
# Evolved cellular automata for 2D video game level generation

Bachelor's Thesis, 15 credits

Adel Sabanovic,
Amir Khodabakhshi

**Bachelor's degree, 180 credits**
**Field of study: Computer Science**
**Programme: Computer Systems Developer**
**Supervisor: Bahtijar Vogel**
**Examiner: Jesper Larsson**
**Final seminar: 2022-05-30**
**Course: DA391A**

# Abstract

Manual design of levels can be an expensive and time consuming process. Procedural content generation (PCG) entails methods to algorithmically generate game content such as levels. One such way is by using cellular automata (CA), and in particular evolved cellular automata. Existing research primarily considers specifically determined starting states, as opposed to randomly initialized ones.In this paper we investigate the current state of the art regarding using CA's that have been evolved with a genetic algorithm (GA) for level generation purposes. Additionally, we create a level generator that uses a GA in order to evolve CA rules for the creation of maze-like 2d levels which can be used in video games. Specifically, we investigate if it is possible to evolve CA rules that, when applied to a set of random starting states, could transform these into game levels with long solution paths and a large number of dead ends. We generate 60 levels over 6 experiments, rendering 58 playable levels. Our analysis of the levels show some flaws in certain levels, such as large numbers of unreachable cells. Additionally, the results indicate that the designed GA can be further improved upon. Finally, we conclude that it is possible to evolve CA rules that can transform a set of random starting states into game levels.

# Acknowledgements

We would like to thank our supervisor Bahtijar Vogel for patiently guiding us through the process of writing this thesis and providing us with valuable feedback and insights.

# Table of contents

# Abbreviations

- CA = Cellular Automata
- FF = Fitness Function
- GA = Genetic Algorithm
- RQ = Research Question
- PCG = Procedural Content Generation

# Chapter 1 - Introduction

## 1.1 Background

As game development costs continuously rise, a costly aspect of game development is the need for lots of artists and designers to manually create game content [1, Ch. 1]. A different approach to manual design is to procedurally generate game content, such as levels, using algorithmic methods. This approach, often referred to as procedural content generation, can lead to a practically infinite number of levels to be generated at a faster rate and potentially lower development cost [1, Ch. 1]. The term procedural content generation (PCG) entails the “algorithmic creation of game content with limited or indirect user input” [2]. Game content is to be understood as levels, items, quests and various other content types for games such as also the rules of the game itself. Having a mechanism that allows for the automated creation of content not only reduces the cost and time spent for development, it also increases replayability [1, Ch. 1].

In this paper we investigate the possibility of using a genetic algorithm (GA) to evolve cellular automata (CA) for the creation of maze-like 2d levels, meant to be used in a videogame as, for example, dungeon levels. Genetic algorithms are a subset of evolutionary algorithms. Evolutionary algorithms are a set of search based methods consisting of the genetic algorithms, genetic programming, evolutionary programming and evolutionary strategies. The term “search based” means that the algorithm is searching through the set of possible solutions for a given optimization problem. This is done by having a population of, initially randomly generated, candidate solutions, typically represented as arrays of numeric values. Each candidate's viability is then assessed by a fitness function (FF) and is assigned a numeric fitness value. Based on some selection method, certain candidates are picked to proceed to the next generation, where higher fitness scores indicate an increased chance of being selected. The selected candidates may undergo crossover and mutation before being added to the new population. Crossover means that the solutions of two candidates are merged together. Mutation is a random change in one or more of a candidate solutions array cells. More explanation of genetic algorithms and its various parts are given in chapter 5. Without crossover and mutation, there would always be the same set of candidates in circulation meaning there would not be any change to the population [3], [4]. One application of evolutionary algorithms is in evolving the rules for cellular automata.

To understand cellular automata one can imagine a square grid of cells. Each cell in this grid has a state, in the case of this paper the states will be on or off, corresponding to a traversable or non-traversable cell of the level. Each cell also has a neighbourhood of cells. Which cells are included in the neighbourhood is given by a neighbourhood definition [5]. There are well known neighbourhood definitions for CA, such as Von Neumann’s neighborhood [6] and Moore’s neighborhood [7]. For each iteration that the CA is applied to the grid, the cells in the grid will synchronously change their state based on the state of the cells in their neighbourhood. How each cell state changes is decided by the so-called CA rule, which defines how a cell's state should be updated depending on the state of its neighbours. As an example, the CA rule might dictate that if 3 out of 4 of a cell’s neighbours are “on”, then that cell should update its state to on as well. In chapter 5 further descriptions of cellular automata are presented. Cellular automata are suitable for generating cave-like dungeons on a 2-dimensional grid [8]. However, a known problem with cellular automata is the difficulty in designing their

rules and predicting their behaviour, due to the fact that they can present very complex behaviour [5].

With the use of a genetic algorithm we aim to evolve the CA rules that are used to create levels. Our aim is to start with a population of randomly generated CA rules, and evolve these using a genetic algorithm. Each CA rule will be applied for a specified number of iterations on a set of different grids, called starting states, in order to generate maze-like 2d game levels. These levels will have a pre-defined start and end point, determining where the player would spawn and where they would have to reach. The initial state of each cell in the starting states will be randomized. The FF will examine each generated level and based on attributes defined in the FF, give each CA rule a fitness score, which over a number of generations will search for better CA rules. Over the course of the algorithm, it will improve the population of CA rules which thus leads to a higher quality of generated levels. The genetic algorithm's output will be a set of game levels. In existing literature, care has been given in defining the starting states that the CA is evolved for, leading to a lack of research regarding whether or not one can evolve CA rules for random starting states.

## 1.2 Problem definition

There is existing research and implementations of cellular automata for game level creation [8]–[11]. The issue with designing CA rules however, is that it can be difficult to design and predict the CA's behaviour, due to their complex behaviour [5]. This grants the question of evolving the rules with GA. There is also some research on evolving CA for level generation [9], [11], [12]. In the case of [9], the algorithm evolved both the CA rules and an associated starting state, resulting in a single level created at the end. Meanwhile in [11], the researchers utilized a so-called goal pattern consisting of solved mazes, requiring a target pattern in the first place. In [12], 2 fixed starting states were employed, namely an entirely traversable one and one where only the center was non-traversable.

We identify a research gap regarding evolving CA rules that work on random starting states. Our approach suggests using a genetic algorithm to evolve only the CA rule, and not the starting states. The use of random starting states simplifies the level generator design, as less effort is needed in configuring the generator. Thus no human intervention is needed for that part of the procedure, further automating the level generation process. This also contributes to making the level generator capable of creating more levels, as the starting states are randomly initialized for each of its runs.

## 1.3 Research question

The aim of this thesis is to answer the following questions:

1. What is the current state of the art with regards to using a genetic algorithm to evolve cellular automata rules for game level generation?

2. How can one evolve cellular automata rules using a genetic algorithm that, given a set of randomized starting states, produces interesting maze-like 2d levels?

In order to answer RQ1, a review of the relevant articles will be conducted. To answer RQ2, a level generator is created that aims to use a GA to evolve CA rules capable of transforming a set of random starting states into game levels.

## 1.4 Expected outcomes

We expect our genetic algorithm to, based on our defined fitness function, evolve and improve CA rules over the course of its run. The best candidate CA rule at the end of the GA's run should transform the set of randomized starting states into interesting 2d maze-like levels. The generated levels are expected to be suitable for use as eg. dungeon or cave levels in video games. This further entails that there should be a solution path for each level, meaning that they are solvable.

## 1.5 Limitations

The results of this paper will be limited by the specifications that we decided regarding the CA and GA. Different representations or neighbourhood definitions for the CA may lead to different results. A probabilistic rule array for the CA, where each output state has a certain chance of happening based on the neighbourhood state, would reasonably also have led to different results. In the same way, decisions regarding the GA used such as the defined fitness functions, selection method or mutation and crossover settings mean that other decisions in these steps could have led to different outcomes.

## 1.6 Outline

Chapter 2 discusses related work and articles of relevance for our thesis work. All from discussing previous research in PCG in a general sense to research about CA's, GA's and evolved CA's in relation to PCG. Chapter 3 describes the methodology used for our research with design science and evaluation by experiment being key parts.
Chapter 4 highlights the current state of the art and the insights we gain from it for our work. The fifth chapter describes our level generator and its various operators and parameters. We present the results and evaluate them in depth in chapters 6 and 7 respectively. Chapter 8 consists of a discussion regarding our level generator, the generated methods and discussion about our analysis of them. Finally in chapter 9 our conclusions are presented together with suggestions for future work.

# Chapter 2 - Related work

In this section we will discuss previous work relating to our research, that is, previous research within PCG and genetic algorithms. Special focus will be given to research regarding using evolved cellular automata for game level generation.

## 2.1 Procedural Content Generation

Previous research relating to the procedural creation of game levels include the use of intelligent agents [13], evolutionary search methods [9], [11], [14], [16], [12] and cellular automata [17], [8]. In [13], intelligent agent of five different types were used to create realistic 3D terrains. The agent types used were coastline, smoothing, beach, mountain and river agents, all responsible for generating different attributes in the terrain. Each agent could also only perform a limited number of actions before being consumed. The type of generated terrain could be affected and controlled through variations in the number of agents that exist of each type, and also through variations in how many actions each agent could perform on the terrain before being consumed [13]. Johnson et al. used cellular automata for creation of infinite cave levels [8]. They also showed that their CA-based algorithm was computationally efficient enough to be used in a game for real-time generation of the cave during gameplay. A different approach to cellular automata is seen in [10], in which a combination of cellular automata and machine learning was used to generate realistic levels with multiple terrain features such as forests, water and sand.

Peter Ziegler and Sebastian von Mammen [17] used Cellular Automata to create height maps for the real time strategy game Supreme commander with the aim to create randomly generated and balanced levels. They did not use a genetic algorithm and evaluated their results by letting a set of participants who were experienced Supreme commander players test the different maps [17]. Procedural generation of strategy game levels was also explored in [14], where they used a multiobjective evolutionary algorithm to generate levels for the real-time strategy game Starcraft.

Research within PCG also includes methods of generating personalized game content for the user. One article generated racing game levels that were personalized based on the users driving style [18]. Users played a racing game on a test track, and using that data the researchers created models of each users driving style. Based on these models they then evolved racing tracks that would fit and be fun for those specific users driving style. Similarly, researchers have also used evolutionary algorithms to evolve weapons based on player preferences in a space action game that they themselves had developed, called Galactic Arms Race [15], [16]. In this game, weapons that were used more often by the player during gameplay were given a higher fitness score, thus the evolutionary algorithm persistently evolved newer weapons based on and suited to the player's preferences. In the multiplayer version, the evolutionary algorithm was guided by the preferences of all the players in the server.

## 2.2 Genetic Algorithms

According to Mitchell [4] GA's in their basic form consist of three "operators", these are the selection operator, crossover operator and mutation operator. Mitchell [4] further states that it is not clear which selection method is most appropriate for a given problem. There exists a number of different selection methods, including tournament selection, roulette wheel selection, elitism and rank selection [4]. Shukla et al. [19]

reviewed multiple research articles that compared different selection techniques. Overall they concluded that tournament selection performed better than the alternatives, based on the papers they had reviewed. It should be noted that one of the benefits of tournament selection mentioned was the fact that it does not require sorting of the entire population prior to selection [19]. One of their reviewed papers comparing tournament selection with roulette selection, found tournament selection to be superior. In their experiments they used identical genetic algorithms with the only difference being the selection methods, and applied these to a set of problems. The results were that the tournament selection algorithm would find the optimal solution faster and more often than the roulette selection algorithm [20]. However, one of their examined articles concluded that deterministic selection and roulette wheel selection performed better than tournament selection [21]. Additionally, Mitchell [4, p. 126] suggested using elitism together with other selection methods to increase the performance of a GA.

Umbarkar and Sheth [22] reviewed various crossover methods, including single-point crossover, k-point crossover and uniform crossover. They state that the choice of crossover method is dependent on the encoding (representation) of the candidates, and the type of problem the GA is being applied to. In choosing the crossover method, one should also review previous work in that area to understand what type of crossover methods have worked well in genetic algorithms in that area [22]. Mitchell [4] also explains that the choice of crossover method is not simple, and depends on various factors of the GA such as the encoding.

## 2.3 Evolved Cellular Automata for game levels

There has also been some research specifically on the application of evolving cellular automata for games [9], [11], [12]. In this section we will present and compare their algorithms and findings in detail.

One such paper investigated the use of interactive, evolving CA for the creation of mazes [9]. In this paper the evolutionary algorithm evolved both a starting state and the rules for the cellular automata, resulting in a single maze. An indirect representation consisting of a 418-bit array was used, with the first 400-bits representing a 20x20 grid and the remaining 18 representing the CA rules. This is considering that there is 18 possible neighbourhood configurations in the indirect representation, with each cell having a range between 0 and 8 filled neighbors corresponding to 18 different states, if one takes into account that the center cell can be both on or off.

The fitness function of the genetic algorithm was in this case based on user evaluation, where users viewed and rated a top-down image of a subset of the generated maps for each generation. The authors findings showed that this method succeeded in evolving more interesting maps, as the average rating of the map from the first generation was 2.75, compared to 3.75 for the map from the final generation [9]. Of note is the authors findings regarding which features of the levels led to higher fitness scores, which were then used for non-interactive evolution in [12].

In [12] they had 3 different fitness functions with the first one being based on the length of the shortest path, the second being the number of dead ends and the third one based on the combination of the first two. These were all derived from the user's interactive evaluation [9]. Two kinds of indirect representations were implemented for the CA rules where one was based on an algorithm which calculated the probability that a cell state

would change and the other one being binary. In both cases the representation consisted of an 18-bit rule array. Both representations were tested and compared with two different starting states for each representation and FF. The result showed that the probabilistic representation did far better with the first and third FF but only slightly better with the second one. In this article they only had 2 fixed starting states [12].

Pech [11] used genetic algorithms to evolve Cellular automata rules . They evolved rules that were based on attributes from mazes that they found desirable. A collection of 100 perfect mazes were used as starting states for the CA. For the fitness evaluation, each candidate CA rule was applied on this collection of starting states. Image processing techniques were then used to extract and analyze certain features and attributes of these levels. These extracted attributes were then compared to a so-called goal-layout, in order to evolve CA rules capable of generating similar levels.
Two representations were used for the CA rules, a direct and an indirect one. These were implemented so that an array which included a neighbourhood of a certain size had the central cell change state depending on the neighbour's states. In the direct representation, the number of possible combinations of the neighbourhood states were counted. For example in a moores neighbourhood, with one layer, there would be a total of 512 unique combinations given that there are 9 cells in the neighbourhood, including the center cell to be updated, each with two possible states which resulted in 2^9 combinations. The central cell was then given a new state which corresponded with the value in a rule table consisting of an array of 512 elements. The indirect representation had an output state for every unique summation of the neighbourhood; this was preferable with larger neighbourhoods since it had a far better time complexity than the direct representation. A technique named flavouring where they allowed for each state to have multiple states was implemented. The result showed that the indirect representation had very similar results among the different generations with different amounts of cell states while the direct representation varied a lot [11].

Different selection methods have been used in the previous works. Tournament selection with a tournament size of 5 was employed in [11], combined with an elitist selection of size five, whilst in [9] fitness proportional selection was employed. In [12] an elitist method was used, where the $n$ parents created $n$ children, leading to a collection of $2n$ candidates. This $2n$ population was then sorted and the top $n$ individuals were selected to form the population of the next generation.

The number of generations that the GA ran for also varied amongst these papers. Pech [11] had a max generation limit of 5000. However, they combined this with two other termination conditions; the first being in case a candidate had reached the highest possible fitness score, and the second being if the population had converged. Convergence was defined as if the highest ranking candidate score had not increased by more than 0.0001 over a span of 100 generations. The two other works had lower max generations, with [12] running for 60 generations and [9] for only 10 generations. The low number of generations in [9] was due to the fact that it used an interactive fitness function, and they thus reasoned that a high number of generations would have led to user fatigue in the rating process.

In [9] they used a two point crossover method with a 70% chance of crossover occuring. However, in their paper they evolved both the CA rules and the starting states, meaning that their representation had a part that encoded the CA rules and another part that encoded the starting states. They used one crossover point for the CA rules part of their representation, and a second crossover point for the starting states part of their

representation. Thus meaning that single-point crossover was employed for the CA rules part of their candidates. Meanwhile [12] used a half-uniform crossover with a 100% crossover probability, with the high percentage being due to their more elitist selection method. In [11] single-point crossover was used with a 60% chance of crossover occuring.

Point mutation was employed in the three works mentioned in this section, where each cell has a certain probability of being mutated. The probability of a cell being mutated was 1% and 5% in [9] and [12] respectively. Pech [11] had the mutation probability set as 1 divided by the length of the candidate, i.e aiming to mutate 1 bit, on average, during the mutation step. In Pech's [11] work certain representations had more than one state for each cell, so-called flavours, so in the case that mutation was to occur for a cell there was another random variable deciding on which state a cell should be set to.

# Chapter 3 - Research Methodology

## 3.1 Design Science

In order to answer RQ2, we aim to create a level generator that uses a GA to evolve CA rules for game level generation. For its creation the design science methodology will be used. Design science is a research approach that consists of designing and evaluating created IT artifacts. It includes designing artifacts for solving problems defined in a research question [23, Ch. 8]. According to Hevner et al. [24], an artifact may consist of constructs, models, methods and instantiations. In the case of this research our artifact consists of method and instantiation. The method is the proposed genetic algorithm to evolve CA rules for game level generation, including their various related parts. The instantiation will be a working level generator consisting of the proposed methods, that will output a number of game levels. The role of the instantiation is to act as a proof-of-concept that the suggested method, in this case algorithm, is capable of producing the expected results, in this case interesting game levels [24]. The created artifact will also serve in answering RQ2. Hevner et al. [24] give seven guidelines for conducting design science research while Vaishnavi and Kuechler [25] provide a five step process. One paper [26] mapped Hevner et al's [24] guidelines onto Vaishnavi and Kuechlers [25] five steps, showing their similarities. In this paper we opt for Vaishnavi and Kuechlers five steps, as visualized in figure 1, to guide us in our design science research process. The steps are: awareness, suggestion, development, evaluation and conclusion [25]. It is important to note that iterations between the steps are possible. Below follows an explanation of how the steps were integrated in our research process.

