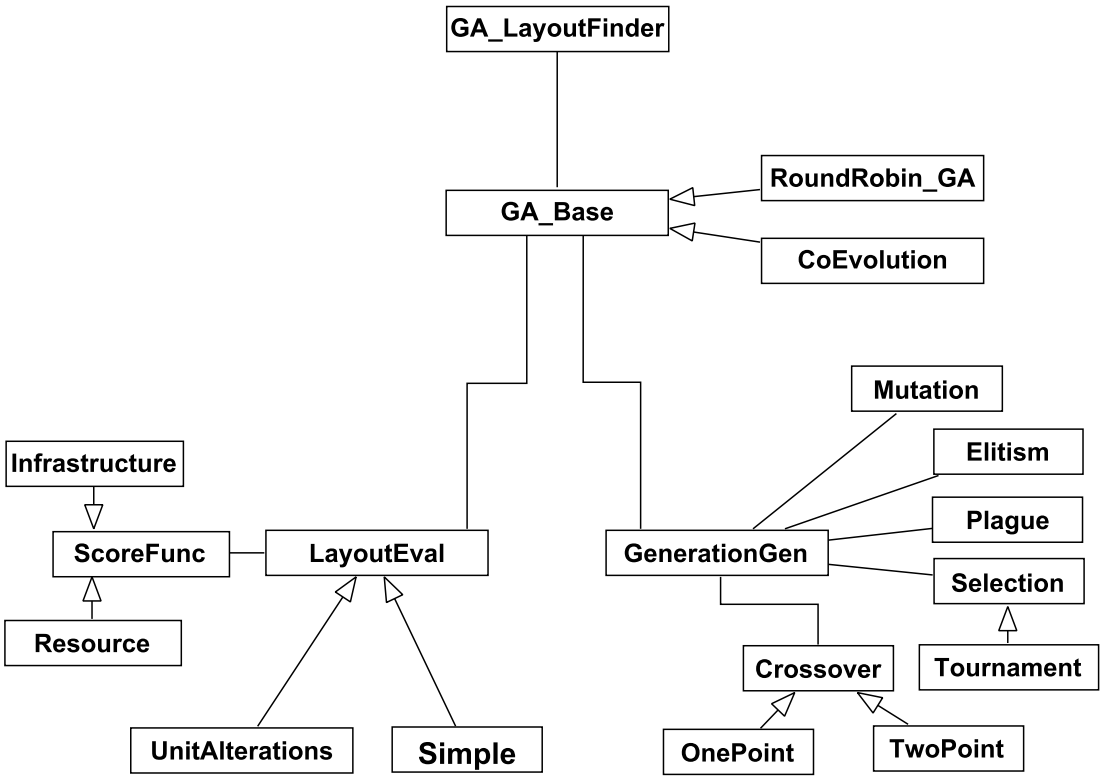

Figure 4.2: Core architecture of our module with GA — FormationsEvolver

RoundRobin GA, which were presented at the end of section 3.4. RoundRobin genetic algorithm evolves the defender's layout against multiple attacker's layouts and CoEvolution gradually improves both defender's and attacker's layouts (we can still use the most basic case of GA with optimization against only one attacker layout because it is a special case of RoundRobin GA where the number of attacker's layouts is equal to one).

The abstract class GA_Base enforces implementation of one method to its derived classes — *evolveNextPopulation()*, which calling corresponds to one GAiteration and its two phases — evaluation and new generation.

The evaluation phase is rather algorithm specific, but the support for this phase is partially facilitated by the instances of class LayoutEvaluator (depicted as LayoutEval). This class receives a tuple $X = (U_d, L_d, U_a, L_a)$ and by performing a SparCraft simulation and its scoring, it returns the score of this tuple, $S(X)$. In the case of RoundRobin GA and its assignment of fitness to chromosome C, the method on LayoutEvaluator is repeatedly called for each tuple (U_d, L_d, U_a, L_i), where $L_i \in L = \{L_1, \ldots, L_m\}$, in order to determine the scores $S = \{S_1, \ldots, S_m\}$. Given this set of scores S the RoundRobin GA may now decide that the resulting fitness of C will be the average of the scores from S or its minimum and so on — this behavior of the algorithm is parameterizable by a functor in order to allow reusability. In the case of CoEvolution, the evaluation phase is rather simple as it calls the LayoutEvaluator for only a single tuple $X = (U_d, L_d, U_a, L_a)$.

The whole logic behind the new generation phase is provided by a single method of a GenerationGenerator class (depicted as GenerationGen)

LayoutEvaluator: this class provides only a single method — *evaluateT-*

woLayouts(). Following the example from the description of class GA_Base, this method receives the tuple X and computes its score $S(X)$. This computation is done in two phases. First, a SparCraft simulation is executed. Then, the result of this simulation is passed into an instance of class ScoreFunc, that implements function for mapping the simulation result to a value, like resource cost formulat from 3.1.

The LayoutEvaluator class is also an abstract one. That is because we can assess the performance of a single layouts based on different types of advanced criteria. We can have one LayoutEvaluator that does not alter the forces it passes to SparCraft in any way. And we can have one that for example alters the forces of each side before the simulation in SparCraft is performed. Therefore, in addition to a Simple LayoutEvaluator, which does not change the tuple X it receives in any way, we may also use a UnitAlterations LayoutEvaluator that may alter the forces.

GenerationGenerator: this class is responsible for the new generation phase. It provides only a single method *createNewGeneration()*. It takes an old generation and returns a new one created by the use of genetic operators. The algorithm for creating the new generation follows the standard loop from the genetic algorithm in 3.1. Each supported operator is detached into its own class — we have ElitismPerformer (depicted as Elitism), PlaguePerformer (depicted as Plague), ChromosomeSelector (depicted as Selection), CrossoverPerformer (depicted as Crossover) and MutationPerformer (depicted as Mutation).

5. Experiments

In this chapter, we will assess the viability of our presented approaches by performing multiple sets of experiments.

