Bachelor thesis
Computer Science

Radboud University

Maze Generation, an L-System Based Approach

Author:
Thijs Voncken
s4223845

First supervisor/assessor:
Dr. Jurriaan Rot
jrot@cs.ru.nl

Second assessor:
Prof. Dr. Hans Zantema
h.zantema@tue.nl

January 23, 2017

Abstract

In this thesis a method for procedurally generating mazes is described. This is achieved by constructing an evolutionary algorithm which mutates L-Systems to generate the desired structure. The fitness function of the evolutionary algorithm gives designers some control over the outcome. In the end it is shown that the presented method is capable of generating mazes which attempt to maximize any of the two given fitness functions.

Contents

Chapter 1

Introduction

*P*rocedural *C*ontent *G*eneration (PCG) is versatile tool, and a popular subject of study [1]. It has the potential to allow architects to generate a mock-up of a building in a matter of seconds [2], or easily create entire worlds for use in video games. Especially video games have made extensive use of PCG, from the old rogue-like games, to the more modern *Minecraft*, or even *The Elder Scrolls V: Skyrim*.

The main goal in using PCG is usually reducing the workload of designers, who would otherwise have to create the objects manually. Since video games have far less limitations than the real world, PCG has also been used to create millions of unique worlds for player to explore, effectively giving each player a new and (almost) unique experience.

Over time, many different approaches have been devised for PCG. Every application usually has its own PCG algorithm tailor-made for its specific purpose. Especially within video games, with the many different requirements for each game, many different algorithms have been implemented.

In this thesis, I will be looking at one of these many facets of PCG: the generation of video game levels. Because video game levels often have a complex set of requirements, not all of which relate to the structure of a level, the more complex level is reduced to a maze, which has far simpler requirements.

For this purpose, a maze is considered to be a 2D structure, consisting of set of interconnecting hallways. The maze has a starting point, where the player begins, and an end point, which the player has to find. All hallways are either directly of indirectly connected to one another, and so every section of the maze can be reached. However, not every segment is situated on a path from start to end: there can be dead ends and cycles.

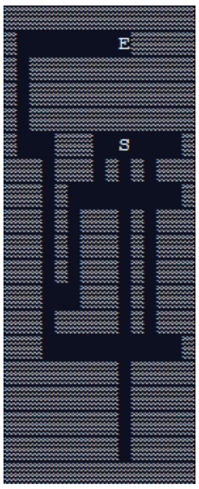

Figure 1.1: An example of a maze.

The goal of a maze is rather simple: provide a fun challenge for players in finding the end, and do so through its structural properties. The exact requirements, however, can still very from case to case, and the designer of a maze still has to keep their audience in mind. It would, for example, not be a good idea to create a fiendishly difficult maze for small children to solve, as they would only get frustrated in doing so. Therefore, allowing the users of such an algorithm to customize the properties of the resulting maze is very important.

In an earlier paper, Ashlock, Lee, and McGuinness [3] describe such a way of generating maze-like levels while giving designers a degree of control over the resulting structures. This is achieved by implementing an evolutionary algorithm, and letting designers set up the fitness function, thereby letting them control which properties the maze would exhibit. Such an approach has the potential of providing a common basic algorithm, which designers can use for a great range of levels, instead of tailor-making a new algorithm for each and every purpose.

Ashlock, Lee, and McGuinness [3] show-case their approach using four different representations of mazes. In their paper they also suggest that a representation based on L-Systems might provide a good alternative to their own representations, due to the fact that L-Systems can be saved in a compact manner compared to a direct representation, and because the resulting maze can be computed with relative ease.

Lindenmayer Systems (or L-Systems) are a string rewriting system, not unlike a context-free grammar, in which the productions are applied in parallel. Originally, they were used in order to describe the growth of biological structures, such as plants and trees, and as such are relatively well suited for generating branching and repeating structures. Since mazes often come down to a complex set of branching paths, L-Systems seem a natural fit for this task. They do, however, have one disadvantage: tweaking the production rules to generate the desired outcome is a difficult task for humans. By using L-Systems as the underlying representation for the evolutionary algorithm used by Ashlock, Lee, and McGuinness [3], this disadvantage is removed.

So, in this thesis, I will attempt to show that L-Systems can effectively be used as the underlying representation for an evolutionary algorithm to generate mazes, roughly based upon the earlier work by Ashlock, Lee, and McGuinness [3].

To achieve this goal, I will first describe the type of L-System used (Section 4.1). Since an L-System only generates a string, I will then introduce the concept of Turtle Graphics, which is used to convert this string into an actual maze (Section 4.2). Then I will describe the evolutionary algorithm used to generate the L-Systems (Section 4.3). Followed by a short description of the three different fitness functions used to test the viability of this approach: one maximizing the shortest path from start to end, one simply summing all path lengths, and one combining the two. After that I will give a short description of the implementation, and where to find the source code, for those curious.

The results of this algorithm will then be compared to the same algorithm executing using plain strings as a representation. These strings will be interpreted in the same way as strings generated by an L-System, using Turtle Graphics. This is done to show that using L-Systems as a representation does have its benefits.

The thesis is structured as follows. In Chapter 2 there will first be a short description of related work in the field of PCG. Next, in Chapter 3 I will cover the underlying concepts that are used in the rest of this thesis. Then, in Chapter 4 I will describe the method used to acquire the results. Followed, in Chapter 5, by the presentation of said results. Then, in Chapter 6 I will discuss any caveats discovered with this method, and pose some possible future research. Finally, in Chapter 7 I will draw a conclusion.

Chapter 2

Related Work

Due to the potential benefits of a good PCG approach, a lot of research has already been done on this subject. Below, I only mention research which is either related to the goal I am trying to achieve in this thesis, or the method by which I am trying to achieve this goal.

In architecture, Wonka, Wimmer, Sillion, *et al.* [4] published a paper describing a way for entire buildings to be generated based upon a shape-grammar. Their main goal being to provide enough control to their users to tweak the generated buildings to their needs. In 2010, Merrell, Schkufza, and Koltun [5] published a method of generated floor plans based on a Bayesian network. They focused more on generating the layout than the outside of the building.

Looking further towards urban planning, Parish and Müller [6] devised a method to generate an L-Systems that procedurally generates road networks and building geometries, while respecting some global restrictions and goals. Later Müller, Wonka, Haegler, *et al.* [2] published a way to generate the visual shells of buildings for use as background scenery. This time their method was based on a generative grammar, and produced realistic looking building shells.

Based on this earlier work Chen, Esch, Wonka, *et al.* [7] devised a way to generate road networks based upon tensor fields. They attempted to provide more user control over the results than Parish and Müller [6], citing the difficulty humans have in manually tweaking L-Systems to fit their needs.

Moving more towards video games, apart from commercial PCG methods, implemented in games such as *Minecraft*, *Skyrim* or old rogue-likes, some academic research has been done as well. In 2010, Dormans [8] described a way of generating levels based on first generating a mission for the player, and then later generating the level around that mission. Both the mission and the level were generated using generative grammars of different types. Also in 2010, Johnson, Yannakakis, and Togelius [9] used another method, cellular automata, to generate infinite cave levels in real-time. Later Dormans [10] refined his method for PCG, by using a rewrite system to generate a mission based on locks and keys a player needs to match up, then using this generated mission to once again generate a level.

Also in 2011, Ashlock, Lee, and McGuinness [3] published their method of generating maze-like levels. This was a search-based approach, using different underlying representations, and attempting to provide control the the level designers through their use of checkpoints and an evolutionary algorithm, allowing tweaking of the fitness function.

In this thesis, the idea by Parish and Müller [6] to use L-Systems for PCG is used. However, keeping in mind the criticism from Chen, Esch, Wonka, *et al.* [7] about human usability, the method is adapted based on the work by Ashlock, Lee, and McGuinness [3]. This results in L-Systems being used for the first time, as far as I am aware, as the representation of a maze within an evolutionary algorithm.

