# 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

# 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[1] 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[2], 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[3], PCG is used to create the entire world and most of what happens in it. Because of the amount of additional content that PCG can generate, there is a potentially near endless amount of novel content that players can consume, making it possible to meet the needs of the players without having to spend more development time on creating content after release.

## 1.2 StarCraft

StarCraft II[4] is a real-time strategy game developed by Blizzard Entertainment. It is the sequel to the original StarCraft[5] 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[1] 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, otherwise one race will dominate the other two, making those two races redundant. The balance comes from tuning the stats of unit's

[1] Countering a unit means using a unit of your own that exploits a weakness in the enemy unit.


*Figure 1.1: Left: A Terran base. Right: A Terran fleet assaulting a Protoss base.*

(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[2] 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[3] 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[4] 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 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

[2]Some maps only utilize two.
[3]Unobstructed vision means that nothing can block the vision in any way.
[4]Overwhelming the opponent (often with cheap units) before they can establish a proper defense.

resources. If multiple expansions are close to a start base, they will be easy to defend and hard to attack, as it is take less time for reinforcements to arrive from the main base. 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 an element of risk/reward to player decision making. The map *Daybreak* is a good example of a balanced map (see figure 1.2).

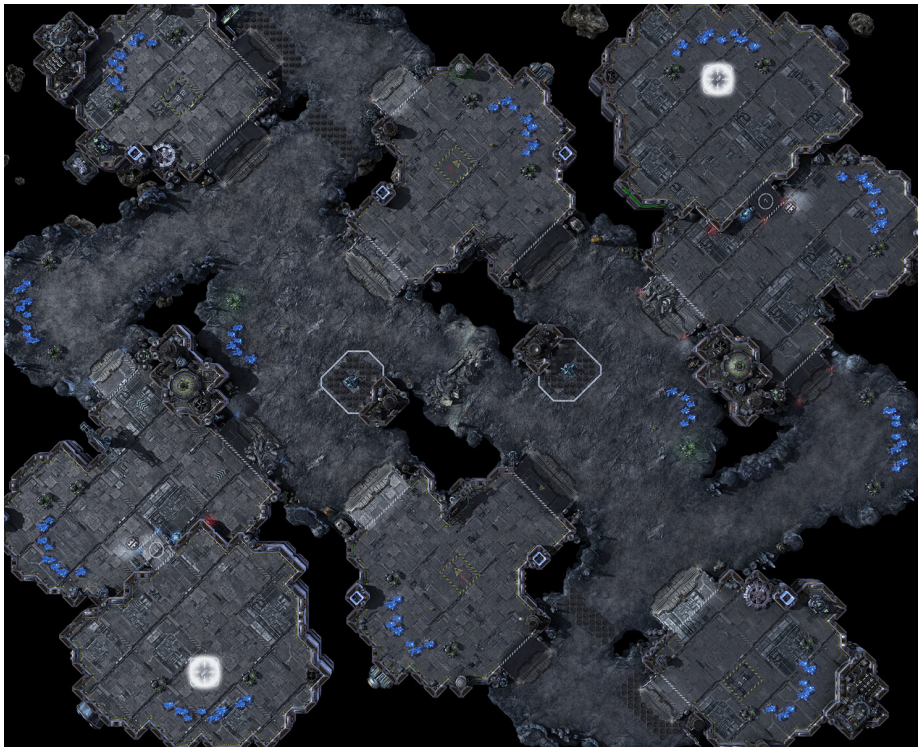

*Figure 1.2: The map "Daybreak". Notice how the start bases (glowing areas) are far from each other and how expansions (the blue minerals) are spread out with only one being close to each start base.*

### 1.2.1 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[5] 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 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. It would allow for some interesting shifts in the meta-game as tournaments could be held where only randomly generated maps were played.

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

This thesis explores the potential of an "on-the-fly" (online) StarCraft II map generation tool and what trade-off between runtime speed, map balance (quality), and novelty is required in order for the generator to finish generation within an acceptable time frame. A cellular automata generates the base maps, and constrained novelty search, a standard genetic algorithm, and a multi-objective evolutionary algorithm is used to search for optimal placements of StarCraft map elements on these base maps. The maps produced by the different algorithms are compared on measures of speed of generation and measures of quality and novelty in order to determine which algorithm is most fit for use as the base of a StarCraft II map generation tool.