To perform our experiments, we have parsed several chokepoints from a few maps from a map pack used in the SSCAI tournament [1]. As explained in section 4.2.5, all of our experiments were performed with two fixed players. Our Defender_AI controls defender's units and our Attacker_AI controls attacker's units. We have prepared experiments with several standard unit compositions standing against each other. Due to the limitations of our AIs, these compositions consisted only of Zealots, Dragoons, Marines, Firebats, Medics, Vultures, Zerglings and Hydralisks.

5.1 Initial experiments

We have started with a simple RoundRobin GA, where the attacker's unit composition U_a is assigned with only a single layout L_a, which is fixed throughout the GA's whole run. We will refer to this type of algorithm as 1RRGA. As there is only a single layout for the attacker, the fitness of each chromosome C with layout L_d is determined only by the evaluation of tuple $X = (U_d, L_d, U_a, L_a)$, with U_d being the defender's unit composition. To evaluate this tuple, first, a battle in SparCraft is simulated. On the result of this simulated battle, we use the resourceCost formula (3.1), which gives us the score $S(X)$, which is also the fitness of chromosome C. In these initial tests, we have used a TwoPoint crossover with rate 0.95, population size of 75, the algorithm terminated after 75 iterations, mutation rate set to 0.25, the random values for mutation sampled from a normal distribution of mean 0 and variance 10, the elitism rate set to 0.1 and Tournament selection.

We have performed several experiments with this configuration, using different unit combinations and different chokepoints. In each of those experiments, the fitness gradually improved by a significant amount and the resulting best layout (best chromosome after the last iteration of GA) offered a layout that greatly improved the combat performance of the defender's units (in contrast to the initial random layouts of the GA). Then, we took the resulting layout L_d for U_d and assessed its performance against a few dozens unseen layouts of the attacker for U_a. The resulting fitness scores of this *fitness comparing experiment* can then indicate the overall robustness of the produced layout as we can look at for example an average achieved fitness or deviances. We will refer to this criterion for assessment of the resulting layouts as a *fitness metric*.

Based on the fitness metric alone, the layouts produced by this simple algorithm seemed to perform quite well and seemed to be reasonably robust. However, assessing the performance of our algorithm based on this metric alone may not suffice and in order for us to assess the robustness of the resulting layouts, it might be also important to observe the layout visually. Then, we can use our knowledge of RTS games and argue, whether the final layout provides a good

[1] http://sscaitournament.com/

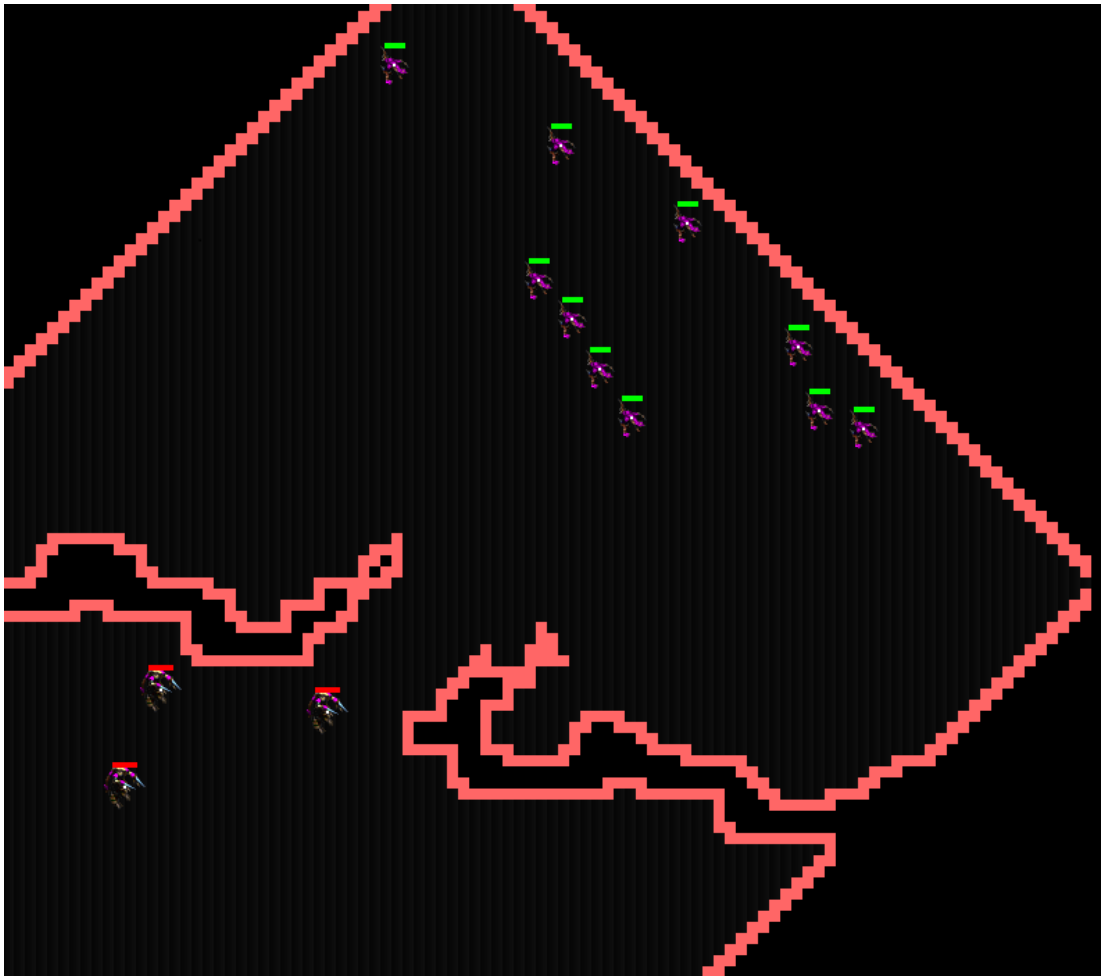

Figure 5.1: Picture showing a typical layout produced by our initial version of 1RRGA in an experiment with 3 Zealots on the defender's side and 10 Zerglings on the attacker's side.

defensive formation against units U_a given that for example the attacker controls his units a bit differently than our simple Attacker_AI.