Chapter 3

Preliminaries

In this section I will give an introduction to the theoretical concepts used in this thesis. First, I will describe how L-Systems work, and how they are defined (3.1). Second, I will describe the Turtle Graphics technique, which is used to interpret the result of an L-System (3.2). Last, I will give a global description of the concept of evolutionary algorithms (3.3).

3.1 L-Systems

L-Systems are mathematical systems originally developed to describe the growth of plants and other biological structures. Essentially, they are a type of string-rewriting where the main property is the fact that rewriting is done in parallel. In non-parallel rewriting systems, whether they are based on strings or any other type of subject, any given part of the subject is often rewritten multiple times, until there are no applicable rewrites left for that particular part [11].

L-Systems apply only a single production (a rewrite rule) to each token (a character in most cases) in the original sequence, then start over with the resulting token sequence as their new input [11]. This means that often L-Systems have no natural stopping point for rewriting, and do not need one. Usually, with an L-System, the string is rewritten a fixed number of times.

There are a few types of L-Systems, distinguished by the type of productions they use. For this paper I will only look at the most basic deterministic L-Systems (also sometimes called D0 L-Systems). The productions in this type of L-System describe the rewriting of exactly one token into a fixed sequence of tokens.

For example, every token a could be mapped to b, and b could be mapped to bc. This is usually written like in Equation (3.1).

$$\begin{aligned} p_1 &: a \mapsto b \\ p_2 &: b \mapsto bc \end{aligned} \tag{3.1}$$

When applying this L-System once, to the string a, this string is then rewritten to b. When applying the productions once more, b transforms into bc, which turns into bbc after yet another iteration.

Formally we describe the productions in a D0 L-System as one function of type $A \to A^*$. Here, A is the set of valid tokens, known as the alphabet of the system. A^* is the set of all combinations of letters, or words, from A. This can also be understood as $\{\lambda\} \cup A \cup (A \times A) \cup (A \times A \times A)$.... λ is used to describe the empty word. From this we can define any valid L-System, l. Lastly, let L be the set of all possible D0 L-Systems l conforming to Definition 1.

Definition 1. *An L-System l is a tuple (A, δ) where A is a set of tokens, and δ is a mapping $\delta : A \to A^*$.*

When looking back at the previously given example, this L-system can now be described as:

$$l_{\text{ex}} = (A_{\text{ex}}, \delta_{\text{ex}}), \tag{3.2}$$

where

$$A_{\text{ex}} = \{a, b, c\},$$

and

$$\begin{aligned} \delta_{\text{ex}} : A_{\text{ex}} &\to A^*_{\text{ex}} \\ a &\mapsto b \\ b &\mapsto bc \\ c &\mapsto c. \end{aligned}$$

The rewriting of a string by any L-System can now be formalized by a function of the form $\mathcal{L} : L \times \mathbb{N} \times A^* \to A^*$. To make this function more readable, we use a helper function $\bar{\delta}$, which applies the relevant productions to a sequence of tokens once, whereas the main function, $\mathcal{L}$, mainly keeps track of the recursion. The first parameter is the L-System used to rewrite the string from the third parameter. The second parameter, r, is the recursion parameter, which describes how many iterations of productions are applied to the string.

Definition 2. *The L-System application function $\mathcal{L} : L \times \mathbb{N} \times A^* \to A^*$ is defined as:*

$$\begin{aligned} \mathcal{L} : L \times \mathbb{N} \times A^* &\to A^* \\ (l, 0, w) &\mapsto w \\ (l, r, w) &\mapsto \mathcal{L}(l, r-1, \bar{\delta}(l, w)), \end{aligned}$$

where

L is the set of all possible L-Systems from definition 1,

$l \in L$ is the L-System that will be used to rewrite the string,

$r \in \mathbb{N}$ is the recursion parameter,

$w \in A^*$ is the string to rewrite,

and the helper function $\bar{\delta} : L \times A^* \to A^*$ is defined as:

$$\begin{aligned} \bar{\delta} &: L \times A^* \to A^* \\ ((A, \delta), \lambda) &\mapsto \lambda \\ ((A, \delta), aw) &\mapsto \delta(a)\bar{\delta}((A, \delta), w), \text{ for all } a \in A \text{ and } w \in A^*. \end{aligned}$$

Using this definition of $\mathcal{L}$, and the earlier description of l_{ex} in Equation (3.2), we can now describe the example application of this L-System to the token sequence a like this:

$$\begin{aligned} \mathcal{L}(l_{\text{ex}}, 3, a) &= \mathcal{L}(l_{\text{ex}}, 2, \bar{\delta}(l_{\text{ex}}, a)) \\ &= \mathcal{L}(l_{\text{ex}}, 2, b) \\ &= \mathcal{L}(l_{\text{ex}}, 1, \bar{\delta}(l_{\text{ex}}, b)) \\ &= \mathcal{L}(l_{\text{ex}}, 1, bc) \\ &= \mathcal{L}(l_{\text{ex}}, 0, \bar{\delta}(l_{\text{ex}}, bc)) \\ &= \mathcal{L}(l_{\text{ex}}, 0, bbc) \\ &= bbc. \end{aligned}$$

It is noteworthy that there are other types of productions, most commonly non-deterministic and parametric [11], which are often used to either allow for less discrete productions and interpretations, or introducing some more variety into the results of the system. One is not limited to using only a single type of L-System, but it is easier to think of each type as a property of the system. And even though these types are not used in this paper, they will be reviewed shortly for a more complete overview.

Non-deterministic L-Systems Normally an L-System is deterministic, which means that the production function δ is deterministic. In this context, any production function δ is considered deterministic if for every token, it always maps to same string, regardless of any other factors. If an L-System is non-deterministic, this means the production function δ is not deterministic. This allows δ to return a different string for the same original token upon different applications of the function. When used in combination with probability weights for each production, these types of L-Systems are called Stochastic L-Systems [11], and are often used to introduced limited randomness into a model.

Parametric L-Systems Parametric L-Systems introduce another property to each token: their parameter list. In a parametric L-System each token has a parameter list which can be used to influence the outcome of the production function [11]. This transforms the function δ from type $A \to A^*$ into, typically, a function of type $A \times \mathbb{N}^* \to A^*$. However, the exact choice of parameters depends, of course, on the way the L-System is being used.

Context-Sensitive L-Systems Another commonly seen type of L-System is the context sensitive L-System. In this case the type of δ changes from $A \to A^*$ to $A \times A \times A \to A^*$, with the first and last A in this signature being the token directly succeeding and preceding the token being interpreted. In the case of an Context-Sensitive L-System the productions are only applied if the succeeding and preceding tokens also match.

3.2 Turtle Graphics

L-Systems alone only describe how to transform a short string, often consisting of one token or a few tokens, into a far more complex string. When modeling a plant, or a maze, these strings alone do not give a good idea of what the result looks like, and a visualization method is needed to transform this string into a representation better suited for human interpretation. The technique most often used for this is called Turtle Graphics.

Turtle Graphics is applied after the L-System has been applied. The most important part of this technique is called the turtle, a virtual actor on a, usually, 2D plane. This turtle is fed the resulting token sequence from the L-System, and interprets each token as a command. What each of these commands means to the turtle is highly dependent on the task at hand, but in most cases the turtle contains at least some simple commands such as “walk forward”, “turn left”, or “turn right”. The path this turtle

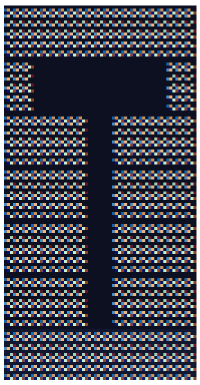

Figure 3.2: Shape resulting from a turtle parsing *ff*[*rff*][*rrrff*].

traversed after it has parsed the entire token sequence is then used as an image representation of the string produced by the L-System.

A turtle could for example be instructed that the token *f* means: walk forward 1 step, and the token *r* means: turn right 90 degrees. If this turtle were to be fed the string *ffrf*, the turtle would take two steps forward, turn right, and take one more step, producing the image that can be seen in Figure 3.1.

