Online Procedural Content Generation of Starcraft Maps

Master's Thesis

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

June 1, 2015

Contents

0.1 Introduction 2
0.2 Decisions about map generation 2
0.3 Map Representation 2
0.4 Cellular Automata 4
0.5 Map Fitness 4

1 Introduction **8**

2 Related Work **9**

3 Goals **10**
3.1 Map Balance 10
3.2 Direct Representation 10
3.3 Optimal Trade-offs 10

4 Methodology **11**
4.1 Choice of Algorithms 11
4.2 Representation Data Structure 11
4.3 Converting Representations To Maps 13
4.4 Constrained Novelty Search 14
4.5 Initialising Evolution With Novel Individuals 15

5 Results **16**

6 Future Work **17**

7 Conclusion **18**

0.1 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.

0.2 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 0.3, 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.

0.3 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.

0.3.1 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?

0.3.2 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.

0.4 Cellular Automata

0.4.1 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.

0.4.2 Rules

Write about rules here when they have been finalized.

0.5 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

0.5.1 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.

1 Introduction

2 Related Work

3 Goals

3.1 Map Balance

3.2 Direct Representation

3.3 Optimal Trade-offs

4 Methodology

4.1 Choice of Algorithms

In order to reach our goal of creating balanced maps, we decided to take a search-based approach to map generation. The method of choice for performing search-based procedural content generation is for the most part evolutionary computation[1](genetic search). Evolutionary computation seeks to optimize the fitness (quality) of solutions in order to find good solutions. The quality of a StarCraft map consists of multiple parameters, such as distance between bases, safety of resources, number of choke points, etc. Because it is not always desirable to add the fitness of these parameters together into a single fitness function value, multi-objective evolutionary algorithms are often used in order to find the optimal compromises between the different parameters. In contrast, the novelty search paradigm seeks to optimize novelty (divergence from past discoveries), which leads to many, often radically, different solutions in the search space. Because it is not reliant on a fitness function, novelty search is effective when dealing with problems where creating an effective fitness function is difficult, such as when the problem is deceptive or difficult to quantify.

Both evolutionary computation and novelty search faces some significant challenges when the search space is constrained[2], such as when generating StarCraft maps. Many solutions are infeasible due to not fulfilling one or more of the strict constraints that good StarCraft maps impose, e.g. a ground path between starting bases. In evolutionary computing, a common method for dealing with constraints is to use penalty scores, while *minimal criteria novelty search* is an extension of novelty search that can handle constraints.

As part of this thesis, we have chosen to implement a standard genetic algorithm, a multi-objective evolutionary algorithm, and constrained novelty search in order to test and compare their runtime, and the quality and novelty of the maps they produce.

4.2 Representation Data Structure

Consider using genotype/phenotype to reduce confusion

It is generally inefficient to perform search on a direct representation of the content that is being generated. There are two important principles to keep in mind when choosing a representation: dimensionality and locality. In most cases, a more direct representation will allow the search algorithm to be more precise in its search, but will also increase the size (dimension) of the search space. This increase in dimension will (in general) make it harder to any specific solution. Locality refers to the correlation between the size of changes in the chosen representation and the content that it represents. It is often desirable to have high locality, meaning that small changes in representation will also result in a small change in the content[3]. In order to reduce the search problem to its core, it is important choose a representation that offers a good compromise between high locality and size of the search space.

In the search algorithms, a solution (StarCraft map) is represented as a two-part data structure.

The first part is a base map generated by our cellular automata**??**, which contains only different height levels, cliffs, and impassable terrain. The second part is a set of *map points* which represent positions where different map elements will be placed on one half of the map. Because only the top half of the map is being generated and then mirrored, the size of the search space is reduced from quadratic to linear in the number of possible tile arrangements. In addition, it means a number of assumptions can be made when checking constraints, e.g. there is no need to check if there is exactly the same number of expansions accessible for each player.