We have therefore also observed the final resulting layouts within SparCraft. A typical layout produced by our GA in an experiment with 3 strong Zealots on the defender's side and 10 weak Zerglings on the attacker's side may be seen in Figure 5.1. On first sight, the produced layout for Zealots seems rather poor overall, even though our fitness metric indicates otherwise. The problem is apparent when one observes the actual course of battle between the units. As alredy presented, our Attacker_AI controls his units according to the NOKDPS script presented in Figure 4.1. Let us reiterate that this player controls his units according to a few sets of rules, one of which is that if a unit is able to attack an enemy in range, it does so. This way, it suffices for the defender to put only a single unit next to the chokepoint. This positioning then causes, that the attacker's units block themselves in the chokepoint when they approach the Zealot inside the chokepoint. This layout then achieves a really high score as the units actually correctly utilized the chokepoint — only a few Zerglings are able to attack at a time.

However, it is apparent that this layout would perform quite poorly in a few

different situations. First, lets consider that the attacker would micromanage his units a bit more cleverly in the chokepoint. For example, if he would moved the front Zerglings through the gaps between the Zealot and wall (before his allies reach the chokepoint), several additional Zerglings would be able to inflict damage. However, our Attacker_AI does not have this behavior implemented and as such, several Zerglings in the back are, during the combat, simply blocked by the Zerglings in the front. In addition, let's consider that the Zealots try to defend this chokepoint to prevent some potential worker harass or any other attack from the attacker that would aim at the defender's production infrastructure. In this situation it would be much better for the Zealots to block the chokepoint, as that would prevent the Zerglings running pass them towards the production infrastructure. But again, the GA we have used in this experiment did not subdue the quality of the defender's layout to any such criterion and therefore it can not distinguish how poorly this layout would perform in this situation.

In order to achieve better results, we have decide to address the issue presented in the second situation.

5.2 Adding an additional fitness criterion

In the previous section, we have seen that using only a single criterion for determining the quality of layouts does not lead to good results and we have decide to add a new criterion for determining the quality of a single layout. In addition to the criterion already used (scenario, in which the units simply battle between each other), the layout is subdued to a scenario in which the defender's units try to guard production infrastructure, while the attacker's units try to attack it.

This additional second criterion will affect our 1RRGA with only one attacker layout L_a in the following way. Let us consider the step in the algorithm when we determine the fitness of a chromosome C with layout L_d. Now, the score $S(X)$, which determines the fitness of chromosomes C will be an average of two components, $S_1(X)$ and $S_2(X)$. The component $S_1(X)$ will be computed in the same way — without any modifications, the tuple $X = (U_d, L_d, U_a, L_a)$ is passed to SparCraft for simulation and the result of this simulation is scored by the resourceCost scoring function. The component $S_2(X)$ will be computed in a similar manner. We will again pass some configuration to SparCraft, gather the result of the simulation and use a scoring function on the result. However, we need to impose several changes to our established evaluation.

First, we need to modify the U_d and L_d that are passed to SparCraft before simulation. That is because we need to add some units, that would represent the production infrastructure. The number of added units as well as their layout may be determined once before the start of the GA. In our case, we have decide to add 5 Probe units to the defender (Probes are worker units for the Protoss race) which mimic the infrastructure. They are always placed into the furthest corners of the defender's side of chokepoint. This way, the attacker's units have to pass the defender's units and travel some additional distance before they reach the production infrastructure — as they would in a real game. In addition, as the Probes represent either workers or structures, and potentially, many of them, their HP's were manually altered inside SparCraft.

However, when we pass the modified U_d and L_d — U'_d and L'_d with Probes

to SparCraft and launch the simulation, our current Attacker_AI is not capable of reacting to the production infrastructure. The presented NOKDPS behavior diagram from Figure 4.1 does not in any way consider this situation and we need to alter it accordingly. We have therefore modified the NOKDPS player. This player now controls his units in such a manner, that they try to deal as much damage as they can to the Probes. If there is a gap between the defender's units, the attacker's units omit attacking them and they try to reach the Probes. The behavior in the situation without any production infrastructure remained unchanged.

The last issue we need to address is choosing the scoring function. The resourceCost scoring function could potentially work in this situation. Instead of modifying the Probes HPs, we could have added more of them. In that case, the scoring function would give us a sensible result — if the attacker is capable of killing more workers, the score produced by this function would be smaller. However, we have decided to design our own scoring function that would score the simulation in a better way. Let us consider each attack of the attacker's units performed during the course of one simulation. Instead of simply summing the overall damage, we will also consider one additional factor — the time of the individual attacks on the Probes with respect to the length of the simulation. The reason for this consideration is that any potential damage dealt to production infrastructure during the course of a time period is more threatening at the start of the time period. Towards the end of this period, the player could have been capable of producing more units, gathering additional minerals or constructing more structures.

Therefore, in addition to each damage d dealt to Probes, we also consider the time it was dealt — t, with respect to the simulation's length T. For each damage d, we will determine a multiplicative factor m which depends on the time t and is computed by function infrMultConst,

$$\text{infrMultConst}(t) = \exp\left(\frac{-t}{\frac{T}{4}} + \log\left(\frac{7}{2}\right)\right) + \frac{1}{2}. \tag{5.1}$$

This function multiplies the damage dealt at the begging of the simulation by a factor of around 2, damage done in time $\frac{T}{2}$ by a factor of about 1 and the damage done in the later stages of simulation, towards the time T is multiplied by a factor of about $\frac{1}{2}$. We have also tried a few different combinations of the parameters and this function seemed to serve our purpose well enough.

The component $S_2(X)$ is then computed in the following way. Given the tuple $X = (U_d, L_d, U_a, L_a)$, we consider tuple $X' = (U'_d, L'_d, U_a, L_a)$ and result of its simulation $\text{sim}(X')$ and the list of pairs $W = ((d_1, t_1), ...(d_n, t_n))$ from $\text{sim}(X')$, where each pair i corresponds to an attack at time t_i and damage d_i. Then we use the function infrastructureHunt defined as

$$S_2(X) = \text{infrastructureHunt}(\text{sim}(X')) = \left(\sum_{(t,d)\in W} \text{infrMultConst}(t)d\right), \tag{5.2}$$

to compute the resulting value. To further distinguish between these two criteria, we will use the following notation. Given a tuple $X = (U_d, L_d, U_a, L_a)$ for evaluation, the component $S_1(X)$ will now be referred to as the score of the *battle*

scenario, or *first scenario*, and the component $S_2(X)$ will now be referred to as the score of *infrastructure hunt scenario* or *second scenario*.