For this paper awareness came in two phases. Initially the authors' curiosity and interest was raised by PCG's potential of solving real-world business needs with regards to game development, including the algorithmic generation of levels, as explained in chapter 1. Following this, a study of related relevant literature was conducted to identify research gaps within the area of PCG, the results of which were presented in chapter 2. After an initial reading of PCG related papers, we identified a research gap in the form of a lack of research with regards to applying evolved CA rules on random starting states for generating game levels. Subsequently, the research questions were formed. In order to answer RQ1, further literature review was conducted on papers relating to GAs and to using GA's to evolve CA rules for game level generation in order to understand the current state of the art. The searching process included using databases such as ACM [27], the PCG Workshop's paper database [28], and searching for papers with the help of Google Scholar [29] and Libsearch [30]. Search terms relating to PCG, GA and CA were used. To answer RQ2, we suggested creating a level generator that would use a genetic algorithm to evolve CA rules that when applied to random starting states would generate interesting game levels.

The third step, development, entails the creation of the level generator containing the CA and the GA. Designing the algorithm, tweaking the parameters and coding a runnable version of it are important parts in the third step. As it is not the coding *process* itself that is the novelty of this research, the actual programming of the generator will not be detailed in this paper. Rather the focus will be on the algorithm design and its parameters. However, ample amounts of pseudocode and descriptions of each part of the algorithm will be given so that both future researchers and practitioners (game developers) may understand our level generator and build upon our work.

In step 4 we evaluate the level generator to ascertain if there are deviations compared to our expected results. The evaluation step serves an important role in understanding the viability of our level generator for generating game levels. This will be done by conducting a series of experiments using the instantiation of the level generator. Evaluation in this thesis will be conducted both between experiments and once all experiments have been conducted. We will have an iterative process between steps 3 and 4, where the results of the experiments will undergo a minor evaluation phase that may lead to changes in the algorithm for subsequent experiments. In the results section, in addition to presenting the experiment results, we will also explain our thought process when it comes to changes to the algorithm for each experiment. Once we are done with conducting the experiments, the algorithm and all of the experiment results as a whole will undergo a major evaluation phase and be analyzed in detail. A minor iterative cycle will also be conducted when setting up the instantiation of the algorithm, as presented in chapter 5.9. Finally, step 5 is the conclusion. In this step our findings as a whole will be presented, serving to communicate our findings both to the research community and practitioners. The conclusion step will be based on our evaluation and analysis of our experiment results. Was the level generator good enough and are there features that could be added to the CA or the GA?  It will also include potential unclarities and suggestions for future work.

In design science research it is important to motivate how the process differs from regular development [23, Ch. 8], [25]. We argue that our level generator contributes to research by taking pre-existing knowledge in the areas of genetic algorithms, cellular automata and PCG and applying it in a novel fashion, in this case by applying evolved CA on randomized starting-states to generate game levels. Additionally, our level generator will be critically evaluated by conducting a series of controlled experiments, which is a crucial step of design science research contributions [24]. We further argue that our proposed research approach differs from normal development in that we in a scientific manner detail and present our work, including the level generator, and the evaluation, results and analysis of it, thus advancing the existing knowledge base within primarily the area of PCG.

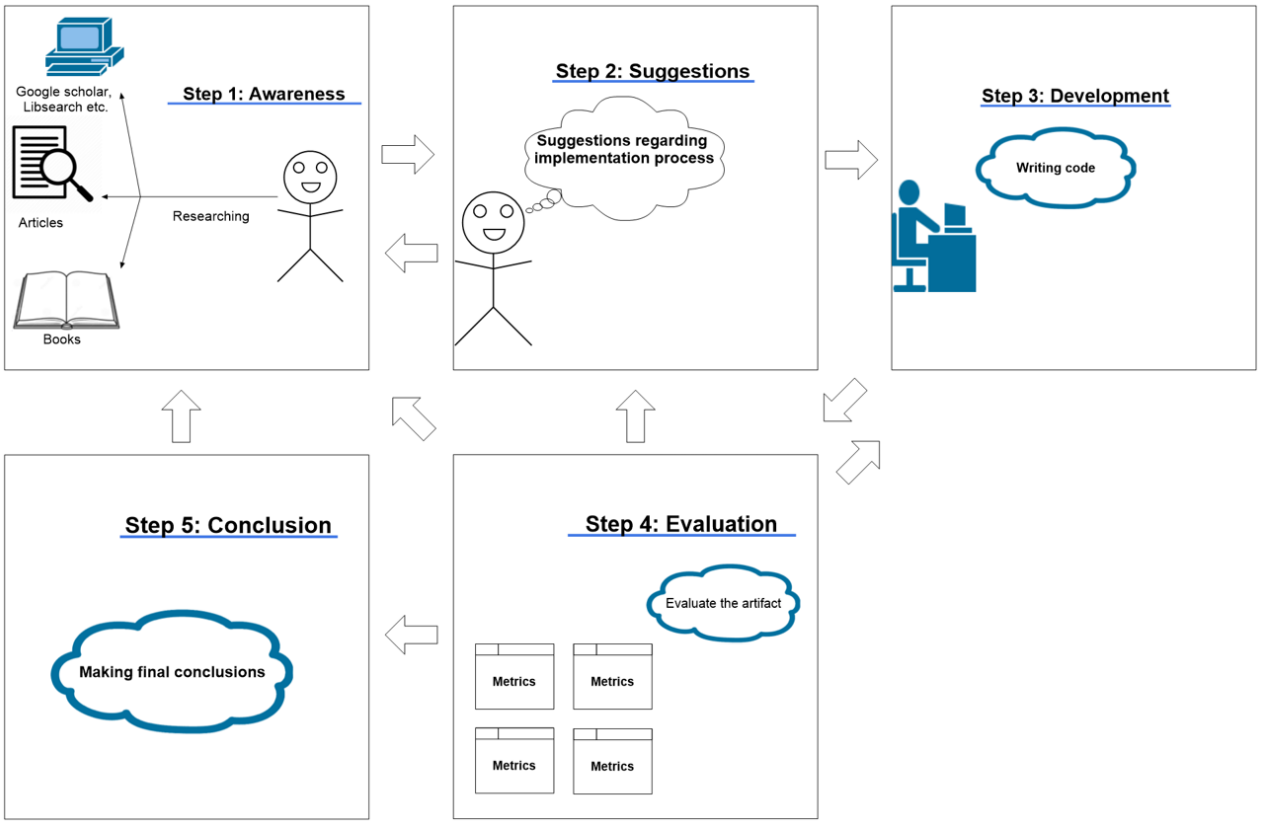



*Figure 1 - A visualization of Vaishnavi and Kuechlers 5 step process*

## 3.2 Evaluation by experiments

Evaluation of the level generator and its output levels takes place through experiments where the observations will be used to generate quantitative data about the levels. An experiment is a strategy where one examines the link between factors and an outcome [23, Ch. 9]. Experiments are also suggested as a potential evaluation method within design science by Hevner et al. [24]. In our case the relevant factors are the parameters of the level generator, such as those relating to the CA, GA and its fitness function, which can be tweaked into steering the algorithm to generate CA rules capable of creating levels with the desired properties. An example being comparing two separately evolved CA rules, each evolved using the same fitness function but with a slight change to the parameters, e.g. having the CA rule run for a different number of iterations on each starting state, to compare outputs. The outcomes are the generated levels. A run of the level generators instantiation will be considered an experiment for this thesis.

The goal of the experiments is also to generate quantitative data on the outputs of the level generator. The levels generated will be measured by a set of metrics. The metrics used in this thesis are: shortest solution path from the start to end of the level, number of dead ends, percentage of traversable cells in the level and the number of unreachable cells. Shortest solution path and number of dead ends are based on the fitness functions, as described in chapter 5.5. Percentage of traversable cells is the number of traversable cells divided by the total number of cells in the grid. Unreachable cells are defined as cells that are traversable, however with no path to them from the start point. For instance, a travesable cell that is surrounded on all sides by non-traversable cells.

The measured metrics help to better understand the quality and type of the generated levels. For example, if a large number of traversable cells in a level are unreachable, that level can be considered to be of lower quality. The percentage of traversable cells metric helps understand the type of the generated level. A higher number of reachable, traversable cells may reasonably lead to more open, “room-like” areas in the level, whereas a lower percentage of traversable cells may lead to more narrow, claustrophobic corridor type levels. Which one of these is prefered depends naturally on the type of game that the generated levels are meant to be used in. The first two of the metrics used in this thesis also relate directly to our proposed fitness functions, in order to assess the fitness functions impact on the levels. In addition, metrics relating to the GA itself will be gathered. The gathered data will be the most fit member and average fitness of the population over the run of the algorithm, in addition to the number of generations that the algorithm ran and the cause of its termination. No data will be gathered regarding the time it takes for the algorithm to run, as optimization of an implemented GA is not one of the goals for this thesis.

An interactive evaluation with a group of specially selected participants, such as game developers or players, could be an alternative, or additional, evaluation strategy. Players who have prior experience with playing video games and especially those who have played top down 2D games before would be prefered since our generator will produce those kinds of levels. Data would be gathered either through interviewing the participants directly, conducting a questionnaire or a combination of the two and then letting them rate the generated dungeons with scores based on how challenging, fun and aesthetically pleasing the dungeon is. This type of evaluation could have been used by itself or as a complement to our main experiment strategy. However, this type of interactive evaluation was passed over in favor of a quantitative approach for this research. One reason was due to the existing literature providing us with some guidance on the type of maze-like levels that users find interesting [9], [12]. In addition, we also had to anticipate the prospect that our suggested method for level generation would not produce viable levels at all. In the case that the evolved CA rules would, for instance, only generate unsolvable levels or highly impractical ones (eg. levels where the only reachable traversable area is a single short, straight solution path) it would be unnecessary and not viable to conduct interactive evaluation. Thus for this article we were more interested in quantitative metrics to guide us in understanding the type of levels that the level generator was capable of creating.

Evaluation could also have been conducted using an expressivity analysis [1, Ch. 12]. In such an analysis the range of content that the generator can produce is examined by running the generator for a large amount of times and measuring the created content by a set of metrics. Using the metrics one could then define expressivity measures, with the help of which an expressivity analysis could be conducted to illustrate the generator's expressive range [1, Ch. 12]. Expressivity measures could be the leniency and complexity of a dungeon. These measures would have been assigned a score based on different features of the dungeon. The leniency score would be determined by difficulty of the level with a range between 0 (lowest difficulty) and 1 (highest difficulty), and the other one will measure how complex it is using the same score interval. The scores could be based on eg. traversable space and solution path length. The next step would have been applying these measures to all of the generated outputs and finally visualizing the generative space. Expressivity analysis was not chosen as an evaluation method for this thesis as we were more interested in the initial viability of our proposed approach of using evolved CA on a collection of random starting states for level generation, rather than showcasing the full range of content that it may generate.

## 3.3 Alternative methods

An alternative to design science was to do an extensive literature review on evolved CA for game levels in order to gather enough existing data to conduct our thesis. In this case no development would be needed, rather the research would be conducted by studying the existing papers and research regarding evolve CA for game level generation. Having a literature review is important in order to gather a knowledge base but it's not enough to sufficiently answer our second research question. A literature review would also mean that no new knowledge would be created rather it would be an accoúmulation of already existing information presented in a unique way and used in our thesis to answer the research question. It would suit better where the research question is formulated differently. Togelius et al. [31] identified that, at the time of their research, no prior textbook or overview paper on PCG taxonomy existed. In their survey paper they made distinctions within different PCG terms and other in depth explanations about different terms within the domain. They also explained games which in some way had PCG features implemented [31]. One can argue that no new knowledge was created, rather a reservoir of basic PCG taxonomy gathered in one article since no such collection existed before. However, as there are not enough existing papers on evolving CA rules for game levels, it would not be viable to conduct our research entirely by a literature review. For this reason the literature review was excluded as the sole strategy.

Another approach would have been to conduct a survey [23, Ch. 7] with game developers and players to further investigate the type of levels that users find interesting and desirable. The results of the survey could have been used to guide
This approach was not selected for two reasons. The first one is that a previous paper using an interactive GA to evolve CA rules for maze generation provided us with some guidance on what properties the levels should have. The second reason was that, given that our approach of using random starting states works, generating other types of levels would be a matter of designing different fitness functions. Changes to the fitness function, such as additional or different ones, can reasonably be expected to lead to generated levels with different properties.

# Chapter 4 - Insights from the state of the art

In this section we will discuss the state of the art regarding using a GA to evolve CA rules for level generation purposes, and their effects on the choices in our research and design of the GA used in this paper. From the previous research we can initially understand that cellular automata are suitable for level generation in games [8],[9],[11],[17],[12]. Of particular interest are the studies where the generated level types (mazes or caves) were similar to our own proposed idea of generating maze-like levels [8],[9],[11],[12]. Previous findings also affirm that evolving cellular automata rules, for level generation, using a genetic algorithm, can produce interesting results [9],[11],[12].

Furthermore, the previous research guides us in our process of designing an appropriate fitness function for our genetic algorithm. Part of the fitness function will be that, the longer a level's shortest solution path is, the higher its fitness rating will be. Additionally, a larger number of dead ends . These decisions were based on findings in previous research showing that users found these to be desirable qualities for maze levels [9]. In addition, the previous research forms the basis of our decision to use a direct representation for the CA rules [11]. Other representation types used were probabilistic [12], and indirect [9],[11],[12]. Probabilistic representation entails that for each neighbourhood configuration, there is a certain chance that decides if the center cell should be on/off, thus providing variety. However, in our work we achieve this variety by means of randomized starting states, whereas in [12] CA's were evolved separately for two specific starting states. The indirect representation on the other hand were explained to be more suitable for when multiple states exist [11], thus not chosen since we only have 2 states.

Regarding the parameter settings and operator types chosen for our genetic algorithm, these settings used in [9] are of the least relevance, due to it using an interactive selection method, which affects other parts of the GA design, such as necessitating a smaller population and max number of generations the algorithm is run, in order to prevent user fatigue. Special relevance is given to Pech's [11] work, in part due to our decision to use a direct representation for our CA rules. In addition, as part of the fitness function of their GA, they ran their CA rules on a collection of 100 starting states and evaluated the resulting levels. This is closer to our proposed idea of running the CA rules on a collection of randomly generated starting states, compared to Adams' [12] work which only had two starting states, with the CA being evolved separately for each.

The genetic algorithm used in this paper will use tournament selection in combination with elitist selection, and a single-point crossover operator with a 60% crossover chance. The previous research shows that these selection and crossover yield good results for this type of problem [11], and using elitism with another selection method is also suggested by Mitchell [4, p. 126] as a way of improving a GA's performance. Additionally, tournament selection was named as a good performing selection method by [19]. The decision to look closely at a similar work in deciding our crossover method is also supported by [22], which stated that in choosing a crossover method one should examine previously applied GA's in the related application area. Another factor in favour of using this crossover method is the fact that we have the same CA representation as Pech's [11] work. It should be noted, however, that while one of the stated benefits of tournament selection was the fact that no prior sorting of the candidates was needed [19], this benefit will be annulled in our algorithm, due to sorting being required for the elitist selection of the top candidates. Additionally, point-mutation with a probability of 1 over

the length of the candidate array will be employed. As this mutation method was used in the three examined articles that used evolved CA for level generation [9],[11],[12], we judge it to be an appropriate mutation method for this type of problem. The mutation probability is inspired from Pech's [11] work, which also had a mutation probability of one divided by the length of the array.

# Chapter 5 - The level generator

In this chapter we present the level generator created for this thesis. The purpose of the generator is to output a set of interesting, 2d maze-like levels. The level generator is meant to be used during the game development process to create levels that can be added to a game and potentially populated with, for instance, items and enemies.

The level generator consists of a genetic algorithm that is used to evolve CA rules. These CA rules are used to generate game levels, and the GA's fitness function will steer the evolved rules toward generating appropriate game levels based on the defined fitness functions. Unlike previous research, our CA will be applied to a set of random starting states. By random starting state we mean that each individual cell in the grid has a random chance of initially being either traversable or non-traversable. In previous research where CA rules were evolved, the starting states were either evolved together with an attached CA rule [9], or a collection of perfect mazes [11], or blank grids or alternatively grids with predefined sections of the grid being traversable [12].

The output levels should be of high enough quality to be used in games as e.g. dungeon levels. Further, as defined by the fitness functions, they should have a long shortest path between the start and end points, while also having a large number of dead ends. Naturally, this means that it becomes a non-trivial task to traverse the levels, compared to if there had been an e.g. straight path from start to end. Based on previous research [9], this contributes to making maze levels interesting to play.

To put our level generator within the larger context of PCG, we use a set of taxonomy defined in [1, Ch. 1] and explain how our level generator fits within these. Accordingly, our generator is meant to be used *offline*, i.e during the game development phase as opposed to during the runtime of the game. Further, it creates *necessary* content, as game levels are a fundamental part of games, as opposed to *optional* content that are not required for the game to function. The generated content is *generic* and not *adaptive*, as it is not specifically made based on a players play style or interactions in the game. However, the users of the generator may steer the generated levels toward certain properties by means of altering the parameters or fitness functions. Furthermore, the level generator is *stochastic* as compared to being *deterministic,* since randomness is involved at multiple stages such as during starting state generation and within the GA's mutation and crossover operators. Thus two runs of the level generator with the same parameters will not result in the same outputs. Further, since the level generator generates a level, tests the quality of it and repeats this process iteratively through the course of the algorithms run it goes in line with generate and test rather than being constructive. Additionally, our generator leans heavily toward *automatic generation*, as the user is only initially allowed control over certain starting parameters before the GA is started. This is contrary to *mixed authorship*, in which a designer is involved during various stages of the generation process and in a sense collaborates with the generator. Another part of the taxonomy is a generator's *degree and dimensions of control*, i.e to what extent the output can be controlled by the user. As previously mentioned, users of the generator have some control over the properties of the output levels as they can alter different parameters or by designing different fitness functions altogether, leading to generated levels with different properties. However, the stochastic nature of it can be considered to lead to reduced controllability.

In the rest of this chapter the various parts of the level generator are explained. As a first step, our definition and representation of cellular automata is described. Then

follows a description of our genetic algorithm and its various parts. Finally we explain the process of setting up the generator before conducting the experiments.

