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 and analyze the quality of the generated maps and compare the trade-offs required in order for each algorithms to generate maps fast enough to be used in online[9] PCG.

1.1 Procedural Content Generation in Games

Games are a widespread form for entertainment and they come in many different forms. One thing they have in common is that the content of a game determines how interesting a player thinks the game is. Thus, if there is not enough content (or enough varied content), a player will grow bored with a game and move on from it.

Game content takes a lot of time and resources to develop, however, so there is a limit to how often a development team can push out new content. This is unfortunate, as many players eventually get bored with the existing content and want something new.

One solution to this problem is to let the game generate new content by itself. This method is called "procedural content generation" (henceforth called PCG) and is used in many games today. There exists a multitude of PCG techniques and they are used to different degrees from game to game. In Diablo 3[1], PCG is used to create weapons and armor (within set parameters) when a player kills enemies and to generate dungeons with new layouts. In Dwarf Fortress[2], PCG is used to create the entire world and most of what happens in it.

needs ending

1.2 StarCraft

StarCraft II[3] is a real-time strategy game developed by Blizzard Entertainment. It is the sequel to the original StarCraft[4] that was released in 1998 and which became a huge success.

StarCraft is set in a military science fiction world, where the player plays the role of a general of one of three races; Protoss, Terran and Zerg. The goal of the game is to build a base, gather resources, raise an army and, in the end, wipe out any opponents that the player is playing against. There is a lot of strategy involved, as each of the three races have access to their own unique buildings and units, with each type of unit having its own strengths and weaknesses. A player must how to predict and counter[5] the opponent's choice of units in order to win. StarCraft becomes even more complex when economy and base management is taken into account.

This level of complexity requires the game to be very well balanced[6], otherwise one race will dominate the other two, making those two races redundant. The balance comes from tuning the

[1] get reference
[2] get reference
[3] See `http://us.battle.net/sc2/en/` and `http://en.wikipedia.org/wiki/StarCraft_II:_Wings_of_Liberty`.
[4] `http://en.wikipedia.org/wiki/StarCraft`
[5] Countering a unit means using a unit of your own that is strong against that unit.
[6] Describe: What is balance?

Figure 1: Left: A Terran base. Right: A Terran fleet assaulting a Protoss base.

stats of unit's (eg. health, armor and damage), tuning the stats of buildings (e.g. health, build time and what the building provides access to) and from the maps the game is played on.

A map in StarCraft is made from two-sided grid of n-by-m tiles (where n and m generally are between 128 and 256) with different features. Some of the most important features are listed here:

- Up to three[7] different **height levels** that can be traversed by ground units.
- **Ramps** that connect height levels and make it possible for ground units to move from one height level to another.
- **Impassable terrain** that only flying units can traverse. It is usually portrayed as a drop off into nothing, lava or water.
- **Start bases** which decide where the players begin the game.
- **Resources** in the form of *minerals* and *gas* that a player needs to gather in order to build buildings and train units. These resources are usually grouped together (eight mineral depots and two gas depots) in such a way that it is easy to set up a base and harvest them.
- **Destructible objects** that block pathing until removed.
- **Xel'Naga towers** that provide unobstructed vision[8] to a player in an area around it while the player has any unit next to the tower.

Balance in maps is achieved by considering the layout of the map and keeping the different units in mind. If the start bases are too close, rushing[9] becomes a dominant strategy. If they are separated only by some cliffs, any unit that can traverse cliffs (e.g. Terran's Reapers, Protoss' Stalkers and

[7]Some maps only utilize two.

[8]"Unobstructed vision" meaning that nothing can block the vision in any way.

[9]Focusing solely on getting a bunch of cheap units out quickly and sending them to destroy the opponent's base before they establish a proper base.

flying units) suddenly become too powerful. How many directions a base can be attacked from is also important, as it determines how easy it is to defend.