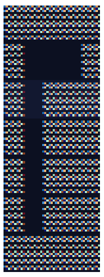

Figure 3.1: Shape resulting from a turtle parsing *ffrf*.

So far, the turtle can only walk one step at a time. This limits the amount of shapes that can be drawn effectively. Especially branching structures are hard to draw, because the turtle would have to receive an inverted sequence of commands to get back to the branching point. For this purpose we introduce a new type of command to the turtle. Commonly the tokens [and] are used as the branching commands [11]. The [command saves the current position and heading of the turtle to memory, and the] command jumps to that position without crossing any other points.

The string *ff*[*rff*][*rrrff*], for example, causes the turtle to take 2 steps forward, save its position, take 2 steps right, return to the branching point and take 2 steps left, producing the image that can be seen in Figure 3.2.

3.3 Evolutionary algorithms

Evolutionary algorithms are a type of algorithm which attempts to mimic the success of evolution in finding viable solutions to a problem. There are many different forms of evolutionary algorithms, but they almost always use a set of solutions, known as the population, and a fitness function. The fitness function describes how "good" any given solution is, often based on how close it is to the ideal solution.

Usually an initial population of minimum viable solutions is created, and these are "bred". Breeding can happen by simply cloning the population member, or by cross-breeding two different population members, and always involves some sort of random mutation during the breeding process. Depending on the needs of the algorithm only the top x population members, scoring highest on the fitness function, are allowed to breed and produce offspring, or only the top x members of the produced offspring are allowed to survive and form the next generation.

This process is repeated multiple times, until the population contains sufficiently viable solutions, and one solution is selected as the top candidate. The result is often a solution which humans could not easily have thought of, and which is highly specialized for one particular purpose. The overall quality of this solution often depends heavily on the chosen fitness function, manner of breeding and mutation, population size, and number of generations. This risks the solution being over-specialized, or under-performing due to the algorithm getting stuck in a local maximum of the fitness function. It is therefore important, when using an evolutionary algorithm, to keep the population diverse, and large enough to allow for relatively radical changes, which might eventually benefit the entire population.

For illustration purposes, a simplified algorithm for sexual evolution can be seen in Algorithm 1. The algorithm takes 3 parameters. P: the current population, from which the next generation stems. b: a breeding/mutation function which is responsible for creating offspring when given two population members, and applying random mutations in that process. f : a fitness function which is used to to select the most viable members of the population.

Eventually, because the mutations are random, this algorithm will keep selecting solutions that score higher and higher on fitness function f with every generation.

Algorithm 1 Evolution algorithm with asexual reproduction.

```
1:  function EVOLVE($P, b, f$)                        ▷ Evolve population $P$.
2:      $P' \leftarrow \emptyset$
3:      for $m_1 \in P$ do                             ▷ This example uses sexual reproduction.
4:          $m_2 \leftarrow rand(P)$                   ▷ Select $m_2$ randomly from $P$.
5:          $m'_1 \leftarrow b(m_1, m_2)$              ▷ Each pair produces two offspring.
6:          $m'_2 \leftarrow b(m_1, m_2)$              ▷ Each offspring is mutated randomly by $b$.
7:          $P' \leftarrow P' \cup \{m'_1, m'_2\}$
8:      end for
9:      $P' \leftarrow sort(P', f)$                    ▷ Sort $P'$ according to $f$.
10:     $P' \leftarrow top(P', |P|)$                   ▷ Select only the top half of the population.
11:     return $P'$                                    ▷ Return a new population of equal size.
12: end function
```

Chapter 4

Research

In this section I will explain how exactly the different theoretical concepts from Section 3 are applied. Starting again with L-Systems, then Turtle Graphics and lastly, the evolutionary algorithm. Because the idea behind this thesis largely comes from the paper by Ashlock, Lee, and McGuinness [3], any differences in approach will be explained where applicable.

4.1 L-System

The type of L-System used is D0 L-Systems. A formal definition of this type of L-System has already been given in Section 3.1. However, for use in the evolutionary algorithm we still need to choose the alphabet of tokens we use, the initial L-Systems, and a sequence of tokens each L-System is applied to.

As an alphabet A, the following was used:

$$A = \{f, b, r, l, [,], s, e, a, b, c, d, f, g, h, i, j, k, l, m, n\}.$$

This alphabet contains all tokens which will be used as turtle commands (f, b, r, l, [,], s, and e), and a few extra commands without any special meaning. These extra commands are used to mitigate the effect of the choice to use D0 L-Systems. This encompasses all randomness into the evolutionary algorithm, while still allowing the L-System to be relatively flexible in the structures it generates. If the L-System were only able to use production from any valid turtle command to another, it would be almost impossible to generate diverse and still complex structures.

As an initial L-System, $l_\varepsilon = (A, \delta_\varepsilon)$, a simple D0 L-System with only two production is used: the first production $p_1 : a \mapsto f$, and the second

$p_2 = f \mapsto ff$. So, the production function δ_ε, looks as follows:

$$\begin{aligned}\delta_\varepsilon : A &\to A^* \\ a &\mapsto f \\ f &\mapsto ff \\ x &\mapsto x, \text{ for all other } x \in A.\end{aligned}$$

The initial L-System, l_ε, will be used for the first population of the evolutionary algorithm described later on in Section 4.3.

In order to effectively fix the string each L-System is tested against, a function $\mathcal{L}'$ was defined. $\mathcal{L}'$ is a variant of $\mathcal{L}$ from definition 2, adjusted to use a pre-defined string, saf, instead of a third parameter. Like in Definition 2, L refers to the set of all possible L-Systems as defined in Definition 1.

$$\begin{aligned}\mathcal{L}' : L \times \mathbb{N} &\to A^* \\ (l, n) &\mapsto \mathcal{L}(l, n, saf)\end{aligned}$$

4.2 Turtle Graphics

For the purpose of this thesis the Turtle Graphics technique was used to create an undirected weighted graph out of the string resulting from an L-System. Typically the result of a applying this technique is an image, as has been explained in Section 3.2. However, in this case it made more sense to let the turtle create an undirected weighted graph, using the path walked by the turtle as the paths in the maze, since the fitness functions that will be used all relate to the length of the path between two points. These problems have already largely been solved efficiently for graphs, and this way the solutions can be reused.

The basic principle is still the same as with typical Turtle Graphics. However, instead of drawing a line with every move command, we insert a vertex into the graph after each move command, and create an edge between the originating and the newly created vertex with a weight of 1. When starting a branch, using the token [, we save the current position and heading (direction) of the turtle on a stack. When ending a branch, using the token], we pop the position and heading of the stack and warp the turtle there without creating an additional vertex or edge in the graph.

In order to formally define this turtle, the type of coordinate system, graph and memory state need to be defined first.

Coordinate System For the coordinate system 2D Euclidean space is used, snapped onto a $\mathbb{N} \times \mathbb{N}$ grid, meaning the coordinates of each point in the space consists only of numbers in $\mathbb{N}$. The set of points $\mathcal{P}$ is defined in Equation (4.1). For expressing the heading of the turtle a vector in the same space is used. These vectors are elements of the set $\mathcal{V}$ as defined in Equation (4.2).

$$\mathcal{P} = \{(x, y) \mid x, y \in \mathbb{N}\} \tag{4.1}$$

$$\mathcal{V} = \{\begin{pmatrix} x \\ y \end{pmatrix} \mid x, y \in \mathbb{N}\} \tag{4.2}$$

Two helper functions: ρ_L and ρ_R are also defined. Function ρ_L for rotating left by 90 degrees and function ρ_R for rotating right by 90 degrees.

$$\begin{aligned} \rho_L : \mathcal{V} &\to \mathcal{V} \\ v &\mapsto \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} * v \end{aligned} \tag{4.3}$$

$$\begin{aligned} \rho_R : \mathcal{V} &\to \mathcal{V} \\ v &\mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} * v \end{aligned} \tag{4.4}$$

Graph A undirected weighted graph g consists of a set of vertices V, the set of the edges between those vertices E, and a function, Δ, from E to $\mathbb{N}$ which maps each edge to its weight. Or, more formally:

Definition 3. *An undirected weighted graph, g, is a tuple:*

$$g = (V, E, \Delta),$$

where

> $V \subseteq \mathcal{P}$ *is the set of vertices,*
>
> $E = \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$ *is the set of edges in the graph,*
>
> $\Delta : E \to \mathbb{N}$ *is a function mapping all edges in E to their weight.*

Then let G be the set of all possible graphs g as defined in Definition 3.

