Using Genetic Programming to evolve an AI for StarCraft

Frans Biström / Marcus Håkansson

Summary

This paper is about the possibility to use evolution to make a StarCraft AI better in some areas by using genetic programming. We aimed to use genetic programming to evolve the numbers of squad units, bunkers and turrets, which are an important part of a successful StarCraft AI.

We have built a separate application for handling the evolution. This application runs in parallel with StarCraft and modifies files based on the data recieved from a played game. This is good for safety, since if StarCraft crashes the evolution is just stalled not lost. Our tests ran over the course of a few weeks.

A combination of a relatively small amount of time, for something very time-consuming, and a lack of experience with genetic programming resulted in a small amount of results. The conclusion is that it is possible to improve an StarCraft AI with genetic programming, however it takes a lot of time.

Name1:   Frans Biström
Adress1: Konstapelsgatan 19
e-mail1: **fransbis@gmail.com**

Name2: Marcus Håkansson
Adress2: Polhemsgatan 33 A Karlskrona
e-mail2: **gexnep@gmail.com**

Mentor: Stefan Johansson
ISSN-nummer:

Contents

1. Introduction

This paper assumes that the reader has a good understanding of programming, is introduced into the area of AI-programming and has a basic understanding of how an Real Time Strategy game works.

1.1 Background

In this section We Introduce the reader to the different components that we used for our project. These include StarCraft, BTHAI and Genetic Programming.

1.1.1 StarCraft

StarCraft is a somewhat older Real Time Strategy game (RTS) . The major components in the game consists of resource gathering, base building, army recruitment and battle. Everything during a game session happens in real time, as the genre name suggests, and requires the player to both be good at planning and forming a strategy as well as to be able to quickly react to changes during a game session.

There are two main resources in StarCraft: Minerals and Gas. Minerals are the basic resource that everything you construct will require to varying degrees, they can be picked up by workers without any special requirement other than having a worker. The other resource is gas. This is a more valuable resource, since the more advanced and powerful buildings and units often require a gas cost in addition to minerals. To harvest gas one must build a refinery, or a similar building, on top of a gas vein and then let workers retrieve the gas from the refinery.

In addition to the two main resources there is also a pseudo-resource in the form of supplies. Supplies set the limit of how many units you can have at a time. The limit of this can be increased by building certain buildings, an example is the Terran Supply Depot. The maximum number of supplies a player can have are 200.

In StarCraft there is three warring races/factions in the form of Terran, Protoss and Zerg. Terran consists of humans, Protoss is a high-tech alien race and Zerg are a feral alien race. There are more to these factions but this is how someone would describe them with just seeing them.

In the game there are, generally speaking, three sub-types of units: Infantry, Vehicles, and ships. These terms apply mainly to Terran, but the other factions have corresponding types.

In our project we have mainly focused on Terran to save time and effort.

Fig.0. Picture of StarCraft, Terrans Assaulting a Zerg base.

1.1.2 BTHAI

BTHAI is a bot for StarCraft that has been developed by Johan Hagelbäck at Blekinge Institute of Technology. It is built upon BWAPI (Brood War API) [6] and implements the bot through a multi-agent system that use potential fields for pathfinding [2]. Most of the things of how a bot behaves on a higher level is found in separate text-files: One for squads, one for build-order and one for upgrades. These things are the main focus of our project.

The squad-files are our main target, since a large part of the success rate depends on squad composition. The representation of a squad in the script file looks like this:

```
<start>
Type=Offensive
Name=Squad0
MorphsTo=
Priority=9
ActivePriority=10
OffenseType=Required
<setup>
Unit=Terran_Marine:10
Unit=Terran_Medic:3
Unit=Terran_Siege_Tank_Tank_Mode: 4
<end>
```

Type signifies what strategic purpose a squad has(Offensive, Defensive, Scout etc.) and OffenseType is if the squad is required to be finished before attacking commences. The units of the squad are declared between <setup> and <end>. A more detailed picture of the grammar for the squad files follows on the next page.

What we cannot change from outside BTHAI is the behavior of an individual unit. This is however not something we aim to do with the project.

```
[squad-setup] ::= <start> [EOL]
                Type=[squad-type] [EOL]
                Move=[move-type] [EOL]
                Name=[name] [EOL]
                Priority=[priority] [EOL]
                ActivePriority=[active-priority] [EOL]
                Offense=[offense-type] [EOL]
                <setup> [EOL]
                [unit-list] [EOL]
                <end> [EOL]
[squad-type] ::= Offensive | Defensive |
                Exploration | Support | Rush | Kite |
                ReaverBaseHarass | ChokeHarass |
                ShuttleReaver
[move-type] ::= Ground | Air
[name] ::= Lurker[string] | Devourer[string] |
                [string]
[priority] ::= 1 | 2 | 3 | ... | 1000
[active-priority] ::= 1 | 2 | 3 | ... | 1000
[offense-type] ::= Optional | Required

[unit-list] ::= [unit]
[unit-list] ::= [unit] [unit-list]
[unit] ::= Unit=[unit-type]:[amount] [EOL]
[unit-type] ::= Terran_Battlecruiser |
                Terran_Dropship | ... | Terran_Wraith
                | Protoss_Arbiter | Protoss_Archon
                | ... | Protoss_Zealot |
                Zerg_Defiler | Zerg_Devourer |
                ... | Zerg_Zergling
[amount] ::= 1 | 2 | 3 | ... | 200
```

Fig. 1. BNF grammar for the squads setup text file [2].

1.1.3 Genetic Programming

Genetic programming is a branch of genetic algorithms. It is a method to solve problems with the help of the computer through an analogue of natural selection. The method automatically solves problems which means that you do not have to know the structure of the problem or the optimal solution ahead of time. You only give it basic building blocks of the solution and a way to analyze the generated solution(s). Genetic programming represents its solutions with a tree structure.

The difference between genetic programming and genetic algorithms is that genetic programming represents is solutions with actual program code while genetic algorithms represents its solution with a string of numbers [1], [3], [5].

We use a so called Tournament Selection method that selects GP programs based on the fitness that lets a fixed number of individuals "fight" each other.

The natural selection of the individuals in the population is based of real life selections.

The three most used ones are Elitism, Mutation and Crossover. Elitism is when the genetic program copies the best existing program to the new generation. Mutation is when you replace either a terminal or function with another terminal or you replace an entire sub tree. The last selection method is crossover. Its when two parents forms two new solutions or an offspring [1], [3], [5].

All the selections are made based on the fitness each solution gets. The fitness is calculated with specific information from the performance of the solution(s). One has to write his/her own fitness functions based on the problem that is to be solved. There are no universal fitness functions [1], [3], [5].

1.2 Purpose and goal

The purpose with the project is to get a general understanding of how genetic programming works and how it can be implemented. This we will achieve through studying third-party libraries. One of the main reasons that we use third-party libraries is to save time, which is important due to how long we expect the evolutionary process to be.

With the help of this we hope to form a solution that helps a StarCraft AI to evolve a strategy that is as optimal as possible and that hopefully will win in most cases.

Some of our challenges is how to calculate the fitness of an AI and how we can mutate it. Another issue is that we may get a great AI with an unbeatable strategy our we might get an AI that builds two marines and attacks, i.e. due to how genetic programming works we do not have complete control over how the end result will turn out. We can give our evolution some guidelines, but we need to be light on the touch as otherwise we will end up controlling the process too much.

1.3 Research questions

- Can we make use of genetic programming to successively improve a StarCraft AI?
- Can we evolve an AI that given a certain game setup can become successful and win most of the time?

1.4 Method

Our projects main focus was the implementation of a genetic programming system, that modified the scripts of the BTHAI StarCraft AI, and the evaluation of the AI:s that it resulted in. In order for us to save time we decided to use an existing GP-library.

After some searching on the internet we settled on JGAP (Java Genetic Algorithm Package) [7], that was implemented in java and was fairly easy to set up. The library is very extensive and contains several examples of how to utilize it, and is therefore a good choice for us.

We have made some small modifications to the BTHAI as well. This mostly concern output of the data we need in order to calculate fitness.

2 Experiment design

What we need to do in the project is to get data from StarCraft, use that data to calculate fitness and in turn use that to evolve the AI. We need to read scripts and run them through the GP-application to receive new scripts.