The placement of other groups of resources (generally referred to as an *expansion*) also plays a huge role in balance, as they are where players want to establish new bases in order to gather more resources. If multiple expansions are close to a start base, they will be easy to defend and hard to attack, as the main base can easily provide unit for defense. Generally there is one expansion close to the main base that is easy to defend (referred to as the *natural expansion*), but remaining expansions are harder to defend and adds some risk-vs-reward balance. The map *Daybreak* is a good example of a balanced map (see figure 2).

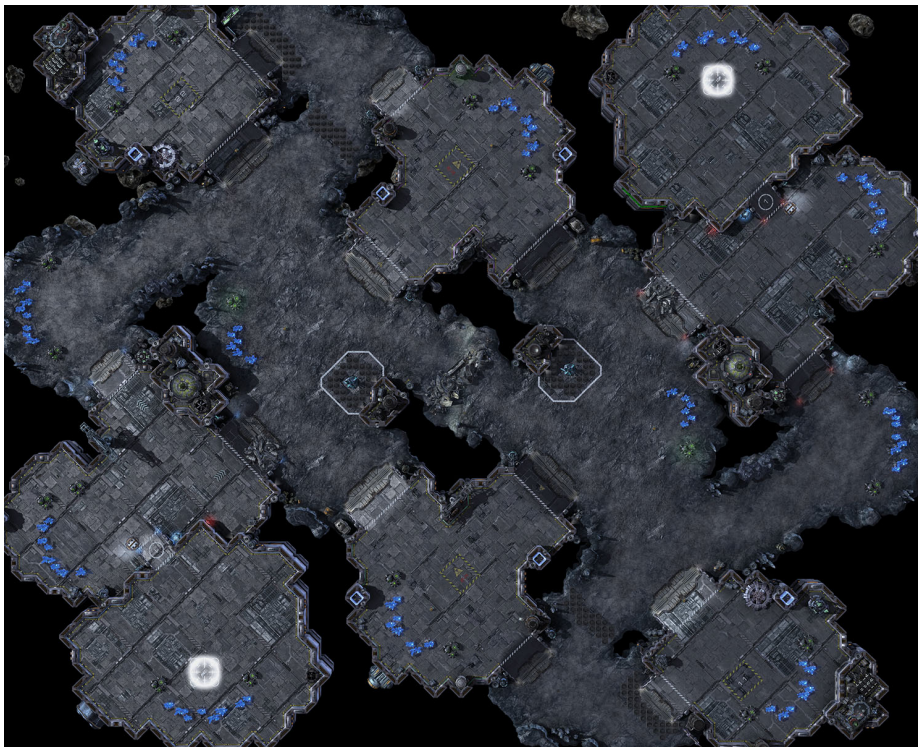

Figure 2: The map "Daybreak". Notice how the start bases (glowing areas) are far from each other and how expansions (the blue minerals) are spread out with only one being close to each start base.

Needs some sort of ending?

1.3 StarCraft and PCG

The StarCraft II Map Editor provides players the tools necessary to create their own maps from the ground-up. These maps can be published to the StarCraft Arcade[10] and, from there, be played by players around the world. This is a good initiative, as it allows players to get a fresh breath of

[10] `http://us.battle.net/arcade/en/`

air, if they have grown bored of the standard maps. The only problem is that new maps require someone to design, create and balance them, which heavily limits the flow of new maps to the pool.

Having a tool that can generate balanced maps would speed this process up and will allow players to have a larger variety of maps to choose from.

Needs something more

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 Map Data Structure

In order to work with maps, we needed a structure that represents the entire map, not just the features in it.

As mentioned above, a map is made up of height-levels and various features the players can use to their advantage. In a map it is possible for a tile to contain a height-level (it always will), a location for a base and destructible rocks that have to be cleared before a base can be built. This situation is the one that had the most impact on the design of our map data structure.

We use three 2-sided arrays to represent the map. One array represents the heightmap itself, height-levels, ramps and cliffs. The second represents features in the map, bases, resources, and Xel'Naga towers. The third represents the placement of destructible rocks.