To use this graph as a maze, it needs to be possible to determine the starting point, and the end of the maze. For this purpose the definition of g is adjusted to also include s and an e, two special vertices.

Definition 4. *A maze m is a tuple:*

$$m = (V, E, \Delta, s, e),$$

where

> (V, E, Δ)*, is a graph as defined in Definition 3,*
>
> $s \in V$ *is the starting vertex of the maze,*
>
> $e \in (\{\perp\} \cup V)$ *is the end vertex of the maze, with* $\perp$ *meaning no end has been set yet.*

Let M be the set of all possible mazes m as defined in Definition 4. Also, let m_ε be the empty maze. This maze only contains a single vertex $(0, 0)$, which is also the starting point, and a Δ which is undefined as of yet, denoted by $\perp$, since there are no edges. This empty maze will be used as the initial maze for the turtle.

$$m_\varepsilon = (\{(0, 0)\}, \emptyset, \perp, (0, 0), \perp)$$

Turtle State A turtle state is used to keep track of the current position of the turtle, and the maze it is building. In this state, $p \in \mathcal{P}$ is the current position of the turtle, $\theta \in \mathcal{V}$ is the current heading (direction) of the turtle, $m \in M$ is the maze the turtle is currently building, and σ' is either another state, or nothing, which is represented as $\perp$. This last member of the state is used to facilitate branching. More formally:

Definition 5. *A turtle state σ is a tuple:*

$$\sigma = (p, \theta, m, \sigma'),$$

where

> $p \in \mathcal{P}$ *is the current position of the turtle,*
>
> $\theta \in \mathcal{V}$ *is the current heading of the turtle,*
>
> $m \in M$ *is the maze the turtle is building,*
>
> σ' *is either* $\perp$ *or any other state which also adheres to the definition of* σ*.*

Let S be the set of any and all possible states σ as defined in Definition 5. Also, let $\sigma_\varepsilon = ((0,0), \begin{pmatrix} 0 \\ 1 \end{pmatrix}, m_\varepsilon, \bot)$ be the empty state the turtle will be in initially. This means the turtle starts at the coordinates $(0, 0)$, facing upwards, with an empty maze.

Turtle Commands Now we define multiple functions, each corresponding to a command the turtle recognizes. Each of the functions C_x for $x \in \{f, b, r, l, s, e, PUSH, POP\}$ has the following type: $C_x : S \to S$.

The first two functions use a helper function $\mathcal{E}$, which is used to add an edge to a graph, while simultaneously adding both vertices the connected by the edge if they are not yet in this graph. It takes 4 arguments, a maze, two edges and the weight for the edge to be created, and returns a new maze with the newly added edge.

$$\begin{aligned} \mathcal{E} : M \times V \times V \times \mathbb{N} \to& M \\ ((V, E, \Delta, s, e), v_1, v_2, n) \mapsto& (V \cup \{v_1, v_2\}, \\ & E \cup \{\{v_1, v_2\}\}, \\ & \Delta', \\ & s, e), \end{aligned} \tag{4.5}$$

where

$$\Delta' = x \mapsto \begin{cases} n, \text{ if } x = \{v_1, v_2\} \\ \Delta(x), \text{ otherwise.} \end{cases}$$

The token f instructs the turtle to take one step forward. This is described in C_f.

$$C_f = (p, \theta, M, \sigma) \mapsto (p + \theta, \theta, \mathcal{E}(M, p, p + \theta, 1), \sigma)$$

The token b instructs the turtle to take on step backwards. This is described in C_b.

$$C_b = (p, \theta, M, \sigma) \mapsto (p - \theta, \theta, \mathcal{E}(M, p, p + \theta, 1), \sigma)$$

The token r instructs the turtle to turn 90 degrees to the right. This is described in C_r.

$$C_r = (p, \theta, M, \sigma) \mapsto (p, \rho_R(\theta), M, \sigma)$$

The token l instructs the turtle to turn 90 degrees to the left. This is described in C_l.

$$C_l = (p, \theta, M, \sigma) \mapsto (p, \rho_L(\theta), M, \sigma)$$

The tokens s and e instruct the turtle to set its starting and end point respectively. This is described in C_s and C_e.

$$\begin{aligned} C_s &= (p, \theta, (V, E, \Delta, s, e), \sigma) \mapsto (p, \theta, (V, E, \Delta, p, e), \sigma) \\ C_e &= (p, \theta, (V, E, \Delta, s, e), \sigma) \mapsto (p, \theta, (V, E, \Delta, s, p), \sigma) \end{aligned}$$

The tokens [and] instruct the turtle to save the current state, or restore a previously saved state respectively. This is described in C_{PUSH} and C_{POP}.

$$\begin{aligned} C_{PUSH} &= (p, \theta, M, \sigma) \mapsto (p, \theta, M, (p, \theta, M, \sigma)) \\ C_{POP} &= (p, \theta, M, \sigma) \mapsto \begin{cases} (p', \theta', M, \sigma') & \text{iff } \sigma = (p', \theta', M', \sigma') \\ (p, \theta, M, \bot) & \text{iff } \sigma = \bot \end{cases} \end{aligned}$$

These command function are combined in a turtle core function Υ. This function is used to generate a function from one state $\sigma \in S$ to another state $\sigma' \in S$ based on an input word in $w \in A^*$. Any tokens encountered in the string that are not recognized as a turtle command, are silently ignored.

$$\begin{aligned} \Upsilon : A^* &\to (S \to S) \\ \varepsilon &\mapsto (s \mapsto s) \\ aw &\mapsto (\Upsilon(w) \circ C_x), \text{ for } a \in \{f, b, r, l, s, e\} \text{ and } w \in A^* \\ [w &\mapsto (\Upsilon(w) \circ C_{PUSH}), \text{ for } w \in A^* \\ ]w &\mapsto (\Upsilon(w) \circ C_{POP}), \text{ for } w \in A^* \\ aw &\mapsto \Upsilon(w), \text{ otherwise, for } a \in A \text{ and } w \in A^* \end{aligned} \tag{4.6}$$

Using Υ from Equation (4.6) we can define the turtle T as a function from words, to mazes. One last helper is used in this definition, π_3, to extract the the maze from a turtle state. These two functions are defined in Equation (4.7) and Equation (4.8) respectively.

$$\begin{aligned} \pi_3 : S &\to M \\ (p, \theta, M, \sigma) &\mapsto M \end{aligned} \tag{4.7}$$

$$\begin{aligned} T : A^* &\to M \\ w &\mapsto \pi_3(\Upsilon(w)(s_\varepsilon)) \end{aligned} \tag{4.8}$$

As a small example the result from using the string *sfe* with the turtle is shown below. This results in a very simple maze, with the starting point on $(0,0)$, and the end on $(0,1)$.