## 5.1 Cellular automata definition

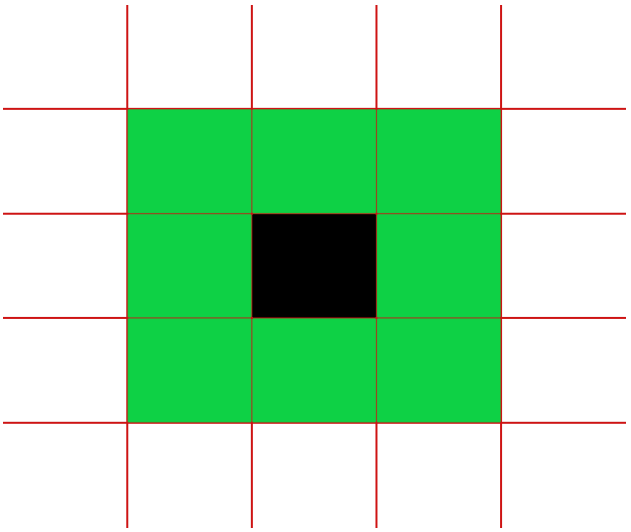


*Figure 2- Moore's neighbourhood definition. The neighbourhood consists of the black center cell and its green neighbours.*

For our neighbourhood definition we use Moore's neighbourhood [7], as visualized in figure 2, with a range of 1. According to this neighbourhood definition, all cells that are either orthogonal or diagonal to the center cell count as its neighbour. For our research we use a range of 1, meaning only the cells closest to the center count as its neighbour, resulting in a total of 9 cells in the neighbourhood. The CA will be run on randomized starting states for a certain number of iterations. In each iteration, each cell in the grid is looped through. For each cell, its neighbourhood configuration is retrieved and compared to the rule array. In the rule array each possible neighbourhood configuration is mapped to an outcome, for example it might be that if the center cell is on and all 8 of its neighbours are off, the center cell should be updated to off as well for the next iteration. The pseudocode for this process is given in figure 3. The rule arrays representation is more clearly explained in the next section. The starting states of the cells in the grid that the CA will be run on will be randomized, with each cell having a 50% chance of being on or off initially. An important part of CA implementation is its representation.

```
applyCellularAutomataRules(grid, rules, iterations)

    temp = copy of grid

    for i=1 to iterations {
        for each cell in grid {
            retrieve cell's neighbourhood configuration
            compare configuration to rule array
            update cell state in temp array
        }
        grid = temp
    }
    return grid
```

*Figure 3 - pseudocode explaining how a CA rule is applied to a grid for a certain number of iterations.*

## 5.2 Cellular Automata representation

The rules of the CA can be represented using a bit array in two ways, direct and indirect [11]. With indirect representation, only the sum total of on or off neighbours is considered by the rules, and not their placement. That is, if 5 of the cells in the neighbourhood are on, this is considered as one state, with no regard to the positions of these 5. As mentioned in [11], this is useful when using a CA with multiple possible states for each cell, in order to have a representation of a manageable size. However, as we in our case only have 2 states for each cell, on (traversable cell) and off (non-traversable cell), we will use a direct representation. In a direct representation, both cell states and positioning of the cells in the neighbourhood matters. Thus to represent a neighbourhood of size 9, where each cell can have one of two states, one needs a 2^9=512 bit rule array in order to represent all possible states. The value stored in the rule array cells define if, for a given neighbourhood configuration, its center cell should be updated to an on or off state. Since the array has 512 cells each with 2 possible states, there are 2^512 possible combinations for the contents of the rule array. This means that the genetic algorithm will have a search space of 2^512 as well. Next we explain the starting states that the CA's will be applied to.

## 5.3 Starting states

Each starting state can be represented through a binary array where an element in the array will correspond to a cell in the 2d square grid that is our level. This means that the size of the binary array is directly relational to the size of the grid. In our case we ran our 6 experiments on 30x30 grids. In the binary array a cell with the state 1 represents a traversable cell, and a 0 represents a non-traversable cell. The starting states were initialized using a random number generator, where each cell was traversed and had a certain chance of being initially on. For each starting state, a start point at the bottom left corner and an end point at the top right corner are pre-defined. These represent the path that the user is supposed to take through the level. The genetic algorithm will evolve CA rules that can transform these random starting states to game levels.

## 5.4 Genetic Algorithm

The genetic algorithm implemented consists of a population of 50 candidate solutions. In this case, each candidate in the population is made up of a CA rule array, represented as described in the previous section. Each element is initialized with a random state, 0 or 1, with the probability of each state being 50%, representing a traversable or non-traversable cell respectively. At this stage a collection of random starting states are also created. The probability of a cell in the starting state being traversable was initially set to 50%.

The GA is then run for a given number of generations. For each generation of the algorithm, the candidates are assessed by the fitness function and given a fitness score. Based on the selection method, candidates are chosen to either directly, or after undergoing the crossover and mutation steps, go through to the next generation. The pseudocode of this process is presented in figure 4.

The genetic algorithm is terminated once a specified termination condition has been reached. In the experiments performed in this paper two such conditions were present, max generations or convergence reached. Max generations is activated when the GA has been run for a certain number of pre-defined iterations. The convergence condition entails that the fitness score of the most fit member of the population has not improved over the course of a specified number of subsequent generations. During the implementation process, a third condition was also used, called max fitness. This condition was true when the most fit member of the population had achieved maximum possible fitness, i.e the optimal solution. However, max fitness was not used during the experiments.

In the parts of the selection, crossover and mutation methods where a random number generator is required, Unity's random number generator is employed [32]. This class is capable of generating floats with up to 7 decimals. The various parts of the GA are explained in more detail in the following subsections, starting with the fitness function.

```
Initialize collection of random starting states
Initialize population of candidate CA rules

while(!termination condition)
{
    Apply each CA rule on each of the starting states to generate levels
    Run fitness function on the generated levels to score each CA rule
    Sort population based on fitness score
    GenerateNewPopulation()
}

GenerateNewPopulation{
    Create new empty population
    Copy elite members to new population
    while(new population < population size){
        Candidate 1 = winner of a tournament selection
        Candidate 2 = winner of a tournament selection
        Perform crossover step on candidates 1 & 2
        Perform mutation step on candidates 1 & 2
        add candidates 1 & 2 to new population
    }
}
```

*Figure 4 - pseudocode of the genetic algorithm's main steps*

## 5.5 Fitness Function

After the initialization all candidates in the population are evaluated by the fitness function (FF) and sorted by their fitness score. A candidate CA rule array is applied to each of the randomized starting states for a specified number of iterations, resulting in a collection of game levels. These generated levels are then assessed by the fitness function in order to determine a fitness score for the given CA rule. This process is repeated for each candidate in the population. Attributes that are measured by the FF in the generated levels are the length of the shortest possible path between the starting point and the end point (FF1) and the number of dead ends (FF2). According to [9] and [12], these are features that users find desirable in maze-like levels. In order to find the shortest solution path between the start and end point, a version of the breadth first search (BFS) algorithm, as described by [33], is used to find the shortest path between the start and end points. Traversing the level can only be done in the directions up, down, left and right, i.e no diagonal movement is possible.

Our definition of a dead end is the same as the described by Adams [12], in which a dead end is a "a map cell that has no neighboring cell with a longer path length to the entrance cell". To calculate the number of dead ends, the previously mentioned BFS algorithm was run on each traversable cell, to find its shortest path to the starting point. Subsequently, each traversable cell is iterated and for each point its shortest path to the starting point is compared to its neighbours. If none of its neighbours have a longer path, it is counted as a dead end. The pseudocode for FF2 is shown in figure 5. The total fitness score of a generated level is given by its total number of dead ends +

shortest solution path. After being assessed by the fitness function, the GA proceeds to the selection step.

```
FF2(level array){

    for each cell in level{
        if cell is traversable{
            run BFS from cell to start point
            store cell's shortest path length to start
        }
    }

    for each cell in level{
        if all 4 neighbours have a shorter path{
            deadEnds++
        }
    }
    return deadEnds
}
```

*Figure 5 - pseudocode for FF2*

## 5.6 Selection

Once each candidate has been given a fitness score by the fitness function, they are then sorted from best to worse. The selection process is constituted of two parts, an elitist selection followed by tournament selection. Initially, an elitist [4] selection, as presented in figure 6, is made of the top 6 candidates of our population. These 6 candidates are *copied*, unchanged, to the new generation, skipping both the crossover and the mutation steps. Elitism guarantees that the candidates with the highest FF in each population have no change applied to them between generations. Thus ensuring that the fitness of the best candidates can not *decrease* between generations.

```
CellularAutomata[] Elitism(CellularAutomata[] oldPop, CellularAutomata[] newPop, int elitismSize)

    for i=0 to elitismSize-1 {
        newPop[i] = copy of oldPop[i]
    }

    return newPop
```

*Figure 6- elitist selection pseudocode*

The remaining candidates will be selected through tournament selection [11].
In tournament selection a number of random, unique, candidates will be chosen from the entire population. In this paper tournament sizes of 5 and 2 were used. They are then sorted and the candidate with the highest fitness score will continue to the crossover stage. Note that a candidate can be selected, and be the winner of, multiple tournaments. Tournament selection is repeated until the new population has been filled and reached the maximum number of candidates. The pseudocode for tournament selection is given in figure 7. The candidates that won in the tournament selection proceed, two at a time, to the crossover and mutation steps.

```
CellularAutomata TournamentSelection(CellularAutomata[] population, int tournamentSize){

    CellularAutomata[] tournamentArray = new CellularAutomata[tournamentSize]

    for i=0 to tournamentArray.Length-1 {
        while(randomCandidate does not already exist in tournament array){
            randomNbr = Random int in range [0,population size]
            candidate = population[randomNbr]
            check if candidate already exists in tournament array
        }
    Sort tournamentArray in descending order of fitness
    return tournamentArray[0]
}
```

*Figure 7, tournament selection pseudocode*

## 5.7 Crossover

After each pair of tournaments, the two winners, called parents, undergo the crossover stage together. In this thesis single-point crossover [4] is employed as the crossover method. The pseudocode for crossover is presented in figure 8. The probability for crossover is set at 0.6, i.e it should happen 60% of the time. A random number is generated between 0 and 1. If this random value is equal to or less than the crossover probability, then crossover will occur. Otherwise the two candidate solutions will continue unchanged to the mutation step. If crossover is to occur, a crossover point is randomly generated between 1 and the length of the array minus one, i.e the last bit of the CA rules array. Two new candidate solutions, called children, are then created. The first child contains all the bits from the first parent from position 0 to the last bit before the crossover point, and all the bits from the second parent from the crossover point until the end of the array. For the second child the process is repeated, except that the parents swap places. Note that the random number generator for the crossover points omits the point 0, as in that case crossover would happen over the entire length of the candidates, resulting in two new candidate solutions that are exact copies of the previous two. The members in the crossover step, irrelevant of whether crossover actually occured between them or not, continue to the mutation steps.

```
singlePointCrossover(CellularAutomata candidate1, CellularAutomata candidate2, crossoverProbability){
    randomValue = Random number in range [0,1]
    if(randomValue > crossoverProbability){
        return (candidate1, candidate2)
    }
    crossoverPoint = Random integer in range [1, candidate.length-1]

    for i=0 to crossoverPoint-1 {
        child1[i] = candidate1[i]
        child2[i] = candidate2[i]
    }
    for i=crossoverPoint to candidate.length-1 {
        child1[i] = candidate2[i]
        child2[i] = candidate1[i]
    }
    return(child1, child2)
}
```

*Figure 8 - Pseudocode over single-point crossover implementation*

## 5.8 Mutation

Point mutation will be used in the GA of this thesis, as shown in figure 9, with a mutation probability of 1/512, i.e one divided by the length of the CA rule array. In theory this leads to an average of 1 mutated cell for each time a candidate is put through the mutation step. The mutation procedure is as follows: For each cell in the array, a random number is generated between 0 and 1. If this random number is equal to or below the mutation probability of 1/512, the state of that cell is flipped. That is, a cell with the value 1 (traversable) is changed to 0 (non-traversable), or vice versa. Note that in the implementation, the number 1/512=0.001953125 is rounded to a number with 7 decimals, i.e 0.0019531. In the next subchapter the process of implementing and setting up the level generator is explained.

```
CellularAutomata Mutate (CellularAutomata candidate)

    mutationProbability = 1/512

    for each cell in candidate CA rule array{
        randomValue = Random number in range [0,1]
        if(randomValue <= mutationProbability){
            flip value in current cell
        }
    }
    return candidate
```

*Figure 9 - mutation method pseudocode*

## 5.9 Implementation Process

The unity 2D [34] game engine and the C# programming language were used for step 3, development of the design science process. Then an iterative process between development and evaluation was done to set up the generator for the subsequent experiments.

A first phase included coding the selection, crossover and mutation operators. Evaluation was done by testing them to ensure that they worked as intended. A second phase involved running the algorithm based on a simple fitness function. This FF aimed to evolve CA rules capable of generating an entirely traversable level, based on a single random starting state of size 30x30, with 2 cells pre-defined as start and end points. That is, the fitness score of a CA rule would be the same as the number of traversable cells in the level it had generated. Each CA rule was applied only once on the starting state. Max fitness was thus defined as when a CA rule would generate an entirely traversable level, i.e 898 for a 30x30 grid, when not considering the starting and end points. The algorithm would stop either when either max fitness or max number of generations was reached.The main goal of this phase was to check that the GA as a whole worked as intended and that the population correctly evolved toward a higher fitness score based on the fitness function. A minor goal of this phase was the beginning of a process of fine tuning the max generations parameter.

Table 1 shows a run of the algorithm that steadily evolves the automata rules, with its output showcased in figure 10. However, it was abruptly ended by the max number of generations being reached, before it had achieved maximum fitness. Thus it was run again with the same parameters as before, with the exception of the max number of generations being set to 1000.

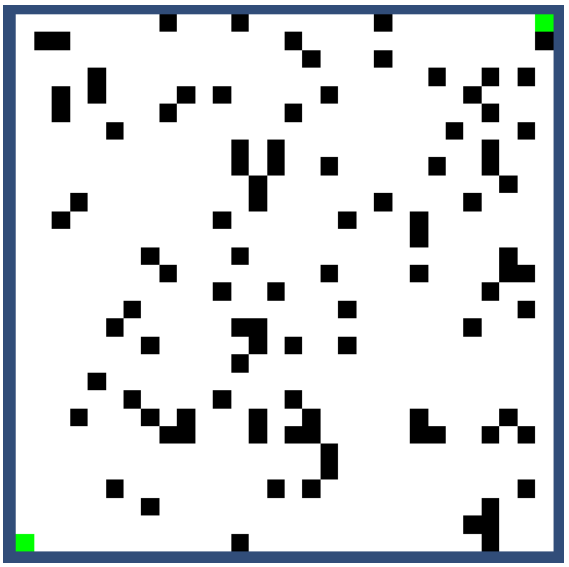


*Figure 10 - Max fitness not reached. Black cells are non-traversable, white cells are traversable and green cells are start & end points. Max generations = 100. Population size = 50, Tournament size = 5, elitism size = 6*

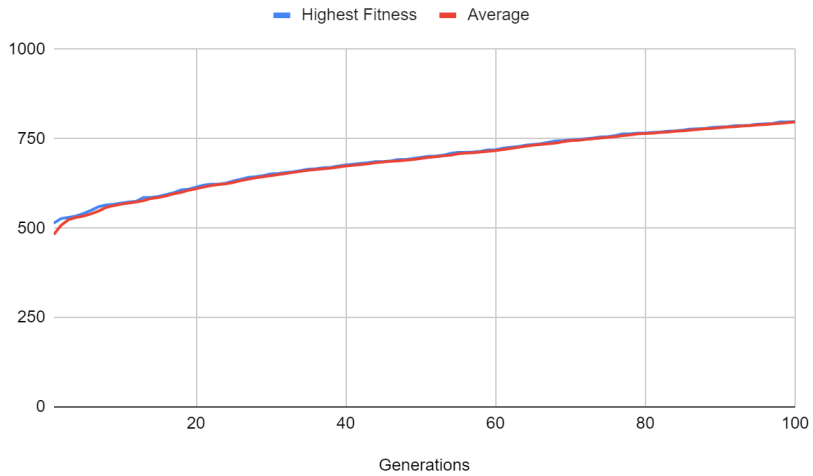



*Table 1 - Both highest and average fitness increased steadily, but the algorithm was prematurely ended by a low number of max generations.*

However, the results, as shown in table 2, indicated that the higher number of generations did not help to improve fitness after about 200 generations. The output level from that GA run is shown in figure 11. This unnecessarily slows down the time it takes for the algorithm to conclude. Thus, inspired by Pech's [11] work, a third termination condition for the GA was introduced. The third condition, convergence, would stop the algorithm if the fitness score of the most fit candidate of each generation had not improved at all over the course of 100 generations. A benefit of thi

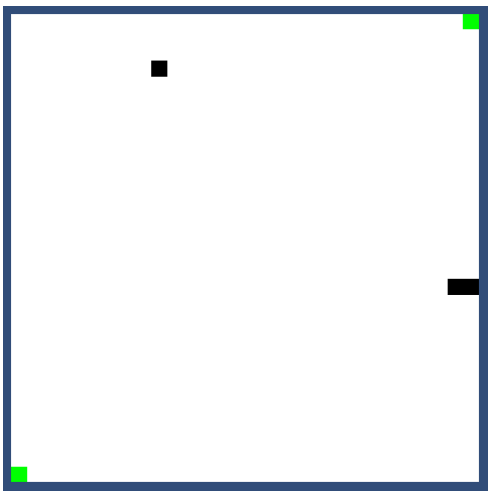


*Figure 11 - Max fitness not reached. Black cells are non-traversable, white cells are traversable and green cells are start & end points. Max generations = 1000. Population size = 50, Tournament size = 5, elitism size = 6*

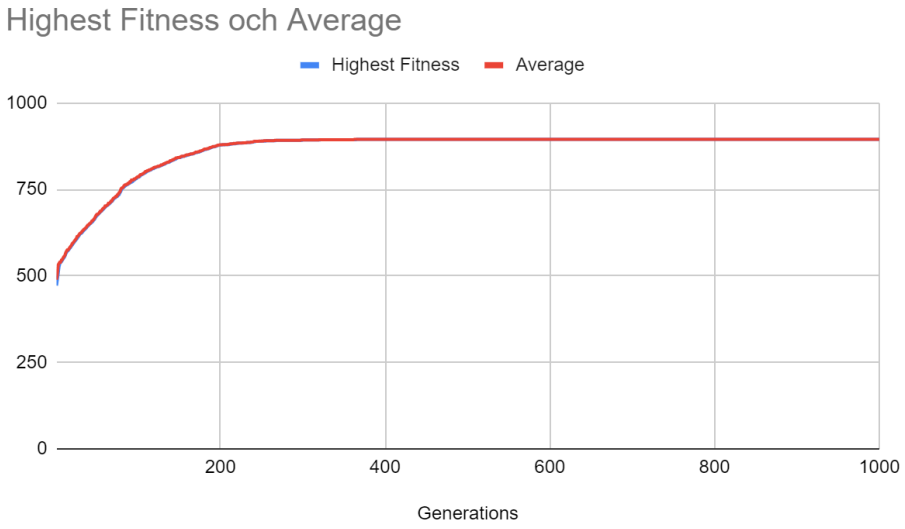
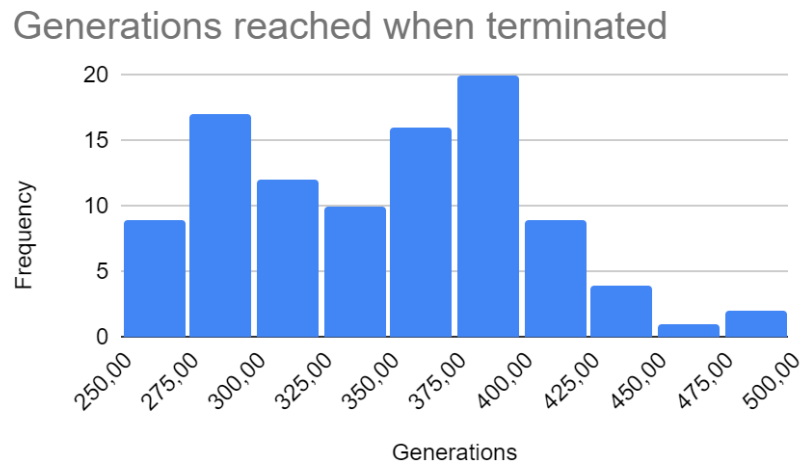
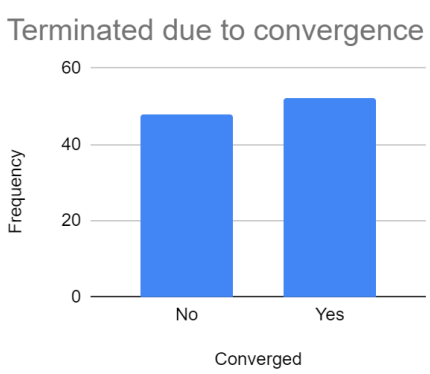



*Table 2 - The population converges after around 200 generations. Due to highly similar values of the metrics, the blue line is hidden.*

Subsequently, a minor experiment was conducted where the GA was run for 100 times, with the aforementioned parameters. The goal was to understand and evaluate the newly implemented convergence termination condition. The results of this experiment, presented in table 3, show that the convergence termination condition was effective in preventing the algorithm from running unnecessarily long. It also showed that no run lasted for longer than 500 generations. Thus the algorithm always terminated for one of two reasons: in 52 cases it was due to convergence, and in 48 cases due to max fitness being achieved. All of the runs came fairly close toward the max fitness, with the lowest recorded fitness value being 888.