With the second criterion introduced, we are now interested in seeing how our enhanced GA performs. We have again performed several experiments with this configuration. Visually, the layouts started to resemble some actual formations and in some experiments, the produced layouts looked really promising. For example, in the experiment with 3 Zealots and 10 Zerglings, this enhanced GA was able to usually produce layout in which the Zealots entirely blocked the chokepoint — a desired behavior with these two units compositions against each other.

However, despite the improvement, we have also observed rather poor layouts similar to the one presented in Figure 5.1. The reason for these poorer layouts were exploits of our current configuration from the defender. These exploits were enabled because of our simple Attacker_AI coupled with only a single, fixed, randomly generated layout for his units — L_a. Given that the attacker's unit composition consists of range and melee units, the random L_a occasionally caused that the range units were positioned in front, closer to the chokepoint while the melee units were positioned in the back. Then, when the attacker's units reached the chokepoint, this poor initial positioning sometimes caused that the melee units got blocked by the range ones. This proved to be an issue mainly in the first scenario, as it allows a rather poor defender's layout to achieve a significant score in the first scenario. This score can then outweigh the insufficiency of this layout in the second scenario. This way, it is difficult for our GA to distinguish between a really good layout and a rather poor one, as the differences are, due to the score of the first scenario, negligible. This way, the GA can produce a layout, that is rather weak with respect to the second scenario. Similar problems occurred even in situations when the attacker does not control any range units. Our GA simply exploited the attacker's poor initial layout. The GA sometimes found a layout, that caused a cascade of blockades among the attacker's units. These blockades again allowed an otherwise poor layout to achieve a high score in both of our scenarios.

We could either improve our Attacker_AI or we can address the issue of a single fixed random layout L_a. We have chosen to latter one as it can be addressed by using the our other presented types of genetic algorithms, which are covered in the following section.

5.3 CoEvolution and RoundRobin GA

In the previous section, we have seen that using only a single fixed layout for attacker during the course of the Genetic Algorithm does not lead to very good results due to occasional exploits. In order to tackle this problem and achieve more robust and consistent results, we can try to use our more sophisticated types of GAs presented at the end of section 3.4 — CoEvolution which evolves the attacker's layout or RoundRobin GA with multiple enemy layouts.

5.3.1 CoEvolution

Let us remind the reader, that this type of GA is based on gradual improvement of the attacker's layout L_a throughout the course of the GA's run. In addition to the defender's generation G_d, the algorithm also keeps a generation of layouts for the attacker — G_a. The algorithm then alternates between optimizing the layout for the attacker and for the defender, each time picking the best layout the current player possesses. The alternation may be based on a few factors. One approach is to choose a fixed number of iterations, another possibility is to wait for sufficient change in fitness values. Disregarding the condition, we will refer to the iterations in which the GA optimizes layout for the defender as the *defender's turn* and to the iterations in which the GA optimizes layout for the attacker as the *attacker's turn*.

In theory, this algorithm could work very well. As the GA gradually improves the initial layouts of the attacker's units, the resulting best chromosome for the defender should represent quite a robust result that was able to withstand even the attacker's improved layouts. If we would improve the attacker's layouts for a sufficient number of iterations, we could find a defender's layout that would perform well regardless of the initial positioning of the attacker's units. Whether this algorithm behaves in this way can easily be recognized by looking at the change in fitness values between the players' turns. As in the other algorithms, we would expect that the fitness of the best defender's chromosome would gradually increase. But in addition, with each new attacker's turn, it should be harder for the attacker to find a layout that would perform well against the defender. This would be reflected in a gradually lower and lower difference between the fitness of the best defender's chromosome and fitness of the best attacker's chromosome.

In order to test this assumption, let us proceed with preliminary experiments.

In the following tests, the fitness of individual chromosomes is determined by the evaluation of a single tuple $X = (U_d, L_d, U_a, L_a)$, where the layouts L_d and L_a depend on the current step of the algorithm. In case of the defender's turn, the layout L_d is taken from the chromosome C_d (of generation G_d) where the layout L_a was produced by the optimization during the previous attacker's turn. During the attacker's turn, we have a fixed L_d and we take layout L_a from chromosome C_a of generation G_a. The score of $S(X)$ will again be the average of two components — score of the battle scenario $S_1(X)$ and score of the infrastructure hunt scenario $S_2(X)$. It is worth pointing out, that we don't need to alter the computation of score $S(X)$ based on the current player's turn. The only imposed change is that during the defender's turn, the algorithm tries to maximize the value of $S(X)$ while during the attacker's turn, the algorithm tries to minimize it.

The condition for alternation from one player's turn to the other was set to a fixed number of iterations. The number of iterations during the defender's turn was set to 20 and the number of iterations for the attacker's turn was set to 5. Total number of iterations was set to 800. The defender's population size was set to 75 and the attacker's population size was set to 50. The reason for the higher number of iterations during the defender's turn is that our primary goal is to find a good layout for the defender and we want to give the GA a sufficient number of iterations to find a reasonable layout against the layout produced in the attacker's previous turn. The arguments for the bigger population size for the defender are similar.