# 2 Related Work

There has been a lot of research on the subject of generation of maps for strategy games. The research has been of the more general type, but research on StarCraft map generation does exist, though it focuses on the original StarCraft.

A search-based map generator for the real-time strategy game *Dune 2* is described by Mahlmann et al.[6]. This generator utilizes an evolutionary algorithm to search for maps and a cellular automata to convert found maps to actual game maps. The results were maps that both looked good and satisfied the specifications that had been set up, which proves that generating strategy maps is possible.

While being able to generate good maps is important, making sure they are diverse is an important quality. Maps that are simply variations of each other will quickly become stale. The research by Preuss et al.[7] focuses on this particular aspect. They found that the distance measure had a crucial effect on diversity maintenance mechanisms and that Novelty search did not reach the same diversity in generated results. That is something we will attempt to improve on through the use of a more specialized constrained novelty search.

Liapis et al.[8] introduces the Sentient Sketchbook, a tool that is intended to help designers create game levels. The Sketchbook alleviates designer effort by checking for playability and evaluating and visualizing significant gameplay properties. Constrained novelty search, an algorithm that searches for "newness", is also introduced and used to provide alternatives to the designer's maps. This program is an interesting take on map generation because it attempts to cooperate with the designer, instead of taking care of everything by itself.

Something more in the direction of what we are doing would be what Togelius et al.[9] did. Knowing that a single fitness function will have a hard time properly describing something as complex as a StarCraft map, they used multi-objective evolutionary algorithms in order to evolve maps instead. The use of MOEAs made the search computationally expensive and they found that it was difficult to find maps that were good on all parameters.

The research on MOEAs with regards to trategy (and StarCraft) maps was continued, again by Togelius et al.[10]. This time, they introduced a generic indirect evolvable representation for strategy maps and used a StarCraft specialized version to prove that it worked. More than a dozen evaluation functions were used and the generated maps were evaluated by skilled StarCraft players. Many of the evaluations partially conflicted, but some of the evaluations were demonstrated to correlate with interesting map qualities. This research was done with regards to offline map generation, and further work would be needed in order to achieve a speed that would be acceptable.

We intend to use normal evolution, MOEAs and constrained novelty search to generate StarCraft II maps and compare the results to each other. Further, we intend to figure out how much balance that has to be sacrificed in order to make the generation fast enough to be used for online map generation.

# 3 Goals

Prior research on the topic of generating maps for StarCraft has been focused on generating the maps in an offline (for during game development) context. Most maps generated by these methods require the developer to implement and polish such that the maps can be used in the game. While this is certainly useful to developers as it reduces the amount of creative work that goes into creating maps, it still takes time for the developer to finish the map before it is accessible to players. If the developer shelves the game, the game will no longer receive new maps from an official source, which means play at higher levels of skill may grow stale. Powerful tools such as the StarCraft II Editor provide tools for the community to develop their own maps, but because such maps are not from an official source, it is not unlikely that they will be rejected by the community. If a procedural content generation tool was accepted or developed by the game developer, it would be possible to provide the game with novel content without spending resources years after the game has been released.

Any tool developed for such a purpose would likely have to be implemented as an online (during runtime) map generation tool. While the idea of generating novel content is great, it is unlikely that most players would have the patience to sit for tens of minutes (or even hours, depending on runtime), even if precise map balance is an integral part of StarCraft gameplay. This means that any implementation would likely have to sacrifice some degree of quality (balance) and/or novelty in order to generate maps. The main goal of this thesis is to explore the trade-off between balance, novelty, and runtime that exists when generating StarCraft II maps. The secondary goal is to evaluate the different algorithms that we implement in order to explore the StarCraft map space (discussed further in section 4.1).

## 3.1 Map Balance

There are many possible ways to define balance in games. The most common balance evaluation in two-player competitive games is whether one player has an unfair advantage over the other. Assuming equal player skill, there are three elements have major impact on who comes out the winning in a game of StarCraft II: choice of race for both players, choice of map, and the strategies chosen by the players[1]. Because choice of strategy is heavily dependent on the map that is being played, the map being played has the largest impact on the result, if there is no imbalanced between the different races available to players. Thus the map must offer equal chances of players winning, else it cannot be balanced.

