Online Procedural Content Generation of Starcraft Maps

Master's Thesis

Jakob Melnyk, jmel@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

June 1, 2015

Contents

1 Introduction

This report is the result of our Master's Thesis at the IT-University of Copenhagen, spring 2015.

The goal of the thesis was to apply both constrained novelty search and multiobjective evolutionary algorithms to do online procedural content generation (PCG) of maps for Starcraft, analyze the quality of the generated maps and compare the trade-offs required in order for each algorithms to generate maps fast enough to b e used in online PCG.

NOTE: The order is totally fucked. I have just been writing whatever came to mind.

2 Decisions about map generation

As the main focus of our thesis was on generating balanced maps, we spent a lot of time on figuring out how, exactly, we wanted to go about generating the maps. We had decided on using Constrained Novelty Search and Multi-objective Evolutionary Algorithms, but no other specifics than that.

We discussed having each algorithm generate everything (heightmap and features) but we had concerns about the runtime of the algorithms if they had to constantly change the terrain, especially on larger maps.

In order to cut down on the time spent changing the terrain, we chose to generate the base heightmap initially and then let the two algorithms take care of placing various features on the map, and in some cases make minor changes to the terrain.

We considered a number of possible options for generating the terrain (fractals, agents and noise), but in the end we decided to use cellular automatas. A Starcraft map is, as mentioned in section 3, represented as a grid which is exactly what most cellular automatas work with. Furthermore, the rules - and therefore results - of cellular automatas are easy to change which allowed us to try a lot of different settings in little time in order to figure out what rules we needed.

3 Map Representation

A map in Starcraft 2 is represented as a map of tiles, where a tile can have have a a height (impassable for ground troops, and 3 heights that ground troops can traverse), cliffs between heightlevels, ramps that connect heightlevels, buildings, resources and Xel'Naga towers (provides vision while a unit is nearby). The map can, of course, also contain units but they are irrelevant for our purposes. The only units a basic map contains are the initial 5 (**6?**) workers that always start next to one's starting base, so we can ignore units in our generation.

We did not have direct access to the map representation used in Starcraft, so we had to create our own map representation. As we wanted to use evolutionary algorithms (a subtype of genetic algorithms), we decided to split our representation into two parts: A genotype and a phenotype.

3.1 Genotype

The genotype contains a list of the items the map should contain and their position on the map. The position uses a a radial system, where the midpoint is the middle of the map. Each item has an angle and a distance. The angle determines the angle from the middle of the map (with 0^o being

right and going counterclockwise (**Check if correct**)). The distance is represented as a percentage of the distance from the middle of the map in a straight line to where the angle will hit the edge of the map.

There were other genotype representations we considered, but decided against for various reasons:

- **Direct representation** - A direct grid representation would be very easy to convert to a phenotype, but it is very close to being a phenotype already which defeats the purpose of having a genotype. It is also more time-consuming to mutate compared to the radial one, as we would have to create a copy of the entire grid every time we mutate.
- **Coordinate representation** - Why did we not choose this one? (I honestly cannot remember)
- **Seed representation** - A seed representation would have been the most simplistic type of representation we could have used, but would also have been the most difficult to code for. What part of the seed controls what, how do we interpret it, how do we mutate it and how do we ensure it does not end up producing completely impossible feature placements?

3.2 Phenotype

Where the genotype represents the genetics that make up an object, the phenotype represents the actual object. As mentioned about, a map is made up of heightlevels and various other things.

For our genotype, we decided to use two 2D arrays to represent the map. One array contains the heightlevels, ramps and cliffs, and the other contains "items" in the map, such as bases, resources, and Xel'Naga towers. We split it into two because that allowed us to place both a height and an item on the same tile in an easy way.

The phenotype has a few functionalities that are used for creating the map: Smoothing the heightmap that has been generated through the cellular automata and placing cliffs in the heightmap.

