ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Varga Zsombor Adorján**
2024 Hallgató törzskönyvi száma: **T/006675/FI12904/N**

Óbuda University
John von Neumann Faculty of Informatics
Software Engineering Institute

THESIS

Name of the student: **Zsombor Adorján Varga**
Registration number: T/006675/FI12904/N
Neptun's code: [redacted]

Title of the thesis:

Creating and validating procedurally generated mazes
Procedurálisan generált labirintusok létrehozása és validálása

Internal supervisor: Miklós László Sipos
External supervisor:

Deadline: 15 December 2024

Subjects of the final exam: Computer Architectures
Cloud service technologies and IT security specialization

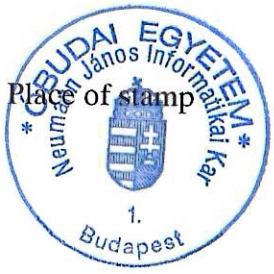
The task:

The task is to develop a labyrinth generator program that can be used as a base for map creation for a variety of games using procedural generation and tested with the A* algorithm. Study procedural generation with a focus on map creation, specifically labyrinth generation, and its latest advancements. Analyse similar solutions of this method that were used in the past. A working solution should be developed using C# and/or JavaScript. The system should be able to serve as a generator for both 2D and 3D maps, depending on what the game requires. The complexity and difficulty should be adjustable, and the system must be able to generate a world accordingly. The outcome should be tested against the A* algorithm to ensure its solvability (or multiple algorithms). A working 2D game demo should be built around this program using its difficulty setting feature to test compatibility.

The thesis must contain:

- the precise description and detailed analysis of the task to be solved,
- high-level structural model of the application,
- possible technologies that can be used for implementing each function,
- plan of the developed system, its block diagram,
- implementation process, encountered issues, and their solutions,
- testing plan, methodology, and results of the testing,
- potential applications of the developed software,
- presentation and evaluation of the results regarding the entire work.

Place of stamp



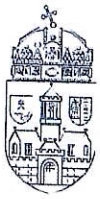
.....
Dr. habil. Zoltán Vámossy
head of institute

This specification is valid until: **15 December 2026**
ÓE HKR Section 54, paragraph (10)

I consider the thesis suitable for submission:

.....
external supervisor

.....
internal supervisor



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2024. December 13.

.....

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: VARGA ZSOMBOR ADORJÁN

Neptun Kód: [redacted]

Tagozat: NAPPALI

Telefon: [redacted]

Levelezési cím (pl.: lakcím): [redacted]

Szakdolgozat címe magyarul:

PROCEDURÁLISAN GENERÁLT LABIRINTUSOK LÉTREHOZÁSA ÉS VALIDÁLÁSA

Szakdolgozat címe angolul:

CREATING AND VALIDATING PROCEDURALLY GENERATED MAZES

Intézményi konzulens: SIPOS MIKLÓS LÁSZLÓ

Külső konzulens: -

Alk.	Dátum	Tartalom	Aláírás
1.	2024.09.29.	**Fejlesztés tervezett menetének revíziója, szakmai egyeztetés, fejlesztési teendők megkezdése.**	
2.	2024.10.20.	**Fejlesztés állapotának és haladásának áttekintése. Technikai problémák és módosítások egyeztetése.**	
3.	2024.11.17.	**Felmerülő problémák egyeztetése és megoldása. Tesztelési fázis megindítása.**	
4.	2024.12.01.	**Dokumentáció véglegesítése. Leadással kapcsolatos teendők egyeztetése.**	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat II. tantárgy követelményét teljesítette, védésre bocsájtható.

A konzulens által javasolt érdemjegy: 4

Sipos

.....

Intézményi konzulens

Budapest, 2024. 12. 06.

Absztrakt

Cím – Procedurálisan generált labirintusok létrehozása és validálása

A tanulmány célja a procedurális tartalomgenerálás területének kutatása labirintusgenerálásra való képessége és potenciálja alapján. Ezek az eszközök egy olyan automatizált labirintusgeneráló szoftver tervezésére és létrehozására lesznek használva, amely magas szintű szabályozást kínál a paraméterei felett, és számos különböző környezetben újra felhasználható. A kutatás céljából híresebb PTG alapú játékok és számos procedurális generálás technika van tanulmányozva, kategorizálva és rangsorolva a disszertációhoz való relevanciájuk alapján. A labirintusok, pontosabban útvesztók, történelme és tipológiája, és ezeknek alkotására és bejárására írt algoritmusai is tanulmányozva vannak. Ezek az algoritmusok szükségesek lesznek program validációjához. A munka a 2-dimenziós labirintusokra fókuszál, viszont 3-dimenziós megoldások is említve vannak, hogy be legyen mutatva a terület potenciálja ezen az ágon is. A kutatásban az átírásos, pontosabban a gráf átírásos módszer, lett a megoldáshoz legmegfelelőbb metódus a magas irányíthatósága, alkalmazkodóképessége és egyszerűsége miatt. Az útvesztő érvényesítéséhez az A* algoritmus a legalkalmasabb, miután a világ egyik leghatékonyabb útkereső algoritmusaként van számontartva. Az irodalomkutatásban szintén kimutatták, hogy egy bizonyos probléma megoldásához egy hibrid megközelítés gyakran a legjobb. A megközelítéshez a gráf átírásos módszer módosítva lett, hogy illeszkedjen a labirintusgenerálás és az alkalmazkodóképesség kritériumaihoz. Ez az átírásos módszer szabályainak és adatbázisának a szétválasztásával van megoldva, a tervezett programot példának véve: egy kezdőponttól kijárathoz vezető utat és egy zsákutca generáló metódus. Ezt kihasználva a rendszerben potenciálisan bármilyen útvonal vagy cella generálása irányítható. A megoldásban először csak az útvonalakat leíró gráf jön létre, amit a rendszer egy rácsra vázol rá. Három tervezett felhasználási módja lesz a rendszernek: a generált labirintusok mentése szerializált objektumként, egy szemléltető kiegészítő, amivel lépésről lépésre lehet elemezni a generálás folyamatát és fejleszteni egy játékot, ami kihasználja a program adottságait.

Abstract

Title – Creating and validating procedurally generated mazes

This study aims to research the field of procedural content generation in relation to its capability and potential to generate mazes. The tools found are used to plan and create an automated maze generation software that offers a high level of control over its parameters and can be reused for several different environments. For this purpose, famous examples of videogames utilizing PCG, like Rogue and Minecraft, and many different PCG methods are studied, categorized, and ranked by their necessity for this thesis. The history and typology of mazes are studied, along with algorithms that have been used to write and traverse mazes. These algorithms are essential to find a suitable validation tool for the generator. The focus of this work is on 2-dimensional mazes; however, some 3-dimensional solutions are also covered to show potential in the field. Dynamic difficulty adjustment is discussed as well, for the same reason. In this search, the rewrite, more particularly the Graph Rewrite method, was deemed most sufficient to the purpose due to their high level of control, adaptability, and simplicity. For traversal and validation, the A* algorithm is most suitable due to it being a staple algorithm for efficient path finding. The findings also showed that creating hybrid methods from the existing solutions to solve a certain problem has great results and are most often used in games. For the approach, the graph grammar method is modified for greater adaptability in maze generation. This is done by separating replacement types by having different rules and set pieces, as in the case of this thesis: start-to-end path rewriting and dead-end rewriting. This method has the potential to be extended by special paths or cells for whatever a problem requires. The graph of the layout is generated first which is then mapped onto a grid. There are three planned deployments of the generator: saving generated mazes as serialized objects, creating a visualizer extension to study the step-by-step generation process, and developing a videogame using the assets provided by the system.

Table of Contents:

1. Introduction

Creating landscapes that are both compelling and immersive is a key component of creating engaging player experiences in the world of digital simulations and interactive entertainment. Maze-like locations have always captivated players because they test their reflexes and spatial awareness. Examples of these environments include the enigmatic passageways of ancient temples and the futuristic, labyrinthine landscapes of science fiction worlds.

The concept of Procedural Content Generation has been around for decades, playing a role in videogame culture since nearly its genesis. PCG, instead of hardcoding, utilizes algorithms for content creation and is used in many fields, from generating textures, maps, landscapes, characters to even music and stories. My main inspiration behind this project is the Binding of Isaac, which is one of the titans of indie gaming. The generation techniques used in the game have always captivated me and made me fall in love with indie games and the roguelike genre. Since childhood it has been my goal to take part in game development in the future and was one of the driving reasons why I wanted to study programming in the first place. As time passed and as I gained programming knowledge this interest shifted towards PCG. My intention with this project is to deepen my knowledge in the subject, create a unique solution that could perhaps lead me to one day expand this part of the gaming industry. The popularity of PCG is and has been on a constant rise since its inception, with many approaches, methods and algorithms extending the concept further along the way.