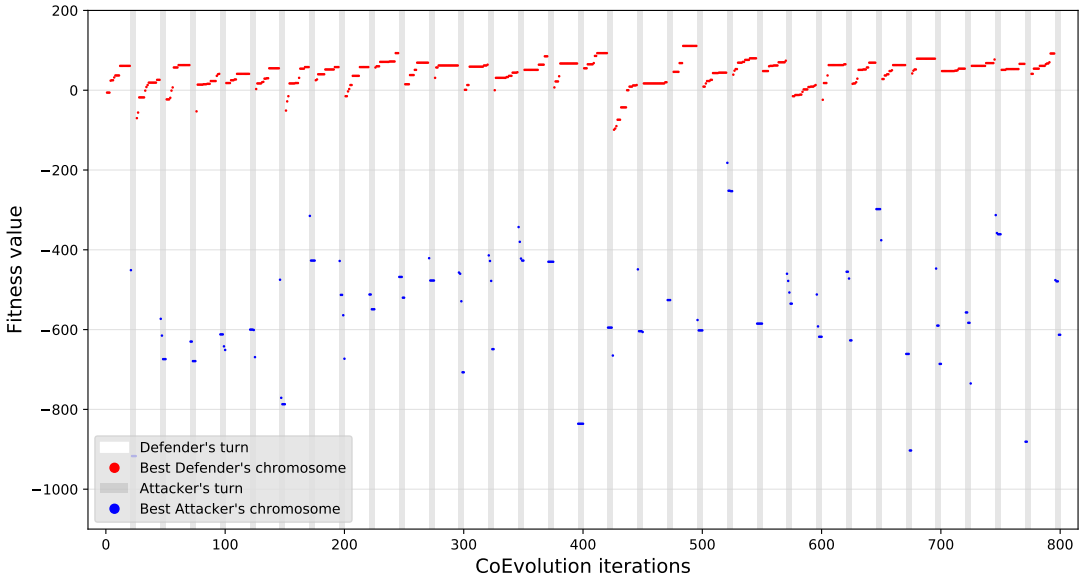

Figure 5.2: Plot showing changes in fitness from one run of CoEvolution. This plot shows the fitness changes from an experiment with 4 Medics and 3 Marines on the defender's side and 2 Zealots and 2 Dragoons on the attacker's side.

The rest of the parameters of the GA were same for both of the players. We have used a TwoPoint crossover with rate 0.95, mutation rate was set to 0.25, with random values sampled from the normal distribution with mean 0 and variance 10, the elitism rate was set to 0.1 and we use Tournament selection.

Using this configuration, we have again performed multiple experiments with different unit compositions and different chokepoints.

Now, we would like to assess whether this algorithm behaves according to our expectations. For that, let us consider a plot showing the changes in fitness from experiments with 4 Marines and 3 Medics on the defender's side vs 2 Zealots and 2 Dragoons on the attacker's side. This plot may be seen in Figure 5.2. Plots similar to this were observed in all of the experiments. From this plot, we can observe that up to the 400th iteration, the algorithm more or less worked as we wanted — with (almost) each turn, the defender is able to find an overall more robust layout, to which the attacker fails to find a good answer for. During the attacker's turn in 400th iteration, we can however see, that the attacker was suddenly able to find a layout that performed really well with a fitness score of about -800. The main reason for this is that the algorithm could have, in the previous defender's turn, produced a rather poor layout by exploiting a layout against which it optimized. This is the main problem of this algorithm as this exploit pollutes the defender's generation with additional poorer layouts.

Despite this flaw, this algorithm has one major advantage over the 1RRGA. If an exploit of the attacker's layout happens and one defender's turn produces a poor layout, the algorithm is able to fairly quickly, in the next attacker's turn, recognize the poorness of this layout. As a result, the layouts produced by this algorithm were quite good overall and robust both visually and according to our fitness metric. More detailed comparison with other presented algorithms is covered in the last section.

5.3.2 RoundRobin GA

Now, let's consider the RoundRobin GA in a generalized settings, with m fixed random layouts for the attacker — $L_a^{(i)}$, $i = 1, \ldots, m$.

This generalization to m attacker's layouts affects only the evaluation phase of the algorithm. The fitness of a single chromosome is now based on the evaluation of m tuples $X^{(i)} = (U_d, L_d, U_a, L_a^{(i)})$ and their scores $S(X^{(i)})$ where $i = 1, \ldots, m$. In our experiments, we again determine the score $S(X^{(i)})$ as an average of two components, score $S_1(X^{(i)})$ of the battle scenario and score $S_2(X^{(i)})$ of the infrastructure hunt scenario. Given the scores $S(X^{(i)})$ for each $X^{(i)}$, $i = 1..m$, the assignment of the fitness to chromosome C_d can now vary. One approach is to take the average of the scores $S(X^{(i)})$, or their minimum. We will refer to the RoundRobin GA which takes the average of scores $S(X^{(i)})$ as *avg_mRRGA* and to the algorithm that takes the minimum as *min_mRRGA*. Intuitively, the min_mRRGA should perform better overall than the avg_mRRGA — min_mRRGA is forced to optimize against the attacker's layout which is best against the defender's current layout.

Let's perform a preliminary set of experiments in order to test this assumption.

First, we need to address the choice of the parameter m. It is clear, that higher values of m should guide the GA towards more robust and flexible layouts, as the chances of clever exploits of the attacker's positioning declines. However, with higher values of m, the computational difficulty of the GA starts to become more important, as it effectively slows down the algorithm by a factor of m (with comparison to the 1RRGA). That is because the evaluation phase of the GA is by far the most limiting factor to the speed of our GA as each SparCraft simulation takes a non-trivial amount of time. For our experiments, we have chosen m to be 7 — this amount of attacker's layouts should provide us with big enough sample for good results and the computational time demands should not limit testability.

With these two algorithms, we also considered whether the use of Plague would have any effect on the algorithm's performance. We were interested in the effect of two parameters — the rate of the Plague (the amount of new chromosomes added to the generation with respect to the population's size) and its frequency. We have tried 9 configurations. Plague with rates 0.25, 0.5 and 0.75 and frequency of 5, 10, and 20 iterations. Our preliminary tests showed, that none of these configurations had any big impact on the algorithm's performance (in contrast to the algorithm with no Plague), and some combinations of the Plague parameters produced even worse results. The only combination of parameters that offered a slight improvement in all of our experiments was Plague with rate 0.25 and frequency 5.