The distinction between map layout and map elements makes it possible to separate the search space into a two-layered search space. The base maps generated by the cellular automata make up the first layer. The second layer consists of all the possible combinations of map points. The layering of search spaces means that the size of the search space depends on the number of base maps that are being searched. Thus if the cellular automata generates base maps that has high potential for being good StarCraft maps, the evolutionary and novelty searches can focus on the placement and amount of map elements on the map. With this approach, it is less likely that the search encounters the problems usually seen when dealing with very large search spaces. However, it also means that the same set of map points are likely to produce very different results when combined with different base maps (see figure **??**).

insert picture of the same set of map points placed onto different base maps

4.2.1 Map Points

A map point in the representation is a combining of a distance, an angle, and a type. The distance is a number between zero and one that is relative to the size of the map that the point will be placed on, where zero is the centre of the map and one is edge of the map. The angle is a number between zero and 180 where zero is east, 90 is north, and 180 is west. The type of the map point determines which map element should be placed at the position. Figure **??** shows a Xel'Naga tower placed at 45 degrees at 0.5 distance on a size 64 by 64 map.

insert picture of a xel'naga tower placed at 45 degree, 0.5 distance

As long as the same base map is being searched, any small change to the degree or distance of a map point will result in small change to the generated map[1]. In contrast, changes to the type of a map point or changes to the amount of map points in a representation has potential for large changes in the generated map. This is mainly relevant for bases, as the number of bases and the distance to them from the start base has a large impact on the quality of the map.

The different types of map points are the following: **StartBase** (a starting base), **Base** (a regular base with regular minerals), **GoldBase** (a regular base with high-value minerals), **XelNagaTower** (a Xel'Naga tower that grants vision), **Ramp** (a ramp between height levels), and **Destructible-Rocks** (destructible rocks that blocks buildings and pathing).

4.2.2 Initialization & Variation

When creating the initial population used for searches, a number of map points with random distance and angles are generated for each individual in the population. The number of map points

[1]With the exception of ramps, as they are placed using a different method than the other map points. This is discussed further in section 4.3

generated is dependent on a set of constraints, e.g. exactly one starting base point and between one and four normal bases are generated. This increases the likelihood of a solution being feasible, since the number of map points in an individual in the initial population never fall below or go above the constraints of each type of map element.

As previously noted, the search space is separated into two layers. A cellular automata is used to generate a base map (a point in the first layer) and the search algorithms traverse the second layer using the map points. In order for this traversal to happen, we apply mutation (a unary variation operator) to the set of map points in order to produce a different one. When a solution is mutated, each map point has a chance of being mutated, producing a new map point with slightly modified angle and/or distance. The new set of map points consist of the map points that were not mutated and the newly produced map points from mutation. From this new set, there is a chance that a random map point is added and a chance that a map point is removed, but without going outside the constraints. Once this process is complete, a set of map points have been mutated.

4.3 Converting Representations To Maps

Converting an individual representation into a StarCraft map requires placing the map points of the representation on the base map that is connected to the representation. The conversion process consists of iterating over the set of map points and attempting to place each map point onto the base map. Depending on a number of factors, placement may not be possible, and so another attempt is made at a slightly displaced location, e.g. one tile directly to the west. If placement is still not successful, or if angle and/or distance values of the map point are outside their constraints, a map point will be registered as *not placed*. Any map point that is registered as having not been placed does not count for fitness or feasibility calculations (but does count for novelty calculation). When the entire set of map points have been iterated over, the completely map is created by mirroring the top half, placing and smoothing cliffs, and finally mirroring the top half again[2].

A map point may fail to be placed due to either an area being occupied by another map point (prevents overlapping map points, e.g. bases) or another placement constraint preventing placement (e.g. ramps cannot be placed if the section of cliff is not wide enough). If placement is possible, the map point is placed by flattening the area around the map point, so that the entire area is the same height as the centre tile of the map point. The area is then marked as *occupied*, such that no other map point can overlap with the placed map point. The exceptions to this method are the destructible rocks and ramps.