$$
\begin{aligned}
T(SFE) &= \pi_3(\Upsilon(sfe)(\sigma_\varepsilon)) \\
&= \pi_3((\Upsilon(fe) \circ C_s)(\sigma_\varepsilon)) \\
&= \pi_3((\Upsilon(e) \circ C_f \circ C_s)(\sigma_\varepsilon)) \\
&= \pi_3((C_e \circ C_f \circ C_s)(\sigma_\varepsilon)) \\
&= \pi_3((C_e \circ C_f \circ C_s)((0,0), \begin{pmatrix} 0 \\ 1 \end{pmatrix}, m_e, \bot))) \\
&= \pi_3((C_e \circ C_f)((0,0), \begin{pmatrix} 0 \\ 1 \end{pmatrix}, m', \bot))) \\
&\text{where } m' = (\{(0,0)\}, \emptyset, \bot, (0,0), \bot) \\
&\qquad = \pi_3((C_e)((0,1), \begin{pmatrix} 0 \\ 1 \end{pmatrix}, m'', \bot))) \\
&\text{where } m'' = \mathcal{E}((\{(0,0)\}, \emptyset, (0,0), \bot), (0,0), (0,1), 1) \\
&\qquad = (\{(0,0),(0,1)\}, \{\{(0,0),(0,1)\}\}, (\{(0,0),(0,1)\} \mapsto 1), (0,0), \bot) \\
&= \pi_3((0,1), \begin{pmatrix} 0 \\ 1 \end{pmatrix}, m''', \bot)) \\
&\text{where } m''' = (\{(0,0),(0,1)\}, \{\{(0,0),(0,1)\}\}, (\{(0,0),(0,1)\} \mapsto 1), (0,0), (0,1)) \\
&= (\{(0,0),(0,1)\}, \{\{(0,0),(0,1)\}\}, (\{(0,0),(0,1)\} \mapsto 1), (0,0), (0,1))
\end{aligned}
$$

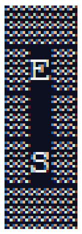

Figure 4.1: The string *sfe* visualized as an image.

So, the string *sfe* results in maze with two vertices, $(0,0)$ and $(0,1)$, an edge between this two vertices with a weight of 1, the starting point on $(0,0)$, and the end point of $(0,1)$. This maze is also visualized in figure 4.1.

4.3 Evolution

I have now defined the L-System and Turtle, but these are only the representation of the maze. Next, I defined an evolutionary algorithm, that uses this representation to find the maze that scores best on its fitness function.

Because of a number of differences between the method in this thesis, and the method used by Ashlock, Lee, and McGuinness [3], and the low amount of detail explained in their paper, it is difficult to directly compare the results of Ashlock, Lee, and McGuinness [3] to the ones found now. So, in order to determine whether or not L-Systems are a viable representation candidate for search-based PCG, a second evolutionary algorithm is run, using the (almost) exact same fitness functions, but a plain token string, instead of one generated by an L-System.

The evolution algorithm comprises a two-step process (a generation) which will be repeated multiple times. First the "breeding pairs" are selected, then they are bred. This is often called sexual reproduction, because multiple members of the population needed to breed together to form offspring.

A breeding pair consists, in this case, of two population members, which are combined to produce a third one. By choosing sexual reproduction over asexual reproduction the resulting population variety is increased, and by this the chance of overcoming local minima during the process. These are selected by designating two fractions of the population, of size x_1 and x_2, with $x_1 + x_2$ being equal to the population size. From the top x_1 members of the population, x_1 pairs are randomly selected. Each population member can be picked for multiple pairs, and some might be unlucky and not get picked at all. From the top x_2 members of the population, again x_2 pairs are randomly selected. This is the extra-fertile fraction of the population, because instead of having two offspring each, each member of the top x_2 has four offspring. By allowing for the top-most members of the population to produce more offspring, the top-layer is focused on while still allowing for some variety, by not reducing the population to only the top x_2 immediately.

Next the pairs are bred. We use two types of population members, a tuple of an L-System and a recursion parameter, and a string of plain tokens from A. The evolutionary algorithm acts the same in both cases, but the breeding function has to be adjusted for each type.

L-Systems L-System tuples are bred by averaging their recursion parameter first. Then, all productions are looped through. For each production in L-System a, a production with the same origin token is looked for in L-System b. If none is found, the production from a is used, otherwise it is a 50-50 chance which is inserted. Afterwards, all productions from b which were not yet considered are inserted as well.

Lastly the new L-System tuple is mutated. The recursion parameter has a 1 in 6 chance to be increased by one, or decreased by one, but never below 1. Each production has a 14% chance to have a random token removed or added in. When added a token, the token is selected from all valid tokens for the turtle, and the tokens a production originates from. Each production also has a 8% chance to be converted into a branch, and a 4% chance to be removed entirely. Each of these mutations is mutually exclusive. Eventually, with a 30% chance a production originating from a random token in A to $\langle F \rangle$ is added.

Token Strings Token strings are bred by looping through each token from a. With a 50-50 chance it is either inserted into the child string, or the token with the same index will be selected from b. If a token from b should be selected, but b does not have a token at the same index (e.g. because it is shorter than a), no token will be inserted. If b is longer than a, each remaining token in b with an index higher or equal to the size of a is inserted into the child string with a 50% chance.

Each token in the resulting child string has a 10% chance to be duplicated, removed or mutated into another random valid token. Valid tokens, in this case, include all tokens for which the turtle has a defined action. The string is limited to these tokens, because this significantly improves performance by reducing the junk which needs to be parsed.

4.4 Fitness functions

As a last ingredient the evolutionary algorithm also needs a fitness function, to determine the best members of the current population. In this section, three fitness functions will be presented. Each of these fitness functions rewards a different property in the maze. One rewards the maze for making its shortest path from start to end as long as possible, another rewards the maze for the total path length it generates, and the third rewards the maze based upon a mix of the first two.

This is where some key differences between the approach in this thesis, and the approach by Ashlock, Lee, and McGuinness [3] appear. Ashlock, Lee, and McGuinness [3] use fitness functions which operate directly on the image of the maze. To simplify this, they used checkpoints placed on the grid in pre-defined positions which are then strung together into a simple graph. Their fitness functions then reward either the number of checkpoints on any given path from start to finish, the number of checkpoints that are not on any path to the finish, or whether a specific checkpoint was visited or not. Because the turtle from Section 4.2 results in a graph directly, and the boundaries of the maze are not fixed beforehand, the fitness functions in this thesis do not use checkpoints. This causes a few differences in approach, but the core idea remains: that the fitness function is one of the most important parts, as it is used to steer the properties of the final maze.

Even though the fitness functions all reward different properties in the resulting maze, they also have some things in common. Each fitness function includes a complexity punishment. This complexity punishment is accomplished by defining a function $c_L : L \times \mathbb{N} \times A^* \to \mathbb{N}$ for L-Systems and a function $c_T : A^* \to \mathbb{N}$ for token strings.

The function c_L sums the length of the resulting token string, with the recursion parameter and the length of the longest production in the L-System. This is to make sure that the evolutionary algorithm does not generate strings with a lot of non-sense tokens, because this can affect the performance of the system adversely. Production length is punished as well to make sure the L-System does not generate its entire maze inside a single production.

$$\begin{aligned} c_L &: L \times \mathbb{N} \times A^* \to \mathbb{N} \\ &((A, \delta), r, w) \mapsto |w| + r + \max_{x \in A}(|\delta(x)|)\}) \end{aligned}$$

The function c_T is just the length of the token string. This complexity function is far simpler, because properties such as the number of iterations do not exist on a token string.

$$\begin{aligned} c_T &: A^* \to \mathbb{N} \\ &w \mapsto |w| \end{aligned}$$

The first fitness function, $\mathcal{F}_1$, rewards the length of the shortest path from start to end, expressed by the sum of the weights of all edges on this path. This length is multiplied by 10, and then the complexity is subtracted from it to get the final score. The shortest path from start to end is denoted here by $min\_path : M \to \mathbb{N}$, and returns the minimum path length in a maze from the maze's start to the maze's end. In the implementation Dijkstra's shortest path algorithm was used as *min_path*. In all of these fitness functions T will be used as well, with T being the turtle from Equation 4.8.

When considering an L-System, the variant $\mathcal{F}_{1L}$ is used.

$$\begin{aligned} \mathcal{F}_{1L} &: L \times \mathbb{N} \to \mathbb{N} \\ &(l, r) \mapsto 10p - c, \text{ if } 10p > c \\ &(l, r) \mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.9}$$

where

$p = (min\_path \circ T)(w)$, the raw score of the system,

$c = c_L(l, r, w)$, the complexity value of the system,

$w = \mathcal{L}'(l, r)$, the token string resulting from the application of the system.