In this thesis, I will cover procedural generation with a focus on videogame maps and layouts, going over influential solutions and popular methods and examining their potential for labyrinth generation. I will dive into the history and types of labyrinths and their PCG's relation to them. The concept of dynamic difficulty will also be examined and put into the context of PCG. After the dissection of the literature covering these topics, I will pick out the necessary technologies and solutions that will aid me in the development of a unique maze generator and write a clear narrative of how this process will transpire. My goal is to find the most efficient, controllable, and scalable combination of these methods to create a flexible labyrinth generator that can be reused for a multitude of games (2D and 3D) with difficulty and complexity parameters that are highly operable and write a working videogame around it using the tools provided by the system.

2. Literature Study

2.1. Similar Systems

In this review I will be going over the evolution of map/dungeon generation starting with the game, that coined the phrase „rogue-like” and originally popularized procedural generation in game development, to modern day examples like The Binding of Isaac, which serves as the inspiration for my thesis, also mentioning other solutions using different methods of PCG.

2.1.1. Rogue (1980)

Rogue is a top-down adventure game developed in 1980 by Michael Toy and Glenn Wichman in C language. Contrary to widespread belief, this is not the first game to use pseudorandom number generators for level creation. Beneath Apple Manor, published in 1978, was the originator of the concept; however, the game's graphics were limited due to the capabilities of personal computers at the time. UNIX systems solved this problem by introducing the curses library, which enabled the development of text-based interfaces by using cursor addressing [1]. Rogue was one of the first games to capitalize on this technology. While not the first dungeon crawler, Rogue would inspire a whole new videogame genre with its nuances. Figure 1 displays a DOS port of the game (1984) emulated on a web browser (the original version was developed for monochrome screens and used different ASCII characters in some places).

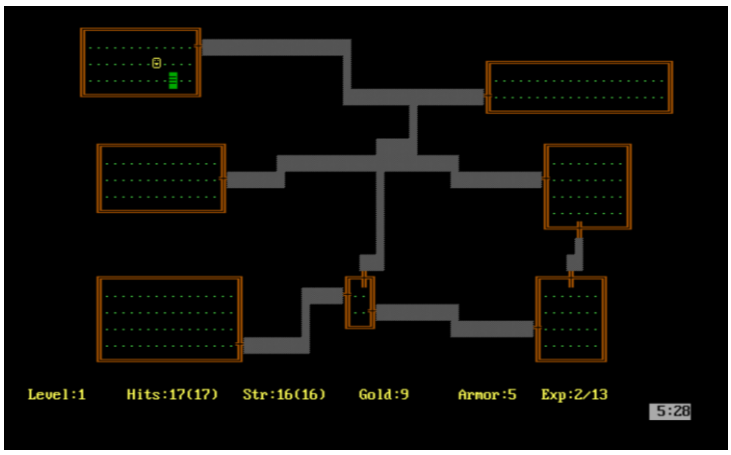


Figure 1: Rogue generated map

The fittingly named Rogue Map Generation Algorithm was simple but very efficient and would be the basis for countless future dungeon generator algorithms. Two main function calls make up the generation process: doRooms() and doPassages() [2]. Based on the given maximum number of rooms, the doRooms() function divides the playing field into cells of a grid, taking the root of the number and estimating upwards to ensure no overflow happens. In each cell, there can be a maximum of one room. Each room’s size is determined randomly within the confines of the cell. This is achieved with the position (upper-left corner of the room) and the maximum (lower-right corner of the room) properties of the room; these values cannot exceed the size of the cell. After the calculations, the rooms are drawn on the map.

The doPassages() function creates a connected graph using graph theory connecting the rooms generated prior, where the vertexes are the rooms and the edges are the passages, where every room is connected to at least one passage. The function adds some random passages as well for variety. Two functions carry out the connections: the mapPotentialConnections() function iterates through the map, checking for neighbours to the right and bottom of each cell, and the connect() function actually draws out these passages. It does this by checking which walls of the two rooms are facing each other, determining where the exact point of connection will be made on the wall, and then choosing a turning point where the two points can be lined up. These passages can be bidirectional as well.

These two functions dictate the average running time of the algorithm, dn^2, where doRooms() is n and doPassages() is dn, d being the average distance between rooms [2].

While the Rogue Map Generation Algorithm was highly effective and efficient, it lacks any real variety in its simplicity. The straightforward map layout and limited connection variations cause the maps to be similar after a couple of plays. The time it takes to generate one of these maps also increases drastically as size increases, making the system unsuitable for scaling.

2.1.2. The Binding of Isaac (Original/Rebirth) (2011/2014)

When it comes to the roguelike genre of games, The Binding of Isaac is often the conversation starter, alongside games like Runescape and Spelunky. The original version was developed by Florian Himsl. It combined the map layout and feel of the original Legend of Zelda (1986) with the random nature of roguelike procedural generation. It was an experimental, part-time project developed in Flash; thus, the original was bare-bones and unbalanced at the start. The game was reworked and greatly extended with the release of The Binding of Isaac Rebirth and subsequent releases of downloadable content with the much more reliable C++ language and a constantly growing team of developers; in fact, add-ons are still being created 10 years later.

A mixed-initiative content generation approach is taken by this game [3], which means that in development, human-made and computer-generated content are mixed to achieve the final product. Much like Rogue, both iterations of Isaac use grids in their level generation, which we can clearly see from their floor layouts. The movement between levels is also nearly identical: moving down trapdoors to get from one floor to another. The human element comes in the form of guaranteed story beats, the floor (coupled with difficulty) range for each beat, room type based on the current floor, even special floor layouts, and other non-map-related entities like certain boss fights, collectibles, etc., with everything in between being generated given these parameters.

Unlike Rogue, there are no passages, and room size isn't randomized within a grid. Instead, every room is interconnected, forming a maze-like structure, merging these concepts into one. At the start of each floor, the player starts in a cell in the middle of the grid [4]. Each dead end (rooms with a single connection point) can be a potential special room (boss, shop, etc.). Secret rooms are also created, having unique properties for each generation. The look and feel of the two Binding of Isaac titles are very similar; however, the approach in which the generation happens is very different, enough to analyse them separately.

In the 2011 version, each map was generated as the floor loaded. Every room takes up a cell on the 9x8 grid, each being numbered. Changing the number by 1 would move it to

a neighbour horizontally, while changing it by 10 would move it vertically [5]. When clicking play or moving a floor down, the algorithm creates the next map on the fly. At first, the topology is decided by assigning doors randomly out of the four possible, starting with the starting room, and moving along the map following pre-set rules based on the current floor. It's quite a memoryless and inefficient model, as said by Himsl himself [4]. Since it calculates everything room by room, the system tends to create doors that lead out-of-bounds, after which the process would restart, making for a very trial-and-error approach. In theory, "winning the lottery" is also possible, resulting in an infinite loading screen [4]. It also doesn't account for the lower limit of rooms to be created, so if the generated map happens to only have 5 rooms instead of 6, the process is repeated. After the layout is finalized, the contents of the room are decided from a set of existing designs categorized by difficulty, with rules regulating how much of each category will be used. As mentioned earlier, only dead ends can be special rooms; for example, the furthest one is guaranteed to be the boss. However, after these steps are done, secret rooms are also generated that can close the gaps (except for the boss room), since they must be connected to at least 2 rooms. The following figure displays a possible floor layout of The Binding of Isaac, generated by an algorithm based on the one used in the game:

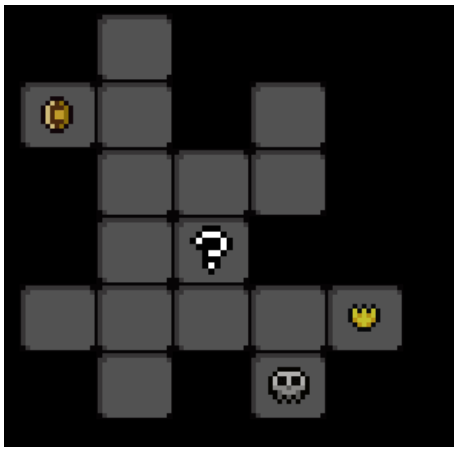


Figure 2: The Binding of Isaac generated map

With a new engine, the Binding of Isaac Rebirth was the ported version of the original to C++ that was extended by the team Nicalis. The new generation process is efficient, scalable, and ensures much more variety.