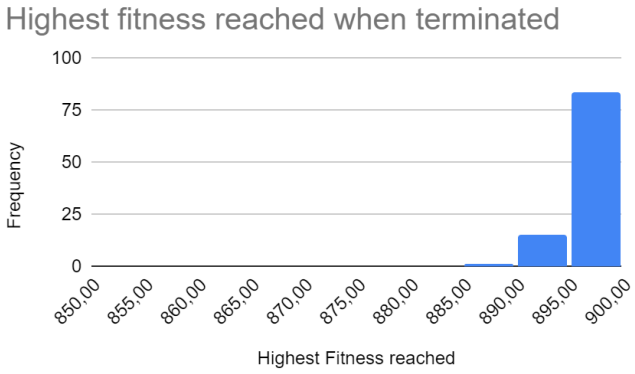




*Table 3- Results of 100 runs of the GA for max generations = 1000, population size = 50, starting state amount = 1, tournament size = 5, elitism size = 6, convergence defined as no improvement of most fit individual in population over 100 generations.*

When we ran the algorithm, with the convergence cap raised to infinity and the maximum number of generations at 5000, we concluded that the majority of them reached their max fitness score after about 300 generations and the rest of them failed to evolve their fitness beyond this point. In *table 4,* the algorithm was conducted 10 times and shows that the GA failed to improve fitness if the target fitness wasn't reached within a certain amount of generations. Five of them reached their maximum fitness after 314 generations on average and the rest converged for an average of 4708 generations, failing to make the last cell in the grid traversable. A test with 10000 generations showed similar results.

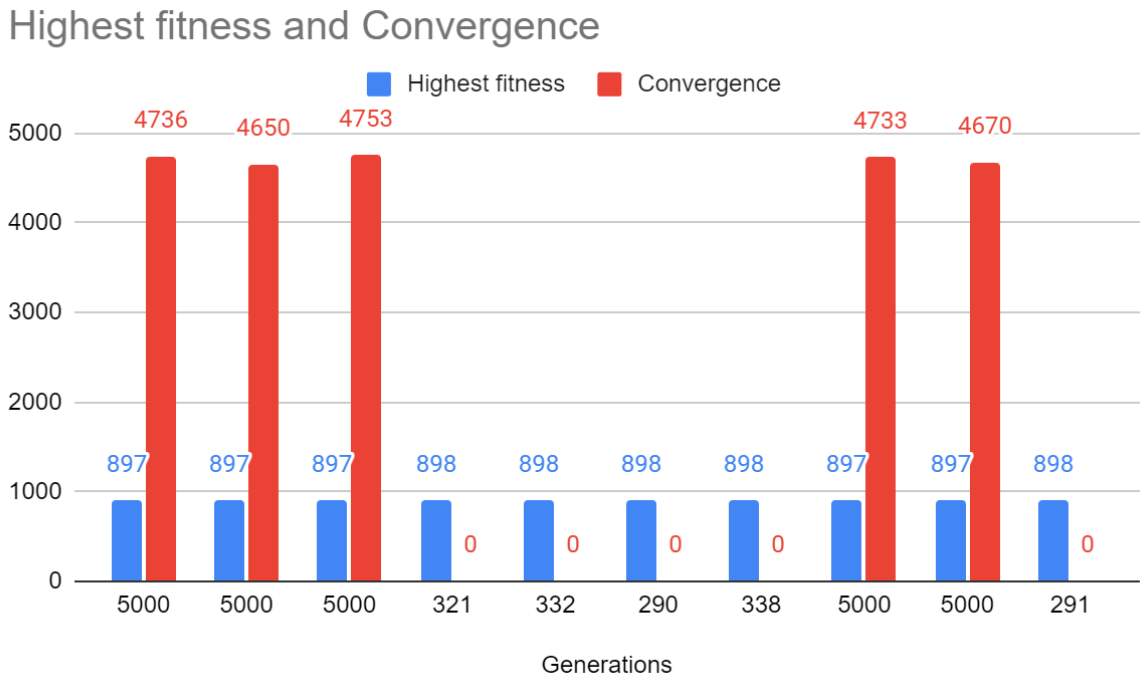



*Table 4 - Result of 10 runs GA for max generations = 5000, population size = 50, starting state amount = 1, tournament size = 5, elitism size = 6, with the convergence cap of 100 removed.*

Subsequently, FF1 was implemented and replaced the previous FF which consisted only of the percentage of traversable cells. It was tried with two starting states, seen in figure 12, both of which resulted in solvable mazes. The only termination condition used was max number of generations, which was set to 1000. For the mazes in *figure 12*, the algorithm had converged after 192 generations and showed no improvement for the subsequent generations. After this, FF2 was also implemented, finalizing the level generator for the performing of the experiments.

*Figure 12 - Two solvable mazes using FF1. Max generations = 1000, Population size = 50, tournament size = 5, elitism size = 6, starting states amount = 2*

# Chapter 6 - Experiment Results

Once the level generator had been completely implemented, a number of experiments were conducted in order to help answer RQ2. In each experiment, the GA evolved CA rules based on different parameters. The size of the starting states for all experiments was set to 30x30. Different parameters were used for each experiment, in order to test the effects of the different parameters on the final levels. The experiments and the levels generated by the evolved CA's alongside their quantitative metrics and related data will be presented in this chapter. The parameters for each experiment are presented in their subchapter.

## 6.1 Experiment 1

In this experiment the algorithm was run for a maximum of 1000 generations with an additional convergence termination condition, set to stop the algorithm if the most fit candidate had not improved over 200 consecutive generations. The algorithm ran on 10 randomly created starting states, with each cell in the starting states having a 50% chance of being on initially. Table 6 shows the full parameter settings for this experiment. The output levels can be seen in figure 13, with their corresponding metrics presented in table 5. Out of the 10 generated levels, only the 2nd one can be considered a failure, due to it lacking a solution path, thus rendering it unplayable. As a consequence of this, it also has an abnormally large amount of unreachable cells (417) and few dead ends (27). For the other levels, the number of dead ends and number of unreachable cells range from 78 to 94 and 52 to 149 respectively. All 10 levels showed similar degrees of traversable space and length of solution path, with the exception of level 8 which stood out with a considerably longer solution path than the rest.


*Figure 13 - The 10 generated levels in experiment 1*

| Experiment 1 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 60 | 59,33333 | 80 | 122 |
| 2. | No path | 59,33333 | 27 | 417 |
| 3. | 66 | 58,22222 | 78 | 112 |
| 4. | 62 | 59,22222 | 88 | 91 |
| 5. | 58 | 61,77778 | 94 | 96 |
| 6. | 62 | 57,11111 | 78 | 149 |
| 7. | 62 | 58,88889 | 91 | 86 |
| 8. | 92 | 59,33333 | 81 | 53 |
| 9. | 58 | 60 | 87 | 52 |
| 10. | 58 | 61,11111 | 89 | 78 |

*Table 5 - Metrics from experiment 1 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 5 |
| | Mutation probability | 1/512 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 50% |
| | Convergence limit | 200 |
| | Fitness score given by | FF1+FF2 |

*Table 6 - parameter settings experiment 1*

Regarding the GA's performance, initially one can conclude that the algorithm as a whole managed to improve the population, as seen in table 7 showing the value of its most fit member and average population fitness over the course of the GA's run. The algorithm failed to improve on its most fit candidate between generations 234-415, but it managed to improve after that before again failing to show any improvement for the last 200 generations, causing the convergence termination condition to stop it. It is possible that the algorithm may have again managed to improve on its most fit candidate had it not been terminated by the convergence condition. While this experiment rendered 9/10 playable levels, the one failed level and large number of unreachable cells were a cause of concern. As a consequence of this, for experiment 3 the number of CA iterations were doubled from 5 to 10, giving the CA rules more iterations to generate interesting levels.

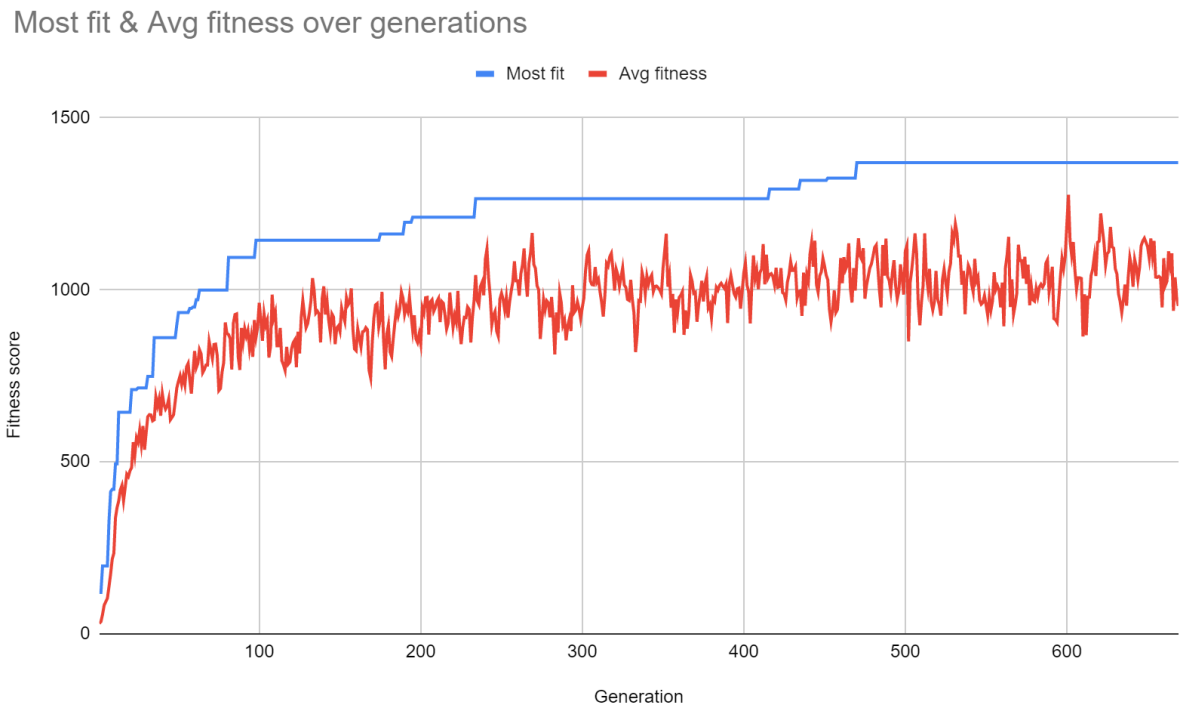



*Table 7 - Fitness score of most fit candidate and average fitness of population over the course of the algorithms run in experiment 1*

## 6.2 Experiment 2

In the second experiment we had the same parameters as experiment 1 with a change to the mutation probability. The main goal of the second experiment was that we wanted to see how a change to the mutation rate affected the algorithm compared to experiment 1. Instead of having a mutation rate of 1/512, we decided to increase it to a 5% chance. At a first glimpse of the 10 generated levels we immediately saw that the mazes felt more open than in experiment 1. Also, the utmost cells in each maze tended to be traversable to a much higher degree than in the first experiment. This gave the maps an outlining of open space thus making them feel easier to traverse from start to finish. One of the maps turned out to have no solution path. Figure 14 displays all levels.


*Figure 14 - The 10 generated levels in experiment 2*

Compared to the first experiment we can conclude an increase in the most fit candidate over all generations. Already after 117 generations the score of the highest fit candidate surpassed the highest fit candidate over all generations in experiment 1. Between generation 187-297 the fittest candidate failed to improve. However for the remainder of the algorithm the fitness increased slightly before converging and then terminating at generation 643. The average fitness showed similar results as in the first experiment. A conclusion based on the two experiments is that the increased mutation rate did little to no improvement to the average fitness score, however it's important to note that not enough data was gathered in order to make a definitive conclusion regarding a change to the mutation rate. The algorithm would have had to run more times to reduce the element of randomness and generate generalizable data. Table 8 displays all metrics from experiment 2 and table 9 all the parameters.

| Experiment 2 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 58 | 66,11111 | 123 | 42 |
| 2. | No path | 65,33333 | 117 | 25 |
| 3. | 74 | 64,88889 | 82 | 150 |
| 4. | 76 | 64 | 105 | 44 |
| 5. | 70 | 64 | 84 | 162 |
| 6. | 82 | 66,11111 | 109 | 31 |
| 7. | 62 | 65,66667 | 110 | 53 |
| 8. | 80 | 63,11111 | 76 | 155 |
| 9. | 58 | 64,44445 | 98 | 67 |
| 10. | 66 | 64,22222 | 97 | 78 |

*Table 8 - Metrics from experiment 2 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 5 |
| | Mutation probability | 0.05 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 50% |
| | Convergence limit | 200 |
| | Fitness score given by | FF1+FF2 |

*Table 9 - parameter settings experiment 2*

Overall both experiments showed similar results. The algorithm in the second experiment tended to generate mazes with more traversable cells given that all generated mazes had a slightly higher percentage of traversable space than in experiment 1. In table 10, the fitness across all generations is displayed.

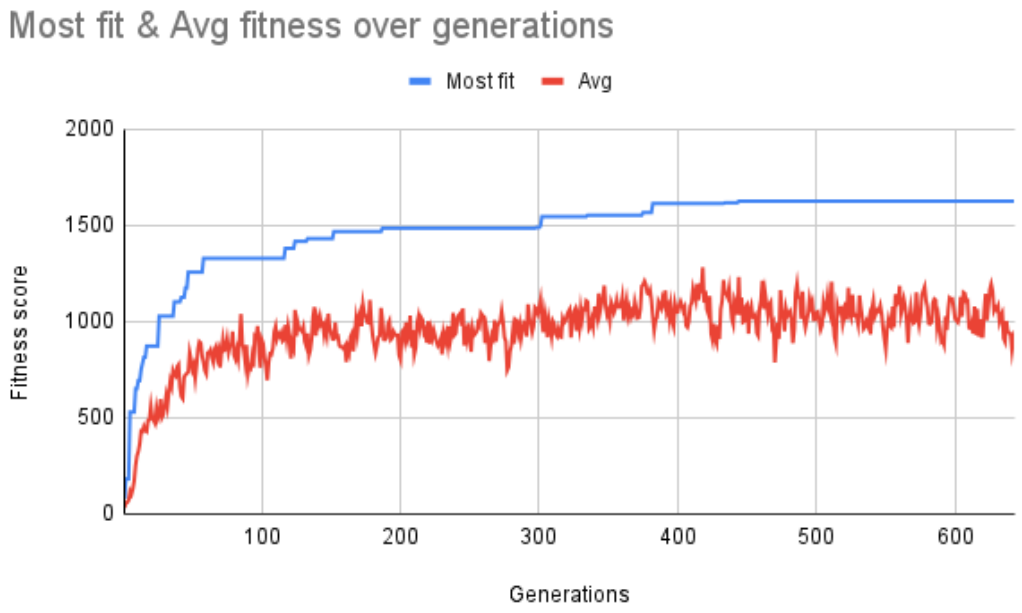



*Table 10 - Fitness score of most fit candidate and average fitness of population over the course of the algorithms run in experiment 2.*

## 6.3 Experiment 3

The third experiment was conducted with the same settings as experiment 1, with two changes, as indicated in table 12. First, an increase in the number of iterations that each CA rule was applied to each starting state, from 5 to 10. The convergence limit was also increased, from 200 to 300. This was done to give the algorithm more time for evolving the CA rules. An initial viewing of the levels, as seen in figure 15, indicate more open levels with less corridor-like areas. The level metrics, presented in table 11, do not show much difference between levels regarding the number of dead ends, ranging from 63 to 81. Much more difference was observed in the number of unreachable cells, ranging from 2 to 40. The length of the solution paths show uniformity, with all but level 9's length of 82, lying between 58 to 62. The percentage of traversable cells for the levels were overall higher than in the previous two experiments.