In the experiment we use two applications BTHAI/ StarCraft and our GP-application that is written in java.

2.1 Getting the data

Our first concern was to identify the data we needed to get from StarCraft in order to calculate the fitness of the bot. We ended up using the squads strength divided by their cost. We fetch a lot more data for utility-purposes as well.

The general data we currently receive are the following:

- If the bot has won the game.
- The race of the bot and the race of its opponent.
- The amount of minerals/gas that the bot has gathered.
- Total number of units spawned by the bot.
- Total score of the bot.

The data we receive from squads are the following:

- ID.
- Name.
- Total number of members.
- Number of living members.
- percentage of the squad that is alive (health).
- How much damage the squad has inflicted (strength).
- If the squad has been completely filled at some point (isActive).
- The priority of the squad.
- The amount of gas and minerals that have been spent on the squad(cost).
- What type of squad it is (offensive, defensive etc.).
- If the squad is required for the bot to move into the attack phase (OffenseType).

We have also modified the BTHAI so that it waits for our GP to create a squad-file before starting the game.

Some of the data we receive is not used, but we chose to gather as much as possible to guard against eventual oversights we might have. It also makes it a bit easier for us humans to read the text files.

2.2 File Editing

Our next step was to be able to read and interpret the data we received from StarCraft. The file-reading/writing was never really an issue and was quickly completed.

The main problem in this area was to create a build-order and an upgrade list that met the requirements and complemented the squad-list.

The issue of the build-order was solved by utilizing a tree that connected buildings with their dependencies. The tree can be seen in Fig. 2.

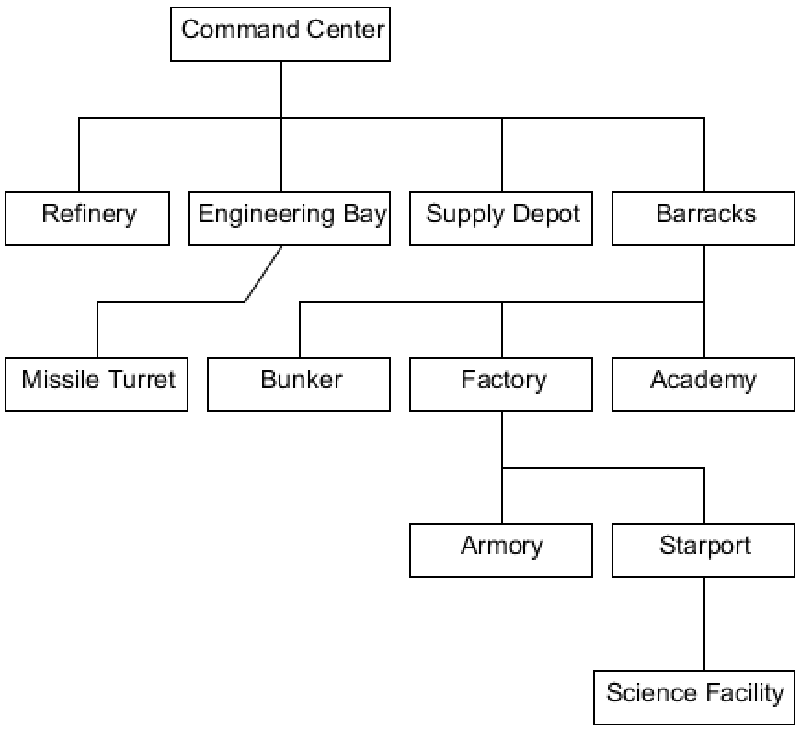

Fig. 2. The relations of the Terran buildings, a leaf/branch requires all buildings that comes before it.

With the help of the tree we could make sure that the generated build-order was a valid one. The build-order is built before the AI-bot starts its match from our GP-program. The quantity of each unit-producing building is decided by the number of units that have to be produced there to fill up the squads. Any possible casualties while building the army are not taken into account. The relations are as follows:

$$\mathbf{Y = X/\ 10}$$

Where **Y** is Number of Barracks and **X** is the amount of infantry with all squads full. If the number exceeds 4 then the number of Barracks is set to 4. The same basic formula applies to the Factories and Starports as well.

The extra unit producing buildings are inserted into the build-order based on where the first occurrence of that building is. The barracks are an exception to this. They are inserted after the first bunker if there is any, otherwise they are inserted after the first barrack. The extra unit producing buildings are all inserted at the same place in the build-order, i.e. all the extra factories are inserted after the first factory, and so on.

The build-order adds one base expansion as well. It is added ⅔ of the way in the build-order, the rest of the expansions are added automatically by BTHAI when resources are running low. The number of Turrets and bunkers are decided by evolution and they are inserted at specific key locations in the build-order.

The upgrades are decided in two different ways. The number of upgrades to armor and weapons are relative to the number of units benefiting from them when all squads are full. The exact relations are as follows:

- If we have any infantry, research infantry weapons and armor to level 1.
  - If we have more than 10 infantry units, research to level 2.
  - If we have 15 or more infantry units, research to level 3.
- If we use Vehicles, research vehicle weapons and plating to level 1.
  - If we have more than 5 vehicles, research to level 2.
  - If we have 10 or more vehicles, research to level 3.
- If we use ships then research ship weapons and plating to level 1.
  - If we have more than 5 ships then research to level 2.
  - if we have 8 or more ships then research to level 3.

The reason for the difference in numbers are due to ships and vehicles being generally more expensive and powerful than infantry which means that they are likely used in lower numbers.

The unit-specific upgrades are made if there are any unit at all that utilize them, meaning that if we use vultures we should research spider-mines as well.

The priority of each individual upgrade are always the same, with the ones we have deemed more essential having a higher priority.

2.3 Calculating fitness

Calculating fitness is an important part of genetic programming and is used to determine how successful an individual is. Essentially it controls if the individual is up for elitism, mutation or crossover. The overall fitness is determined by the bots performance in the game.

The formula we use to determine the fitness is to take **strength /cost** for each squad and add them together and do this for all 10 of the squad setups to get a better and more overall fitness for the used individual. Strength in this case is the total kill-score of all the units in the squad(it is a value that was available to get from a squad in BTHAI when we started our project). Cost is the total mineral + gas cost of all the units that have been recruited into the squad.

Pseudo-code for the fitness calculation can be seen in Fig. 3.

```
foreach (Squad Setup)
    foreach(Squad)
        totalCost = GetCost();
        totalStrength = GetStrength()*100;
        if(totalCost & totalStrength > 0)
        (If totalCost and totalStrength is greater than 0)
            SquadFitness = totalStrength / totalCost;
        else
            SquadFitness = GetStrength();
    TotalFitness += SquadFitness;
return TotalFitness;
```

Fig. 3. Pseudo code for fitness calculation

2.4 Design

As mentioned earlier our main tool for evolving the AI is to edit the text files for the squads, build-order, and upgrades. To do this we utilize a second program that we have written in java. The program's purpose is to take in the data from the finished match, evaluate it, evolve/modify and then write to the three text-files.

A simple UML for our program can be seen in Fig. 4.

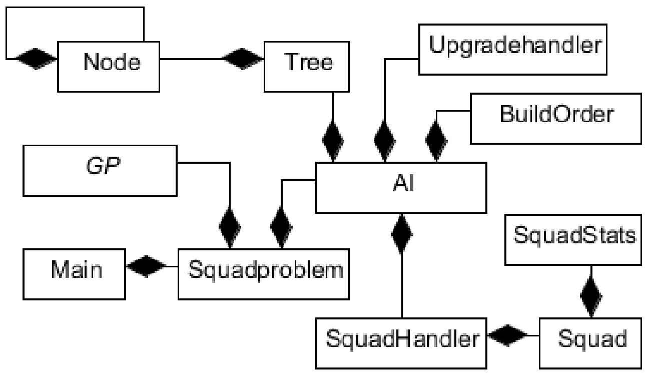

Fig. 4. Simple class diagram for our GP Program

2.5 GP-system

The GP-System evolves the individuals to better modify the squad setups. It uses simple mathematical operations to do the modifications.