For both our min_7RRGA and avg_7RRGA, we used the following configuration. The termination condition was after a fixed number of 75 iterations, population size was set to 75, mutation rate was set to 0.25 with values sampled from a normal distribution with mean 0 and variance 10, we used a TwoPoint crossover with rate 0.95, Tournament selection, elitism of rate 0.1 and plague with rate 0.25 and frequency of 5 turns.

With this configuration, we have performed several experiments with different chokepoints and different unit compositions.

According to the fitness metric, both of these algorithms performed, with rare exceptions, better than the 1RRGA. Surprisingly, min_7RRGA did not offer

the improvement we had expected and typically, it performed comparably to avg_7RRGA. Even though the avg_7RRGA is more prone to exploits — a factor which should be reflected in the fitness metric, we observed that the min_7RRGA tends to, despite the Plague, soon diverge into a local optimum. This is caused by the rather strict condition for assignment of fitness to chromsomes and the algorithm tends to throw away the chromosomes that could potentially lead to much better layouts. Nevertheless, our test showed that both, avg_7RRGA and min_7RRGA are viable options that offer an improvement over the 1RRGA and we will further assess their performance in the last set of experiments covered in the last section.

5.4 Final experiments

In our last set of experiments, we were interested in comparing the presented approaches — 1RRGA, CoEvolution, avg_7RRGA and min_7RRGA to two baselines — random layout and our manual layout.

We have assessed the resulting layouts both visually and according to our fitness metric. In order to perform the fitness comparing experiment, we have generated 30 random layouts for the attacker and evaluated the resulting layout the same way we did in the genetic algorithm — by taking the average of battle scenario and infrastructure hunt scenario. In order to preserve equal conditions, we have generated the final 30 layouts with same seed (14052017) for each experiment.

Moreover, for each combination of units and each type of approach, we have performed 5 runs in total (this includes the random layout). Then, we have considered the fitness values produced by the fitness comparing experiment for each individual run. From the 5 sets of fitness values corresponding to each run, we have taken an average of each one and considered the layout with highest average as the resulting layout for the corresponding approach.

For our 4 approaches, we have used the same settings outlined in the previous sections. All of the experiments were performed on 3 different chokepoints taken from 2 SC:BW maps.

In our first experiment, we considered 3 Zealots on the defender's side and 10 Zerglings on the attacker's side. The results of the fitness comparing experiment may be seen in Figure 5.3. From the plot, we can see that all 4 of our approaches offer a significant improvement over the random layout. In the case of CoEvolution, the resulting layout even achieved similar result to our manual layout (depicted as "Human"). When we observed the layout produced by CoEvolution, it looks almost identical to ours — all of the 3 Zealots were placed inside the chokepoint in order to block the Zerglings. The layouts produced by the RoundRobin-type algorithms were in all cases slightly worse, which is also visible on the presented plot.

In the next experiment, we considered 3 Dragoons and 2 Zealots on the defender's side and 3 Zealots with 2 Dragoons on the attacker's side. The resulting box plot may be seen in Figure 5.4. As with the previous experiment, we can see a significant improvement over the random layout and interestingly even over our manual layout. In our manual layout, we have placed the 2 Zealots more towards the chokepoint and 3 Dragoons slightly behind them into a concave for-

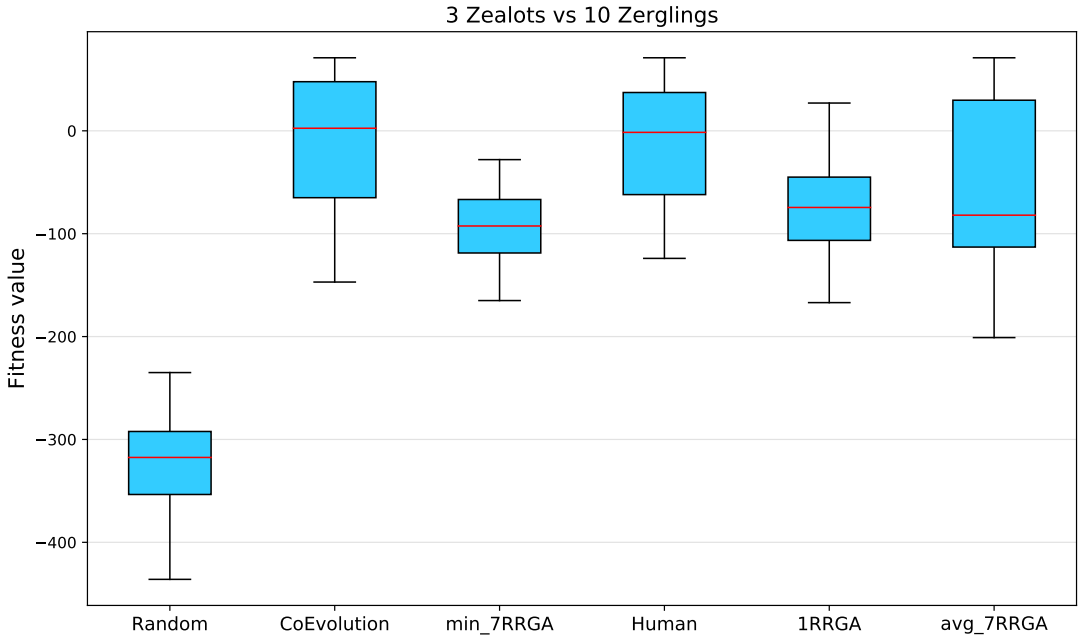

Figure 5.3: Box plot with fitness values from the fitness comparing experiment of individual approaches in experiment with 3 Zealots on Defender's side against 10 Zerglings on Attacker's side.

mation. This layout is able to achieve a good score in the battle scenario and it also performs quite well in the infrastructure hunt scenario.