When considering a token string, the variant $\mathcal{F}_{1T}$ is used.

$$\begin{aligned} \mathcal{F}_{1T} : A^* &\to \mathbb{N} \\ w &\mapsto 10p - c, \text{ if } 10p > c \\ w &\mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.10}$$

where

$p = (min\_path \circ T)(w)$, the raw score of the token string,

$c = c_T(w)$, the complexity value of the token string.

The second fitness function, $\mathcal{F}_2$, rewards the total length of all paths, no matter the start or end of the path. This length is once more multiplied by 10, and then the complexity is subtracted from it to get the final score. The total path length is computed by summing the weight of all edges in the maze.

When considering an L-System, the variant $\mathcal{F}_{2L}$ is used.

$$\begin{aligned} \mathcal{F}_{2L} : L \times \mathbb{N} &\to \mathbb{N} \\ (l, r) &\mapsto 10p - c, \text{ if } 10p > c \\ (l, r) &\mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.11}$$

where

$p = \sum_{e \in E} \Delta(e)$, the raw score of the system,

$c = c_L(l, r, w)$, the complexity value of the system,

$w = \mathcal{L}'(l, r)$, the token string resulting from the application of the system,

$(V, E, \Delta, s, e) = T(w)$, the maze created by the turtle.

When considering a token string, the variant $\mathcal{F}_{2T}$ is used.

$$\begin{aligned} \mathcal{F}_{2T} : A^* &\to \mathbb{N} \\ w &\mapsto 10p - c, \text{ if } 10p > c \\ w &\mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.12}$$

where

$p = \sum_{e \in E} \Delta(e)$, the raw score of the token string,

$c = c_T(w)$, the complexity value of the token string,

$(V, E, \Delta, s, e) = T(w)$, the maze created by the turtle.

Lastly, the third fitness function, $\mathcal{F}_3$, combines both $\mathcal{F}_1$ and $\mathcal{F}_2$, and attempts to strike a balance between the total path size, and the shortest path from start to end. $\mathcal{F}_3$ emphasizes the shortest path from start to end a little bit more, to prevent the evolutionary algorithm from discarding the shortest path entirely, and simply generating a maze with the start and end on the same vertex.

When considering an L-System, the variant $\mathcal{F}_{3L}$ is used.

$$\begin{aligned} \mathcal{F}_{3L} : L \times \mathbb{N} &\to \mathbb{N} \\ (l, r) &\mapsto (6p_1 + 4(p_2 - p_1)) - c, \text{ if } (6p_1 + 4(p_2 - p_1)) >= c \\ (l, r) &\mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.13}$$

where

$p_1 = (\mathit{min\_path} \circ T)(w)$, the raw score of the system according to $\mathcal{F}_{1L}$,

$p_2 = \sum_{e \in E} \Delta(e)$, the raw score of the system according to $\mathcal{F}_{2L}$,

$c = c_L(l, r, w)$, the complexity value of the system,

$w = \mathcal{L}'(l, r)$, the token string resulting from the application of the system,

$(V, E, \Delta, s, e) = T(w)$, the maze created by the turtle.

When considering a token string, the variant $\mathcal{F}_{3T}$ is used.

$$\begin{aligned} \mathcal{F}_{3T} : A^* &\to \mathbb{N} \\ w &\mapsto (6p_1 + 4(p_2 - p_1)) - c, \text{ if } (6p_1 + 4(p_2 - p_1)) >= c \\ w &\mapsto 0, \text{ otherwise,} \end{aligned} \tag{4.14}$$

where

$p_1 = (min\_path \circ T)(w)$, the raw score of the token string according to $\mathcal{F}_{1T}$,

$p_2 = \sum_{e \in E} \Delta(e)$, the raw score of the token string according to $\mathcal{F}_{2T}$,

$c = c_T(w)$, the complexity value of the token string,

$(V, E, \Delta, s, e) = T(w)$, the maze created by the turtle.

The numbers used in these fitness functions were acquired through experimentation in balancing the path-score against the complexity punishment. Weighing the path-score 10 times as heavily as the complexity punishment made sure that some expansion was no problem, but the complexity of the system was not allowed to grow rampant.

4.5 Implementation

In order to obtain the results presented in the next chapter, an implementation of this algorithm was created in C++. This implementation follows the mathematical definition given here as closely as possible. The source code of this implementation can be found on GitHub at `https://github.com/tjvoncken/bachelor-thesis`.

Chapter 5

Results

The evolutionary algorithm from Section 4.3 has been executed several times, with different values for the number of generations, and the size of the population. For each fitness function, the evolutionary algorithm was executed with both a population size of 100 and 10000 generations, and a population size of 500 and 1000 generations. Because randomness is involved, each of these trials has been run 100 times, and their results aggregated.

The graph in Figure 5.1 shows the results when running with fitness function $\mathcal{F}_3$, the function combining the other two, and population size 100. The graph in Figure 5.2 shows the results when running with the same fitness function, but a population size 100. The graph in Figure 5.1 shows 10000 generations, whereas the graph in Figure 5.2 only shows 1000. The lines represent the average score over 100 iterations for each generation. The gray area shows the standard deviation of this score.

The token string method seems to stabilize at around a score of 100 with a population of 100, or around 250 for a population of 500. The L-System method does not seem to stabilize within the calculated generations. It does however, beyond a certain number of generations, seem to consistently score higher than the token string method.

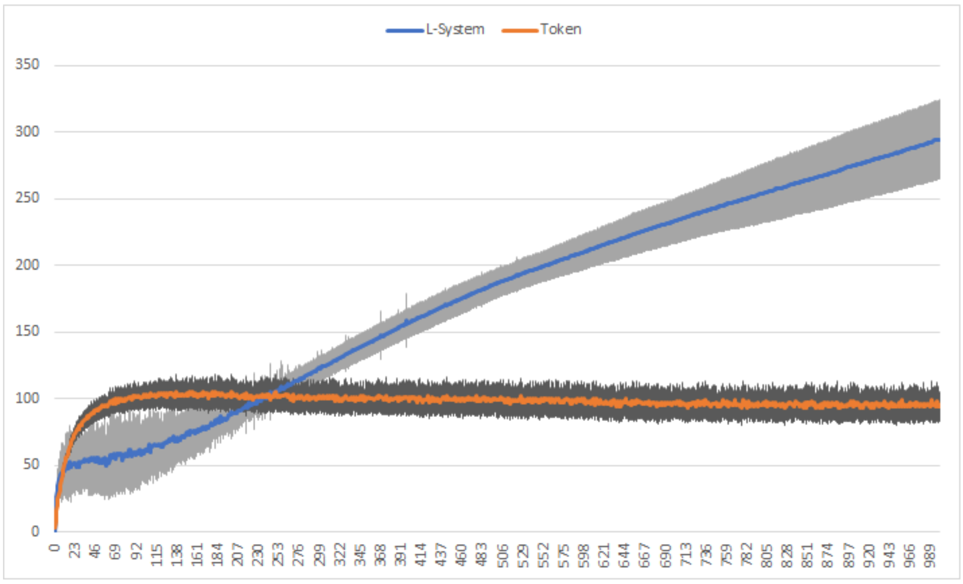

Figure 5.1: Fitness function $\mathcal{F}_3$, population size 100.