The team introduced the seed system, where each playthrough, or "run," is labelled, allowing players to share or replay earlier attempts. Each run is represented by an 8-digit hexadecimal number, with some hardcoded special seeds. This also means that, when pressing play, the entire run is generated instead of on a floor-to-floor basis. The pre-defined room types have more variety: corridors (0.5*1), long rooms (2*1), L rooms and large rooms (2*2) were added, and the regular square rooms also have types where doors can't generate on certain walls. Clearly, the already inefficient method would be unusable with these additions. The new approach generates in a puzzle-like way where the set pieces (rooms) are decided first, then pieced together by the algorithm [5]. Naturally, rules also play a role in how often room types can be generated; for example, if a big room is already added, the chance of another one is 5% [5]. The size of the grid was also increased, allowing for an expanded difficulty spectrum. The following figure gives an example of what one of these arbitrary maps looks like:

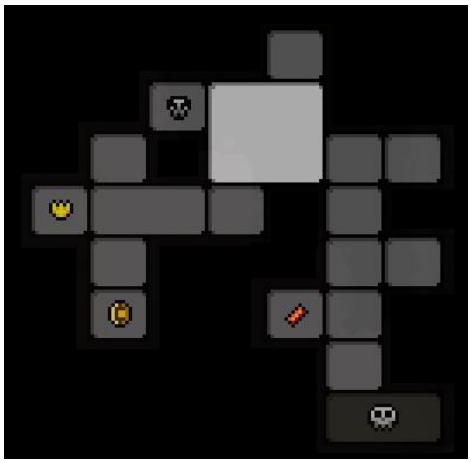


Figure 3: TBoI Rebirth generated map

The Binding of Isaacs procedural generation algorithm made a game that was originally designed to fail grow out of proportion in popularity, becoming one of the most successful indie projects ever. The simplicity paired with how efficient and large-scale the output is leads young developers to emulate, recreate and expand it to this day to gain a greater understanding of proc gen or to create projects of their own.

2.1.3. Minecraft (2009-)

Minecraft is an open-world sandbox video game, developed by Mojang in Java in 2009 and later acquired by Microsoft in 2014, that rose to popularity in its demo phase, later becoming the most bought game of all time. It works on a 3D grid divided into 1*1*1 blocks with a pseudo-infinite map-generating method heavily inspired by the rogue-like genre (Dwarf Fortress in particular) [1]. Sandbox games usually lack linearity in their play, focusing on freedom and creativity, and while Minecraft still has these attributes, it still has a linear gameplan with a start and a finish, assigning the key elements in the story with a composite PCG code. The algorithm creates biomes, cave systems, mountains, villages, etc., each with their own set of rules. Its open-source nature and relatively simple and understandable coding also make the game stand out for its endless possibilities.

Minecraft's world generation is made up of three main steps: mapping the landscape, creating it according to the map, and adding trees and other structures to it [6], with the random parameters saved into a seed so the world can be shared. The following figure shows the final product:

Figure 4: Arbitrary world generated in Minecraft

2.1.3.1. Mapping

Biome mapping is used to determine land and ocean temperatures, biomes, hills, and rivers. The system does this with multiple stacks of layers of commands, starting with

large cells of chunks and incrementally zooming in while adding details and definition. The first command, Island, generates noise with two colours: one for the ocean and one for land, the latter having a 10% chance of being chosen [6]. At this point, each chunk is 4096*4096 blocks in size and with each zoom, the dimensions are halved. The process avoids large square landmasses by moving these cells around with each Zoom iteration, which also makes sure to connect nearby lands at the same time. The function Remove Too Much Ocean speaks for itself in that it switches some ocean chunks that are in the vicinity of another with a 50% chance [6]. Later, this same neighbourhood is used to determine ocean depth.

After this, temperature zones are assigned to the land chunks, ranging from coldest to hottest: freezing and cold with a 1/6 chance each, and warm with a 4/6 chance [6]. To avoid unrealistic weather changes moving from area to area, if adjacent chunks have opposite temperatures, the function creates transitions between them. If warm is adjacent to cold or freezing, the chunk changes to temperate, which will populate most of the map, and after this is done, freezing temperatures will change to cold if adjacent to warm or temperate blocks. Following this, biomes are added. Every temperature type has biome variants with certain weights; special variants of these biomes have a low chance of replacing the regular ones as well. A thing to note here is that special biome decisions can take place at separate points in the code at different Zoom iterations, meaning their size can vary.

A different layer of white noise is used to map out hills and rivers, then smoothing them out and connecting them with another algorithm, River Mixer. If a hill is placed in shallow waters, it becomes a small island. Lands next to oceans get turned into beaches. Lastly, the Ocean Mixer layer is executed, creating precise water temperatures and depths using Perlin noise. In short, Perlin noise is like white noise but much smoother due to its higher pixel density, because of which it gives off a more natural look. At the end, a special Zoom layer upscales each block and smooths out everything by breaking up biome edges further, resulting in the final layout.

Figure 5 displays the difference between random noise and Perlin noise, and Figure 6 shows the stages of map layout generation in Minecraft:

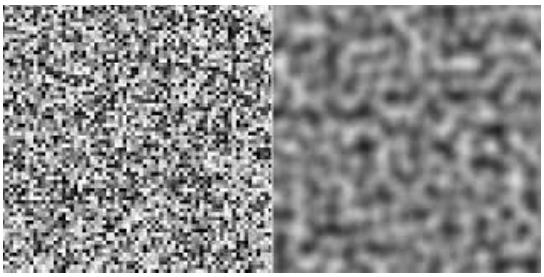


Figure 5: Random noise, Perlin noise

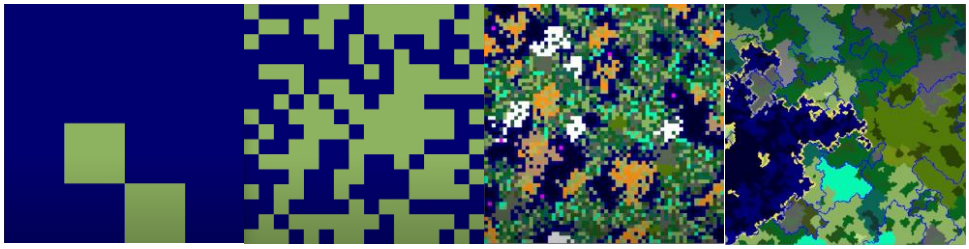


Figure 6: Stages of Biome Mapping in Minecraft

2.1.3.2. Terrain and Structure Generation

The terrain generation process's first output is responsible for determining heights and adding the lower edge of the world and is made up of only one type of block at this point. The method responsible for calculating heights combines the biome map's values with three additional fractal noise maps in different resolutions and mapping it all to the x, y, z values of each block [6]. Fractal noise is made up of many Perlin noise maps layered on top of each other, resulting in a 3D lifelike landscape. This generation process is, of course, regulated by rules; for example, a beach cannot be part of a mountain range. The surface is added on as an extra layer; its attributes are determined by the biome map. Water is added to the surface below Y = 63 if there are no blocks placed there [6]. Cave systems are carved out by Perlin worms, which is a method that moves a spherical shape through the map in randomly changing directions [6]. Pre-existing assets in the form of villages, structures, trees, ores and other decorations are then put in place based on biome or other rules pseudo-randomly. For example, if an abandoned mine structure is generated, there is a much higher chance for ores to appear in its walls. When all the calculations are done the game generates 16x16 sized chunks on the fly as the player is changing positions.

This information covers Minecraft until version 1.17. The version 1.18 changed almost the entire generation process and made it much more complex, outside the bounds of this thesis. However, some updates can be mentioned, like 3D temperature mapping, the use of many more noise maps, the new complexity of caves working with a completely different approach, and a much more realistic temperature zone mechanic, where, for example, distance from the nearest ocean plays a role.

2.1.3.3. GDMC AI Settlement Generation Challenge

The Generative Design in Minecraft challenge, particularly the AI Settlement Generation Competition, aims to extend Minecraft and gives opportunities to competitors to show off their work for industry professionals every year [7]. The competition focuses on fine-tuning mountain and layout generation, finding paths, creating larger settlements that adapt to their landscape, variety and tree placement using AI and other state-of-the-art technologies to achieve these goals. Settlements in regular Minecraft are, as mentioned earlier, pre-defined and are carved into the map, sometimes causing buildings to glitch and merge with existing hills.

An interesting solution in city generation came from two university students, Albin Esko and Johan Fritiofsson, who utilized a multi-agent approach. In short, agents perceive the environment and take actions based given objectives and constraints [8], but this subject will be discussed more in detail later. First, he students' method maps out the area in which the city will be built using a height map as well as a liquid map, creating a weighted graph, where the coordinates of the height map are the values of the nodes and the height differences are the edges, all water aeras being left out of the calculation [9]. A centre point is assigned to the area with largest plain surface to make a settlement. Then the roads are generated first using 2 agents: extendor and connector agents [9]. The extendor agent moves randomly and, if it finds another node, it connects it to the system. The connector agent searches for ways to connect already established roads so most nodes are connected to each other. Buildings are grown using methods from several PCG approaches like cellular automata and grammars, both explained later.

The only PCG technique related to Minecraft using Machine Learning is World-GAN, GAN being short for generative adversarial networks, which uses training pipelines to change biomes after generation [10]. The project is way more complex, the reason it's included is to show the potential of AI and Machine Learning use in procedural world generation.