The initial heightmap is generated by a cellular automata (see section 4.2 on page 6). The map structure is then in charge of smoothing out any irregularities in the heightmap, placing cliffs and smoothing out cliffs. The map structure also has the ability to print the map it represents to a .png file for an easy visual representation.

Heightmap smoothing is done through the use of a cellular automata, using a set of rules created specifically for smoothing out the heightmap. The rules check if a tile fits in with the surrounding tiles and if it does not, it will be changed to the height that occours the most in the surrounding neighbourhood.

Cliff placement is done after the heightmap has been smoothed. The map structure iterate over all tiles on the map and check their 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, unless something will prevent that (gas, minerals, a base, etc). In that case, the tile we are at will become a cliff instead.

Cliff smoothing was implemented in order to avoid situations where remnants of cliffs would be spread around the map, without bordering up to two different height levels (which is where cliffs should be). Such excess cliffs are removed by iterating over all tiles in the heightmap and, if a tile

is a cliff, looking at its neighbours (using the Moore neighbourhood). If there is only one other height-level in its neighbourhood, the cliff is changed to that height-level instead.

After these processes have been run, the map is ready to receive data from the representation structure and be turned into a complete 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 - including destructible rocks - is drawn on top of the already-existing bitmap, in order to avoid items being overwritten by heightlevels. When the items have been drawn, the map is saved as a .png file.

Picture of generated map

4 Cellular Automata

4.1 General Overview

A cellular automata is a form for model that can simulate artificial life and is often used in computational tasks (e.g. procedual contant generation[8]), mathematics and biology.

Cellular automatas are made from a regular grid of cells, where each cell can have one state out of a finite set of states. A cellular automata also contains a set of fixed rules that determines how a cell's state is affected by the cells around it, known as a cell's *neighbourhood*. The rules are applied to all cells in the grid at once[11] in order to create a new generation. New generations are created either until a certain criteria is reached, or until a chosen number of generations have passed.

The number of states in a cellular automata can range from two different states (e.g. *active* and *not-active*) all the up to a practically infinite number. The more states, however, the more complex the cellular automata will be. Every state will need to be linked to at least one rule (otherwise there is no point in having the state) and the more states that exist, the more complex their interactions will become.

Cellular automata rules use *neighbourhoods* to determine what happens to a cell (e.g. if five of the surrounding cells are active, change the state of this cell to active). These neighbourhoods determines which of the surrounding cells whose states are considered with regards to changing the current cell. There are two different neighbourhoods that are used in most cellular automatas; *von Neumann* and *Moore*. *von Neumann* covers the cells horizontally and vertically in either direction from the current cell. *Moore* covers not only horizontally and vertically, but also diagonally. Both neighbourhoods only consider cells one tile away from current cell, but they can be *extended* to cover a larger area. Figure 3 shows the two neighbourhoods along with their extended versions.

4.2 Our Cellular Automata

We use a cellular automata to generate the initial heightmap for the phenotype. In order to save time, the cellular automata only work on the top 60% of the map, which later is mirrored to form a complete map. We work on 60% instead of 50% in order to avoid that the cells at the bottom of the working area are biased by the empty cells below. This still happens with 60%, but as the map is mirrored, the bottom cells will be overwritten and the bias will have no influence.

[11] Not all cells may have a rule that applies, in which case it keeps its state.