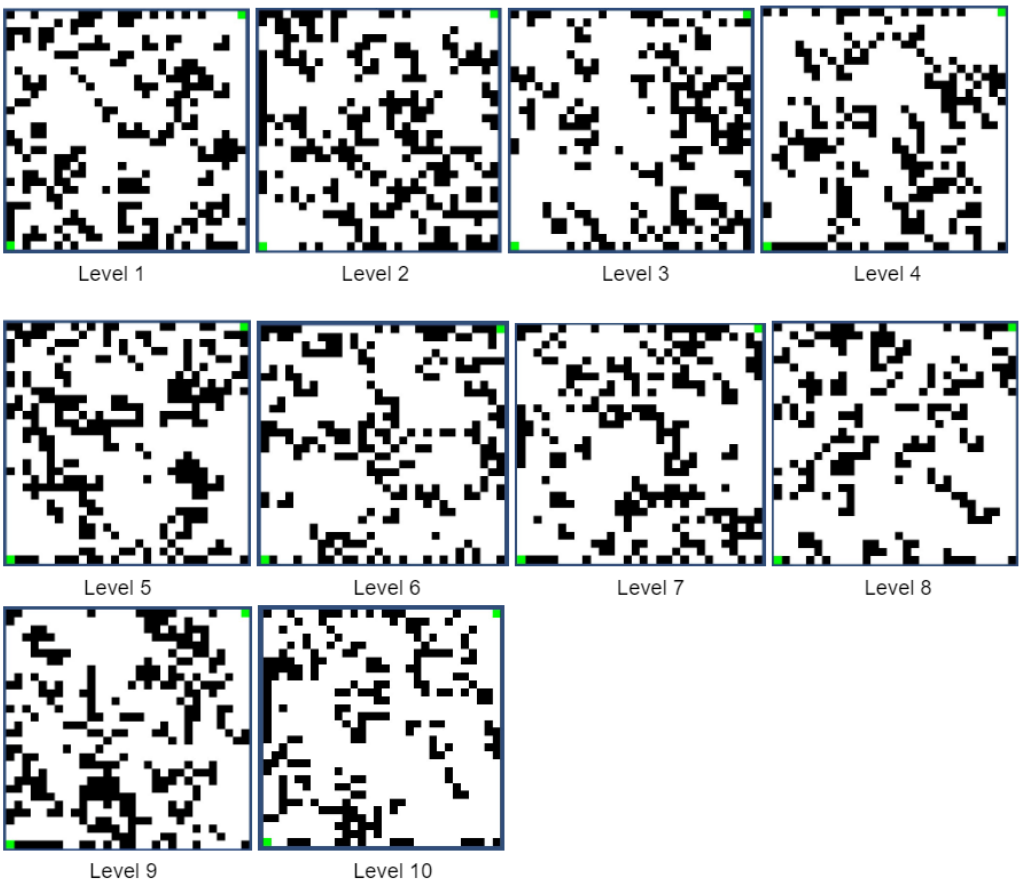



*Figure 15 - The generated levels in experiment 3*

| Experiment 3 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 58 | 72,66667 | 80 | 6 |
| 2. | 62 | 69,11111 | 80 | 40 |
| 3. | 60 | 75,33333 | 73 | 35 |
| 4. | 58 | 75,55556 | 78 | 11 |
| 5. | 58 | 70,77778 | 69 | 20 |
| 6. | 58 | 75,33333 | 81 | 14 |
| 7. | 60 | 73,22222 | 68 | 24 |
| 8. | 58 | 77,33333 | 63 | 7 |
| 9. | 82 | 71,44445 | 75 | 26 |
| 10. | 58 | 76,77778 | 70 | 2 |

*Table 11 - Metrics from experiment 3 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 10 |
| | Mutation probability | 1/512 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 50% |
| | Convergence limit | 300 |
| | Fitness score given by | FF1+FF2 |

*Table 12 - parameter settings for experiment 3*

The GA's performance, as presented in table 13, shows a rapid increase in fitness followed by two minor improvements, before being terminated by convergence at generation 599.

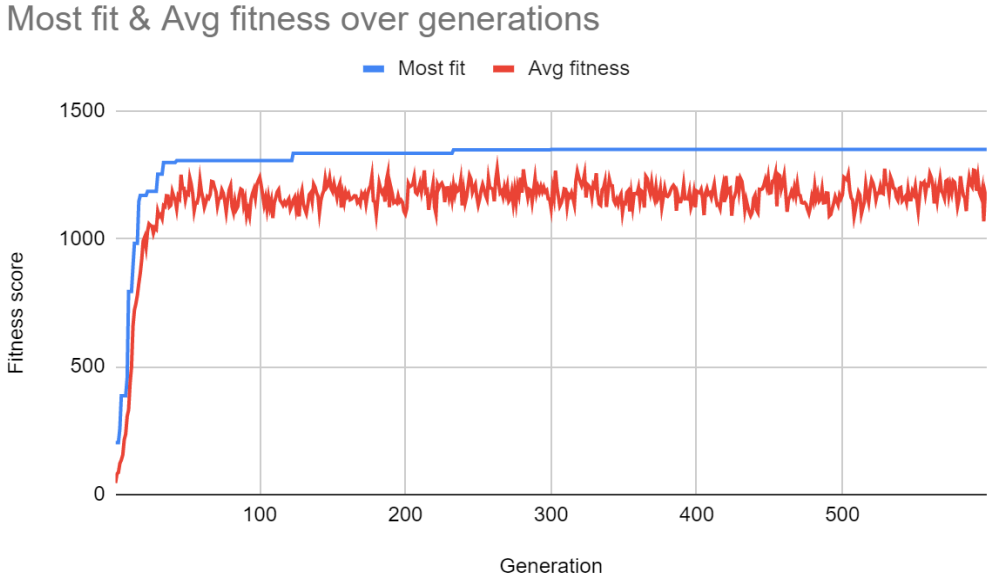



*Table 13 - Fitness score of most fit candidate and average fitness of population over the course of the algorithms run* in experiment 3

## 6.4 Experiment 4

For experiment 4 we wanted to investigate the effects of changes made to the starting states. In this experiment, the chance of each cell being on initially in the starting states was lowered from 50% to 25%, as shown in table 15. The expectation was that the generated levels would be more narrow and corridor-like. The generated levels can be seen in figure 16, with their metrics shown in table 14. As previous experiments, there is uniformity amongst the levels with regards to the percentage of traversable cells. However, traversable space was not as low as expected. The number of dead ends and unreachable cells ranged from 93 to 125 and 12 to 59 respectively. Length of the solution paths were in the interval 58 to 82 for the levels generated in experiment 4.


*Figure 16 - The generated levels in experiment 4*

| Experiment 4 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 62 | 61,55556 | 100 | 29 |
| 2. | 60 | 63,77778 | 120 | 26 |
| 3. | 60 | 64,88889 | 102 | 27 |
| 4. | 82 | 62,11111 | 111 | 59 |
| 5. | 58 | 63,77778 | 94 | 12 |
| 6. | 66 | 62,88889 | 98 | 59 |
| 7. | 64 | 63 | 93 | 20 |
| 8. | 66 | 62,22222 | 110 | 46 |
| 9. | 72 | 63,66667 | 125 | 33 |
| 10. | 66 | 63,33333 | 118 | 16 |

*Table 14 - Metrics from experiment 1 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 5 |
| | Mutation probability | 1/512 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 25% |
| | Convergence limit | 300 |
| | Fitness score given by | FF1+FF2 |

*Table 15 - parameter settings for experiment 4*

As in the previous experiments, the GA was terminated before reaching 1000 generations, due to the convergence condition being reached at generation 714. The last improvement for the most fit member was seen at generation 415. The GA's performance can be seen in table 16. We can also see that the population average increased steadily together with the most fit members' fitness score.

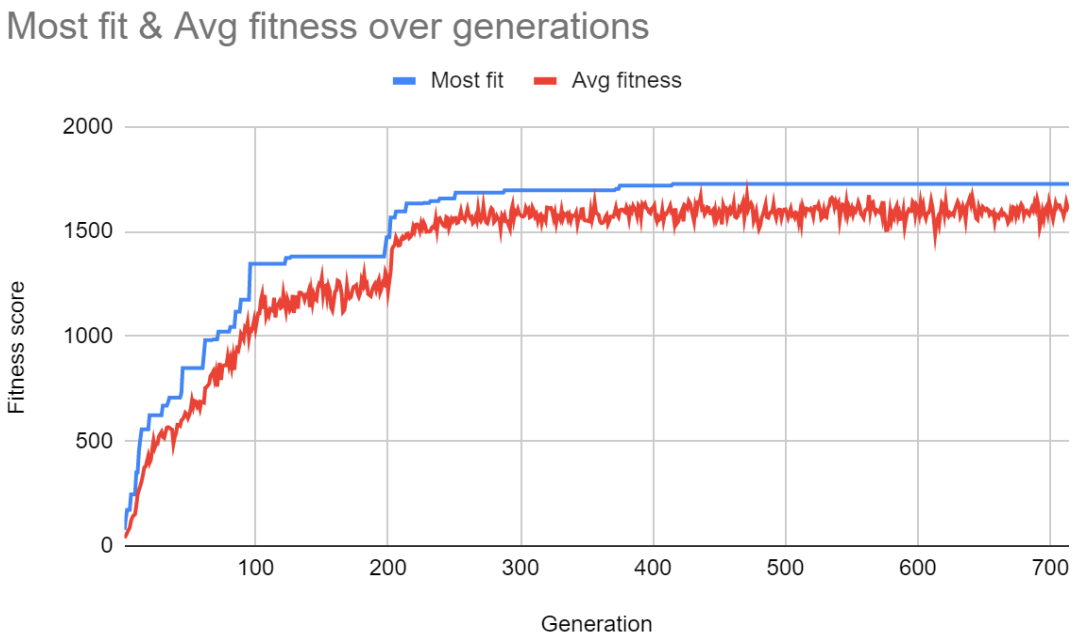



*Table 16 - parameter settings for experiment 4*

## 6.5 Experiment 5

This experiment followed the same parameters as experiment 4, with a change to the fitness function. In this case, the fitness score of a level was given by: fitness score = shortest solution path length + (number of dead ends / 2), i.e FF1+FF2/2. The full parameters are shown in table 18. The aim was to give FF1 more weight, and thus aiming to generate levels with a longer solution path compared to previous experiments. Chance of each cell in the starting states being initially on was set to 25% to make the levels less open. The generated levels can be seen in figure 17. An initial reading of the metrics, presented in table 17, do not indicate a significant increase in the length of the solution path, which was between 58 and 66 for eight of the levels. The number of reachable cells of the levels lies within a large range, from 22 to 158. The amount of dead ends were within a narrower interval, from 73 to 107. The percentage of traversable space of the generated levels showed similar numbers to those from experiments 1,2 and 4.


*Figure 17 - The generated levels in experiment 5*

| Experiment 5 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 64 | 60,66667 | 107 | 22 |
| 2. | 64 | 60,33334 | 96 | 70 |
| 3. | 76 | 57,88889 | 90 | 51 |
| 4. | 58 | 60,22222 | 76 | 150 |
| 5. | 96 | 58,22222 | 84 | 115 |
| 6. | 58 | 61,44444 | 96 | 32 |
| 7. | 66 | 58,66666 | 92 | 46 |
| 8. | 68 | 58,77776 | 98 | 48 |
| 9. | 58 | 58,11111 | 73 | 158 |
| 10. | 58 | 58,44445 | 90 | 76 |

*Table 17 - Metrics from experiment 1 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 5 |
| | Mutation probability | 1/512 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 25% |
| | Convergence limit | 300 |
| | Fitness score given by | FF1+FF2/2 |

*Table 18 - parameter settings for experiment 5*

The GA itself ran for 377 generations before converging, as seen in table 19. Of note is that it converged early before reaching 100 generations.

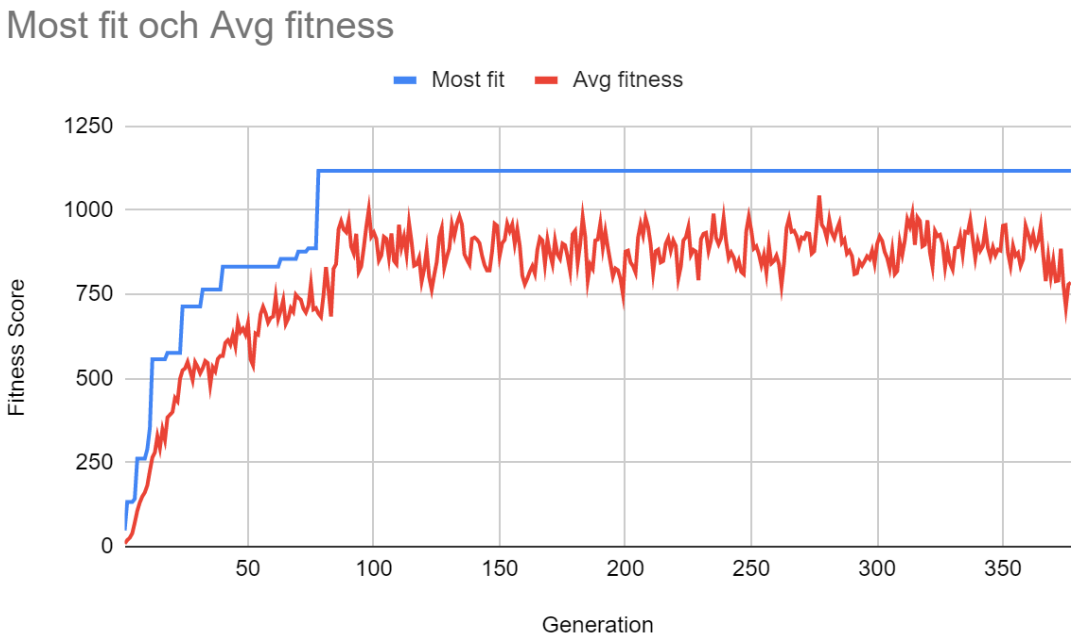



*Table 19 - parameter settings for experiment 5*

## 6.6 Experiment 6

In this experiment we changed the chance of the starting state cells being on to 75%, as shown in table 21. It had the same parameters as in experiment 4 with the only difference being the starting state chance, which was set at 25% in experiment 4. Our goal was to see how such a change would affect the output levels. In *figure 18* we see the algorithm generated quite interesting levels with all of them being solvable. Noteworthy is that there was no remarkable difference in the metrics as opposed to experiment 4 with all the metrics, except the number of dead ends, showing similar results. It is reasonable to assume that the percentage of traversable space would be greater. In table 20 all metrics are displayed.

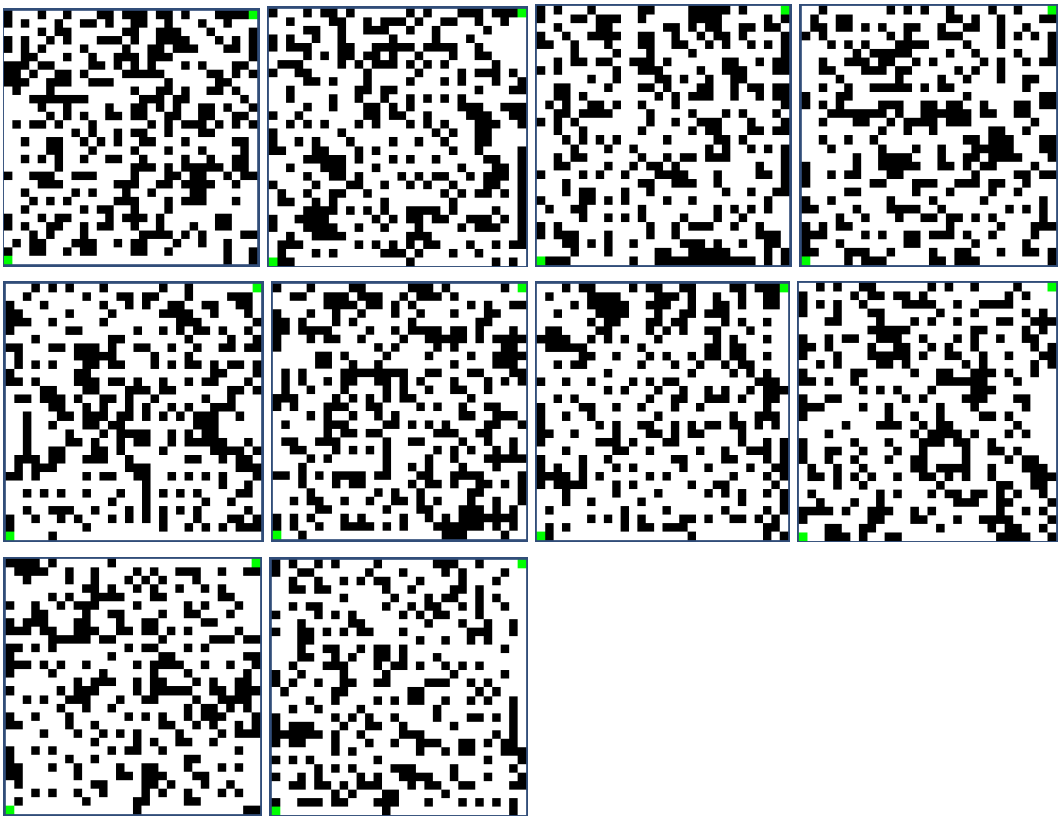


*Figure 18 - The generated levels in experiment 6*

| Experiment 6 metrics |
|---|

| Level # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| 1. | 84 | 62,55556 | 88 | 55 |
| 2. | 62 | 64,55556 | 101 | 45 |
| 3. | 58 | 64,88889 | 102 | 14 |
| 4. | 58 | 65,11111 | 104 | 14 |
| 5. | 60 | 65,88889 | 86 | 37 |
| 6. | 64 | 62,77778 | 105 | 33 |
| 7. | 64 | 67,22222 | 79 | 50 |
| 8. | 58 | 66,11111 | 99 | 27 |
| 9. | 58 | 68,44444 | 86 | 17 |
| 10. | 58 | 69,66667 | 95 | 7 |

*Table 20 - Metrics from experiment 1 with the rows marked in red being mazes with no solution path*

| Parameters | | |
|---|---|---|
| | Max generations | 1000 |
| | Population size | 50 |
| | Cellular automata iterations | 5 |
| | Mutation probability | 1/512 |
| | Crossover probability | 0.6 |
| | Elitism size | 6 |
| | Tournament size | 2 |
| | Number of starting states | 10 |
| | Chance of each cell being initially on in starting states | 75% |
| | Convergence limit | 300 |
| | Fitness score given by | FF1+FF2 |

*Table 21 - parameter settings for experiment 6*

In table 22 we see that the population, after a sharp increase in fitness, slowed down before converging soon after 100 generations.