2.1.4. Conclusions

Rogue in this thesis serves as history in the world of procedural generation, but also as a basis for my project going forward. Its algorithms, while outdated, give a basic understanding in the subject matter that can be built upon to understand the more complex solutions of today. The Binding of Isaac is the most solid candidate for me to take inspiration from. The layouts that it generates are already referred to as maze-like and are created efficiently. There is as much to change and extend as there is to take; in 2.3. I will go over some methods that can be used for this purpose. Minecraft wasn't mentioned due to its relevancy in maze generation, since its approach is too random for such a controlled environment, however the tools that it gives, and the potential of artificial intelligence use in its world generation could be a base for generating 3 dimensional mazes. Other possible methods will also be discussed in 2.3.

Category	Output	Generation Control	Variety	Strengths	Weaknesses
Rogue	2D Dungeon	Minor difficulty control	Very low	Simplicity, efficiency	Simplicity
The Binding of Isaac	2D Maze-like Dungeon	Moderate difficulty control	Low	Simplicity, efficiency	Scalability, glitches, limitations of Flash
The Binding of Isaac: Rebirth	2D Complex Maze-like Dungeon	Room placement and difficulty control	Moderate	Scalability, efficiency, control, C++	Extension limitations
Minecraft 1.17	3D Sandbox World	-	High	Scale, flexibility, Variety	Control
Minecraft 1.18, community extensions	3D Sandbox World	Highly controllable	High	Control, scale, increased variety	Possible balancing issues

Table 1: Mentioned videogame properties in terms of world generation

2.2. Mazes

Before trying to understand the methods with which a maze generation process could be initiated, a grasp on the concept of mazes will be needed. This part is to give a formal definition of mazes, get rid of common misconceptions, briefly analyse the history and working of these structures and specify the basics in their relevancy in Procedural Content Generation.

A clear distinction needs to be made between labyrinths and mazes. Labyrinths can be a considered a subset of mazes. Mazes are designed to disorientate, and its paths lead to many dead ends, either having a single path or interconnected paths to exit it, while labyrinths have more of a value in spiritual, entertainment and religious purposes, having one or multiple paths leading to the destination, without any dead ends. In Figure 7. the differences between the two can be examined.

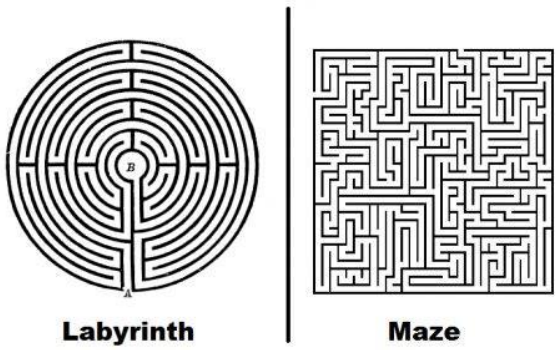


Figure 7. Arbitrary Labyrinth and Maze side-by-side

Mazes date back thousands of years to Greek mythology in the well-known story of Theseus and the Minotaur, the earliest reference to it being around 8th century BCE. The myth contains the Labyrinth, which, as I established, is not the exact structure this thesis is sought to analyse. The mazes that we know today were a part of German tradition in the Middle Ages, whereby solving one signified a boy becoming a man. The earliest man-made occurrence however can be found in Egypt with the layout of many of its ancient architecture. While never being unpopular, nowadays the rise of videogame culture gave the concept an even larger audience, alongside with art and movies as well.

The most basic definition for mazes is that they are puzzle, constructed with path or tunnels, the aim being to get to a certain location from the start. From this hundreds of types of mazes can be differentiated, based on end goal (exit, the centre of the maze), one or multiple paths to the goal, use of dead-ends, use of other objectives (keys and locks), and what can be the influence behind these attributes: purpose. An interesting type of mazes are multi-level 3 dimensional mazes, as I mentioned earlier, which work more as conceptual or artistic endeavours. They can be imagined as pipeline or as structure that uses staircases. Some interpretations of these mazes barely, if at all, fit into the definition. While I won't be deep diving into the subject, it can be a great extension to the project in the future and is an interesting concept in general. Figure 8. displays a couple of renditions of 3D mazes:

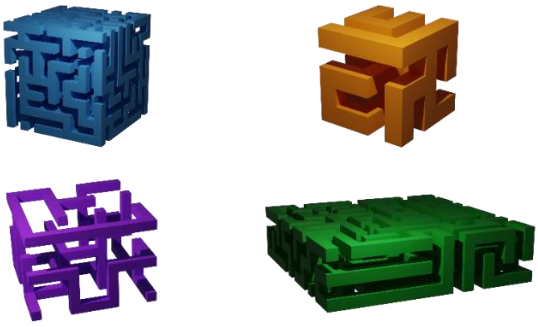


Figure 8. Examples of 3D mazes

2.3. Maze Generation Algorithms

Generating of mazes is not a new concept in any way, in fact the algorithms this part will be going over originate in graph theory [11]. In fact, other links between maze generation and graphs are plenty in other methods as we'll see later. For clarity, the mazes generated by the following algorithms are all have a single path for exit, include dead-ends, have diagonals paths, were created on grids, and fill out mentioned grids, resembling the most well-known form of puzzling mazes. The solutions and testing parameters are a part of Peter Gabrovsek's work [11].

Recursive backtracking (Figure 9. Left), or depth-first search, originally works by going in a direction as much as possible then, then backtracking to an unvisited path to continue the function. It repeats this process until every cell is covered. For the sake of maze generation, the author randomized the path of the algorithm. The backtracking approach is quick and efficient, however the mazes created by it pose the least challenge in the examples [11].

Prim's algorithm (Figure 9. Middle) was originally used to find the minimum spanning tree in a graph. It's based off the randomized breadth-first search [11]. The cells work with two states: passage and blocked. The algorithm randomly picks a starting cell, then assigns its frontier cells. Frontier cell are blocked cells in the neighbourhood of a passage state cell with an arbitrary distance, that, if chosen, are then connected to the starting cell creating a path. Only already connected frontier cells can have frontier cells of their own. The connection is determined randomly. The algorithm runs until it can't find more frontier cells. The output of the algorithm is filled with short dead-ends [11], but the method itself is very efficient.

Kruskal's algorithm (Figure 9. Right) was originally used to find the minimum spanning tree in a graph [11]. Instead of finding moving along existing paths the algorithm views the entire grid and picks a random wall from the wall list (area separating two cells), deletes the and labels the connected path. If two paths connect one of them will inherit the label of the other path. This process will continue until every cell has the same label. While Kruskal's algorithm is the slowest out of the three [11], the mazes it creates are fun and varied, and have many dead-ends.

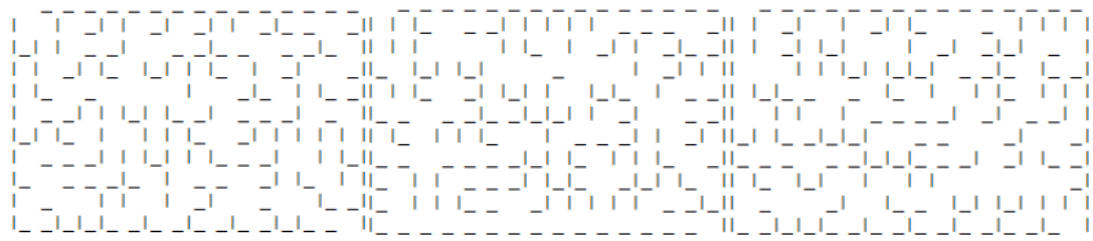


Figure 9. Difference between maze generation algorithms

2.4. Procedural Level Generation Methods

In this section I will examine some popular procedural generation methods and components and grade them based on their usefulness regarding level/maze generation. Procedural content generation has many use-cases, in most open-world games nowadays trees, details and textures are generated to shorten development time for example. The methods mentioned here are either purely for Procedural Level Generation (PLG) or closely linked to it, with the primary focus being on dungeons. In terms of PCG, dungeons

refer to a closed, maze-like layout, perfect for purposes of this paper. With the information gathered the scope can be narrowed down to the potential for maze generation.

2.4.1. Controlling Generation

A procedural dungeon generation method normally consists of three elements: a representational model, a method for creating it and creating the actual geometry of the dungeon based on the representational model [12]. One of the most important aspects of dungeon generation is the control it provides over the output. Control in this case means how much the developer of the program can alternate the process, how difficult it is, and how much that changes the output, while maintaining consistency. The focus before with PCG was on what it can create. Use of noise (random/Perlin) is popular for this philosophy, and it led to complete randomness without much control (e.g.: Minecraft, as mentioned earlier). More recently as complexity grew the main concern became how can something be generated, and the connection between machine and man became closer. Two techniques helped shift the paradigm: controllable agents and declarative modelling. There is no consensus on the meaning of software agents, and it works more as an umbrella term for its many uses. “When we really have to, we define an agent as referring to a component of software and/or hardware which is capable of acting exactingly in order to accomplish tasks on behalf of its user.” [13] Declarative modelling, as the name suggests, is approach where the developer can declare/specify properties and/or constraints for the system to follow. While the use of these techniques are the building blocks for the more complex methods, which aim to give more control over the output of a system.