The smoothing works by iterating over every single tile and checking the neighbouring tiles (using the Moore/extended Moore neighbourhood). If the tile does not fit in, it is changed to something that fits better. The changes are made in a clone of the heightmap, so changes do not affect subsequent tiles until next iteration. This smoothing is repeated over a number of generations, at which point the clone is saved as the actual heightmap in the phenotype.

Cliff placement is a simple procedure. We again iterate over all tiles on the map and check its neighbouring tiles (using the Von Neumann neighbourhood). If any of the neighbouring tiles are of a lower heightlevel than the current tile, the neighbour tile is transformed into a cliff.

The phenotype is also used when we create a visual representation of the map. The visual representation is created in two steps: First we draw the heightmap as a bitmap, where each heightlevel has a different tile icon. After that, every item is drawn on top of the already-existing bitmap, in order to avoid items being overwritten by heightlevels. When the item have been drawn, the map is saved as a .png file.

4 Cellular Automata

4.1 What it is

A cellular automata is a form for model can simulate artificial life and is often used in complexity studies. Cellular automatas are often used in computational tasks (e.g. procedual contant generation), mathematics and biology.

A cellular automata consists of a grid of cells of any size and a set of rules. The grid evolves over a number of iterations(generations?) by applying the rules to tiles that fulfill the requirement for a given rule. This result in a semi-controlled evolution that is able to create very varied results.

Our cellular automata is used to generate the initial heightmap of the phenotype. This map is smoothed out, but forms the basis of the search as it ultimately is what is used to check if a map is feasible or not.

The cellular automata is initialized by randomly seeding the map with the different height levels through the use of random number generation. The default odds are the following, but can be changed by by the user:

- **Height2**: 0.0 - 0.3
- **Height1**: 0.3 - 0.6
- **Height0**: 0.6 - 1.0

The CA starts at the top of the list and works its way down until it reaches a heightlevel where the random number fits into the range. Then it moves on to the next tile and repeats the process.

When we create maps, we only generate half the map and then turn/mirror it onto the empty map in order to save time. We do something similar in the CA, where we seed half the map plus 10% of the map's height/width (whichever is relevant for the half). Each iteration works on the same area.

Had we only worked on exactly half the map, we would have had issues where the tiles in the middle row of the map would have "empty" neighbours. This would mean that the middle row of tiles would behave in unintended ways, which in turn would mean that the middle of the map would look weird.

4.2 Rules

Write about rules here when they have been finalized.

5 Map Fitness

For both our algorithms, we needed some way to determine whether a map was good or not. The novelty search needs some way of evaluating the maps it finds, in order to actually chose a map at the end of a search. The evolutionary algorithm needs some way (often called a fitness function) to

work properly in the first place. Without one, there is no way for the evolution to select candidates from (and for) any generation.

The evolutionary algorithm was the algorithm that we considered most when figuring out what was important for the fitness function. During evolution a lot of maps are generated and evaluated, so we needed the fitness function to be fast. Spending even one whole second on evaluating a map would drastically reduce the number of iterations our evolution could run without being too slow (ref to part about speed).

While speed was important, a poor fitness function would be at least as bad as a slow one, as our evolution would create poor maps in order to satisfy the fitness function. Our goal was therefore to create a fitness evaluation that was relatively fast but also covered what we felt was important in a StarCraft map.

There are three main ways of creating a fitness function (Togelius2010Multiobjective): *interactive, simulation-based* and *direct*. An interactive fitness function requires humans to give feedback, which is then used to determine whether a map is good or not. A simulation-based fitness function runs one or more simulations of the game on the map and the result of these simulations are used to judge the map. A direct fitness function purely looks at the phenotype and runs calculations on various parameters in order to determine the fitness.

In order to create an interactive fitness function, we would not only need humans willing to look at maps, we would also go against the intention of the project. So that was not an option. For a simulation-based fitness function, we would need to write an AI to play the game. Writing the AI itself would take a long time, it would most likely not be very good and even if we managed to create a good AI, each simulation would be slow due to the nature of the game.