A **StartBase** map point flattens and occupies a 24x24 area, places a 5x5 starting base in the centre of that area, and places blue minerals and gas geysers around the starting base in a specific pattern. **StartBase** map points are always placed first, as they are required for the map to be feasible. A **Base** flattens and occupies a 16x16 area, places a base marker (used as an indicator on the map) slightly to the south-west of the centre of the area, and places blue minerals and gas geysers in the same pattern as the start base. A **GoldBase** is the same as a regular base, except the minerals are gold instead of blue. A **XelNagaTower** flattens and occupies a 4x4 area, in which the tower itself is placed. **DestructibleRocks** do not occupy or flatten an area, but can be placed on already occupied areas. However, they cannot be placed on cliffs, impassable terrain, start bases, Xel'Naga

[2]This second mirroring happens due to cliff placement often creating a symmetric line through the middle.

towers, minerals, or gas. Upon placement, they are randomly chosen to be either 2x2, 4x4, or 6x6 tiles large. **Ramp** placement is handled slightly different from the other map points. Because it is quite unlikely that a cliff will be at the exact spot calculated from the angle and distance of the map point (and locating the nearest cliff is inefficient), ramp positions are chosen using a hash function of the angle and distance to select a position from a set of known cliff positions in the map.

4.4 Constrained Novelty Search

While objective evolutionary algorithms seek to optimize one or more objectives in order to reach an optimal solution, novelty search seeks to optimize diversity and novelty. Novelty search is a recent algorithm that evolves like any evolutionary algorithm, but instead of basing the survivor selection on objective fitness, novelty search selects survivors based on their novelty, thus diversifying the population[2]. A novel archive keeps track of novel solutions from each generation, ensuring that solutions are novel both compared to current and historic behaviour. The main benefit of novelty search is that it avoids the deception and local optima problems that is frequently experienced in objective optimization approaches[4].

Constrained novelty search applies constraints to the novelty search algorithm in order to direct evolution towards a desired outcome[2]. The algorithm uses the concept of *two-population novelty search* introduced by Lehman and Stanley [5]. *Two-population novelty search* maintains one population of *feasible* and another for *infeasible* individuals. The two types of individuals are kept separate as it is preferable to avoid comparisons between them[2] as a constrained novelty search will quickly kill off infeasible solutions if they are compared to feasible solutions. This is not desirable behaviour as it is likely that the infeasible solutions can have feasible children when performing novelty searches. Instead, any feasible child of an infeasible solution is migrated to the feasible population and vice versa.

Liapis et al. [2] presents two different versions of constrained novelty search: *Feasible-infeasible novelty search* (FINS) and *feasible-infeasible dual novelty search* (FI2NS). Feasible-infeasible novelty search attempts to maximize the novelty score of feasible individuals and minimizing the distance of infeasible individuals from feasibility. In feasible-infeasible dual novelty search, each population performs novelty search separately and contains their own separate feasible and infeasible populations. We have implemented FINS with the suggested *offspring boost* enhancement.

4.4.1 Distance to Feasibility Measure

The distance to feasibility for the representation used in this thesis is measured in two parts. The first part is to calculate the number of map points that were successfully *placed* during map conversion. If there are too few or too many placed map points of a given type, the distance to feasibility is increased by a penalty value multiplied by the number of items missing and in excess. The second part is to check if there is a path between the start bases; if no ground path is found, the map is not feasible as a StarCraft map (too many strategies would be excluded), and a penalty is applied.

4.4.2 Novelty Measure

The novelty of an individual is the average distance between its k-nearest neighbours in both the novel archive and the current population. To calculate the distance between two solutions A and

B in the context of this thesis, each map point in solution A is compared to every map point in solution B. The absolute difference in angle and distance (distance from centre of map, not novelty) of each of these comparisons are summed to form the distance between the two solutions.

4.5 Initialising Evolution With Novel Individuals

Search algorithms often have an initial population that is randomly initialised in order to ensure diversity in the search. This diversity in the search could also be provided by seeding an evolutionary algorithm with novel solutions found using novelty search. In addition to testing evolution and novelty search separately, two different tests will be performed to test the combination of the two paradigms for improvements in the trade-off between speed, novelty, and quality of the generated maps. The first test will be to seed evolutionary algorithms with the most novel of the solutions found with novelty search, while the second test will seed the evolutionary algorithms with the solutions from the novel archive that have the most fitness.

5 Results

6 Future Work

7 Conclusion

Bibliography

[1] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[2] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Constrained novelty search: A study on game content generation. 2014.

[3] Julian Togelius, Noor Shaker, and Mark J. Nelson. The search-based approach. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[4] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[5] Joel Lehman and Kenneth O Stanley. Revising the evolutionary computation abstraction: minimal criteria novelty search. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 103–110. ACM, 2010.