2.4.2. Cellular Automata

Cellular Automata works in a self-organizing matter with applied rules and is one of the most used methods in dungeon generation. The method consists of cells in a grid in a finite number of dimensions, each cell having a state dependent on its initial state (t=0) and the neighbouring cells’ state. These neighbourhoods and states are defined as part of the rules. A famous cellular automaton solution is credited to L. Johnson et al., where the method works with Moore neighbourhood (8 neighbours instead of 4, like the Neumann model) and 3 states: floor, rock, and wall [12]. Walls function as a barrier within floor and rock, floor is the play area and rock are out-of-bounds. The method works with 4 parameters: initial state (random percentage of rock cells populating the map, r), number iterations (n), neighbourhood value threshold (T) and Moore neighbourhood size (m). First the system populates the map with rocks based on the initial state parameter, then iterates through n, assigning cells rock if their neighbourhood value is greater or equal to T and floor otherwise. Then if a rock cell has a floor cell adjacent, it becomes a wall cell. Figure 10. displays a random output of L. Johnson et al.’s formula:

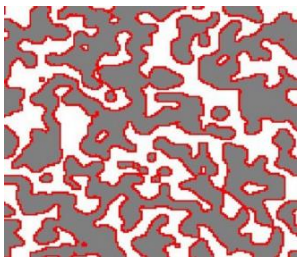


Figure 10. CA generated map

This solution is very simple, only using 4 parameters, 3 states and 2 rules, because of which it is also efficient even a larger scale, the generated layout looks natural and cave-like as intended and has a high level of variety. It's limited to 2D, which wouldn't be a big problem, however, the output is not very controllable, meaning if a desired map would need a specific number of rooms the only possible solution would be trial and error.

2.4.3. Grammars

Grammars, or rewrites, are used to generate structured content dynamically with sets of rules. These grammars can define patterns, configuration, and constraints. The most basic of grammars is generative grammar, which starts out with a linear chain of phrases, then, based on the ruleset, iteratively replaces them with other phrases in a list or adds new links to the chain of phrases. The selection method takes these phrases as terminal symbols. This method originates from describing linguistic phrases, but also has implications in videogame development. Take for example if we have a chain like {greeting > noun}, out of the possible list of phrases we could get {hello world}. The implications in gaming are quite limited due to its linearity, but more nuanced versions have been developed like graph grammars or shape grammar.

Graph grammars opt for a more complex and diverse directed or undirected graph solution [14] instead of a linear chain. This method could seem familiar, because of the description I gave for Rogue, and solutions could even look like The Binding of Isaac, however neither uses this approach. Nodes represent rooms and edges the connections between them. Rules can govern what structure the graph will take by ways of replacing nodes with other nodes or node structures (like Figure 11.), shifting around connections, making connections directed or undirected, recognizing shapes and restructuring them and other decisions based on the attributes of the nodes. The method runs until a fix number of iterations or until a condition is met.

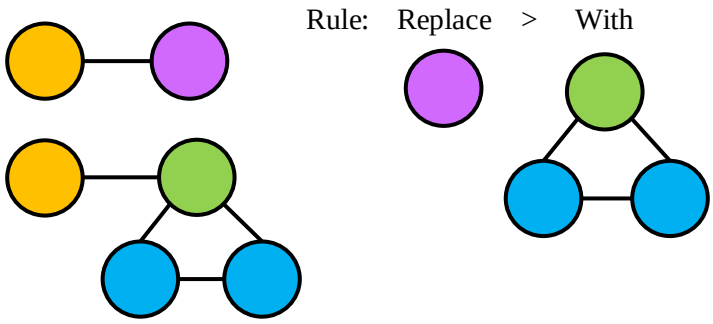


Figure 11: Simple presentation of an arbitrary rewrite

Nodes themselves can have interesting properties, for example in David Adams' project defines difficulty and fun as properties and governs global size as parameter [14]. Search algorithms are often used with grammars and s is the with Adams' who uses it checks through the results to match up with the parameters [12]. This solution while working great with his project is not very extendable without changing the existing code. However, it does go to show how universal graph grammars can be, since Adams' game doesn't follow the usual dungeon-crawler formula, opting for a first-person shooter game instead. Joris Dormans' solution is interesting in its approach in generating adventure games. In his algorithm he uses directed graphs to model the tasks of the player, then a shape grammar uses the same structure to formulate a layout. The popular key and lock

mechanism is also used in the method, where one node contains a key, before the door needed to be unlocked in another node [12].

The great part about grammars is that they can be integrated into a lot of systems, even each other and can produce very different results. In map generation they have potential to create somewhat lifelike dungeon, a cell structured maze, or can even be put into a physics simulation to create something chaotic.

2.4.4. Genetic Algorithms

Genetic algorithms are inspired by the process of natural selection and are search-based algorithms designed to find an optional solution to a problem. The approach uses virtual organisms in the form of strings with different values and goes through them iteratively while a fitness function measures their fitness, in other words their effectiveness to solve an issue. After this a selection process starts, where a certain number of genes are dropped for the children of the remaining genes. With each generation the child strings inherit attributes from the parents. This can happen by duplicating the survivors of the selection with a small chance for mutation, where arbitrary values of each gene randomly change to allow variation. A genetic operator, known as crossover (Figure 12.), can be introduced for potentially higher efficiency, where two surviving genes' (parents) values are cut at a chosen point to then be combined into two other genes (children), mutations also being part of the operation. Genetic algorithms are a great optimization tool but have been used in the generation of maze-like structures.

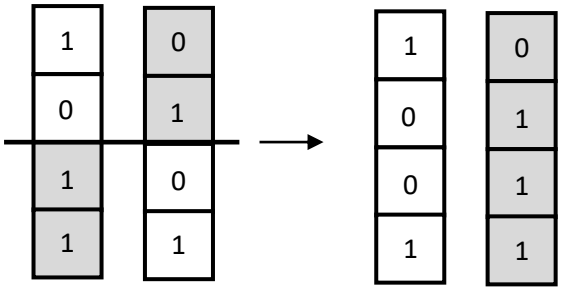


Figure 12. Instance of crossover operation

Daniel Ashlock et al.'s [15] generator aims to create maze-like structures with variety and a high level of control over the mazes generated. Dynamic programming is a big part of the study, and it works by traversing a network node by node and recording the cost of getting there from the starting node. It does this until it finds the minimal cost to get to a certain node. The team distinguishes 4 types of maze representation: first direct, chromatic, indirect positive and indirect negative. The first direct representation the genes can represent accessible and inaccessible areas bits. Chromatic uses colours represented as strings on a grid, where an agent can move to adjacent cells if the cell either has the same colour or is a neighbouring colour in the list. The indirect approaches start out with an empty or a full grid and draw out or delete the walls, depending on if we're using the positive or the negative approach. The following Figure shows the outputs of Ashlock's approaches.

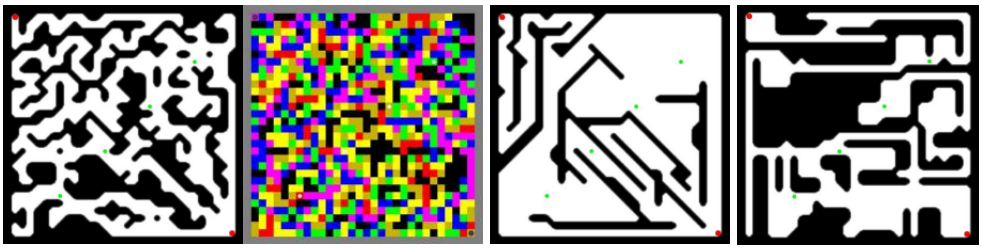


Figure 13. First direct, chromatic, indirect positive and negative approaches

The fitness functions that are used by the authors change the output of the generator completely, each focusing on a different aspect of level creation. These fitness functions can focus on the accessibility of checkpoints, which can really make the out diverse and unique (used in Figure 13.). They can maximize the path between start and exit, a good way to "extend" a smaller map. Also, it can encourage path branching around checkpoints, which can lead to interesting layouts. Using checkpoints can really increase the size of a level generated, a drawback being, if the branching option is chosen the path between the start and exit points becomes shorter.

Overall, the tools that genetic algorithms, in particular, Ashlock et al.'s solutions give are highly controllable in their area, especially considering that mutation and crossover parameters weren't even mentioned.

2.4.5. Constraint-Based 3D Graph Method

Roden and Parberry [16] describe a method for purely 3-dimensional underground level generation that is considered one of the only ones to do so [12]. The method relies on a 3D undirected graph that maps out the structure of the level with constraints put on the edges and nodes. There are only two fixed nodes to start with: the starting room and the exit, and from there a constraint solving mechanism solves the rest. The constraints can use known topologies, like tree or star, but can also connect subgraphs. Rules include distance between start and finish, connections, and geometry of a node. While this solution shows potential for 3-dimensional mazes the lack of gameplay control, like setting difficulty, and knowledge about its inner workings are some huge drawbacks. Figure 14. displays a level created by this method.