We chose a direct fitness function as it was the only one possible. Even had the others been possible, we would still have chosen a direct one due to the speed compared to the others.

In the fitness function, we used two main techniques to calculate fitness: Counting of features and pathfinding between points. For the pathfinding we used Jump Point Search (**reference https://harablog.wordpress.com/2011/09/07/jump-point-search/ + http://gamedevelopment.tutsplus.co to-speed-up-a-pathfinding-with-the-jump-point-search-algorithm–gamedev-5818**) over A* (**discuss both**).

Probably not done. Remember direct distance vs. ground distance

5.1 Parts of the fitness function

Our fitness function consists of 11 parts. They are the things we consider the most important when evaluating a StarCraft map. Every part is calculated individually, normalized (between an optimal and worst-case value that we have determined) and then multiplied by a significance value based on how important we consider the part to be.s

Open space around a start base

Open space around the start base is needed to place buildings in order to train units. The open space is calculated by selecting the middle point of a start base and Breadth-First Search to find all traversable tiles no further than 10 tiles from the start base.

The height level the base is located at

The higher the start base is positioned, the easier it is to defend. From analyzing the competitive maps, it is also clear that the start bases are always at the highest level in the map (some maps only have 2 height levels)[1]. This value is calculated by comparing the hight the start bases are located at to the highest level in the map.

The path between the two start bases

The distance from one start base to another heavily influences which strategies that can be utilized. It should be short enough that rushing[2] is possible, but long enough that rushing is not the only possible winning strategy. The fitness value is the distance of the path between the two start bases normalized between $(0.7 * mapWidth) + (0.7 * mapHeight)$ ([3]) and 0.

If there is no path, the map is not a good one (as many units travel on the ground) and it receives such a huge penalty to its fitness score that it will not recover.

How many times a new height is reached on the path between the start bases

Defending against an attack from high ground offers a sizable advantage. It is therefore important that there is some variation in height when traveling between the two start bases. The fitness is calculated by counting how many times a new height levels is reached on the path between the two bases.

The number of choke points on the path

Choke points allow a defender to funnel an attacker's units and prevent the attacker from swarming him. They are important in defenses (and sometimes during attacks), as they can effectively reduce the opponents strength for a while. The fitness value is the number of those points[4] that are found on the path between the two start bases.

The distance to the natural expansion

A natural expansions is the closest base to a start base. The distance to it should not be long, as players want to be able to get to it quick and be able to defend it and their start base well. The fitness value is the distance of the path from the start base to the nearest other base.

If there is no ground path to the natural expansion, a player cannot expand to it by normal means which means the map is bad. It will receive a huge penalty to its fitness score and will not be able to recover.

The distance to the non-natural expansions

The non-natural expansions should be spread well around the map. If they are grouped too much, it allows a player to defend all of them at once. Expanding to a new base should, apart from with the natural expansion, mean that it is not that safe. The fitness is calculated by finding the distance of the shortest path to each expansion and normalizing the distance depending on what number of expansion it is.

[1] Need analysis somewhere.
[2] need reference
[3] remember default value
[4] size of choke points

The number of expansions available

Expansions are necessary for players to continue building their economy. There should be enough expansions that players will not run out of resources too fast, but also enough that they will not have to fight over every expansions because there are too few in the map. The fitness value is the number of expansions divided by the number of start bases.

The placement of Xel'Naga towers

Xel'Naga towers provide unobstructed vision of an area in a radius of 22 tiles centered on the tower when a player has a unit net to the tower. They should offer a view of the main path between the start bases in order to be a valuable asset early on. The fitness is the number of tiles of the path between the start bases a Xel'Naga tower covers.

How open the start bases are

How open a base is determines how many directions it can be approached from. A start base should not be completely open, as it would be too difficult to defend properly. The fitness here consists of two parts: How many tiles in an area around the start base that are not blocked, and how many of the compass' directions (north, north-east, etc.) that are open in a direct line.

How open the expansions are

Expansion openess works the same as start base openess.