Our definition of what constitutes a *balance StarCraft map* is based on the definition used by Uriarte and Ontanón[11]:

> "... we will define a *balanced map* as one that satisfies two conditions: a) if all players have the same skill level, they all have the same chances of winning the game, and b) in the case of StarCraft, no race has a significant advantage over any other race." - Uriarte and Ontanón[11]

[1] It can be argued that choosing the correct race and strategy is part of player skill. They are, however, still worth noting on their own.

In addition to giving players of equal skill an equal chance of winning and allowing no race to have a significant advantage over any other race, we expand the definition of balance by taking the strategies available to players into account. A map must have more than one viable strategy (no single strategy can dominate others) so that players who favour different strategies are not left at a huge disadvantage when playing the map.

We aim to achieve this map balance by making symmetrical maps and by aiming to equalise the number of opportunities that each type of strategy has. Creating symmetrical maps for real-time strategy games guarantees that one starting position is not more favoured than the other. In order to equalise different strategies, we aim to make sure that the most direct ground path between starting bases is not too short (favours rushes) and not too long (favours playing the macro game). Additionally, we aim to place the nearest expansion base at a relatively close distance and other bases somewhat further away, such that strategies focused on the mid-game are viable without making it too easy to secure multiple bases. To prevent mobile units have too much of an advantage (or disadvantage), we aim to avoid having too few or too many choke points on the path between start bases and to not have too few or too many angles of approach for bases. A low number of choke points or very open areas around bases will benefit mobile units heavily, as they can run around and/or past less mobile units, and vice versa in the case of a high number of choke points and/or very closed bases.

## 3.2 Direct Content Generation

Part of the initial goal of this thesis was to produce actual StarCraft II map files in order to provide players with a "play & plug" option after running the generator. This is not possible, however, since there exists no functionality to do so within the StarCraft II Editor, and the map file format has, to our knowledge, not yet been reverse engineered in a way that is applicable in this context. While it is beyond the scope of this thesis to produce actual StarCraft map files, all players have access to the editor in which they can manually create maps for play. We intend to make use of this feature by generating an image file that shows a tile-by-tile layout of the generated maps, so that players (and developers) can create the maps manually in the editor.

## 3.3 Optimal Trade-off

In order to reach an optimal trade-off between balance, novelty, and runtime, it is important to consider what minimum requirements exist for each part of the trade-off. Because the gameplay of StarCraft relies heavily on the balance and fairness between players and the in-game races, any generated must be balanced as described in section 3.1. If a map does not conform to the requirements for it to be balanced, it cannot be considered as a solution to the problem of StarCraft map generation. It is imperative that generated maps are balanced within predictable parameters, as the tool will not be useful in any type of competitive environment if even a small number of output maps are unbalanced.

As previously mentioned, the generation of a new map should not take longer than players are willing wait for it be generated. Based on personal experience, we have chosen two different time limits for map generation: two and five minutes. These time limits constitute a very short and short break, respectively. These breaks happen naturally when playing StarCraft in ladder games, as it takes time to find an opponent between games. If players are patient enough to wait for an

opponent to be found, it is not unlikely that they would be willing to wait the same amount of time for a map to be generated.

The search algorithms must also provide maps that are novel when compared to other maps generated by the search algorithms. If generator can only produce a very limit number of novel maps, then it does not live up to its functionality as a creator of novel content and thus is unusable for that purpose. Therefore it is a requirement that the generator is capable of generating, at least, several hundred novel maps of medium to high quality. The StarCraft map space is immense, so it is likely that any algorithm capable of traversing a wide area of that search space will be able to produce the required level of novelty in maps.

While it is likely that there exists a number of combinations that could be considered optimal trade-off, the focus of this thesis is to tend towards trade-off where balance is a dominating factor.

# 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[12](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[13], 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

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[14]. 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 4.1).