2.5.1 Individual and Operations

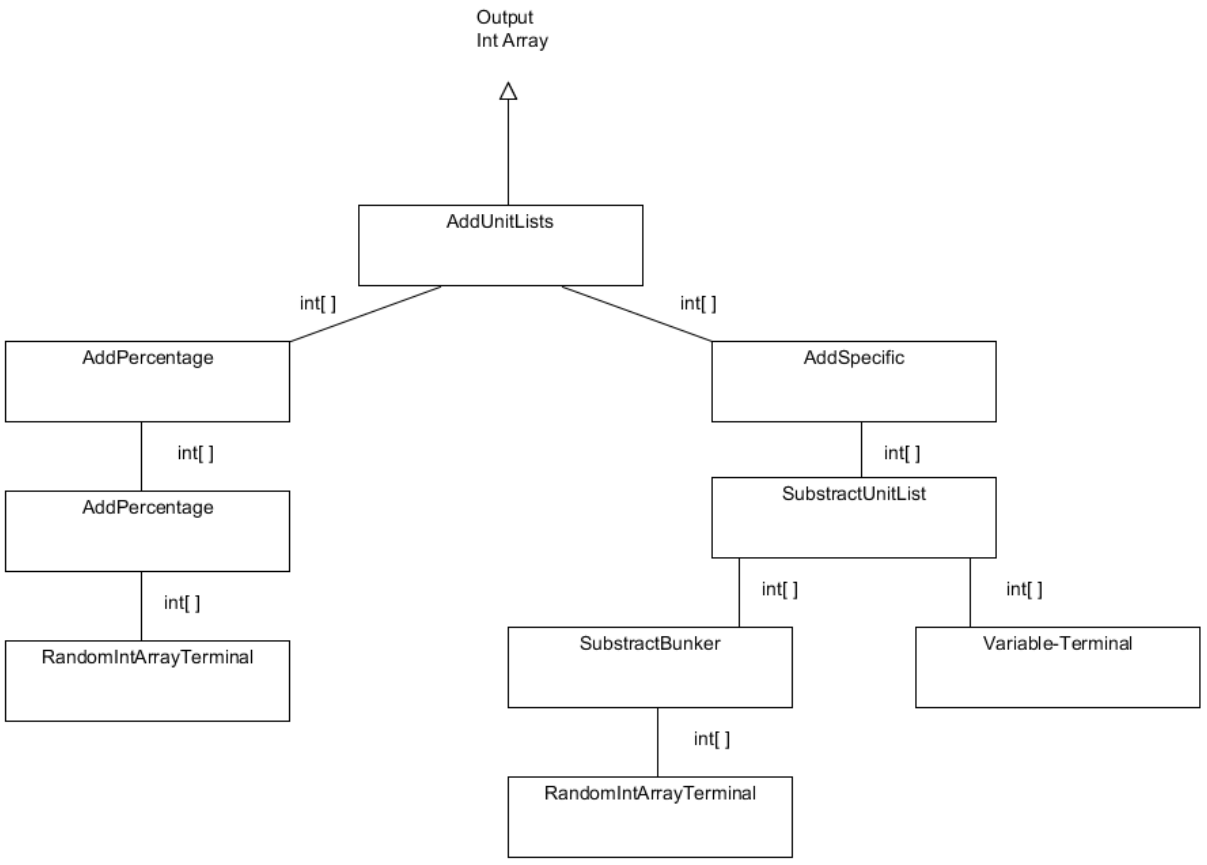

Fig. 5 .This is a representation of an individual.

Each individual modifies the values of the integer array. Fig. 5 illustrates an individual. Each slot in the integer array represents the unit type and the value in the slots represents the amount of units. There is a total of 11 unit types and the integer array has a 13 slots in size. The last two slots are for bunker- and missile turret-frequency. The integer array for each squad is then sent into a Variable-Terminal and is then changed when it goes through different possible operations.

In the system we have initial squad lists, bunker- and missile turret-frequencies. The initial squad-lists are created manually by us. We have tried to keep the initial squad-lists at least somewhat sensible in order to give the evolution a strong start.

The squads are represented as integer arrays (int[]). The slots in the arrays represent the quantity of the different units.

We let different operations change up the units in the squads in a number of different ways. Bunker- and Missile turret-frequency is changeable as well.

All function operators have a return type int[] (integer array) and have at least one input value as int[]. The same sort of operator can be used more than once in a solution. Every function and terminal operator, that have the possibility to mutate, has its own description of how it is going to mutate.

Function Operators: (non-Terminal)

The function operators are one of the building blocks for an individual and they each do a specific task. It is represented as a non-leaf node in the tree structure and has one or more child nodes, these can be both functions and terminals.

The different function operations possible are: AddPercentage, AddSpecific, AddUnitlists, SubstractUnitlists, Add/SubstractBunker and Add/SubstractMissileTurret.

AddPercentage: Adds a percentage between 0.5(-50%) and 1.5(+50%) to either all infantry, all vehicles or all ships. You multiply all the specific units of the target type(infantry, vehicles or ships) with a random value between 0.5 and 1.5.

The mutation is based on a value the GP-system generates. The mutation changes the target unit type that is to be modified to a different one.

a[x] = a[x] * Percentage-value

AddSpecific: Sets a specific value to a unit of a specific unit type. The value can be between 0 and 15. For example, if you have 3 Marines and the specific unit is Marine and the value is 10. The 3 marines is replaced by 10 marines.The mutations work the same as in AddPercentage.

a[x] = value

AddUnitlists: Adds half of the squad list unit count to itself. If you have 2 Marines you will have 3 when the operation return its value. The mutation changes it to SubstractUnitlist.

a[x] = a[x] + (b[x] * 0.5)

SubstractUnitlist: Substract half of the squad list unit count to itself. If you have 2 Marines you will have 1 when the operation return its value. The mutation changes it to AddUnitlist.

a[x] = a[x] - (b[x] * 0.5)

Add/Subtract Bunker/MissleTurret: Randomly adds or subtracts a value between 0 and 10 to/from the current frequency.
The mutation of these operations goes from Add to Subtract and Subtract to Add.

We have a maximum and minimum value for Bunker and Turret frequency. The maximum is set to 10 and the minimum to 0.

Terminal Operators:
Terminal operators are one of the building blocks that represent a value(10, true, 1.03 or in our case an integer array). A terminal can randomly create the value or you can set the value manually. A terminal operator is represented as a leaf node in the tree structure.

The different terminals the GP can use is: Variable, RandomIntArrayTerminal.

Variable-Terminal: This variable is set to the currently used squad from the initiated list of squads. You let each Squad from the squad setup to go through the GP solution program.

RandomIntArrayTerminal: Creates an integer array with random values between 0 and 10. There is a 50% chance if a slot in the array should be filled with a value.

All operations are cloneable. All values in the operations are saved and cloned.

Operation Probabilities
The probability that a crossover operation is chosen during evolution is around 90%.
The probability that a reproduction operation is chosen during evolution is around 10%.
The probability that a node is mutated during growing of a program is 10%.
Percentage of the population that will be filled with new individuals during evolution are around 30%.
In crossover: If a random value between 0.0 and 1.0 is less than 0.9, then it will choose a function over a terminal. For more detail read Section 2.5.3.

2.5.2 Overview of the evolution process
For a general understanding of this look at appendix B.

2.5.3 Crossover Operations
The method of crossing two chromosomes is branch-typing.

It is when you choose a random point in the first chromosome(of the two you have chosen). It has a certain probability to be a function with a rest probability it will be a terminal.

The second chromosome is chosen the same way as the first but with a extra requirement. That the chosen node in the first chromosome is the same type as in the second chromosome.

If a good point in the second chromosome isn’t found, then the crossover won’t happen. If a resulting chromosome's depth is larger than the maximum crossover depth then that chromosome is simply copied from the original rather than crossed.

2.6 The experiment

The experiment is set up so that we have an instance of StarCraft running matches indefinitely (or until a crash occurs). In parallel with this we are running our java-program, “SquadGP”, that receives data after each match and use this to evolve and modify squads, buildings and upgrades.