However, in our current configuration, this layout can be potentially enhanced in both of the scenarios. In order to understand the potential improvements, let us remind the reader that the Defender_AI controls his units according to the NOKDPS diagram from Figure 4.1 with one additional condition. The units remain in their initial positions until one of two conditions is satisfied — either there is an enemy unit in the range of one of our units or one of the allied units received any damage. In a battle scenario, Attacker_AI is controlled only by the NOKDPS player and in the infrastructure hunt scenario, the player is enhanced by a simple rule which effectively checks if there is a path towards the enemy infrastructure. If there is, the unit takes this path. Now, in the battle scenario, it is not that beneficial for the defender to place his units directly inside the chokepoint — if the enemy Dragoons approach the chokepoint and attack the Zealots, the rest of the defender's units "wake up" and they march towards their enemies. This way, the attacker may be able to lure out the defender from an advantageous positions This way, we can see that it would be better for the defender to place his Zealots behind the Dragoons. When the defender's Dragoons find themselves in range of the attacker's units, they attack and the Zealots are still able to reach the chokepoint fast enough in order to achieve a really good formation similar to the one we employed manually. In the infrastructure hunt scenario we could also improve this layout by forming a wider blockade, that would include a few Dragoons.

When we examined the layouts produced by the algorithms which layouts performed better, we observed some form of this clever improvement by all four of them — the produced layouts typically placed the Zealots slightly behind the

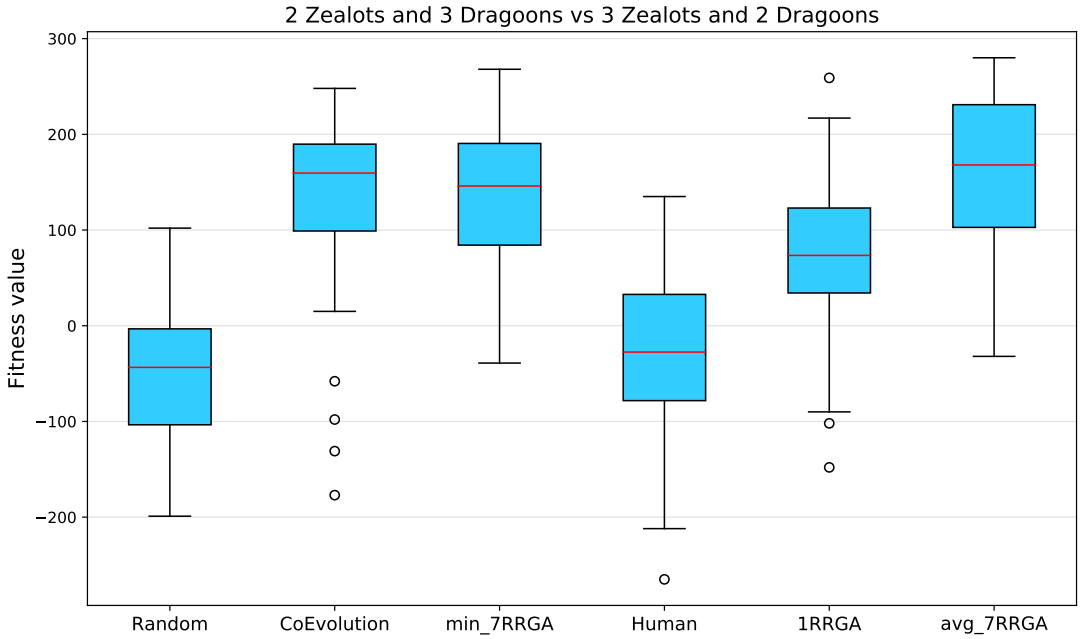

Figure 5.4: Box plot with fitness values from the fitness comparing experiment of individual approaches in experiment with 3 Zealots and 2 Dragoons on Defender's side against 3 Zealots and 2 Dragoons on Attacker's side.

Dragoons and the algorithms that performed best also included the Dragoons in the blockade.

In the rest of our experiments, this pattern repeated. According to our fitness metric, all of our presented approaches were able to perform significantly better than random layouts. In comparison to our manual layouts, we have observed that in each experiment, at least one of our approach was able to either cope with the layout, or even outperform it.

If we compared our approaches among each other, we have observed that our more sophisticated methods (CoEvolution, avg_7RRGA and min_7RRGA) were overall able to outperform the 1RRGA. However, we need to bear in mind that those algorithm also took considerably longer to finish and whether the benefit outweighs the time demands is debatable. Among CoEvolution, avg_7RRGA and min_7RRGA it is hard to assess which approach works the best, but overall, CoEvolution seemed to be the most consistent approach.

Let's also contemplate the visual component of the resulting layouts. If we consider the first simple scenario with 3 Zealots and 10 Zerglings, we have seen that CoEvolution was able to produce layout that looked almost identical to our manual layout, which blocks the chokepoints with Zealots. This is the correct solution that will perform well (almost) regardless of AIs we use in the simulations. This also reflected in the fitness metric — CoEvolution and our manual layout performed the best. However, if we consider more complex situations with, for example, range units on both sides, we have observed that the layouts produced by our algorithms did not seem as compact as our manual layouts and yet they performed better. The reason for the poorer performance of manual layouts are of course our simple AIs. This gives us an interesting observation as we can see how much is the notion of "good" layout affected by the way both AIs play. This

poses a question whether there even is an actual one universally good layout for the situation of chokepoint defense. As we have observed, our AI was not able to leverage good layouts created by our human expert knowledge but rather, it has benefited much more from layouts that were generated specifically for its behavior and specifically against the opponent it was trained.

Conclusion and Future work

In this work, we have addressed the problem of initial positioning of units around chokepoints before battle. We have observed that current state-of-the-art bots could greatly benefit from chokepoint utilization as it could improve their performance in all stages of the game.

We have presented four approaches to deal with this problem that are based on Genetic algorithms, and we have assessed their performance in a set of experiments against two baselines — our manual layout and a random one. All of our four presented approaches proved to be superior when compared to random layouts. In comparison with our manual layouts, the layouts did not visually seem as compact and robust. However, according to our simulations, the found layout was, surprisingly, able to outperform, or at least cope with the human one. Our methods typically outperform the human layout in more complex scenarios with e.g. range units on both sides. The reason is that in these types of scenarios, the way the individual AIs control their units starts to play a significant role. Therefore, a really favorable layout according to human expert knowledge can actually offer quite a poor performance when we take into account the behavior of both AIs.