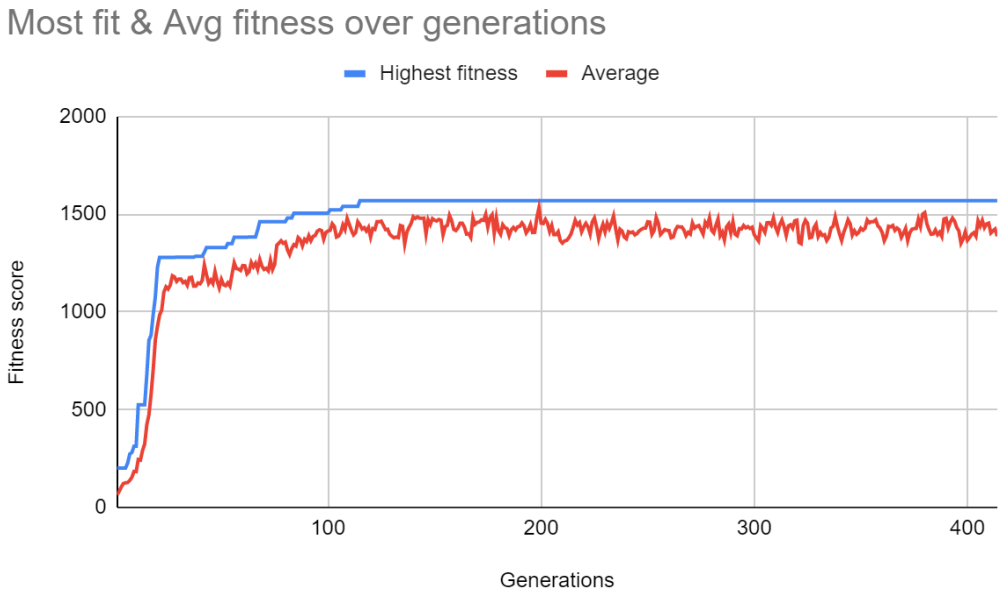



*Table 22 - parameter settings for experiment 6*

# Chapter 7 - Evaluation and Analysis

In this chapter we will analyze the level generator by evaluating the experiment results. In each experiment 10 levels were generated. We have defined four metrics, the length of the solution path, the number of dead ends, percentage of traversable cells and the number of unreachable cells. These will be used in order to gather quantitative data about the levels and measure the quality of the generated final mazes.

## 7.1 Experiment 1

In the first experiment we see that level 2 has metrics that deviate greatly. The GA did not produce a solution path for it, it has an abnormal number of unreachable cells and deviates in the number of dead ends compared to the metrics in the other levels. There are no deviations in the percentage of traversable cells. Table 23 and 24 illustrate this.

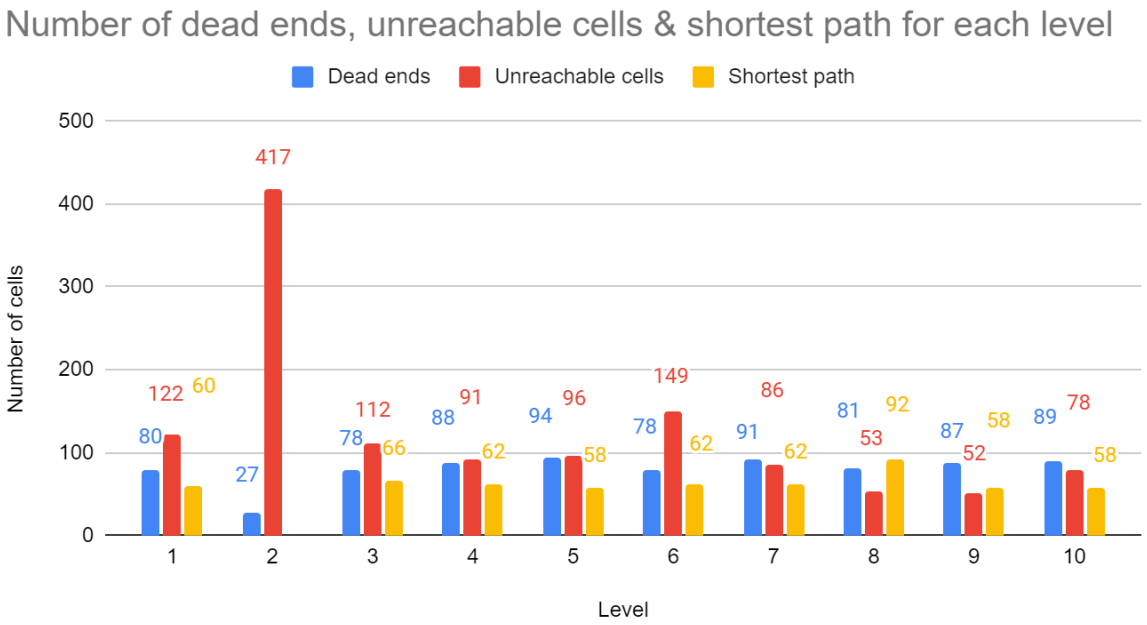



*Table 23 - Level metrics excluding percentage of traversable cells in experiment 1*

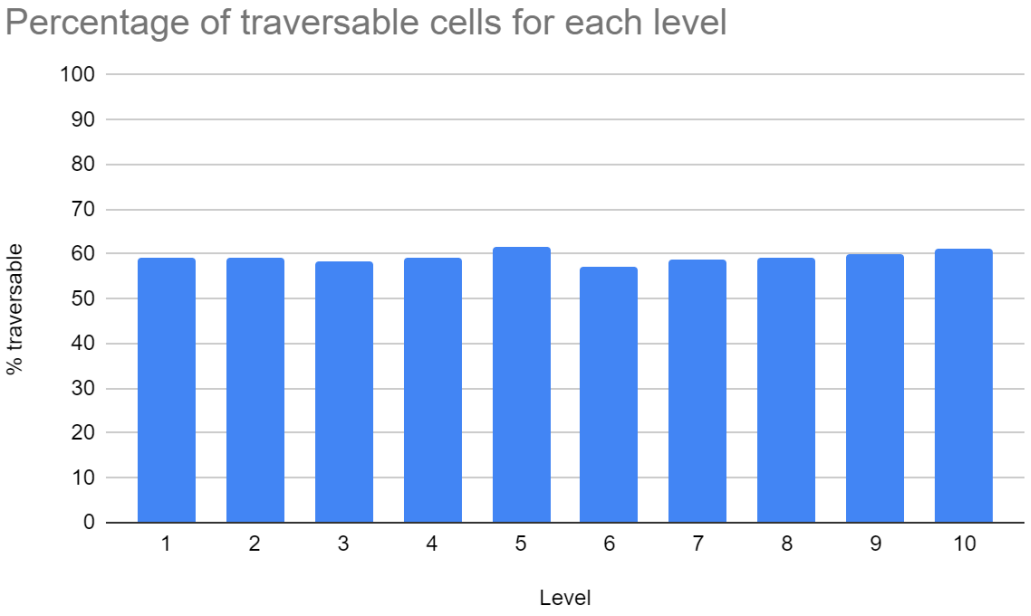



*Table 24 - Percentage of traversable cells for each level in experiment 1*

## 7.2 Experiment 2

In this experiment we can find interesting results. The number of unreachable cells among the levels is varying a lot. With level 2 and 6 having few while level 3, 5 and 8 having 150 or above. The shortest path metric is similar across all levels with the exception of level 2 where the GA failed to produce a solution path. The other two metrics are quite similar with dead ends varying slightly more than the percentage of traversable cells. This is illustrated in table 25 and 26.

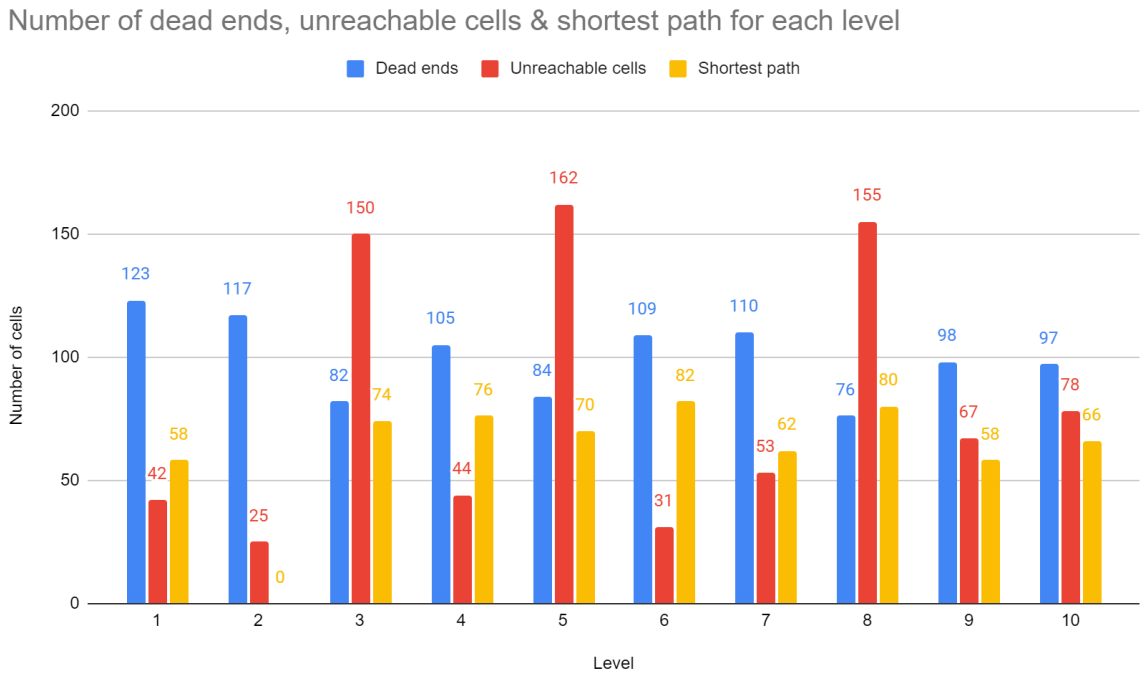



*Table 25 - Level metrics excluding percentage of traversable cells in experiment 2*

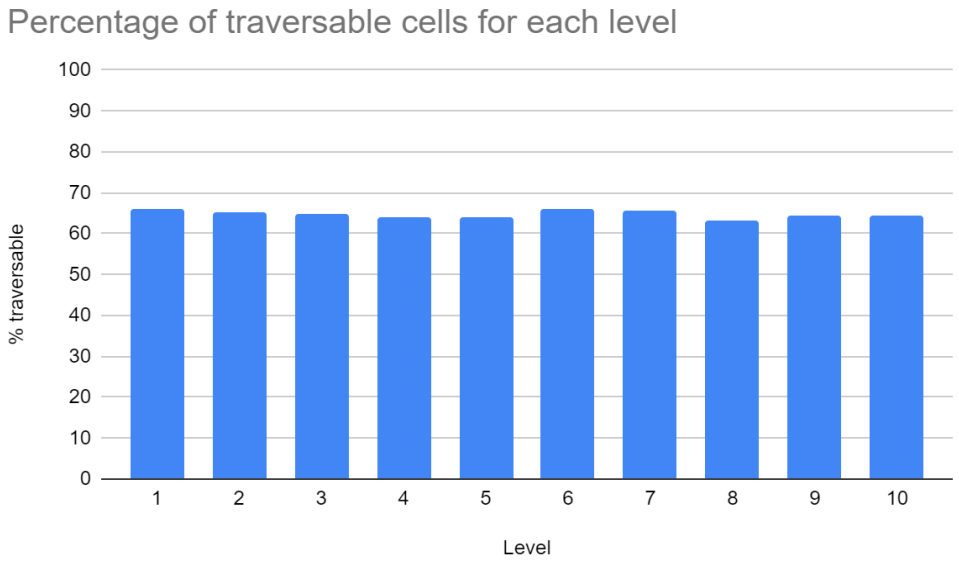



*Table 26 - Percentage of traversable cells for each level in experiment 2*

## 7.3 Experiment 3

Across the 10 levels generated table 27 and 28 illustrate similar values for all metrics with the exception of the numbers of unreachable cells. It shows that level 1, 8 and 10 only have 6, 7 and 2 unreachable cells while level 2 and 3 have 40 and 35 respectively. The percentage of traversable cells range between 69 and 77 meaning that there is no major noteworthy deviation.

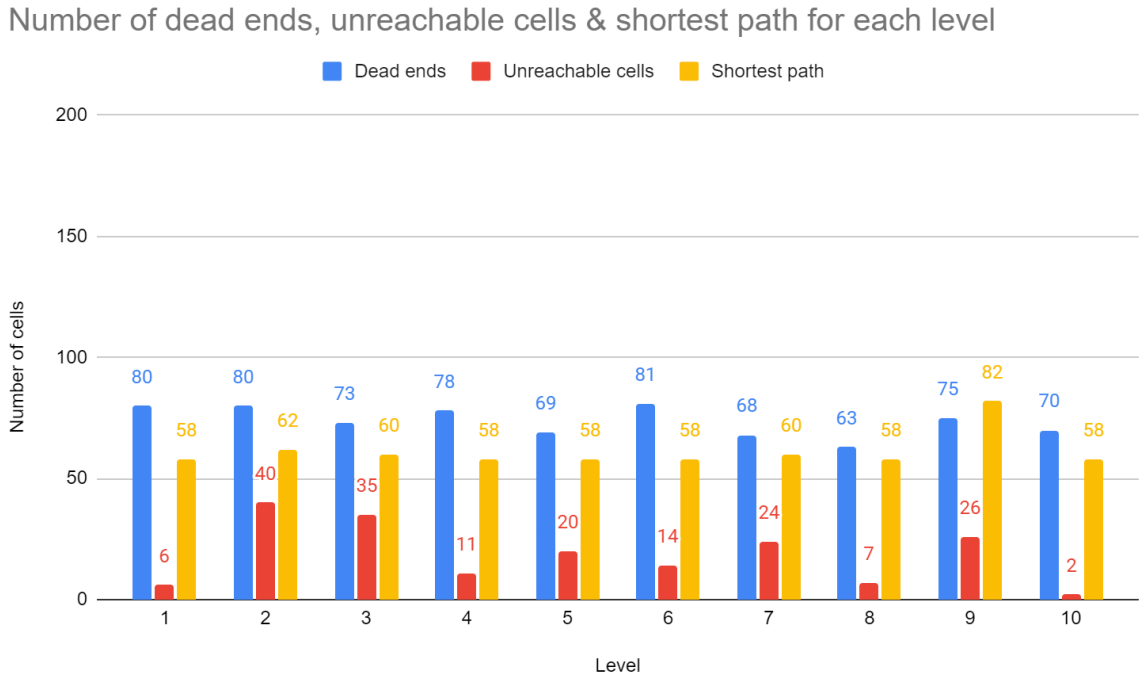



*Table 27 - Level metrics excluding percentage of traversable cells in experiment 3*

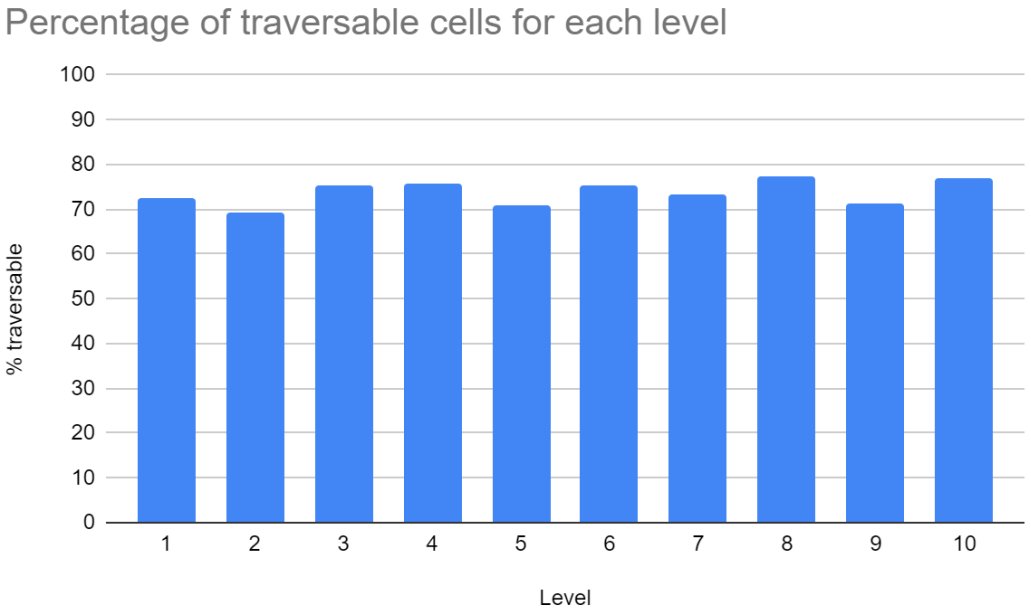



*Table 28 - Percentage of traversable cells for each level in experiment 3*

## 7.4 Experiment 4

Experiment 4 shows, as seen in table 29 and 30, that all generated levels have a similar percentage of traversable cells, length of shortest path and the number of dead. All levels produced by the GA have slightly more than 60% pf traversable cells. The number of dead ends range between 93 to 125 and the shortest path is around 65 for most levels. The metric that we can find deviations in are the number of unreachable cells where values ranging from 12 to 59 exist with the values being spread out within this range.