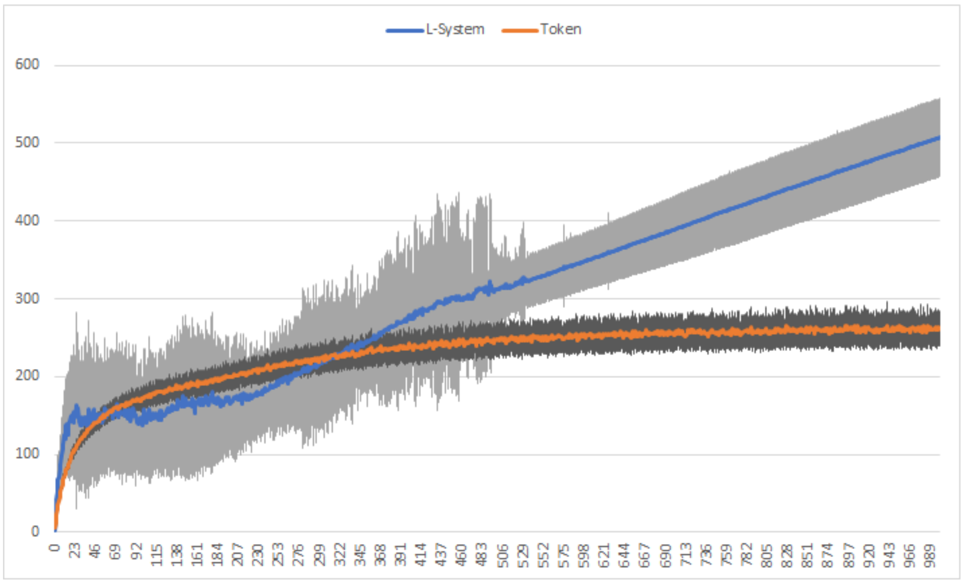

Figure 5.2: Fitness function $\mathcal{F}_3$, population size 500.

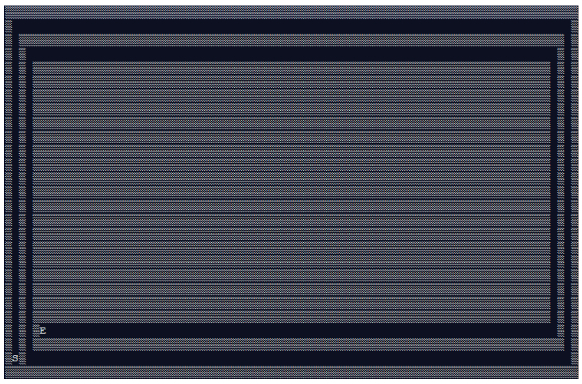

Figure 5.3: Based on an L-System, fitness function $\mathcal{F}_1$, population size 100.

When looking at one of the resulting mazes in Figure 5.3 it can be seen that the mazes all seem to attempt to maximize the length of the shortest path by making a straight path, with as few branches as possible. This supports earlier conclusions, also cited by Ashlock, Lee, and McGuinness [3], about using the shortest path between start and end as a fitness function.

The graphs for fitness functions $\mathcal{F}_1$ and $\mathcal{F}_2$ show similar results. The exact resulting values differ, of course, but this is due to the fitness function not being directly comparable. The graphs do, however, show a consistent tendency for the L-System method to outrun the token string method after only relatively few generations. The result graphs for the other fitness functions can be seen in Appendix A, as Figures A.1 and A.2 for $\mathcal{F}_1$, and Figures A.3 and A.4 for $\mathcal{F}_2$.

As for the resulting mazes, none look really like a maze. This is probably caused by a relatively poor choice of fitness function, as the results do seem to maximize the different fitness functions quite effectively. One more promising result can be seen in Figure 5.4, this result did not however score very well on its fitness function, and should probably be regarded more of an accident, than a success.

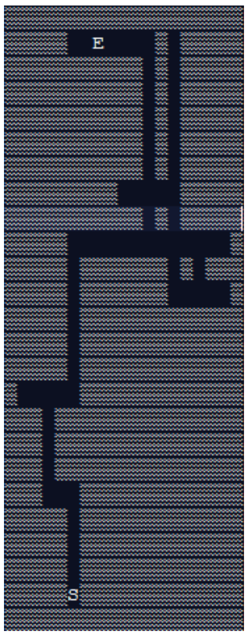

Figure 5.4: Based on a token string, fitness function $\mathcal{F}_3$, population size 100.

Chapter 6

Discussion

While the results from the previous chapter do suggest that an L-System based approach at least outperforms a token string based approach, this does not directly mean that L-Systems are a good fit for this kind of PCG work. To strengthen the conclusion from this thesis, further research will be needed to eliminate other factors. Some of these factors will be discussed in this chapter.

Performance First of all, with any evolutionary algorithm the speed of calculating the next generation is extremely important. Since evolutionary algorithms depend on random and often small mutations to progress towards their goal, usually many iterations are needed to make any meaningful progress. Furthermore, using a large population is also important in order to keep a varied population and be able to overcome local maxima and other obstacles. So, a balance has to be struck between the number of generations and the size of the population. A larger populations means a longer computation time per generation, which in turn makes it less viable to use a large number of generations, but a smaller generation might mean the algorithm gets stuck on a local maximum before it reaches a good solution. In order to mitigate the effect of this on the conclusion, the experiment was run both with 10000 generations and a population size of 100, and 1000 generations and a population size of 500.

This need for performance also forced the introduction of the complexity punishment in each fitness function. Since the L-Systems and token strings would otherwise become so cluttered with non-sense tokens or return to the same paths so many times that computing the fitness of the resulting maze becamse very expensive. This does not necessarily have a negative effect on the results, but does introduce another factor in the fitness functions which can influence them.

Another way to improve system performance, and eventually get better results, would be to attempt execution of the fitness functions directly on the string, instead of computing a maze first. This would eliminate the need for a turtle in the implementation, and allow the evolutionary algorithm to skip this, now very expensive, step. This would, however, require some additional research into how a fitness function based on a maze could be reliably translated into a fitness function based on a string.

In the end, the performance of my implementation does not seem great. Due to this, the results all have either a relatively low population size, or a relatively low generation count. In the end, the algorithm runs once in roughly 900 seconds, and this is not suited for any kind of real-time generation. This could somewhat skew the results, and a better optimized version of the code might generate more conclusive results. On the other hand, since the token string method does seem to stop improving relatively early on, it does seem that the results can be used for comparison between the two methods.

Applicability Second, the fitness functions used during this thesis, while based upon earlier work by Ashlock, Lee, and McGuinness [3], are not directly conducive to generating a good maze. When attempting to maximize the shortest path from start to end, an easy solution is to make a straight path as long as possible. Generating branches and circles only adds more complexity, while not increasing the path length. The same can be said for the other two fitness functions, since both of these are almost directly based upon path length within the maze.

Ashlock, Lee, and McGuinness [3], for example, had a fitness function rewarding the amount of cul-de-sacs (dead ends) within the structure for example. A similar approach, rewarding the number of branches not leading to the end, was not tried due to time constraints, but might give additional insight in whether an L-System backed evolutionary algorithm is actually suited for generating mazes. As is, the results merely allow the conclusion that an L-System backed evolutionary algorithm can effectively maximize the fitness functions used in this thesis.

Chapter 7

Conclusions

The data from chapter 5 does seem to support the goal of this thesis, to show that L-Systems are a viable representation for generating mazes using an evolutionary algorithm. And keeping the caveats from chapter 6 in mind, it can be stated with relative confidence, that the L-System based approach will consistently outperform the token string method, and generate a maze which maximizes the chosen fitness function.

While this does not directly support the claim that L-Systems can be used to randomly generate a maze, and this was also not observed directly in any of the results, all resulting structures did effectively maximize their respective fitness functions. This suggests that upon tweaking of the fitness function, to reward more maze-like properties, a proper maze can be generated using the approach described here.

Bibliography

[1] N. Shaker, J. Togelius, and M. Nelson, *Procedural content generation in games*. Springer, 2014.

[2] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," in *Acm Transactions On Graphics (Tog)*, ACM, vol. 25, 2006, pp. 614–623.

[3] D. Ashlock, C. Lee, and C. McGuinness, "Search-based procedural generation of maze-like levels," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 260–273, 2011.

[4] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, *Instant architecture*, 3. ACM, 2003, vol. 22.

[5] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," in *ACM Transactions on Graphics (TOG)*, ACM, vol. 29, 2010, p. 181.

[6] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, 2001, pp. 301–308.

[7] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," in *ACM transactions on graphics (TOG)*, ACM, vol. 27, 2008, p. 103.