This indicates that there are no universally good layouts for chokepoint defense that would significantly improve performance of every bot, but the bots would rather benefit much more from layouts specifically generated according to their behavior and the opponent they face. However, our framework and methods may easily be reused in order to achieve this.

Future work

We can see several ways how can one build upon our work.

First, it would be interesting to see the differences in performance when a SC:BW bot employs unit layouts generated by our methods. To achieve the best results, one would need to implement unit control behavior of his bot into SparCraft as well as implement the possible opponent's unit control behavior.

Our sub-system with genetic algorithms may be easily extended in order to allow further testing — one can easily implement additional components of the fitness score or new types of genetic algorithms. There is one particular modification that we think would be worth trying out: it would be interesting to see experimental evaluation of a genetic algorithm that combines CoEvolution with RoundRobin GA. First, we would use CoEvolution in order to generate good layouts for the attacker and then follow with one of the RoundRobing-type algorithms. This way, we could increase the chances of eliminating easily exploitable attacker's layouts that tend to degrade the performance of our algorithms.

Finally, even though we have implemented several abilities and advanced units into SparCraft, it does not contain any AIs that can take advantage of these improvements. Their implementation could further extend the usability of SparCraft in more complex scenarios. Furthermore, one can easily use our ability infrastructure in SparCraft and implement additional abilities.

Bibliography

Henrik Alburg, Filip Brynfors, Florian Minges, Björn Persson Mattsson, and Jakob Svensson. Making and Acting on Predictions in StarCraft: Brood War. Master's thesis, Chalmers University of Technology, University of Gothenburg, June 2014.

Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Building Placement Optimization in Real-Time Strategy Games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014.

Michal Čertický. Implementing a Wall-In Building Placement in StarCraft with Declarative Programming. ArXiv preprint, June 2013. URL `http://arxiv.org/abs/1306.4460`.

David Churchill and Michael Buro. Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft. In *IEEE Conference on Computational Intelligence in Games*, pages 1–8, 2013. doi: 10.1109/cig.2013.6633643.

Holger Danielsiek, Raphael Stuer, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent Moving of Groups in Real-Time Strategy Games. In *IEEE Symposium On Computational Intelligence and Games*, pages 71–78, 2008. doi: 10.1109/cig.2008.5035623.

Antonio Fernández-Ares, Antonio Miguel Mora, J. J. Merelo, Pablo García-Sánchez, and Carlos Fernandes. Optimizing Player Behavior in a Real-Time Strategy Game using Evolutionary Algorithms. In *IEEE Congress on Evolutionary Computation*, pages 2017–2024, 2011. doi: 10.1109/cec.2011.5949863.

Reuel R. Hanks. *Encyclopedia of Geography Terms, Themes, and Concepts*. ABC-CLIO, 2011. ISBN 9781598842951.

John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

Siming Liu, Sushil J. Louis, and Christopher Ballinger. Evolving Effective Micro Behaviors in RTS Game. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, August 2014. doi: 10.1109/cig.2014.6932904.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker. *Science*, 2017. ISSN 0036-8075. doi: 10.1126/science.aam6960.

Jacob Kaae Olesen, Georgios N. Yannakakis, and John Hallam. Real-Time Challenge Balance in an RTS Game using rtNEAT. In *IEEE Symposium On Computational Intelligence and Games*, pages 87–94, December 2008. doi: 10.1109/cig.2008.5035625.

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. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013. doi: 10.1109/tciaig.2013.2286295.

Luke Perkins. Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition. In *Sixth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

Florian Richoux, Alberto Uriarte, and Santiago Ontañón. Walling in Strategy Games via Constraint Optimization. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, January 2016. doi: 10.1038/nature16961.

Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N. Yannakakis. Multiobjective Exploration of the StarCraft Map Space. In *IEEE Conference on Computational Intelligence and Games*, pages 265–272, August 2010. doi: 10.1109/itw.2010.5593346.

Chang Kee Tong, Chin Kim On, Jason Teo, and Aroland MConie Jilui Kiring. Evolving Neural Controllers using GA for Warcraft 3-Real Time Strategy Game. In *Sixth International Conference on Bio-Inspired Computing: Theories and Applications*, pages 15–20, 2011. doi: 10.1109/bic-ta.2011.70.

Jay Young, Fran Smith, Christopher Atkinson, Ken Poyner, and Tom Chothia. SCAIL: An Integrated Starcraft AI System. In *IEEE Conference on Computational Intelligence and Games*, pages 438–445, September 2012. doi: 10.1109/cig.2012.6374188.

List of Figures

List of Abbreviations

AI	Artificial Intelligence
AIIDE	Artificial Intelligence and Interactive Digital Entertainment (StarCraft AI competition)
API	Application Programming Interface
ASP	Answer Set Programming
BWAPI	Brood War API
BWTA	Brood War Terrain Analyzer
CSP	Constraint Satisfaction Problem
DPS	Damage Per Second
EMP	Electro-Magnetic Pulse
GA	Genetic Algorithm
HP	Hit Point
NOKDPS	No Overkill, Damage Per Second
ORTS	Open Real-Time Strategy
RRGA	Round Robin Genetic Algorithm
RTS	Real-Time Strategy
SC	StarCraft
SC:BW	StarCraft: Brood War
SSCAI	Student StarCraft AI
SSCAIT	Student StarCraft AI Tournament

Attachments

CD / online zip attachment structure

- **bin** — folder with precompiled executables of our programs. In order to execute them, please refer to SparCraftDocumentation.pdf and FormationsEvolverDocumentation.pdf
- **Code** — Folder that contains code of our two main projects — SparCraft and FormationsEvolver as well as a few scripts for easier recompilation on Linux
  - SparCraft
  - FormationsEvolver
  - Scripts
- **Libraries** — folder with code for a few libraries we use in our work that are required for recompilation
- **Documentation** — folder with documentation for both of our projects. It contains SparCraftDocumentation.pdf and FormationsEvolverDocumentation.pdf
- **Thesis** — folder with this text and abstracts
  - Thesis.pdf
  - Abstrakt_cz.pdf
  - Abstract_en.pdf