The AI is mainly fighting against Blizzard’s own Terran AI, we have chosen to do this due partly to time-constraints and partly to maintain as much stability as possible with StarCraft, since it seems that it is more prone to crashing when it faces off against other factions. Incidentally this may also help us evolve an AI faster, but with a strategy that is more limited in where it will be successful.

We will save the important data after each match in order for us to better present the result of our experiment.

Most of the GP-related work, other than our defined operators, is handled by the library JGAP and we should not have to do much other than keep an eye on the process and analyze the changes once in awhile.

For the configuration of the Genetic Programming system we use a fitness evaluator that sees a greater value as a better fitness.

We also have a max population size of 50. Which is the total number of solution programs, more solutions equals more memory and time.

We have put a maximum on the depth of the solutions at 12. The solution programs is a tree structure with the top node as depth zero and its children at 1 and so on.

The matches are always Terran vs. Terran, 1 on 1 and on the map (4)Lost Temple.scm. Which is a 4 player map with natural expansions and 2 islands in the top left and bottom right corners. The map is illustrated in Fig. 6.

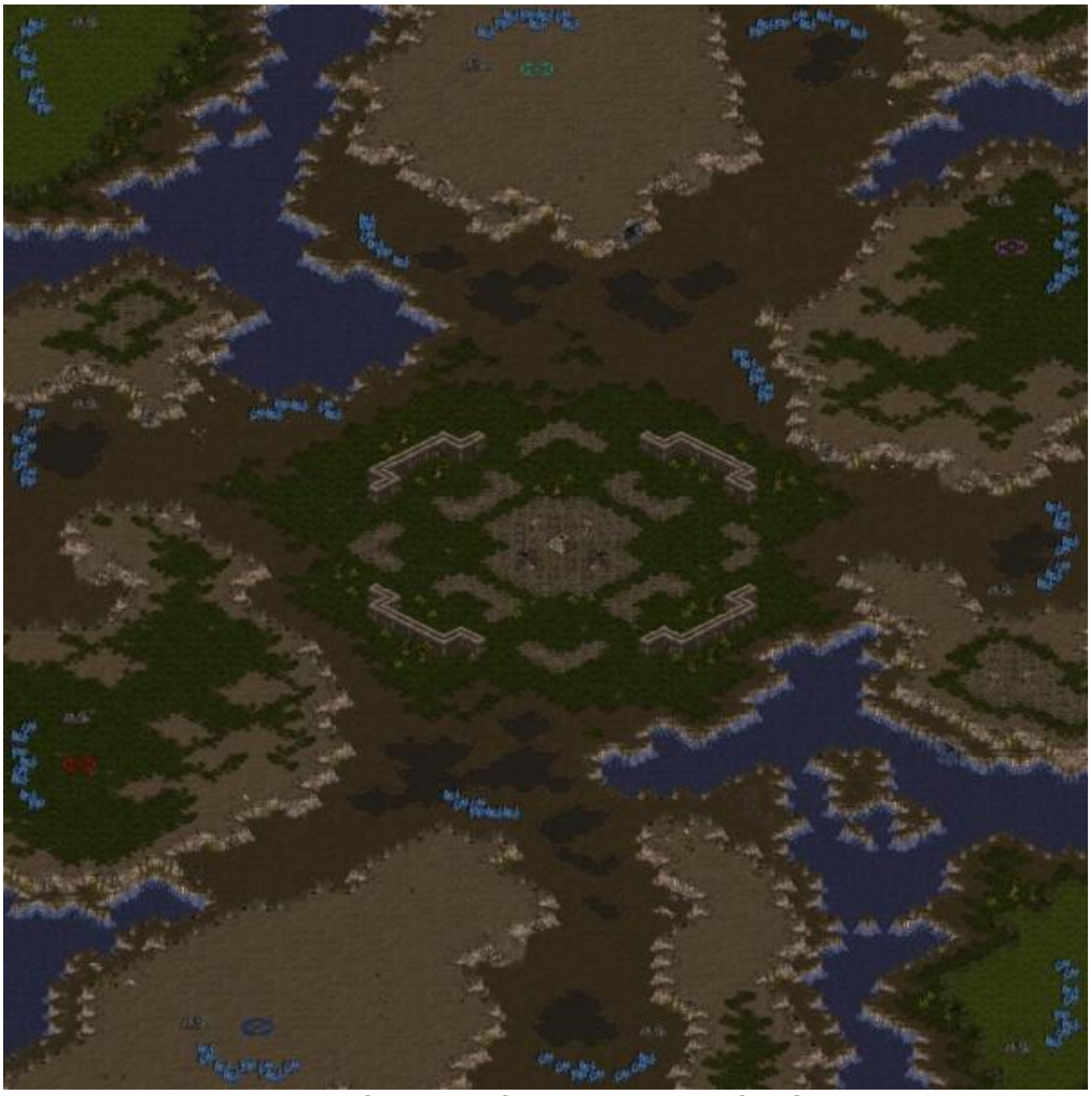

Fig. 6. The map [4], the experiment was played on.

3 Results

We started with a base squad setup population that had 10 different squad setups with different combinations of units in them. Each squad setup have 6 squads, with a variable amount of units in them. To see the setup of the original Squad Setups see appendix A.

Some of the setups were very similar, this to see if the same setups could end up at different places.

For Fig.7, 10, 13; The values on the x-axis is the number of generations.
The values on the y-axis is the number of total units of all the squads that generation.
Each line represents a squad setup, and how it is modified during the process.

For Fig. 8, 11, 14; The values on the x-axis is the number of generations.
The values on the y-axis is the total fitness for that generation.

For Fig. 9, 12, 15; The values on the x-axis is the number of generations.
The values on the y-axis is the total of wins that generation. The colors represents in the graph represents the setup that won.

3.1 First trial

Our first proper iteration of the experiment was off to a good start, with a mix of wins and losses for the individuals.

However, as the generations went on the “add percentage”-operator was over-used which led to the individuals using barely any units. Some bots did not even have any units at all. This led us to make the decision to abort the evolution and restart the process.

Some key data from the first trial follows:

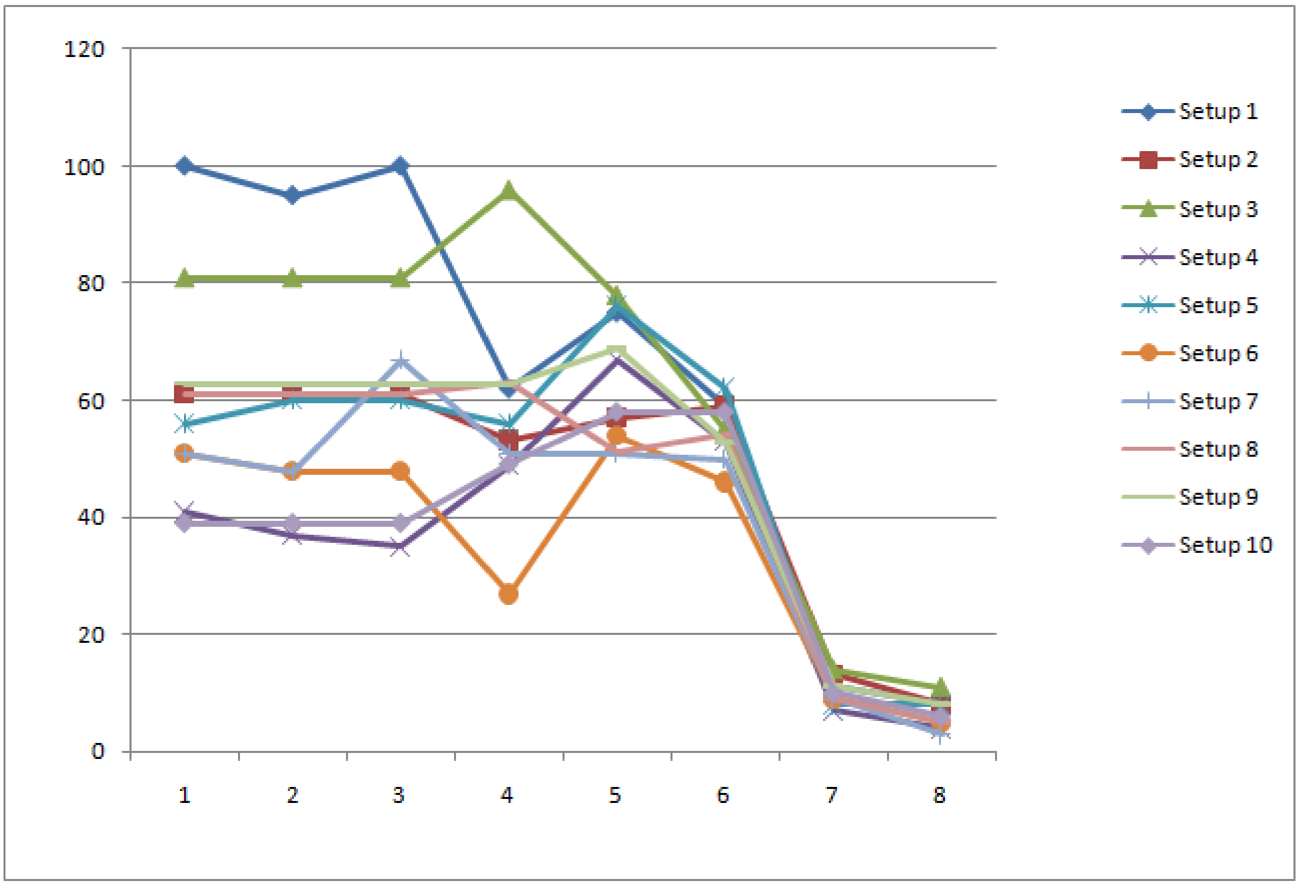

Fig. 7. This chart shows the change in number of units when all squads are full.

As you see in Fig. 7. The numbers are given for each setup. As is shown, the number of units almost instantly shrunk to extremely low and inefficient numbers.

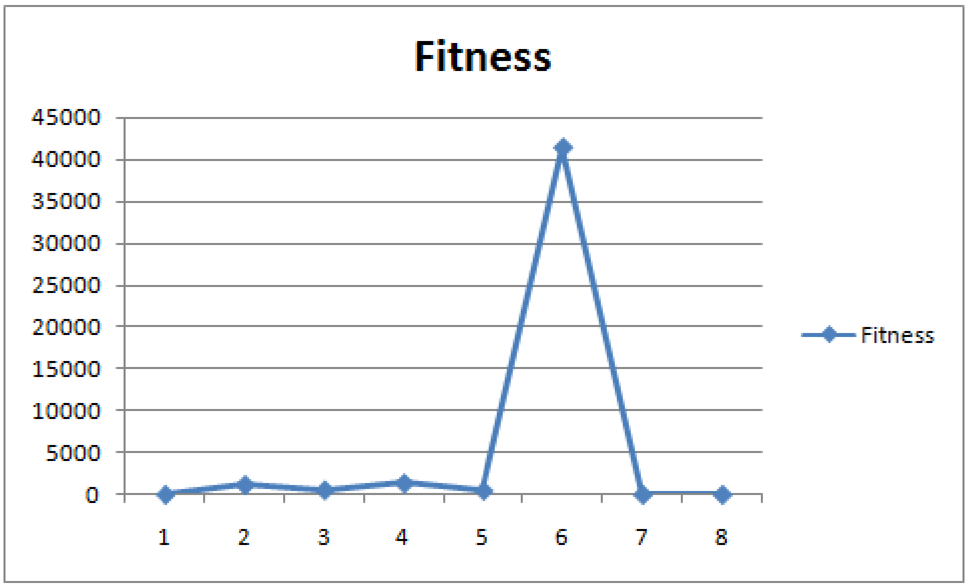

Fig. 8. This figure shows the fitness change for each generation.

As seen in Fig. 8. The fitness of 43000 in generation 6 is likely due to a bug that led to the extreme decrease in the number of total units.

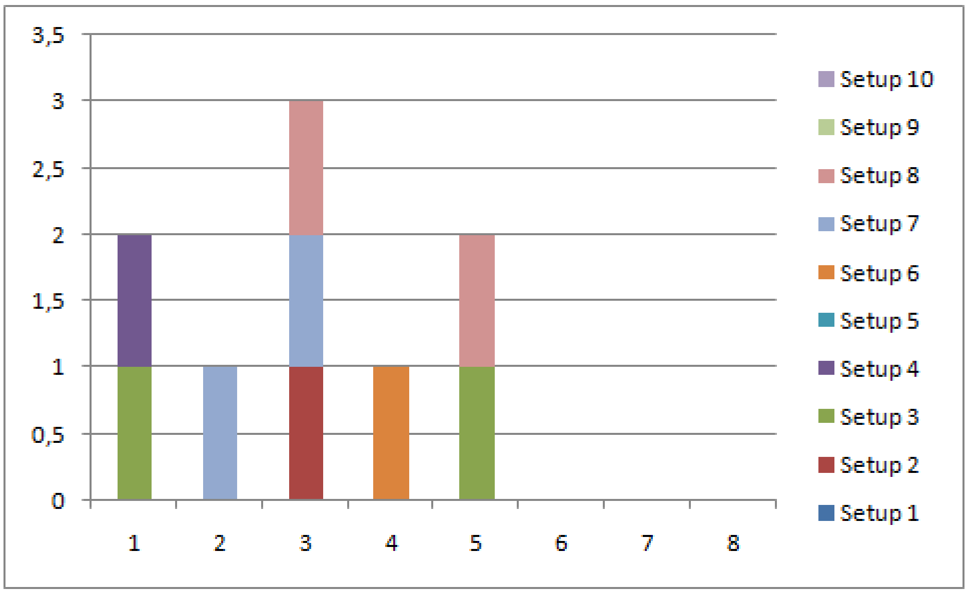

Fig. 9. This chart shows if a specific AI has won its match in that particular generation.

As seen in Fig. 9. Setup 8, 3 and 7 won twice over the course of 8 generations. Setup 6 and 4 won once. After generation 5 no one won.

3.2 Second trial

For our second trial we did not make any modifications to our code, we just restarted the process and hoped for a better progress/ result.

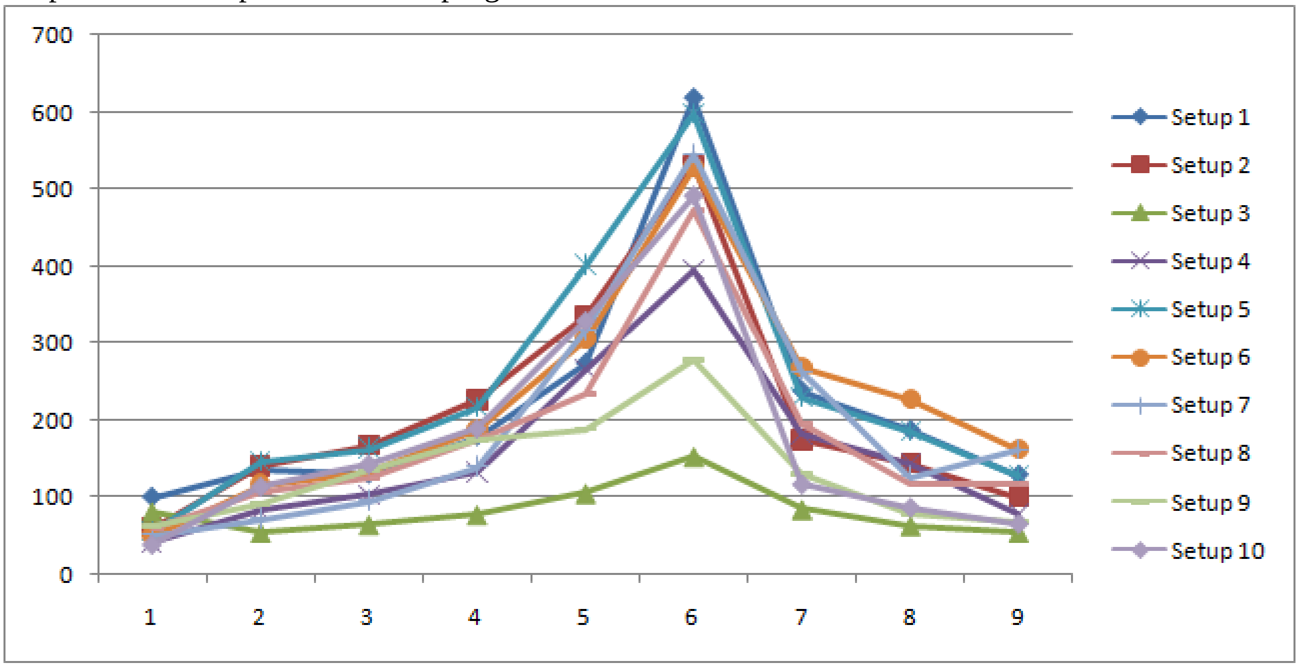

Fig. 10. This chart shows the change in number of units when all squads are full.