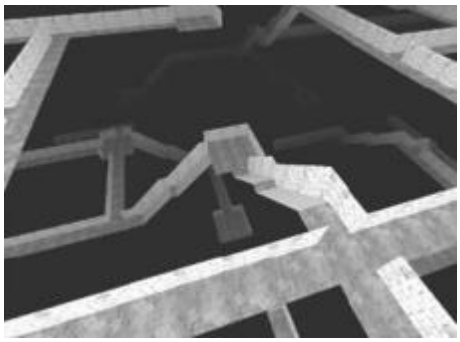


Figure 14. 3D map constructed with constraints

2.4.6. Composite Methods

While the methods mentioned so far goes far in the study of PCG most games and generators (if not all) use some combination of them to meet the requirements of

games/programs. Some methods benefit greatly from these integrations, cellular automata for example showed great efficiency in what it was designed for, which is detailed lifelike dungeons, but suffered in the gameplay department. The solutions I'll look at combine the strong suits of CA with other PCG methods, that lack in other departments to create specialized systems.

2.4.6.1. Evolutionary Cellular Automata

An interesting hybrid method created by Chad Adams and Sushil Louis [17] combines the versatility of cellular automata with the already flexible and highly controllable genetic algorithms to make a generator for maze runner games, fine-tuned for entertainment purposes, the aim being to generate mazes that are not easy to complete, but also not too hard to take away enjoyment from the experience. It does this by evolving the ruleset of CA with GA to create maze like patterns to then string together with a region merger.

First, the rules of cellular automata are evolved, then optimized through 3 fitness functions. The cellular automaton used applies the Moore neighbourhood in a 2D 30x30 grid [17]. Cells have two states meaning every gene will have two rulesets, nine rules each (one for the cell and for each neighbour) [17]. The authors used 50 iterations; each rule is applied every time. After the generation is done the merger algorithm detects the empty state regions and connects them starting from the bottom left at the narrowest point of connection. Figure 15. shows the region merger algorithm's working.

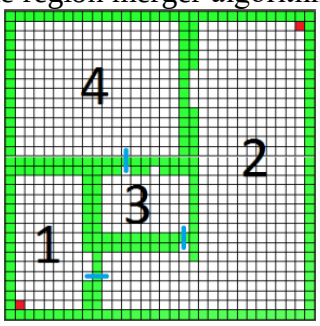


Figure 15. Region Merger algorithm

The method uses elitist genetic algorithm that selects the top 'n' participants from a population of size '2n' for crossover, resulting in n offsprings. These same participants are also preserved in the next generation, resulting in a population of size 2n [17]. Two gene representations were tested: binary and probabilistic.

In the binary representation an empty cell will become filled if it must or three neighbours and a filled cell will become emptied if it has 2, 3 or 4 neighbours. The crossover rate is 100% with a single point crossover and a mutation rate of 0.5%. The values were chosen to avoid early convergence and increase exploration [17].

The probabilistic representation works very similarly to the binary, but instead of changing states linearly, state change is dependent on percentages given with each gene holding a value between 0 and 127 (7 bits for crossover purposes). The examples given in the paper are: if an empty cell is adjacent to a filled one it has 100% chance of converting and if a filled cell is adjacent to another filled cell, it has a 38.6% chance of converting. As for the AG crossover rate is 90% using half uniform crossover with a mutation rate of 1%. The values were chosen to increase exploration and to slow the convergence.

Their tests involved three distinct fitness functions: shortest solution path length (F1), total dead ends (F2) and sum of path lengths and dead-end count without weights (F3). F1 is the same as Ashlock's with the same name (2.2.4.) and follows the same motive, that the larger the map gets the more interesting shape the pathing takes, resulting in a snaking path to the exit. F2 generates a more intriguing maze solving experience with more path cells and less connection between them. While, once again, inspired by Ashlock there's a key difference: Chad and Sushil's fitness function identifies individual dead-ends instead of groups of dead-ends at the end of a path. Lastly, F3 is a combination of the other two fitness functions, each one's effects being less pronounced. The following figure show an output using F3 fitness function with both binary (left) and probabilistic (right) representations:

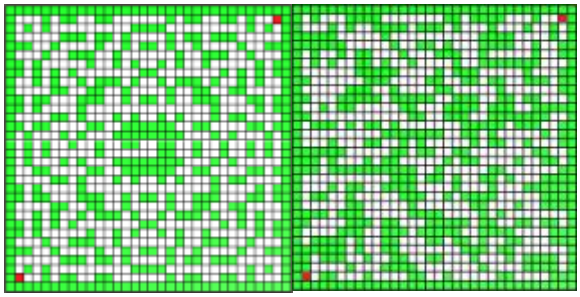


Figure 16. Mazes generated with F3

The authors' work show how PCG methods can be used as puzzle pieces to create a system specialised for one's needs, where CA serves as the chaotic, infinite generator and GA as the glue and optimization tool along with the region merger. This solution builds on Ashlock's work and adds layers of controllability to it.

2.4.6.2. Hybrid Roguelike Solution

Roguelikes, as I alluded to earlier, are a stable when it comes to procedural content generation. The examples mentioned, while popular, leave a lot of doors open for the subject and after some time (depending on the videogame) still lack in variation and extendibility. As the roguelike(/lite) genre is such an open term and often associated with indie developers a lot of experimental, but well thought out work can be found in the field. Gellel and Sweetser [18] show promise for hybrid approaches in this genre with a solution that is simple, efficient, varied and most importantly controllable and extendable.

As is often the case with rogue-like PCG systems, a grid is used as the basis of the map. The authors take inspiration from Dormans' grammar approach created for adventure games that combines graph and shape grammars. These grammars give a description string of the map that is then used by the generator working in a cellular auromata-esque fashion. This approach is not a traditional CA due to generation process not only being influenced by the cell neighbourhood. Not only can the mission plan, but the size of the map and the distribution of input rooms can be controlled through the grammar strings. The way the solution works also gives opportunity to extend this with custom strings. When the generator starts, it places rooms one-by-one in the order of the mission string, also accompanied by pseudo-random rules.

The first rule (Figure 13; Left) gives the location of the start room, marks it as active and attempts to randomly move to another cell in its neighbourhood, marking it as active. It does this until all the "missions" in the string have been used up or until it's impossible

to make another move. An important note is that this rule does not have knowledge of the entire grid, just the active and its neighbouring cells. The second rule (Figure 17; Right), "Subsection Search, evaluates a subsection of the map and picks out a direction to continue generating from the room closest to the sections centre. Using this, the generator avoids too many straight lines to be created, encourages clustering, and produces a more natural map with these attributes (than Rogue for example).

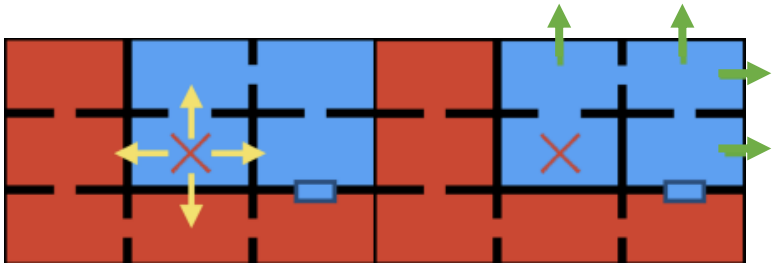


Figure 17. The first and second CA rules

Two separate approaches are proposed by the authors using the rules and mechanisms mentioned. Persistent Subsection Search applies the second rule with each iteration, while Subsection Search on Halt only uses it when a dead end is found. In addition, a third rule is added that can overwrite the current active cell and force an exit to be generated, replacing a non-special room. The Subsection Search on Halt approach is a great failsafe, ensuring that the process will not restart, and deals with compatibility even further.

After the layout has been generated, the system decides the pathing/door placement. It does this with a path difference heuristic. This heuristic defines two paths: shortest path to complete the level (all missions considered), also known as Critical Path, and shortest path between start and finish, also known as the Spine Path. The generator decides the path difference by subtracting the Spine Path length from the Critical Path length. The product tells us how much traversal will be needed to complete the level. Figure 18. shows one of many outputs that Gellel and Sweetser's solution can generate using the Subsection Search on Halt approach:

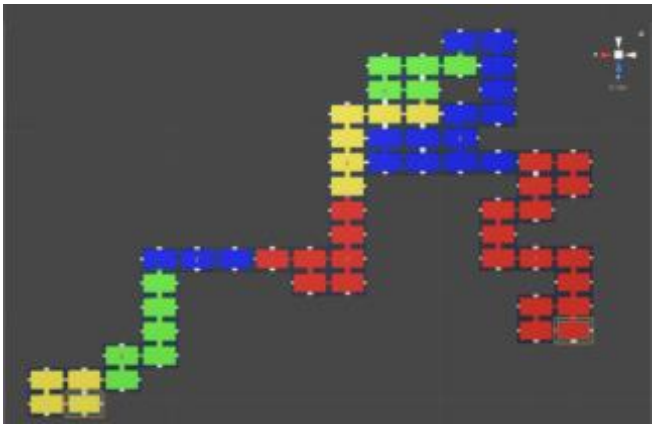


Figure 18. Arbitrary map using the Subsection Search on Halt approach

This hybrid method shares a lot of benchmarks that I hope to achieve with this project, putting an emphasis on reusability and interchangeability. While being more complex than a regular CA solution it still achieves to be efficient and scalable with the number of possible variations it can provide.