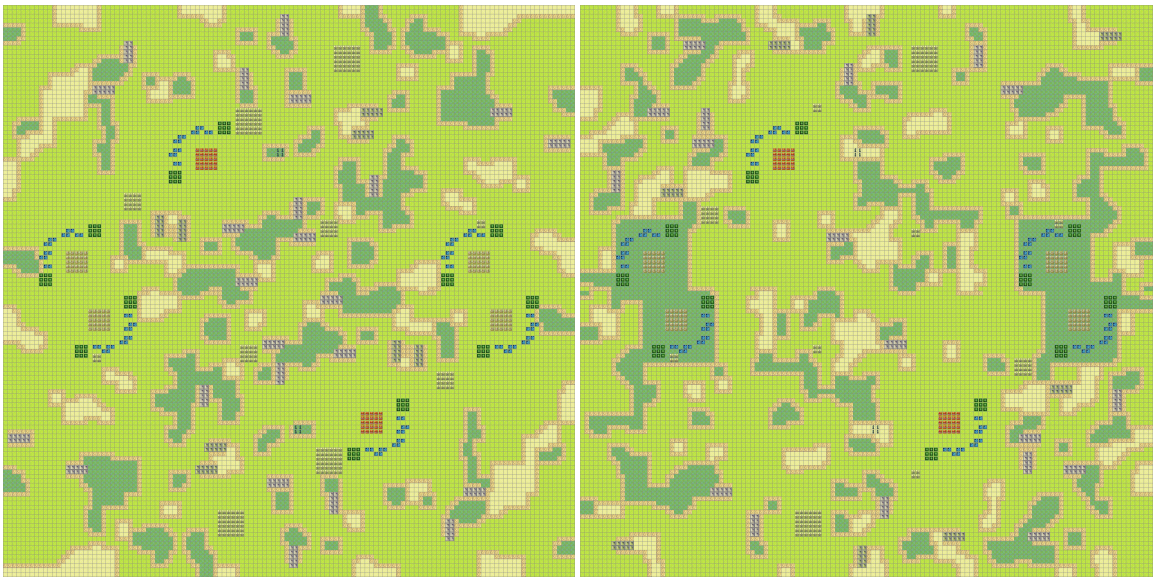

*Figure 4.1: Two generated maps that share the same map points, but have a different base map.*

### 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 4.2 shows a Xel'Naga tower placed at 45 degrees at 0.5 distance on a size 64 by 64 map.

As long as the same base map is being searched, any small change to the degree or distance of a

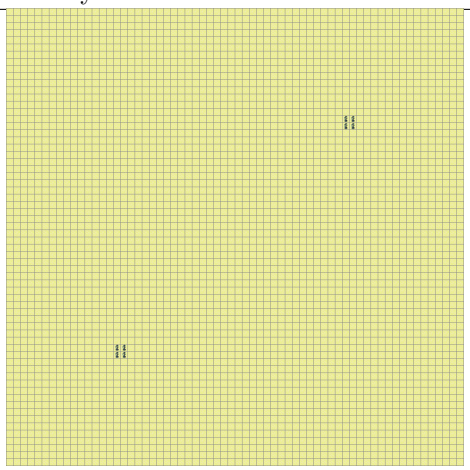

*Figure 4.2: The placement of a Xel'Naga tower at 45 degrees and 0.5 distance on a 64-by-64 map. The two grey 2x2 square indicate a Xel'Naga tower.*

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 **DestructibleRocks** (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 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

[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 4.4

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 Map Data Structure

In order to test the quality of maps and have something that potentially could be put into the StarCraft II Editor, a structure that can represent the entire map is required.

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 a single cell to contain a height-level (it always will), a location for a base[2] and destructible rocks that have to be cleared before a base can be built. This situation require three $n$-by-$m$ grids in order to properly represent:

- The first grid represents the heightmap. It has the following states: Impassable terrain, height 0, height 1, height 2, ramp or cliff.
- The second grid represents the actual position of different map features; Bases, minerals, gas and xel'naga towers.
- The third grid represents where destructible rocks are positioned in the map.

### 4.3.1 Functionality of the Structure

Generation of the *initial heightmap* and *smoothing of the heightmap*[3] are both handled by a cellular automata (see section 4.5) and will not be discussed in-depth here.