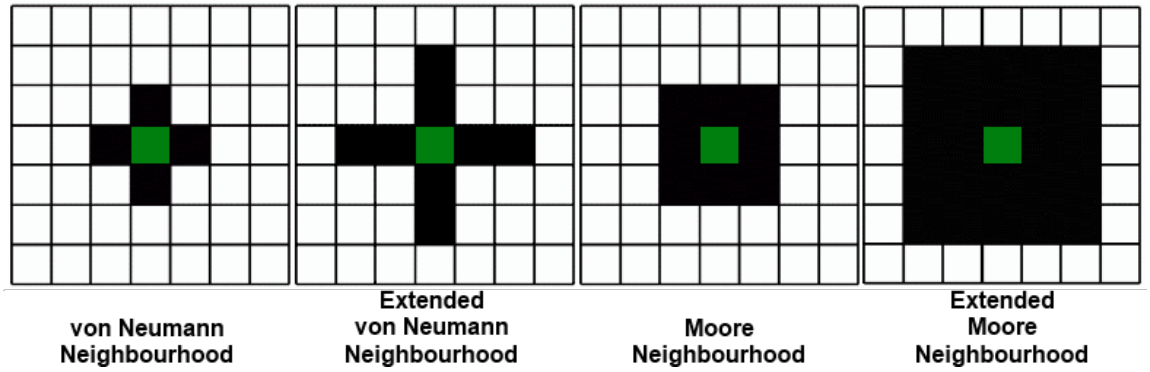

Figure 3: The neighbourhoods used in cellular automatas.

4.2.1 Initial Seeding

The cellular automata is initialized by randomly seeding the map with one of the different height levels through the use of random number generation. The default odds are the following:

- **Height2**: $odds < 0.25$
- **Height1**: $0.25 \leq odds < 0.50$
- **Height0**: $0.50 \leq odds$

Before the seeding process begins, **x** *focus points* are randomly determined. During seeding, for every cell a random number between 0.0 and 1.0 is generated and the state of the cell is set to the corresponding height-level, as shown above. If the cell is within 15 tiles of any of the *focus points*, the generated number is reduced[12] depending on how close the cell is to a focus point. The reduction follows the calculation

$$odds := \langle odds \rangle - \frac{15 - \langle actualDistance \rangle}{15 \times 3} \tag{1}$$

where *odds* is the original chance generated and *actualDistance* is the actual distance to the closest of the focus points. This results in a reduction of up to 0.33. Furthermore, if the cell less than $\frac{15}{2}$ from the closest focus point, the following calculation is applied after calculation 1.

$$odds := \frac{\langle odds \rangle}{2} \tag{2}$$

[12]This heightens the odds of a higher height-level occouring

4.2.2 Ruleset

The default rules we use focus on creating connected areas of the same height-level. This is done in order to mimic how StarCraft maps actually look. The default ruleset does leave single tiles of odd heights around, which is later cleaned by smoothing the terrain out using another ruleset. **Wrap this up somehow.**

5 Evolution

Evolutionary programming[3, 2] is based on the "survival of the fittest" way of nature. If we can determine the quality of an object through a function (also called the *fitness function*), it is possible to improve on an object over time.

We do this by creating an initial population of objects. From this population, we choose candidates according to the *selection strategy* (usually based on the fitness function) and use them to seed the next generation. When the next generation comes around, *offspring* will be spawned from the chosen candidates (called "parents") and added to the population. Candidates for the next generation are selected from the current population, new offsprings are created and a new generation comes around. This way good candidates will survive and reproduce continuously, until a set quality, or some other condition (e.g. number of generations or barely any quality improvement over the last few generations), is reached.

A simple *selection strategy* would be to always select the highest scoring candidates. The problem with this strategy, is that the evolution may get stuck in a *local optima*. This essentially means that no matter how the best candidates are changed we will never find better solutions and because we always select the best candidates, we never get to try other opportunities. Rocha and Neves[7] suggests that parents for the next population should be chosen using a stochastic method, where a candidate's chance of being selected is based on its ranking compared to other candidates. While it does not completely remove the problem of hitting a local optima, it does alleviate it to some degree.

When it comes to spawning *offspring*, there are two ways to do it[9, Chapter 2]. *Recombination* selects two or more parents at random and combine their values (according to some function) to form one or more new offspring. *Mutation* selects a parent and creates an offspring where one or more pieces have been randomly changed (usually within set boundaries).

While our main idea was to compare MOEA and CNS, we decided that we wanted to see how a normal evolution compared to the other two methods. For this purpose, we implemented an evolutionary algorithm that allows for choosing between two selection strategies (highest fitness and chance based) and the two offspring strategies (mutation and recombination).

The fitness function we use is discussed in section 6. The fitness the EA attempts to optimize is the sum of all the part of the fitness function.