As seen in Fig. 10. The numbers are given for each setup. As is shown, the number of units almost instantly grow to extremely high and inefficient numbers.

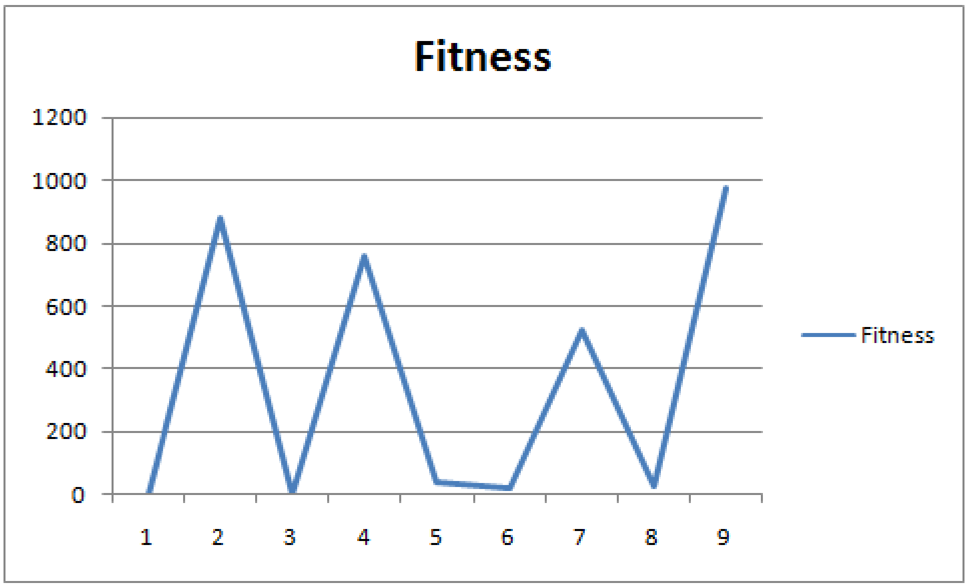

Fig. 11. This figure shows the fitness change for each generation.

As seen in Fig. 11. Their is no consistent values so it is hard to analyze the graph.

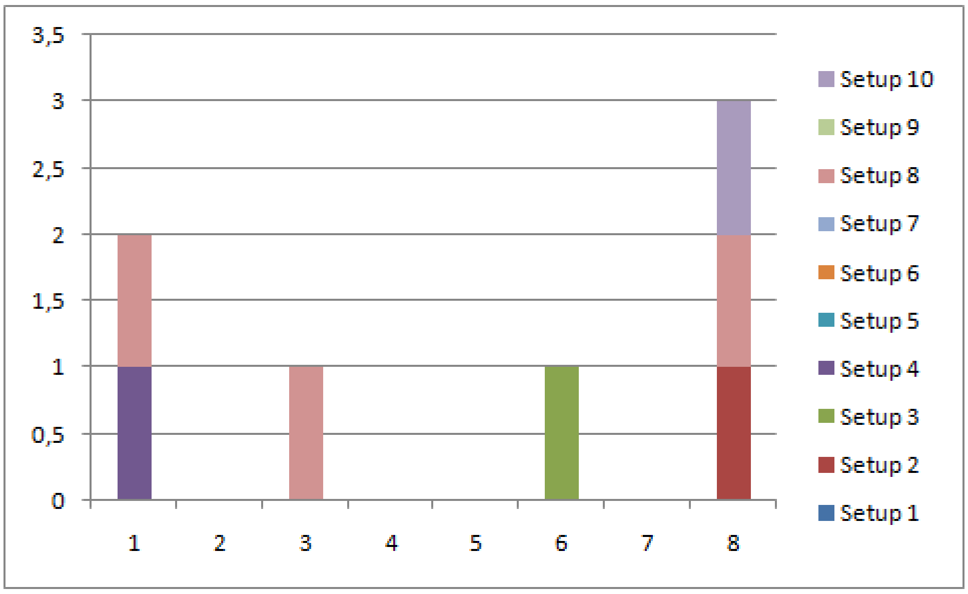

Fig. 12. This chart shows if a specific AI has won its match in that particular generation.

As seen in Fig. 12. Setup 8 won three times. Setup 4, 10, 2 and 3 won once.

3.3 Third trial

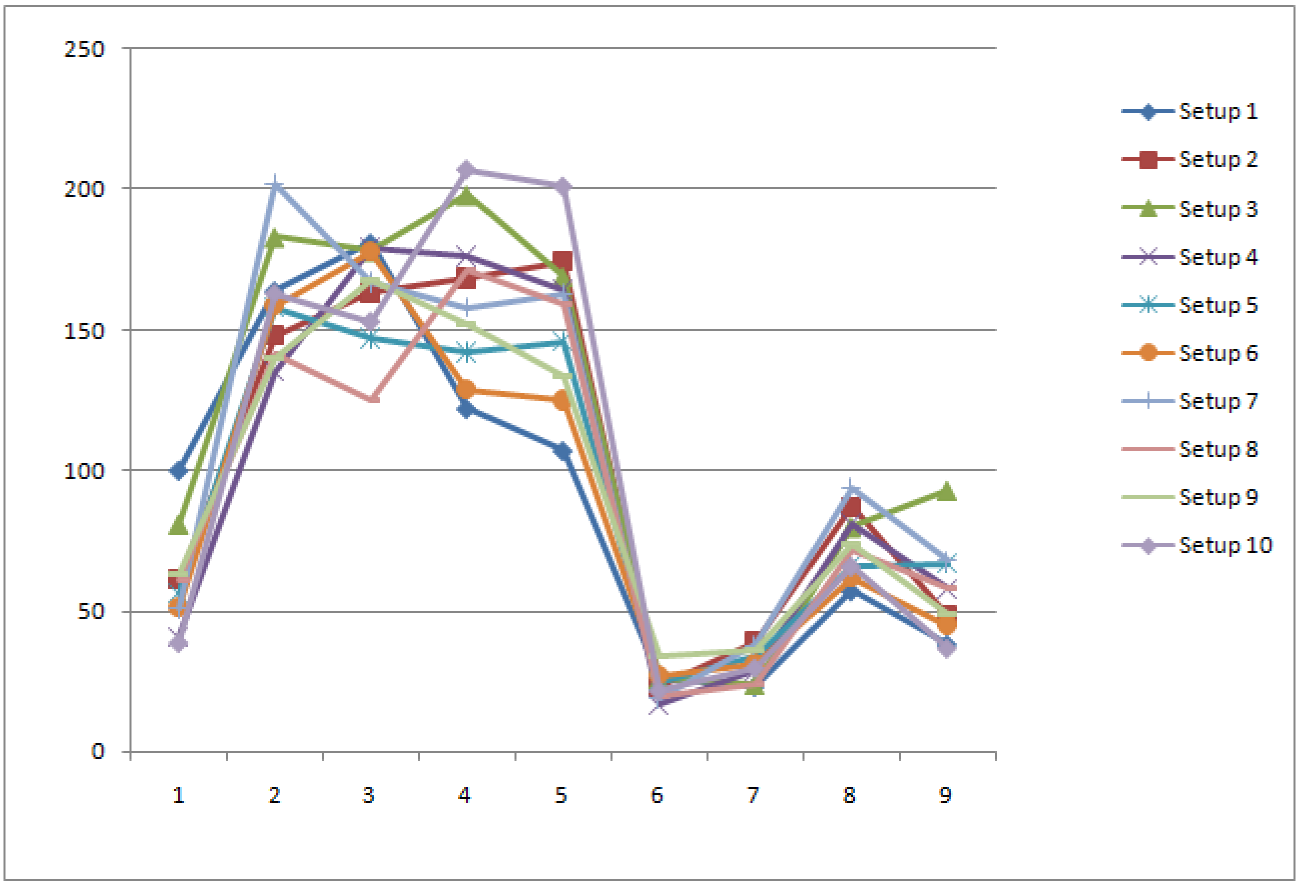

Fig. 13. This chart shows the change in number of units when all squads are full.