**Cliff Placement**

*Placement of cliffs* is a straight-forward process. In StarCraft, cliffs form the natural border between two height-levels (see figure 4.3). We find, and create, the cliffs by iterating over all tiles in the heightmap and finding every tile's neighbours (using the von Neumann neighbourhood, see figure 4.4 on page 16). If any neighbour is of a lower level, that neighbour is chanced to become a cliff[4]. In the event the tile has two, or more, neighbours of lower level, the tile is a corner of that section of height-level. The tile, and its neighbours, are changed to cliffs, as that is how StarCraft represents the corner of a height-level (as see in figure 4.3).

**Cliff Smoothing**

After cliffs have been placed, it is possible that cliffs have been placed in such a way that they do not border up to form a border between height-levels. This can happen in situations where one of the two height-levels only take up a small area. After placing cliffs, we iterate over the heightmap once more. If a cliff is found, its neighbours are checked (this time using Moore neighbourhood) and if it does not have two different height-levels in its neighbourhood, the cliff is changed to whatever height-level it bordered up to.

During the smoothing process, if any tile is found that is only connected to one other tile of the same height-level[5], that tile is transformed into a cliff and the position is saved. For every saved position, the area in a 3-by-3 square around it is run through the smoothing process again. This

[2] We save the centre locations of bases for use with the fitness function.

[3] Both generation and smoothing are only focused on height-levels, not on cliffs or ramps. The placement of cliffs are handled by the map structure and placement of ramps is handled by the representation structure.

[4] Unless something is in the way, such as minerals, gas or xel'naga towers.

[5] This is checked using the von Neumann neighbourhood.

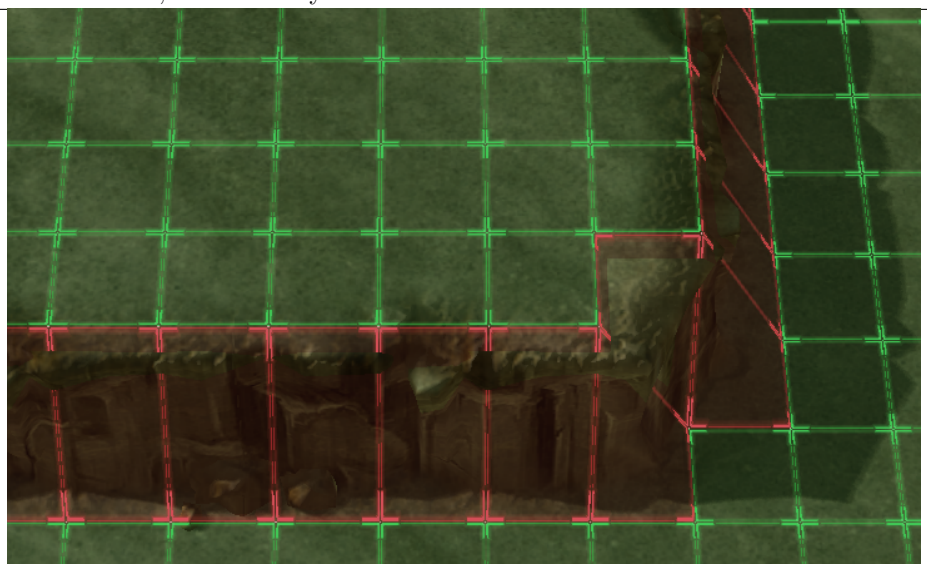

*Figure 4.3: A section of cliffs in the StarCraft map editor. Notice how the corner of the cliff appears to be on the higher level.*

may seem counterintuitive, but in StarCraft the minimum size of a height-level area is 2-by-2. Any height-level tile **must** therefore have at least two other tiles next to it, otherwise it is not a valid height.

**Visual Representation**

When a map has been fully created (that means after the representation has been converted, see section 4.4), a visual representation can be created. Every state in the heightmap, the feature map and the rock map has a picture of a tile mapped to it. These tiles are draws on a bitmap, in the order of heightmap, features and rocks. This map is saved to a .png file which can be opened and analyzed. For an example of the visual representation, see figure 4.1.