[8] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proceedings of the 2010 workshop on procedural content generation in games*, ACM, 2010, p. 1.

[9] 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*, ACM, 2010, p. 10.

[10] J. Dormans, "Level design as model transformation: A strategy for automated content generation," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, ACM, 2011, p. 2.

[11] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

Appendix A

Result Figures

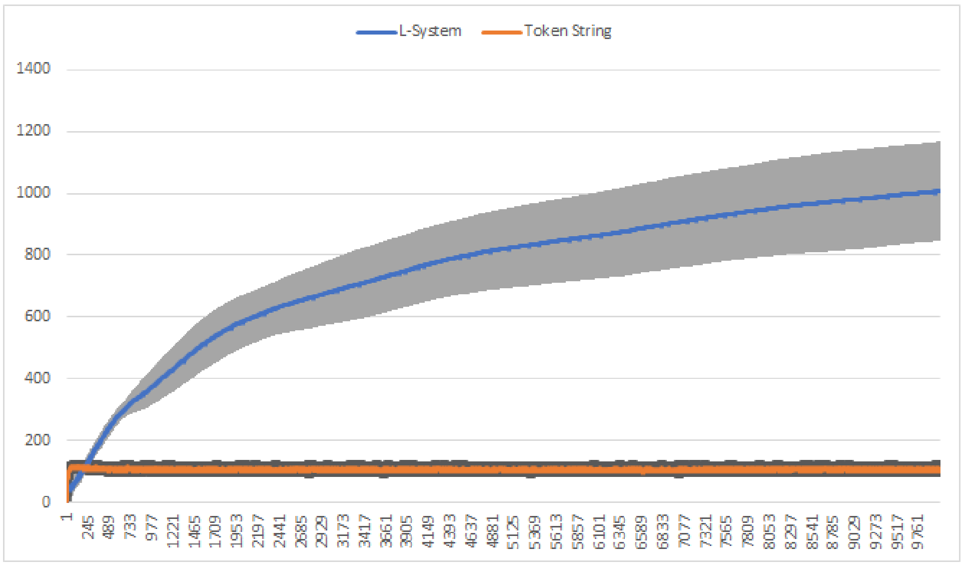

Figure A.1: Fitness function $\mathcal{F}_1$, population size 100.

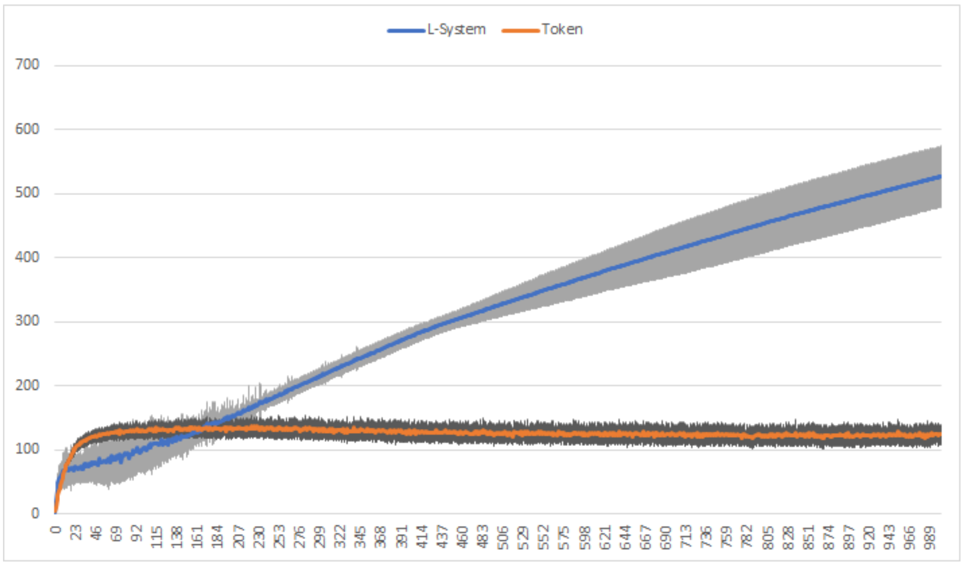

Figure A.2: Fitness function $\mathcal{F}_1$, population size 500.

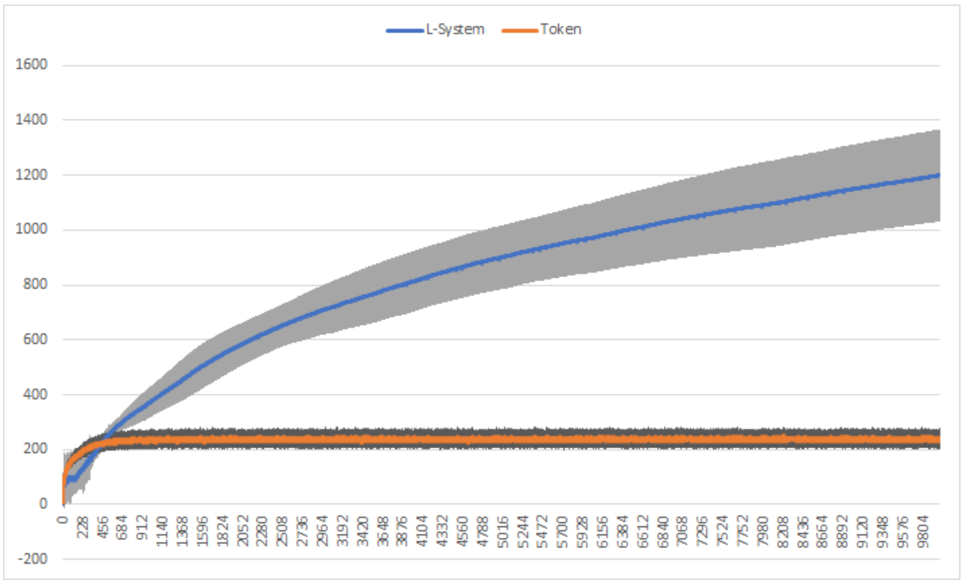

Figure A.3: Fitness function $\mathcal{F}_2$, population size 100.

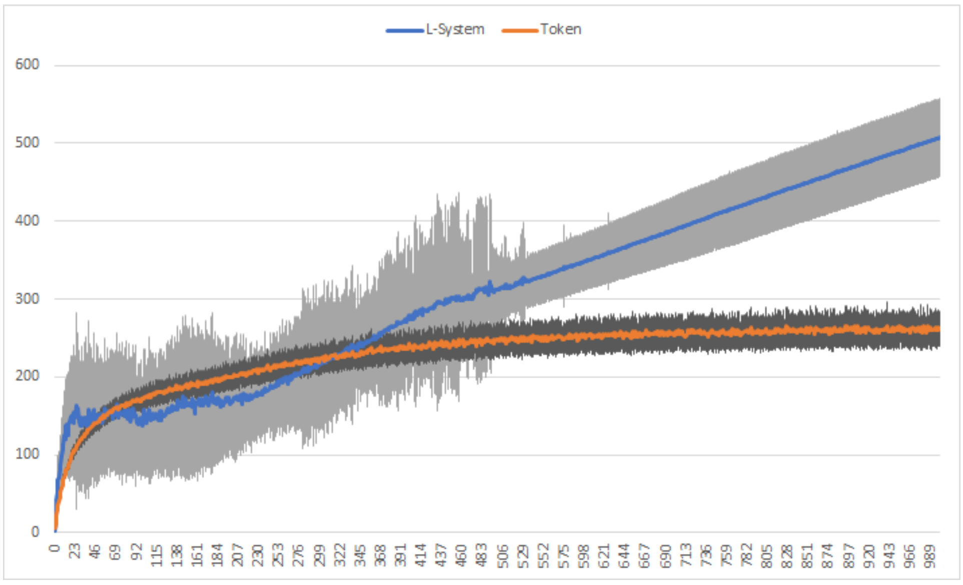

Figure A.4: Fitness function $\mathcal{F}_2$, population size 500.