6 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 it would impact our two evolutions negatively. 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 the distance between points. For the distance, we used Jump Point Search[4, 5, 6] in order to find a ground path and then count the number of tiles that make up the path. When we judge the distance between points, we also consider the direct distance due to the flying units each race has available.

6.1 Preserving Speed

As mentioned above, we wanted the fitness function to be fast. We have made some optimizations to the fitness calculations in order to help with this.

When the fitness function is started, it iterates over all cells in the map and saves the position of every start base, other bases and Xel'Naga tower. This process has a time complexity of $O(n \cdot m)$, where n and m is the width and height of the map respectively. Finding these positions at the start saves time later when we need the positions for calculating different parts of the fitness (see section 6.2).

The path between the two start bases is found, and saved, immediately after as it is used for

multiple purposes. We use JPS for the pathfinding [13], which makes the pathfinding take roughly 60 milliseconds[14].

As the map is mirrored, any fitness calculation that involves a start base does not need to be run for both start bases. They will both score the exact same value, so we only run such calculations once. This is mainly important when open space around a start base is calculated, as that part relies on using Breadth-First Search, an inherently slow algorithm.

6.2 Parts of the fitness function

Our fitness function consists of eleven separate parts. These eleven parts cover what 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.

Open space around a start base

Open space around the start base is needed to place buildings and be able to maneouver units.

The fitness is calculated by selecting the middle point of a start base and using 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)[15].

The fitness 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[16] 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 $(0.7 * mapWidth) + (0.7 * mapHeight)$ ([17]) and 0. If there is no path, a penalty is given instead (see section 6.2.1).

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.

[13] **cannot find anything about JPS time complexity**
[14] On our computers.
[15] Need analysis somewhere.
[16] need reference
[17] remember default value

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 choke points[18] 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 path, a penalty is given instead (see section 6.2.1).

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.

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 and expansions are

How open a base is determines how many directions it can be approached from. A 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.

[18] A choke point is any point that is only 3 tiles wide.

6.2.1 Penalty function

There are some maps that are infeasible for playing, e.g. maps with no ground path between the start bases. Such maps should not dominate the fitness rankings, as it would lead to a lot of infeasible maps being generated.

We chose to give infeasible maps a penalty that reduces their fitness. We did not want to give give them a *death penalty*[1], however, as they may have some interesting features if one looks away from what makes them infeasible. Instead, we calculate a penalty for the map as follows.

$$penalty := \frac{\langle maxTotalFitness \rangle}{3} \tag{3}$$

This penalty ensures that the map's fitness will not dominate feasible maps, but it may still be chosen over feasible maps that are horrible.

References

[1] Carlos Artemio Coello Coello. Constraint-handling techniques used with evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 849–872. ACM, 2012.

[2] A. E. Eiben. Evolutionary computing: the most powerful problem solver in the universe? 2002.

[3] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2007. URL `http://www.cs.vu.nl/~gusz/ecbook/Eiben-Smith-Intro2EC-Ch2.pdf`. Accessed May 15, 2015.

[4] Daniel Damir Harabor and Alban Grastien. Online Graph Pruning for Pathfinding On Grid Maps. In *AAAI*, 2011.

[5] Daniel Damir Harabor and Alban Grastien. The JPS Pathfinding System. In *SOCS*, 2012.

[6] Tomislav Podhraski. How to Speed Up A* Pathfinding With the Jump Point Search Algorithm, 2013. URL `http://gamedevelopment.tutsplus.com/tutorials/how-to-speed-up-a-pathfinding-with-the-jump-point-search-algorithm--gamedev-5818`. Accessed May 15, 2015.

[7] Miguel Rocha and José Neves. Preventing premature convergence to local optima in genetic algorithms via random offspring generation. In *Multiple Approaches to Intelligent Systems*, pages 127–136. Springer, 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.8662&rep=rep1&type=pdf`.

[8] 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.

[9] Julian Togelius, Noor Shaker, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.