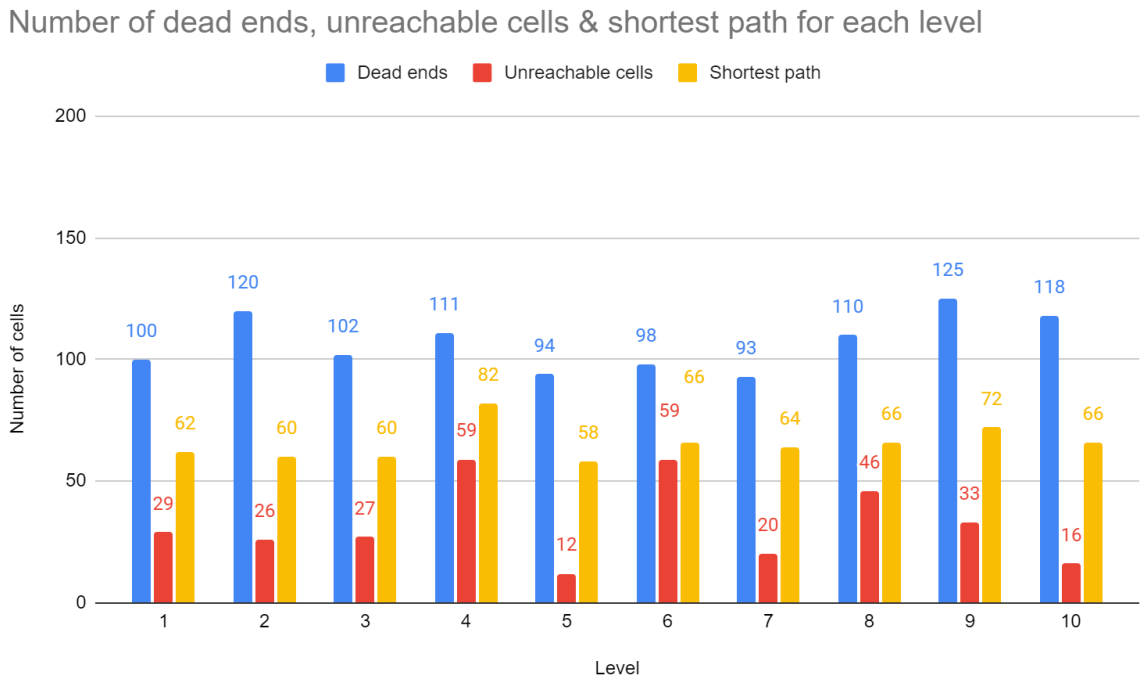



*Table 29- Level metrics excluding percentage of traversable cells in experiment 4*

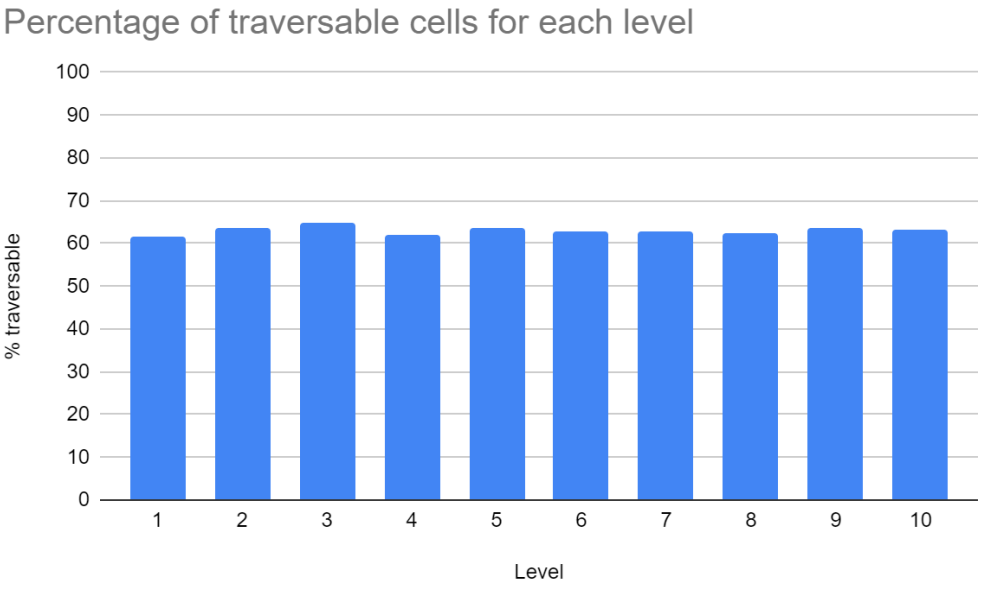



*Table 30 - Percentage of traversable cells for each level in experiment 4*

## 7.5 Experiment 5

In table *31* and *32* the levels illustrate the metrics from this experiment and display similar results across all levels with regards to the number of dead ends and percentage of traversable cells. The other two metrics vary between the levels with the unreachable cells metric going from having very few in level 1 to having way over 100 in level 4,5 and 9 and shortest path ranging from 58 to 96.

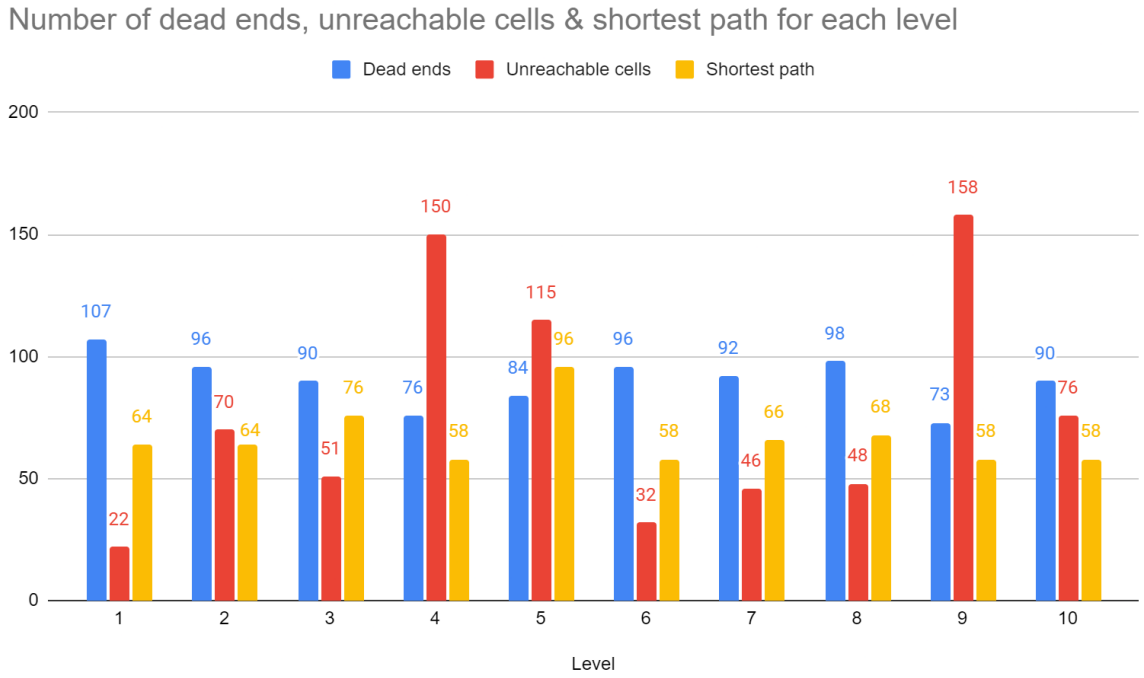



*Table 31- Level metrics excluding percentage of traversable cells in experiment 5*

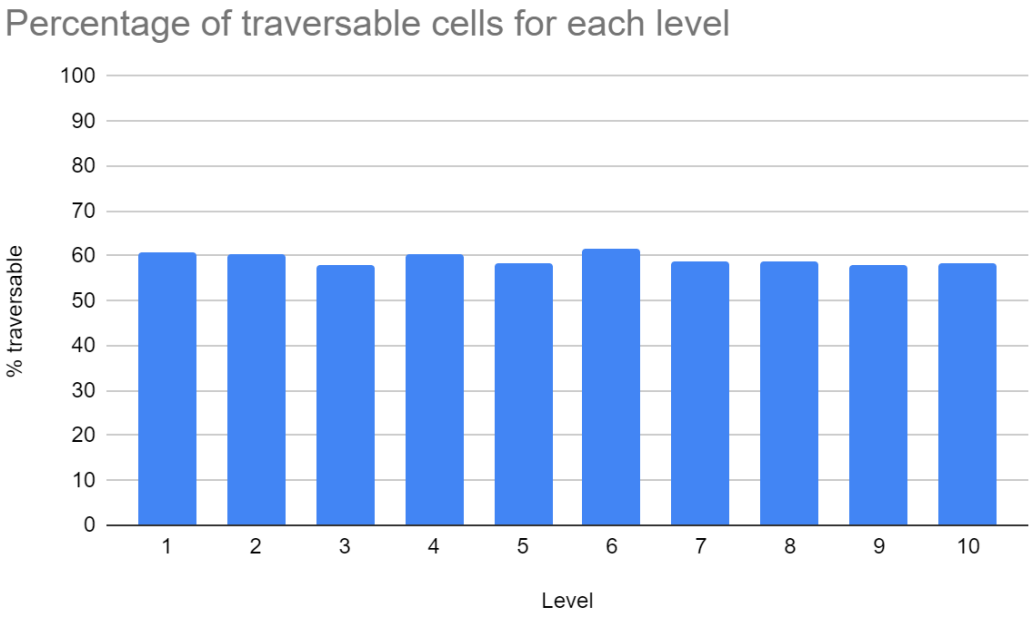



*Table 32 - Percentage of traversable cells for each level in experiment 5*

## 7.6 Experiment 6

For this experiment the number of dead ends is fairly similar across all levels. There is no remarkable deviation that is worth further examination. Most levels have about a 100 dead ends with some having slightly below that. The percentage of traversable cells and the shortest path shows similar results across all levels with the exception of the first level's shortest path. It displays a moderately higher metric compared to the other levels. The number of unreachable cells shows varying results across all levels with not a single being close to 100 cells and some levels having less than 10 unreachable cells. This means that this experiment generated levels that to a large extent are traversable. Table 33 and 34 show this.

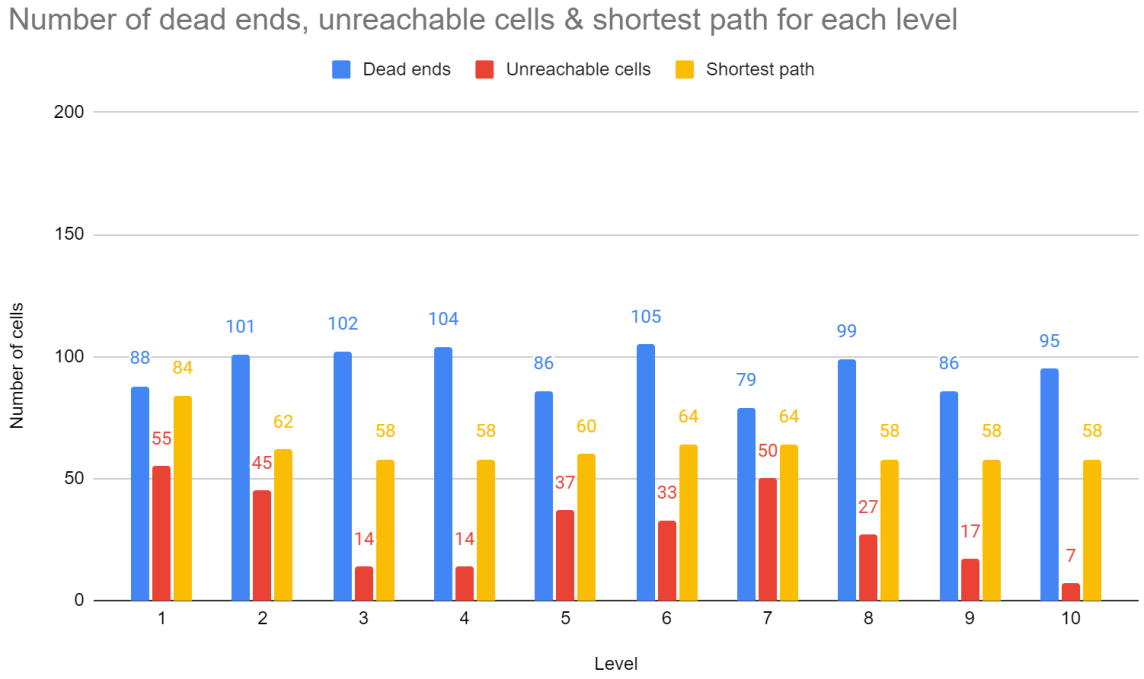



*Table 33 - Level metrics excluding percentage of traversable cells in experiment 6*

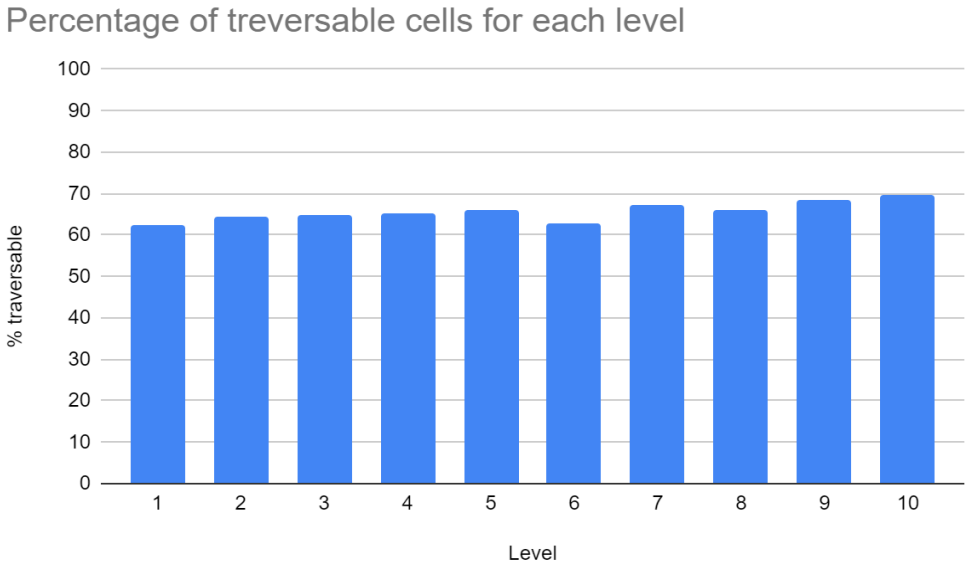



*Table 34 - Percentage of traversable cells for each level in experiment 6*

## 7.7 Experiment Comparisons

The average level metrics for the experiments is presented in table 35. In experiments 1 and 2, the metrics of the two failed levels, i.e ones with no solution paths, were not included. Based on this, we see that the length of the solution paths remained consistent throughout the experiments, lying in the range of 61,2-69,6. The shortest possible solution path in our grids is 58 cells long. Out of the 60 generated levels, 21 of them had a solution path of 58 cells.
The difference in length indicates that for some levels, longer paths could have been found. The levels in experiment three, which were the most traversable, had the shortest paths. In experiment 5, FF1 was given more weight in an attempt to increase solution path length. However, it is only on average 1 cell longer than experiment 4, which except for the FF changes had the same parameters. Table 36 shows all the experiment parameters for easy lookup.

| Average of all experiments |
|---|

| Exp. # | Length of solution path | Percentage of traversable cells | Number of dead ends | Number of unreachable cells |
|---|---|---|---|---|
| | Avg. | Avg. | Avg. | Avg. |
| 1.* | 64,2 | 59,4 | 85,1 | 93,2 |
| 2.* | 69,6 | 65,5 | 98,2 | 86,9 |
| 3. | 61,2 | 73,8 | 73,7 | 18,5 |
| 4. | 65,6 | 63,1 | 107,1 | 32,7 |
| 5. | 66,6 | 59,3 | 90,2 | 76,8 |
| 6. | 62,4 | 65,7 | 94,5 | 29,9 |

*Table 35 - An average for all metrics in each experiment. *The metrics for the unsolvable levels are not included.*

Regarding the percentage of traversable cells, they range from ~59% to ~74%. Experiment three has on average the largest number of traversable cells. This indicates that the increased number of CA iterations in that experiment lead to the generator outputting more open levels. Experiment 6 had 75% chance of each cell being on in the starting states and CA iterations set at 5, compared to 50% and 10 for experiment 3. Experiment 6 did not have a larger percentage of traversable cells than experiment 3, thus showing that if the goal is to generate more open levels, an increase in CA iterations may be more effective than an increase in percentage of each cell being on in the starting states. Although and even larger increase in chance of each cell being on, eg to 90%, may also have been effective for producing open levels. Conversely, experiments 4 and 5 had the chance of cells being on set at 25% to investigate if this led to narrower, more “claustrophobic” levels. However, similar to the increase in starting state chance, this seems to have had little effect on the final levels compared to the other experiments.

Experiment 5 had 59,3% traversable cells, while experiment 1 had 59.4% with a 50% chance of cells being on initially. Experiment 4 had 63,1% traversable cells, a value close to experiment 2's 64,8% and experiment 6's 65,7%. Of note is that experiment 6 had a chance of 75% for cells to be on initially, whilst averaging only about 3 more traversable cells compared to experiment 4. But similar to before, perhaps a further decrease, eg to 10%, may have had a clearer impact on the layout of the levels.

The second fitness function appears to work well, with the number of dead ends lying between 73,3 and 107,1 on average, contributing to making the levels interesting to play. Experiment 3 has the least number of dead ends and unreachable cells, something that can be attributed to its openness and suggesting that there is a tradeoff between these two attributes. The number of unreachable cells on average lies between 18,5 and 125,6. These are parts of the level that were not properly utilized, as had they been reachable, they could have for instances contributed to more dead ends.

| Parameters |
|---|

| | Experiment | #1 | #2 | #3 | #4 | #5 | #6 |
|---|---|---|---|---|---|---|---|
| | Max generations | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| | Population size | 50 | 50 | 50 | 50 | 50 | 50 |
| | Cellular automata iterations | 5 | 5 | 10 | 5 | 5 | 5 |
| | Mutation probability | 1/512 | 0.05 | 1/512 | 1/512 | 1/512 | 1/512 |
| | Crossover probability | 0.6 | 0.6 | 0.6 | 0.6 | 0.6 | 0.6 |
| | Elitism size | 6 | 6 | 6 | 6 | 6 | 6 |
| | Tournament size | 2 | 2 | 2 | 2 | 2 | 2 |
| | Number of starting states | 10 | 10 | 10 | 10 | 10 | 10 |
| | Chance of each cell being initially on in starting states | 50% | 50% | 50% | 25% | 25% | 75% |
| | Convergence limit | 200 | 200 | 300 | 300 | 300 | 300 |
| | Fitness score given by | FF1+FF2 | FF1+FF2 | FF1+FF2 | FF1+FF2 | FF1+FF2/2 | FF1+FF2 |

*Table 36 - Parameters of all experiments*

## 7.8 Metric correlations

To understand the relation between the various metrics, and gain insight into the kind of tradeoffs between them, we conducted a correlation analysis on the gathered level metrics. For the purposes of this analysis, the two failed levels with no solution path were excluded, meaning that the correlation analysis was conducted on the metrics of the remaining 58 levels. We calculated the Pearson correlation coefficient $r$ [35], which is suitable for ratio data [23, Ch. 17], between pairs of metrics for the 58 levels, which gives a value between 1 and -1. A value of 0 indicates that there is no linear correlation between 2 metrics, whereas a value of 1 implies a linear correlation, i.e that if the value of one of the metrics rises, the value of the other one will as well. A coefficient of -1 indicates that as the value of one metric increases, the value of the other one will decrease. In interpreting the coefficient values we follow Oates [23, Ch. 17] guidelines, stating that a coefficient value in the range of ±0,3-0,7 indicates "a reasonable correlation". Additionally, an associated p-value will be calculated. This value indicates whether the observed correlation is statistically significant or if it could have been due to randomness. The p-value threshold for this paper is set at 0,05,meaning that we consider correlations with a p-value less than 0,05 to be statistically significant [23, Ch. 17].