2.4.7. Conclusions

Category	Approach	Control	Gameplay Features	Output Types	Variety
Cellular Automata	Grid cells, state modification	Initial states, number of iterations	-	Chaotic 2D dungeon	Little
Grammars	Generative Grammars	Size, parameters	Missions	Linear Mission Control	Little due to linearity
	Graph Grammars	Difficulty, size, fun	Difficulty and fun parameters	2D room graph	Dependent on coded strings
	Graph plus Shape Grammars	Input Missions	Missions	2D dungeon with mission items	Dependent on coded strings and parameters
Genetic Algorithms (Ashlock)	Combining multiple representations and fitness functions	Fitness function, gene parameters	-	2D maze	Large due to the variations
Constraint-Based	Constrained Graph generation	Topology, node placement	-	3D maze-like underground level	Little
Hybrid Methods	Evolutionary Cellular Automata (CA + GA)	States, iterations, fitness function, gene parameters	Layout structure type	2D maze	Extended Ashlock approach, Large
	Hybrid Roguelike generator (CA + Grammars)	Size, mission input, distribution	Missions, difficulty, size	2D maze-like structure with missions	Dependent on input, can be extended, Large

Table 2. Overview of the approaches examined [12]

2.5. Traversal Search Algorithms

In the area of procedural generation and especially the generation of dungeons, maze-like structures and mazes themselves can't be manually checked for validity. This validity can stem from the ability to solve a proposed challenge to any other kind of failsafe or optimization, like checking for a desired design. Traversal search algorithms provide

systematic approaches for this problem, exploring these environments to fix any potential problems. As was the case with many other aspects of maze generation these traversal searches are rooted within graph theory, the first two displaying that exactly.

2.5.1. Depth-first and Breadth-first searches

Although depth- and breadth-first searches were in covered earlier in the context of their ability to generate mazes, they can also function as a maze optimization algorithm. Since their basic functions have already been mentioned, this part will only show their functions when used as an optimization/traversal search tool. Naturally, using them as search algorithms are much closer to their roots.

The depth-first approach starts at the starting cell and marks every possible direction it can continue its path. Then, randomly starts going down a route marking each interconnection it comes across. If it hits a dead-end it reverts recursively to the nearest interconnection and starts the process again. The algorithm continues this until it has reached a pre-determined goal or every route, depending on what we need, and after, from the information gathered, determines the shortest path from start to finish. This approach shines when the targeted cell can be found without a complex route, it is a linear search method and tends to be one of the slower ones when it comes to maze traversal [19].

The breadth-first approach, as remarked earlier, uses frontier system, where every potential continuation of a path is marked and queued. It marks a new frontier based on the order of queues, so it's guaranteed to traverse the entire map while also traversing in every direction it can simultaneously. The algorithm does this until the queue becomes empty, meaning there is no path to continue. This is one of the faster search algorithms for maze traversal even at larger scales.

An intriguing traversal algorithm is the iterative deepening depth-first search. It combines the benefits of depth- and breadth-first search by increasing its depth limit by one with each iteration [19].

2.5.2. Dijkstra's Algorithm

Dijkstra was a Dutch scientist who developed the greedy pathfinding algorithm named after himself. The algorithm aims to find the shortest path possible in a weighted graph, always choosing the lightest or least costing node next to continue, making it a greedy algorithm. Dijkstra's algorithm is used in many fields of science: graph theory, OSPF (open shortest path first) in networking and, of course, maze solving.

The algorithm uses the linear distance between the current and the target node, basing off its decisions on the value. Out of the examples it can be the quickest and most efficient, however when it comes to maze solving statistical data oscillates a lot, sometimes being the best and other times one of the worse performers. This is due to the fact it chooses the most optimal cell in a current situation instead of the entire maze. The function struggles to see larger obstacles, and in most cases, it approaches these obstacles and then moves around them. Nonetheless, Dijkstra's algorithm is still a great tool, and its approach extended with the benefits of BFS would create the most optimal of these methods.

2.5.3. A* Algorithm

The A* algorithm is an informed search algorithm, meaning it has extra information on the system for traversal, making it more efficient. It is a fundamental technique in artificial intelligence and pathfinding, it's effectiveness and efficiency in finding the most optimal

path graph-like structures make it stand out in the group of traversal algorithms. It's a heuristic function that estimates the remaining cost of getting to a certain node from any position. It is well known as the most efficient and used method for finding the shortest path on a graph and solving a maze.

When determining best path options the A* algorithm gives three values to the surrounding cells and adds them to a list. These values are G, which is the distance from the starting node based on the path, H, which is a heuristic distance from the end node (admissible, never overestimates), and F, which is the sum of these distances. It evaluates the cells and chooses the ones with the lowest F value. If multiple candidates have the same lowest value, the cell with the lowest H will be evaluated first. The algorithm will revert to another path if its value is lower after some iterations on another path. If a cell is visited twice on two different paths its G value will update to the most optimal.

From the studies done by Iloh [20] the A* algorithm is great tool for maze solving. It doesn't find the exit cell the quickest in every test, however when it comes to larger scale levels it outperforms the other algorithms. The great strength of A* is in its balance between a greedy algorithm and the fact that it's an informed search, always finding a path around obstacles if possible. Figure 19. shows the difference between Dijkstra's and the A* algorithms' ability to detect and find a path around a wall:

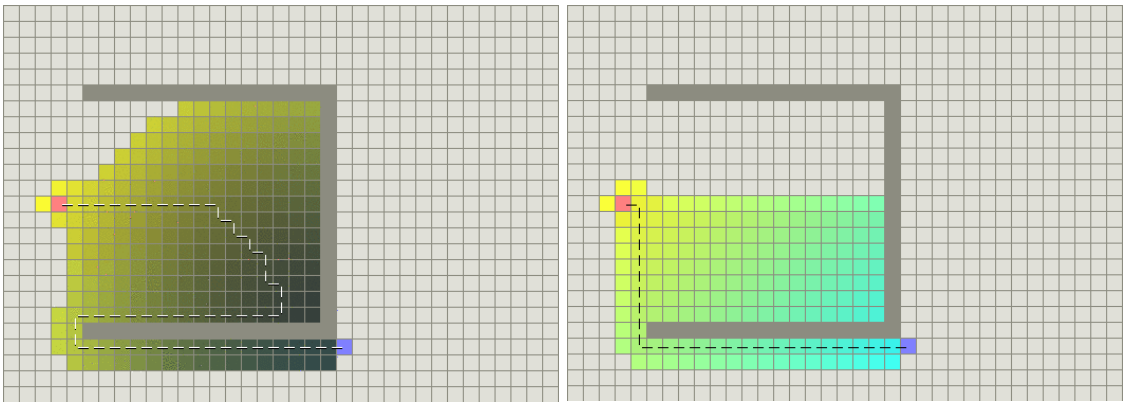


Figure 19. Difference between Greedy (Left) and A* (Right) algorithms

2.6. Dynamic Difficulty Adjustment (DDA)

Videogames that rely heavily on PLG can still suffer from lack of a personal difficulty variation sometimes due to the hard coded nature of the game design. The aspect of fun has been mentioned earlier, with a design that hits the golden middle ground between too hard and too easy, but that changes from person to person and with experience. Dynamic difficulty adjustment (DDA), in theory, helps craft a tailored experience to every group, and could increase the replay value of a game even further.

While changing difficulty on the fly can't be fully automated, the structure of the systems, the underlying mathematics of the algorithms can be used to alter level generation and other factors based on the statistics of the player. These statistics change based on what the underlying difficulty is: win/lose, life points lost, and completion time could all be factors. There are plenty of approaches to this problem and I will cover a few here.