## 4.4 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[6].

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

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 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.

## 4.5 Cellular Automata

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[12]), 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[7] 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

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

neighbourhoods only consider cells one tile away from current cell, but they can be *extended* to cover a larger area. Figure 4.4 shows the two neighbourhoods along with their extended versions.

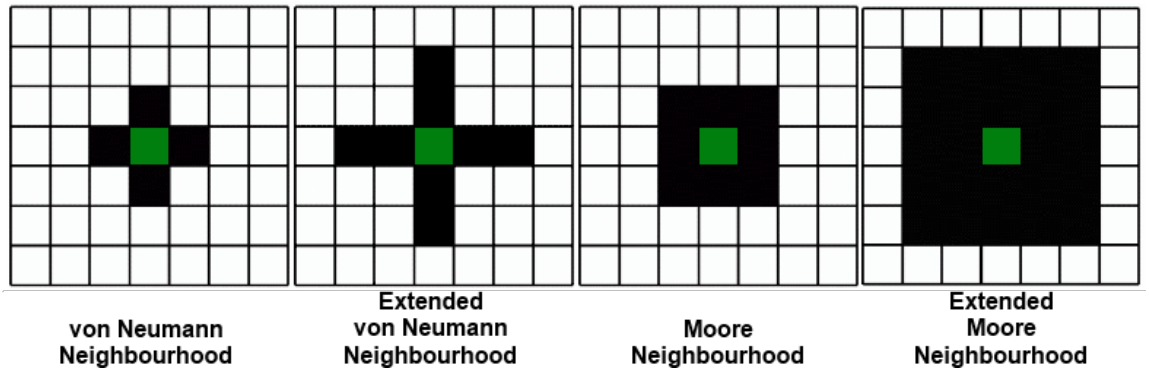


*Figure 4.4: The neighbourhoods used in cellular automatas.*

### 4.5.1 Our Cellular Automata

We use a cellular automata to generate the initial heightmap for the phenotype, to smooth out the heightmap and to place impassable terrain. 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.

**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[8] depending on how close the cell is to a focus point. The reduction follows the calculation

$$odds := \langle odds \rangle - \frac{15 - \langle actualDistance \rangle}{15 \times 3} \tag{4.1}$$

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

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 4.1.

$$odds := \frac{\langle odds \rangle}{2} \tag{4.2}$$

**Rulesets**
The ruleset we created for the *initial heightmap generation* uses two types of rules:

One type looks at the bigger picture. It looks at the extended Moore neighbourhood and upgrades a height-level to the one above, if at least 75% of the neighbours are of the higher level. This makes areas of height-level more circular, which is how many maps in StarCraft are created, and it helps avoid pockets of low height-levels inside areas of higher levels.

The second type looks at the immediate area and acts upon that. It uses the normal Moore neighbourhood. If 60% of its neighbours are of a higher level it upgrades to that level. If it is a lower level that is stuck between groups of higher levels, it is changed to the lowest of those levels in order to smooth out "holes".

The *smoothing ruleset* focuses solely on making areas more uniform. Every tile checks both the normal and extended Moore neighbourhood and is changed to fit better in with the surroundings. For example, if a height2 is surrounded by height1, it is transformed into a height1 itself.

**Impassable Terrain Generation**
Impassable terrain is generated after the heightmap has been smoothed. By default, four sections of impassable terrain is generated. For every section, two points are randomly chosen on the map, no further than 50 tiles apart. A direct line is traced and split into 10 points, which are then displaced to avoid robotic-looking areas. Impassable terrain is placed in areas of varying size around every point, which creates chasms or large pools of terrain that blocks ground unit movement.

## 4.6 Evolution

Evolutionary programming[15, 16] 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 offspring 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[17] 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[1, 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).

As part of the goal of exploring both evolutionary and novelty search methods, a standard genetic algorithm was implemented. The tests will include two types of selection strategies (highest fitness and chance based) using mutation as the variation operator. The fitness value of an individual in the evolutionary algorithm is the sum of all of the different fitness measures described in section 4.8.