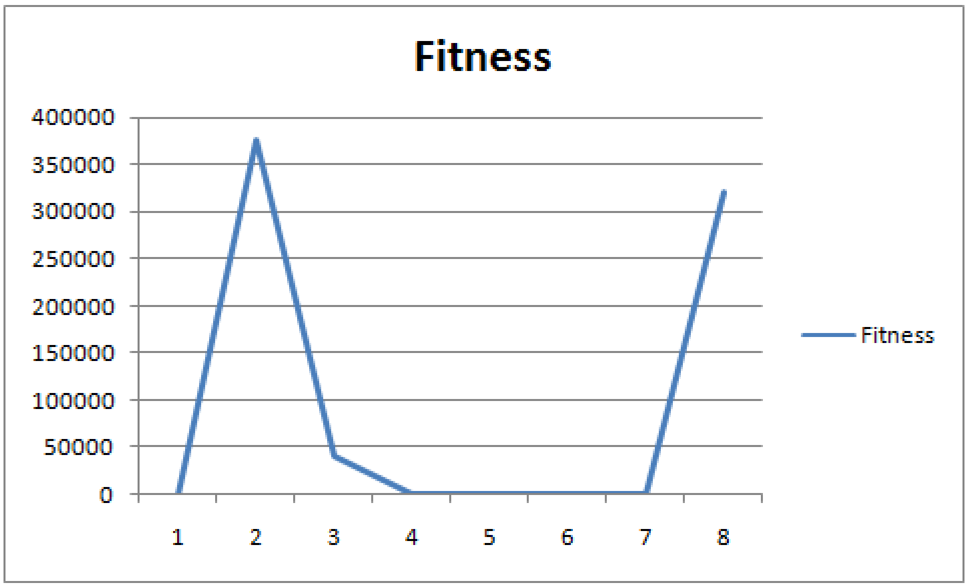

Fig. 14. This figure shows the fitness change for each generation.

As seen in Fig. 14. Their is no consistent values so it is hard to analyze the graph.

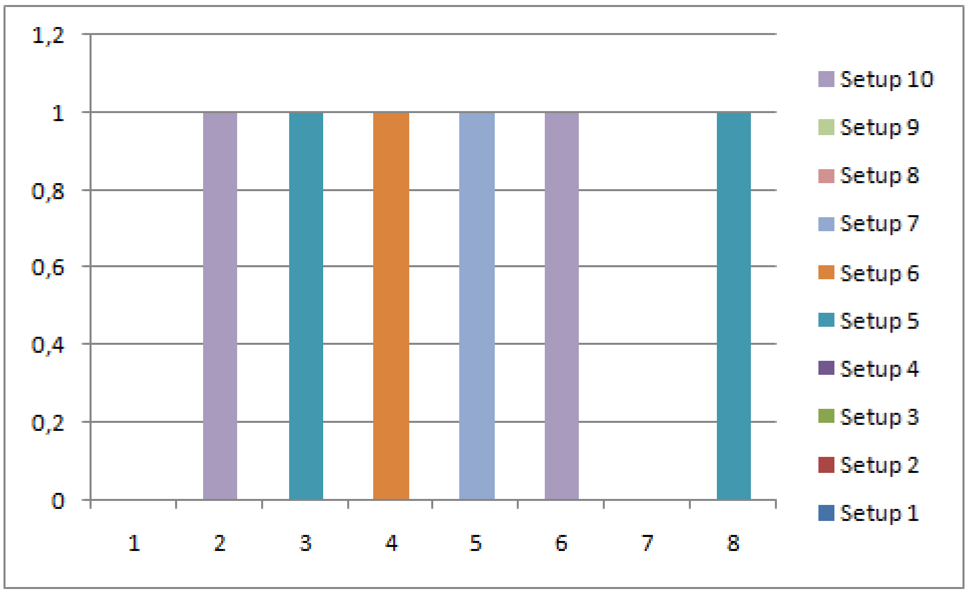

Fig. 15. This chart shows if a specific AI has won its match in that particular generation.

As seen in Fig. 15. Setup 10 and 5 won twice. Setup 6 and 7 won once.

3.4 Issues / bugs

There has been some bugs and crashes related to StarCraft. Here we list the ones we found that affected us:

- The getHealthPcr() in the squad-class sometimes divided by zero, this was resolved to returning 0 when the variable in question is 0.
- We also found out with too many units in the squads the AI don't have enough resources to build more Command Centers. Therefore the SCV:s stop gathering of minerals and you lose the match.
- If the build in AI builds an expansion on an island it is hard and often impossible for the AI to find it, resulting in a stalemate where no one wins.

There were some seemingly random crashes as well that we could not trace. This slowed down the evolving process and limited the number of generations since we couldn't have StarCraft under constant surveillance.

4 Discussion

In this section we first discuss and analyze the results from the three trials separately, and after that we have a more general discussion around the project as a whole.

4.1 First trial

In our first trial we can see that the amount of units was about to stabilize around generation 5, the generation before something strange happened. However, before that happened the evolution seemed to move in a general direction and eventually was about to stabilize around 40-65 units. The whole evolution was not really able to kick off, due to the strange “bug” and us aborting the process.

Our fitness-values was never really able to show how well it performed. This is mostly due to us aborting the process because of the “bug”.

One of the reasons we might not get as good result as we want is the shortage of time. For a great result we would probably need at least a 3-digit number of generations, perhaps even more. Unfortunately this is something we do not have time for.

4.2 Second trial

In this trial we can see a fast growing unit count for each setup. With peak values for each setup at generation 6. At the peak we can see ridiculous values stretching up over 600 units in some cases.

After generation 6 the unit count took a nosedive down to better values but not good. If we had continued the evolution after this point we would maybe have landed on more suitable values eventually. We had to end this trial because we found vital flaws that we had to change in our GP program.

Once more the lack of time is one of the key factors for the limited results.

4.3 Third trial

The result of the third trial is that initially the values are higher to eventually shrink and get a more reasonable total. If the setup had won more matches we could probably have said that a, relatively, lower number of units is more efficient than large numbers. What supports this is the fact that massing a large army takes a lot of time and resources that may not be available. However, generally speaking, the setups did not win a lot of matches.

That is partly due to the fact that our number of generations were very low in all trials, but there are also factors that we do not have any control over. Some of those things are mentioned in section 3.1, but also include the fact that the enemy quite often is fighting dirty with nukes. Something that was a big part of the reason for the losses, for the second trial at least, was that the number of total units far exceeded the maximum number of supplies!

4.4 Final discussion

To conclude the discussion we can say the following: We probably could have evolved a viable StarCraft AI, if we had been given more time. This is partly to be able to fix the flaws in our implementation, but evolving something like this takes a very large amount of time on its own.

The reason we evolved between Terran and Terran was because the StarCraft AI didn't crash as often as against other factions in the game when playing versus Terran. We also wanted the evolution to be more stable and easier to maintain.

Because of this a squad setup could possibly end up being perfect for fighting Terran but against against Protoss, or Zerg, may be completely worthless. If we had evolved against all factions we might have gotten bad fitness values because of that. If we had evolved against all factions a squad setup that may be good against Terran might mutated into something else that is completely worthless, and we would have lost a good candidate.

When we look back on the experiment we realize that we could have structured the individuals a bit differently to give more room to what attributes that could have been mutated in the evolution. For example our number of squads is locked to six, allowing this to be variable might have given us a bit different or more interesting result.

The squad type is locked to a specific type in our solution. This is also something that one might want to be able to mutate in order to get a more varied result.

These are two things that one that attempts this may want to consider in order to give the evolution a wider spread. We, unfortunately, did not think of these things while implementing our program, which is something that might have impaired our results.