In table 37, we see that only two of the correlation coefficients $r$ showed statistical significance, meaning that their p-value was less than 0,05. Firstly, a negative correlation is implied between the number of dead ends and unreachable cells. Meaning that levels which had more dead ends had fewer unreachable cells and vice versa. This suggests that CA rules that are better at creating dead ends also result in levels with fewer unreachable cells. One can also view isolated areas of unreachable cells as potential dead ends, given that a path is carved to them. Additionally, a reasonable correlation can be seen between percentage of traversable cells and the number of dead ends, suggesting that levels with more traversable cells also had more dead ends.

| Pearson correlation coefficient |
|---|

| Metric | Shortest path | % of traversable cells | # of dead ends |
|---|---|---|---|
| % of traversable cells | $r$ = -0,0188<br>$p$ = 0.893 | | |
| # of dead ends | $r$ = 0,0553<br>$p$ = 0.681 | $r$ = 0,393<br>$p$ = 0,00231 | |
| Unreachable cells | $r$ = 0,234<br>$p$ = 0,0772 | $r$ = -0,242<br>$p$ = 0,0672 | $r$ = -0,528<br>$p$ = 0,00002 |

*Table 37 - Correlation coefficient and p-value between level metrics*

## 7.9 Genetic algorithm performance

It is also important to analyze the performance of the GA itself, and whether it was good enough or if it can be improved. Table 38 shows the generation after which the most fit member of the population stopped improving in each experiment, together with the value of the most fit member at the end of the GA's run. In experiment 5 the value of FF2 was halved in order to give more importance to FF1, thus making comparisons between it and the other experiments difficult. The other five experiments had the same FF's, and thus had the same aims regarding what type of levels they aimed to create. Out of these five, experiment 3's most fit member has the lowest value at 1349, closely followed by experiment 1 at 1370. Meanwhile experiment 4 has the highest with 1727. Looking at table 35, the levels of experiment 4 had, on average, around 28 and 33 more dead ends than the levels in experiments 1 and 3 respectively. For the solution path this difference was 8 and 5 cells. This disparity can be a cause of concern, as the higher fitness in experiment 4 shows that there was still room for improvement in the other experiments. In addition, the GA in all experiments terminated due to convergence. The last generation at which the most fit member showed improvement being between generations 78 and 470 for the experiments, as shown in table 38. To amend this, other decisions regarding the GA, such as different mutation rates or selection methods have to be considered in the future in order to achieve more effective exploration of the search space.

| Genetic algorithm performance | | |
|---|---|---|
| Exp. # | Convergence start at gen. | Most fit candidate at termination |
| 1. | 470 | 1370 |
| 2. | 444 | 1626 |
| 3. | 300 | 1349 |
| 4. | 415 | 1727 |
| 5. | 78 | 1117 |
| 6. | 115 | 1569 |

*Table 38 - The generation at which the GA stopped improving and its most fit member at termination*

# Chapter 8 - Discussion

## 8.1 Discussion

The results and analysis show that the proposed approach as a whole worked. The GA correctly evolved CA rules that could transform 10 random starting states into interesting game levels. The criteria for "interestingness" was defined in the fitness function as levels with long shortest solution path and a large number of dead ends, properties that the output levels did generally have. However, some levels only had the minimum possible solution path, something that is discussed further below. Our results also indicate that the level generator can output game levels for different sets of 10 random starting states. This is due to the fact that, despite random starting states being used, and different ones for each experiment, the generator could still transform these into game levels. This is of note especially with regards to previous papers in which special care was given in determining the starting states [9], [11], [12]. Our results thus downplay the importance of choice of starting state when evolving cellular automata rules for level generation. This simplifies one part of the level generation process, as designers of these may focus on other parts of the generator while using random starting states instead of having to design specific ones.

Given that the algorithm worked for our defined FF's, it is also reasonable to assume that it should work for alternative FF's, given some care in defining them and possibly balancing their weights against each other in order to achieve the desired results. The approach of using random starting states could also be combined with other methods, such as the image processing techniques used for the fitness function of [11], which could allow for the implementation of more advanced features such as clearly defined rooms. One may also use our method for alternative game genres, such as 2d platformers. As a simple example, one may imagine a platformer game where the aim is to "climb" from the bottom row to the top row by means of jumping on non-traversable cells. Here fitness function(s) that aim to create platforms within jumping distance from bottom to the top could be used. In similar fashion the fitness function could be designed to generate levels for 2d platformers similar to the ones in Super mario bros. In [9], evolved CA were partially evaluated by means of a first-person view "maze running" game; our generated levels may also be used for such a game or other first-person games where maze-like levels are desired, if converted to 3d where the non-traversable cells would constitute walls.

Our analysis also shows that there is room for improvement regarding the GA's performance. This is based on the fact that there were large differences in the fitness values of the "winning" candidate in each experiment. Thus indicating that in some experiments, more optimal solutions could have been found. For the GA to produce more optimal levels, changes to the GA's operators and parameters, such as different selection methods or mutation rates may have to be considered and investigated. Overall regarding the GA design, we can conclude that while it was good enough to produce viable levels, improvements to it can still be achieved. Of the created levels, two levels out of the generated 60 lacked a solution path. This should not pose a major issue in itself, as the level generator is supposed to be used offline, i.e during game development. Thus the developer can discard the failed levels or manually edit the levels to ensure there's a solution path. Alternatively one could add a constraint that does not allow the GA to terminate if all 10 levels do not have a solution path, or algorithmically carve a

solution path in levels that lack one after the GA's termination. Overall these results further show that there is still room for improvement regarding the GA's design.

A second issue was that the GA generated 21 levels with the shortest possible solution path of 58, which is also the shortest possible path on our 30x30 grids. These results show that the generator in about a third of the levels only generated the shortest possible path, whereas longer solution paths would generally mean more interesting levels. Of note, however, is that these paths weren't necessarily in a straight line from start to end. Rather we saw different variations of only having to move up and right to reach the end. In figure 19 we see the shortest solution through level 5 of experiment 4, which was one of the levels where the solution path was only 58 cells. It still, however, remains a non-trivial task to solve the level as the player can be misguided in the intersections, and either pick a longer path or run into a dead end.

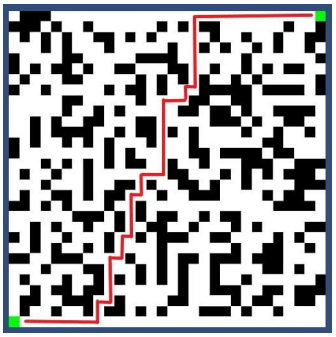


*Figure 19 - A level with the shortest path of 58*

Another cause for concern is the number of unreachable cells, which often were fairly high in number. These are cells that are technically traversable but without a path to them from the start point. In a top-down view the player might see these levels and falsely assume that there is a path to them. Additionally, as the levels might be populated in a game, for instance with items and enemies, these cells impose a problem. Additional care has to thus be given to not spawn e.g. enemies or important items in unreachable cells. Different ways of fixing this are possible. One is to include a third FF who's aim is to decrease the number of unreachable cells. However, this could lead to issues in balancing and weighing the three FF's against each other. A simpler and perhaps more effective approach would be to "dig" a path to these cells. For each unreachable cell, the nearest reachable, traversable cell could be found and the state of the cells in the way to it would be changed to traversable. There are existing region merging algorithms that can be used, as was employed in [9] and [12]. A third avenue would be to conduct a breadth-first search from the starting point, and for each traversable cell that is unreachable, its state could be flipped to non-traversable. This issue, combined with the unsolvable levels, shows that our level generator may not be suitable for directly putting levels in a game. However, it can be used as a basis for procedural generation of levels, where some work is still required from the developer before the levels are used in a game.

When evaluating the results, one must also consider the randomness involved in initializing the starting states. This makes it difficult to draw definitive conclusions about the effects of single parameter changes on the final levels. For two experiments where the chance of a cell being on in the starting state was 50%, there are still differences due to the stochasticity involved. However, we consider the experiment results to still give us some initial insights into the impact of parameter changes and the type of levels that can be created by the generator. These insights can be used as a

starting-point for game developers using our method to generate levels with certain characteristics. In addition, another valuable insight from our results is that, for the given changes in parameters, the generator was still capable of outputting viable game levels based on random starting states. In order to fully explore the types of levels that our generator is capable of producing, a comprehensive expressivity analysis would have to be conducted. For various parameter configurations, the algorithm would be run a large number of times. Thus more definitive conclusions could be made regarding the effects of single parameter changes with regards to the output levels.

# Chapter 9 - Conclusion and future work

## 9.1 Conclusion

In our research we designed a GA, based on the findings of previous research using GA to evolve cellular automata rules for maze-like video game level creation. Our GA operators were tournament selection with tournament size of two and elitism of size 6 as the selection methods, one-point crossover with a 60% probability and one-point mutation. We implemented two fitness functions, aiming to generate levels where the shortest solution path was as long as possible and one aiming to include a large number of dead ends in the level. The FF's were based on previous research showing that users found these to be desirable traits in this type of level.

With RQ1 the literature review that we conducted not only provided us with the right knowledge and insight about the current state of the art regarding PCG in general, it also helped us in the task of answering RQ2. From the literature review we accumulated knowledge about what work was previously done with regards to evolving CA rules with a GA and identified where there were research gaps within the field. More specifically previous research stated that levels with longer solutions paths and a larger number of dead ends were considered more interesting and challenging. We used this knowledge to design our two FF's, aiming to generate interesting levels with long solution paths and many dead ends. Additional insights that we gained from the current state of the art helped form our GA design, resulting in tournament selection and elitism selection, one point crossover, one point mutation and also a direct representation of the CA.

As for the RQ2 our results showed that it is possible to use a GA to evolve CA rules that would, when applied to a collection of 10 random starting states, output interesting maze-like game levels. This can be confirmed based on the experiments that were conducted along with the metrics that were evaluated, showing levels with generally long solution paths and many dead ends. Over six experiments, 60 levels in total were generated, out of which 2 levels were considered failures due to a lack of a solution path. In the remaining levels, the evolved CA had correctly created levels with a large number of dead ends while in the case of 21 of them, had solution paths that were equal to the shortest theoretical path on a square grid. This did not go in line with what was defined in FF1 nevertheless it wasn't a major problem since the terms of RQ2 were to a certain degree still satisfied. Those levels also did not have a straight path from start to end, thus potentially making solving them non-trivial. However, from what we can see in the generated levels is that the ones with longer solution paths tended to have less linear and more zig-zacky solution paths making them more challenging and consequently more interesting. The levels with 58 cell solution paths were still interesting in the sense that they granted challenging levels where the solution path wasn't necessarily easy to find. We can also conclude that the parameter that regulated the CA iterations had the most significant visual impact on the levels. The results also indicated that more work may need to be done in designing the GA in order to find CA rules that generate more optimal levels. An additional cause for concern was the at times large number of unreachable cells, where we suggested some methods for amending this.

With this study we believe that we made a contribution to a deeper understanding on how CA can be used together with GA. By implementing an additional way of evolving CA rules with GA and more specifically by doing so on random CA starting states. We proved that it is possible to generate interesting levels given our approach. Our research

results could be utilized by game developers and other researchers within the field of PCG.

## 9.2 Future work

An interesting avenue for future work would be to increase the number of starting states, and investigate if the given method still works. Taking this one step further, one could see if it is possible to evolve CA rules that are capable of generating interesting levels based on any given randomized starting state, and not just those that were initialized at the start of the GA's run. The benefit of this approach is that, given that such a CA rule is found, only the CA rule would have to be included in a game. Thus the game could at runtime generate a potentially infinite number of levels by applying the rule to randomized starting states. This could be used in games like the infinite cave game of [8], but with CA rules that are specifically evolved to output a certain kind of levels. One may also go in the opposite direction, and investigate the pros and cons of using only 1 random starting state, but instead re-running the entire algorithm multiple times. The interest here would be in comparing these approaches time wise but also to compare the level characteristics, trying to understand if this approach leads to higher quality levels and ce whether the levels will still show some similarities.

An additional approach could be to experiment with increased starting state grid sizes to produce bigger levels. One trait of using CA with the intention of creating game levels is that it is capable of generating organic looking ones. However, in order for a level to achieve an organic look, the starting state grid size would need to be increased. By increasing the size of the grid one could assume that the levels produced after such an altercation would be less "blocky" since each individual cell would be smaller in relation to the grid. Consequently levels would have a smoother and more organic feel. In such a case more care may have to be given to optimizing the GA and its implementation, in order to run the GA in a reasonable amount of time.

One could also try different GA parameters and operators in order to further optimize the GA's performance, or try alternative optimization algorithms in combination to find suitable CA rules for level generation. Finally, playtesting of the levels based on an improved version of the generator could provide further insights into the viability of the levels for different types of games.

# References


[1] Noor. T. Shaker and M. J. Julian. Nelson, *Procedural Content Generation In Games*. Place of publication not identified: SPRINGER, 2018.

[2] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is procedural content generation?: Mario on the borderline," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames '11*, Bordeaux, France, 2011, pp. 1–6. doi: 10.1145/2000919.2000922.

[3] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects," in *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, Jalgaon, India, Dec. 2016, pp. 261–265. doi: 10.1109/ICGTSPICC.2016.7955308.

[4] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996.

[5] M. Mitchell, *Complexity: a guided tour*, 1. iss. as an paperback. New York, NY: Oxford University Press, 2011.

[6] E. W. Weisstein, "von Neumann Neighborhood." https://mathworld.wolfram.com/ (accessed Feb. 26, 2022).

[7] E. W. Weisstein, "Moore Neighborhood." https://mathworld.wolfram.com/ (accessed Feb. 26, 2022).

[8] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames '10*, Monterey, California, 2010, pp. 1–4. doi: 10.1145/1814256.1814266.

[9] C. Adams, H. Parekh, and S. J. Louis, "Procedural level design using an interactive cellular automata genetic algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Berlin Germany, Jul. 2017, pp. 85–86. doi: 10.1145/3067695.3075614.

[10] Z. Wu, Y. Mao, and Q. Li, "Procedural Game Map Generation using Multi-leveled Cellular Automata by Machine learning," in *Proceedings of the 2nd International Symposium on Artificial Intelligence for Medicine Sciences*, Beijing China, Oct. 2021, pp. 168–172. doi: 10.1145/3500931.3500962.

[11] A. Pech, "Using genetic algorithms to find cellular automata rule sets capable of generating maze-like game level layouts," Bachelor of Computer Science (Honours), Edith Cowan University, Joondalup, Australia, 2013. [Online]. Available: https://ro.ecu.edu.au/theses_hons/95/

[12] C. Adams, "Evolving Cellular Automata Rules for Maze Generation," Master Thesis, University of Nevada, Reno, Reno, Nevada, USA, 2018.

[13] J. Doran and I. Parberry, "Controlled Procedural Terrain Generation Using Software Agents," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 2, pp. 111–119, Jun. 2010, doi: 10.1109/TCIAIG.2010.2049020.

[14] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, and G. N. Yannakakis, "Multiobjective exploration of the StarCraft map space," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, Copenhagen, Denmark, Aug. 2010, pp. 265–272. doi: 10.1109/ITW.2010.5593346.

[15] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the Galactic Arms Race video game," in *2009 IEEE Symposium on Computational Intelligence and Games*, Milano, Italy, Sep. 2009, pp. 241–248. doi: 10.1109/CIG.2009.5286468.

[16] "Galactic Arms Race." http://galacticarmsrace.blogspot.com/ (accessed Mar. 07, 2022).

[17] P. Ziegler and S. von Mammen, "Generating Real-Time Strategy Heightmaps using Cellular Automata," in *International Conference on the Foundations of Digital*

*Games*, Bugibba Malta, Sep. 2020, pp. 1–4. doi: 10.1145/3402942.3402956.
[18] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *2007 IEEE Symposium on Computational Intelligence and Games*, Honolulu, HI, USA, 2007, pp. 252–259. doi: 10.1109/CIG.2007.368106.
[19] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Greater Noida, India, Feb. 2015, pp. 515–519. doi: 10.1109/ABLAZE.2015.7154916.
[20] Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu, "Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms," in *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Vienna, Austria, 2005, vol. 2, pp. 1115–1121. doi: 10.1109/CIMCA.2005.1631619.
[21] S. Mashohor, J. R. Evans, and T. Arslan, "Elitist Selection Schemes for Genetic Algorithm based Printed Circuit Board Inspection System," in *2005 IEEE Congress on Evolutionary Computation*, Edinburgh, Scotland, UK, 2005, vol. 2, pp. 974–978. doi: 10.1109/CEC.2005.1554796.
[22] Walchand College of Engineering, U. A.J., S. P.D., and Government College of Engineering, Karad, "Crossover Operators In Genetic Algorithms: A Review," *ICTACT J. Soft Comput.*, vol. 06, no. 01, pp. 1083–1092, Oct. 2015, doi: 10.21917/ijsc.2015.0150.
[23] B. J. Oates, *Researching information systems and computing*. London ; Thousand Oaks, Calif: SAGE Publications, 2006.
[24] Hevner, March, Park, and Ram, "Design Science in Information Systems Research," *MIS Q.*, vol. 28, no. 1, p. 75, 2004, doi: 10.2307/25148625.
[25] V. Vaishnavi and B. Kuechler, "Design Science Research in Information Systems." Jan. 20, 2004. [Online]. Available: http://www.desrist.org/design-research-in-information-systems/
[26] R. Helms, E. Giovacchini, R. Teigland, and T. Kohler, "A Design Research Approach to Developing User Innovation Workshops in Second Life," *J. Virtual Worlds Res.*, vol. 3, no. 1, Apr. 2010, doi: 10.4101/jvwr.v3i1.819.
[27] "ACM Digital Library," *ACM Digital Library*. https://dl.acm.org/ (accessed Apr. 18, 2022).
[28] "PCG Workshop | Paper Database." http://www.pcgworkshop.com/database.php (accessed Apr. 18, 2022).
[29] "Google Scholar." https://scholar.google.com/ (accessed Apr. 18, 2022).
[30] "Bibliotek | Malmö universitet." https://mau.se/bibliotek/ (accessed Apr. 18, 2022).
[31] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-Based Procedural Content Generation," in *Applications of Evolutionary Computation*, vol. 6024, C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcazar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preuß, J. Togelius, and G. N. Yannakakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 141–150. doi: 10.1007/978-3-642-12239-2_15.
[32] U. Technologies, "Unity - Scripting API: Random." https://docs.unity3d.com/ScriptReference/Random.html (accessed Apr. 04, 2022).
[33] "Shortest distance between two cells in a matrix or grid," *GeeksforGeeks*, Oct. 13, 2017. https://www.geeksforgeeks.org/shortest-distance-two-cells-matrix-grid/ (accessed Apr. 11, 2022).
[34] U. Technologies, "Unity Real-Time Development Platform | 3D, 2D VR & AR

Engine." https://unity.com/ (accessed Apr. 07, 2022).
[35] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability & statistics for engineers & scientists: MyStatLab update*. 2017. Accessed: May 11, 2022. [Online]. Available: http://www.myilibrary.com?id=947904