## 4.7 Multi-Objective Evolutionary Algorithm

While the standard genetic algorithm seeks to optimize a single objective function, multi-objective evolutionary algorithms bases its fitness on two or more fitness functions. This is because it is infeasible, for some problems, to combine all interesting features into a single fitness function. An example of this is the basic need for food and water for humans. Most fitness functions would reduce the problem to *amount of food* + amount of water. This is infeasible, however, as humans cannot function without water and/or food. Therefore for any solution to be feasible, it is necessary for it to provide both enough food and water for a human to survive. During the evolution process, multi-objective evolutionary algorithms focuses on finding solutions that are *non-dominated solutions* (a solution is dominated when there is at least one other solution "that is better in at least one objective and worse in none."[10]). In the case of food and water for a human, for any solution to dominate another, it must provide at least as much food and water as the solution it dominates, and must provide more of either food or water.

Using multi-objective evolutionary algorithms is a solid approach to StarCraft map generation, as there are multiple objectives to optimize when generating a good map, such as distance between starting bases, number of choke points, how easily it is to defend a starting position, and the number of expansions available to players.

## 4.8 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[**?** ]: *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[18, 19, 20] 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.

### 4.8.1 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 4.8.2).

The path between the two start bases is found, and saved, immediately after as it is used for multiple purposes. We use Jump Point Search[20] for the pathfinding[9], which makes the pathfinding take roughly 60 milliseconds[10].

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

[9] **cannot find anything about JPS time complexity**

[10] This test was performed on our relatively powerful modern laptops, so results may vary.

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.

### 4.8.2 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. Looking at the maps that are part of the high-level competitive map pool, it is clear that the start bases in those maps are always at the highest level in the map (some maps only have 2 height levels). 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 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. If there is no path, a penalty is given instead (see section 4.8.3).

**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 choke points[11] 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 4.8.3).

[11] A choke point is any point that has a width of three tiles or less.

**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.

### 4.8.3 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 them a *death penalty*[21], 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.

$$penalty := \frac{\langle maxTotalFitness \rangle}{3} \tag{4.3}$$

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.

## 4.9 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[13]. 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[22].

Constrained novelty search applies constraints to the novelty search algorithm in order to direct evolution towards a desired outcome[13]. The algorithm uses the concept of *two-population novelty search* introduced by Lehman and Stanley [23]. *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[13] 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. [13] 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.

### 4.9.1 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.

### 4.9.2 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.

## 4.10 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. Ensuring that the evolution will be

seeded with individuals that are as novel as possible will force the search to visit a larger area of the search space, but it is likely to quickly converge towards the best of those very novel individuals. Using the novel individuals with highest fitness ensures that the search will find

# 5 Results

# 6 Discussion

# 7 Future Work

# 8 Conclusion

# Bibliography


[1] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[2] Blizzard Entertainment. Diablo iii. `http://us.battle.net/d3/en/`, May 2015. Accessed May 15, 2015.

[3] Tarn Adams and Zach Adams. Dwarf Fortress. `http://www.bay12games.com/dwarves/`, 2006. Accessed May 15, 2015.

[4] Blizzard Entertainment. Starcraft ii. `http://us.battle.net/sc2/en/`, May 2015. Accessed May 15, 2015.

[5] Blizzard Entertainment. Starcraft. `http://us.blizzard.com/en-us/games/sc/`, May 2015. Accessed May 15, 2015.

[6] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Spicing up map generation. In *Applications of evolutionary computation*, pages 224–233. Springer, 2012.

[7] Mike Preuss, Antonios Liapis, and Julian Togelius. Searching for good and diverse game levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[8] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *FDG*, pages 213–220, 2013.

[9] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, J Hagelback, and Georgios N Yannakakis. Multiobjective exploration of the starcraft map space. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 265–272. IEEE, 2010.

[10] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N Yannakakis, and Corrado Grappiolo. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277, 2013.

[11] Alberto Uriarte and Santiago Ontanón. Psmage: Balanced map generation for starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[12] 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.

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

[14] 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.

[15] AE Eiben and JE Smith. Introduction to Evolutionary Computing. 2008. Accessed May 15, 2015.

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

[17] 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`.

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

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

[20] 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.

[21] 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.

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

[23] 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.