One possible reason our implementation is as flawed as it is, is likely due to our inexperience working with Genetic Programming. However, if we today retried the project we would hopefully be more successful. This is partly due to having learned a lot from this unsuccessful attempt. We already have a few good ideas of how we could improve it, as we already have discussed.

So to answer our questions:

- Can we make use of genetic programming to successively improve a StarCraft AI?

Yes, we believe that we can [8]. But due to lack of primarily time, we did not achieve much in the form of results. However with the right technique and enough time it is fully possible.

- Can we evolve an AI that given a certain game setup can become successful and win most of the time?

It might be possible [8], but we could not achieve any meaningful results on this matter. Again, evolution takes time, unfortunately time that we did not have.

5 Conclusions

The conclusion of the trials is that with more time and a bit of better understanding of genetic programming. The trials could have shown better values and maybe we could have seen some better improvements of the StarCraft AI.

The short answer to our research questions are that both a most likely possible, but we lack the time to achieve anything meaningful.

6 Future Work

In future work we would want to change the GP structure so that each part of the individual representes a part of the bot configuration. That the mutations change the values of the individual and by that get a simpler and better structure with less limitations.

To do this project with a wider time frame so more time can be spent on getting the knowledge of genetic programming and evolve longer.

Adding more information to evolve the bot gets better on a greater level.
To let the bot go against other opponents than Terran so the bot can get unique squads and setups for specific opponents.

To speed up the evolutionary process perhaps looking into how one could parallelize the process might be worthwhile. Since we have to actually run the matches, which is time consuming on its own, this would hopefully speed things up.

7 References

1. Schwab, Brian.(2009),AI game engine programming second edition, Charles river Media.
   ISBN -10: 1-58450-572-9.
2. Hagelbäck, Johan. (2012). “Multi-agent potential field based architectures for real-time strategy game bots”. Blekinge Tekniska högskola.
3. J. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.
4. Lost Temple, Map information, **http://wiki.teamliquid.net/starcraft/Lost_Temple**
5. Genetic Programming Source, Evolutionary Computing at GeneticProgramming.us, **http://www.geneticprogramming.us/What_is_Genetic_Programming.html**
6. BWAPI, An API for interacting with Starcraft: Brood War, **http://code.google.com/p/bwapi/**
7. JGAP, Java Genetic Algorithms Package, **http://jgap.sourceforge.net/**
8. Genetic Programming/Algorithms Possible, **http://lbrandy.com/blog/2010/11/using-genetic-algorithms-to-find-starcraft-2-build-orders/**

Appendix

Appendix A

here we list the base population in the form of tables. Each setup is represented as a list of squads.

Setup 1	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**5**	**15**	**5**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad1	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**
Squad2	**5**	**5**	**5**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**2**	**0**	**5**
Squad4	**10**		**3**	**0**	**3**	**0**	**6**	**0**	**0**	**0**	**0**
Squad5	**0**	**0**	**0**	**0**	**7**	**0**	**0**	**0**	**0**	**0**	**0**

A setup using a mixed combination of units that aim for superiority in numbers.

Setup 2	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**10**	**0**	**3**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
Squad2	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**0**	**0**	**5**
Squad4	**10**	**0**	**3**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad5	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

A basic setup using a mixed combination of units

setup 3	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**

Squad0	**30**	**0**	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad1	**1**	**0**	**0**	**0**	**0**	**0**		**0**	**0**	**0**	**0**
Squad2	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**10**	**0**	**0**	**0**	**0**	**0**		**0**	**0**	**0**	**0**
Squad4	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad5	**10**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

This setup uses marines in large numbers.

Setup 4	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**10**	**0**	**3**	**0**	**0**	**4**	**4**	**0**	**0**	**0**	**0**
Squad1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
Squad2	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**0**
Squad4	**0**	**0**	**0**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad5	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**0**	**0**

Another well-balanced setup.

Setup 5	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**0**	**0**	**0**	**0**	**3**	**5**	**7**	**0**	**0**	**0**	**0**
Squad1	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad2	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**3**	**5**	**7**	**0**	**0**	**0**	**0**
Squad4	**0**	**0**	**0**	**0**	**3**	**5**	**7**	**0**	**0**	**0**	**0**
Squad5	**0**	**0**	**0**	**0**	**7**	**0**	**0**	**0**	**0**	**0**	**0**

A setup that is heavy on the vehicle side

Setup 6	**Marin**	**Fireb**	**Medic**	**Ghost**	**Vultu**	**Golia**	**S.T.***	**Wrait**	**S.V.***	**B.C.***	**Valky**

	es	ats			re	th		h			rie
Squad0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**	**0**	**0**	**10**
Squad1	**8**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**
Squad2	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**0**	**0**	**5**
Squad4	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**	**0**
Squad5	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

This setup focuses on ships for air superiority.

Setup 7	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**	**0**	**0**	**10**
Squad1	**8**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**
Squad2	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**0**	**0**	**5**
Squad4	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**	**0**
Squad5	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

This one is identical to the last one, this way we might find that identical base-individuals may turn out differently.

Setup 8	**Marines**	**Firebats**	**Medic**	**Ghost**	**Vulture**	**Goliath**	**S.T.***	**Wraith**	**S.V.***	**B.C.***	**Valkyrie**
Squad0	**10**	**0**	**3**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
Squad2	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Squad3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**0**	**0**	**5**
Squad4	**10**	**0**	**3**	**0**	**0**	**0**	**4**	**0**	**0**	**0**	**0**
Squad5	**0**	**0**	**0**	**0**	**7**	**0**	**0**	**0**	**0**	**0**	**0**

This is also an individual with a good mix of units.

Setup 9	Marines	Firebats	Medic	Ghost	Vulture	Goliath	S.T.*	Wraith	S.V.*	B.C.*	Valkyrie
Squad0	10	4	3	0	0	0	0	0	0	0	0
Squad1	0	0	0	0	0	0	0	0	1	0	0
Squad2	0	0	0	0	0	0	0	0	1	0	0
Squad3	10	4	3	0	0	0	0	0	0	0	0
Squad4	10	4	3	0	0	0	0	0	0	0	0
Squad5		10	0	0	0	0	0	0	0	0	0

A setup that uses a balanced mix of infantry.

Setup 10	Marines	Firebats	Medic	Ghost	Vulture	Goliath	S.T.*	Wraith	S.V.*	B.C.*	Valkyrie
Squad0	0	0	0	0	12	0	0	0	0	0	0
Squad1	0	0	0	0	0	0	0	0	1	0	0
Squad2	0	0	0	0	0	0	0	0	1	0	0
Squad3	0	0	0	0	12	0	0	0	0	0	0
Squad4	0		0	0	3	0	0	0	0	0	0
Squad5	10	0	0	0	0	0	0	0	0	0	0

This individual focus on using mainly vultures, a gigantic minefield is a possible result.Its weakness is ships.

Appendix B

Evolve Stages	Description
0:Start Evolve	Overall: Evolve the population by one generation
	Step 1: Select two GP programs to be crossed over
	Step 2: Cross over two GP programs
	Step 3: Randomly (instead of crossing over) do reproduction of GP programs
	Step 4: Randomly add new GP programs
1: Population Create	Create a single GP program
2: Program Create	Create a single GP program
	1: Instantiates the GP program
	2: Fills the GP program with nodes
3: Program Grow	Initializes a GP program with nodes
	1: Construct as many nodes (=chromosomes) as needed
	1a: Instantiate a single chromosome
	2: For all nodes
	2a: Grow the node by populating it with sub nodes
	3: If cache enabled and program found in cache: read fitness; otherwise: calculate fitness of program
4: Program Chromosome Grow	Initializes a node within a program
	1: If custom initialization strategy is registered, use it to preset the first node
	2: Recursively build the node and its subtrees
	3: Recalculate the depths of the nodes in the tree
5: Program Chromosome Grow/Full Node	Create a tree of nodes
	1: Select an appropriate node
	2: Validate node, if validator registered
	3: Randomly mutate node if node supports this
	4: If node is marked as unique, remove type of node from list of available functions
	5: Clone node and add it to the tree of nodes
	6: Call self recursively (goto step 1 in this section) if nodes are missing
0: Back in evolve	5: Set evolved population as current