2.6.1. Probabilistic Methods

A study by Su Xue et al. [21] proposed the idea that DDAs can be viewed as a problem of optimization [22]. The team modelled a graph to show the probabilistic tendencies of a player's progression that can be seen in the following figure:

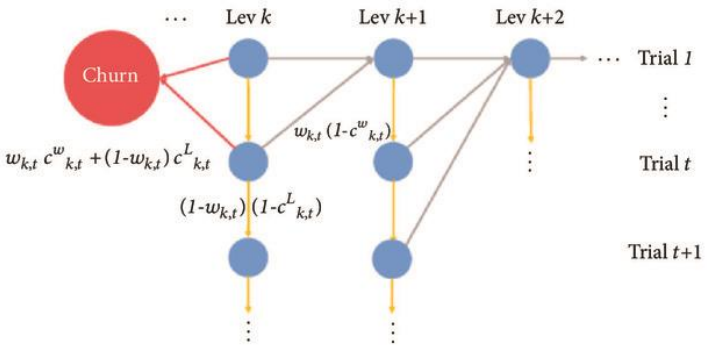


Figure 20. Probabilistic graph showing a player's potential progression

This implementation was used on a mobile game with a clear increase in user retention and engagement [22]. This method can be used on a variety of game genres if they are constructed with a similar progression model, the main factors for implementing it for level-based games being level and trial. The paper also defines the churn state as the player leaving the game behind. This is naturally the worst-case-scenario from a developer's perspective, so the aim of the method is to maximize playtime before a player reaches the churn state, while also maximizing the trials needed to get to the end so that the game is not too easy, optimizing the graph to be suitable for an individual.

This probabilistic method collects these engagement values from all players and creates an expected outcome for each of them [22]. In case a player does not match this, either by completing a level in less or more trials, or by churning, the game adjusts accordingly.

2.6.2. Perceptrons

The solution of Pedersen et al. was created for platformer games, more specifically Super Mario Bros [23], and utilized recorded player experience and neuroevolutionary preference learning [21]. The data was gathered after inviting users for the test recording both their gameplay and emotion through a questionnaire. The resulting function created with this information was very noisy since a player's style and choices can vary dramatically. A single neuron was used to study the relationship between choices and emotion. This is referred to as a single-layer perceptron and the reason it was used over the more precise multilayered method is because its output is much easier to analyse [21]. The systems predictions in challenge were highly accurate in the tests that followed [23].

The authors concluded that this method could be used for personalized level generation and would release studies on the subject utilizing a multilayered model and AI agents [21]. These adapting levels were also tested by people, where the player would change after 12 levels, the solution adapting to the current one each time [21].

2.7. Summary

Based on the literature I've gathered, the development of dungeons and more importantly mazes can vary a lot. For a clear narrative of workflow to be formed these options need to be categorized and narrowed down to the potential candidates for the exact system I'm going to build. These categories are the technologies that are needed the development, methods that can potentially extend the concept later and the approaches more suitable for other kind of generators.

First, the kind of mazes I want my approach to create are puzzle mazes of today and the paths are going to be mapped on a grid. This means that a grid must be filled with paths unlike the maze-like structures of The Binding of Isaac and Rogue. I will avoid the clustering of paths creating room-like structures and as part of the definition of mazes every path must be reached. This would mean that a cell must have a minimum of one wall and a maximum of three. I want to leave the question of multiple paths to the exit opened, so if the generator is used, the developer can opt for a single path or interconnectivity. Also, the map generated must be extendable and controllable for convenience of further development.

For the level generator methods like cellular automata in its natural form and the noise and Perlin maps used in Minecraft will not be used. The chaotic nature of these techniques does not fit well in a highly control-based environment like this. Cellular automata are more suited for pure dungeon games and artistic endeavours, although algorithms could be taken from it. Evolutionary CA is also great for maze generation, however it's not solution I will use. After the generation process noise maps can be used to generate the floor decoration but are not suitable for the level generation itself.

The approach I chose for this solution is rooted deep within graph theory. Grammars, more appropriately graph grammars would serve as the best approach for the purposes of my thesis. The graph generated by them can be mapped on a grid, they offer plenty of control, like difficulty, size, and potential for extensions, for example if a creator would like to make a key and lock system, procedurally allocate spawners for enemies, graph rewriting is a simple and efficient way to do so. I will also use this rewriting for a similar map building process as I explained in the Binding of Isaac Rebirth, where set pieces like long hallways, L turns, and 4-way interconnections can replace existing cells. This does not guarantee the grid to be filled, but graph-based search methods can be used side-by-side with the grammars to find empty cells, calculate the shape that is missing and put the matching set piece in its place. This approach may also guarantee a path to the exit if a node is marked as "path to the exit" and generates accordingly, with rules to observe the process. A difficulty setting option will dictate the size of the grid, the spectrum of dead-ends to generate and the distance between start and exit. The percentage of certain set pieces to be placed are also governed by this scale. Further controllability can be added, so the ratio between path length and dead ends can be set. The A* algorithm will be used for testing purposes, even if theoretically a path to the exit is warranted, and if the level is created with interconnection, the solution could show the player if he found the shortest path.

As I said a couple of times, graph grammars can be paired with a lot of other methods. I've talked about 3 dimensional mazes before, and although I'm not going to include it in my solution, the constraint-based method or even the machine learning and GAN techniques from Minecraft's competition extensions can be used in tandem with my PLG

to make 3D mazes possible. The idea of the constraint-based method is especially fitting since it also uses graphs to create its maze-like structures. As an optimization tool, genetic algorithms come in handy for most generation processes, and in my example, they would do the same. Dynamic difficulty adjustment is an interesting feature in games that utilize it. I would love to incorporate it with time since it would fit perfectly and would be a fine addition for even more intriguing gameplay.

3. Requirement Specification

With the methods I described, the system must be able to generate mazes according to the parameters given. These parameters should include a difficulty setting that controls the size of the grid that the maze will be mapped on to, the heuristic distance limits between the start and the exit cells and the number of potential dead ends, separate settings for these parts for fine-tuning and testing purposes. Also, the placement of the starting and finish cell should be manually decidable as well.

The system must utilise graph rewrites for the starting parameters to be controllable, and to make it extendable with the intention of what it will be used for. For the purpose of flexibility, the code must be easy to read and segmented. It must be an asset that developers can add code onto or take the generated map from it to build a level or a game around it. A database holding potential routes must be created for the rewriting. This database must be accessible and the assets in it extendable, as an example: if a developer wants to add a key and lock system, they should be able to do so by adding rewrite paths to the database.

The generated layout must be validated via the A* algorithm to make sure that the level is completable. The generation process will restart if either the algorithm doesn't find a way to the exit (theoretically this shouldn't be possible) or if the generated path to the exit doesn't match the minimum length dictated by the input parameters. This could only happen if the path blocks itself in a corner.

A demo game must also be built on this generator. Using the parameter system a slider must be provided for the player to set the difficulty. A game mode should also be selectable based on the start/end rooms' placements. Minimal in-game graphics ("sprites") should also be included. The game should be a puzzle game with the focus of showing off the capabilities of the generator. Controls need to be added for the player.

4. System Plan

As I discussed earlier procedural generation methods and game developing in general nearly always have a multitude of approaches, even if the end goal is seemingly the same. For the purposes of creating this maze generator I'll take a similar approach to how The Binding of Isaac Rebirth was developed with how the data is partitioned and how the generation process happens in a puzzle-like matter, with the extra layer of graph rewriting and a full-fledged maze structure as an output, instead of a maze-like one.

4.1. Maze Generator Plan

The system will use graph rewrites similar to the one described in 2.4.3. in figure 11, but in this case, it will create a tree graph structure. By changing the rules of rewriting, this is not difficult to achieve. The generator will have a data structure that have the graph parts that will overwrite a selected node in a list-like manner. This list of graph parts will be widdled down by checking the neighbourhood of the current "frontier" node. This neighbourhood check will be carried with the earlier mentioned Neumann neighbourhood by first checking the direct neighbours (r=1) to decide what direction a path can be generated, then, an extended checking the extended neighbourhoods (r=2) to see what set pieces are viable to replace the targeted node. In Figure 21. this check is visualized on grid, however this is only a representation, since this will be carried out on the graph:

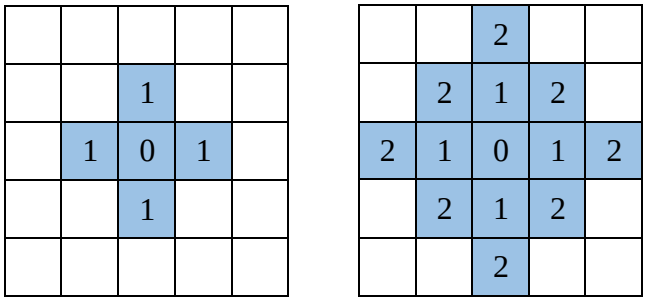


Figure 21. Neumann neighbourhoods (r=1, r=2)

The graph part candidate list is decided by the following factors: is the node to be replaced the starting node, the finish node, or a random general path node, the direction/directions it can extend to and how far it can extend. The graph pieces hold transient data about where they are going to be place. Due to how a maze works, when moving down to a connected node an x or y value is incremented or decremented by one. The final decision of what piece will rewrite the target node is random with weighted values based on the requirements. Figure 22. shows an example of how this rewriting process happens:

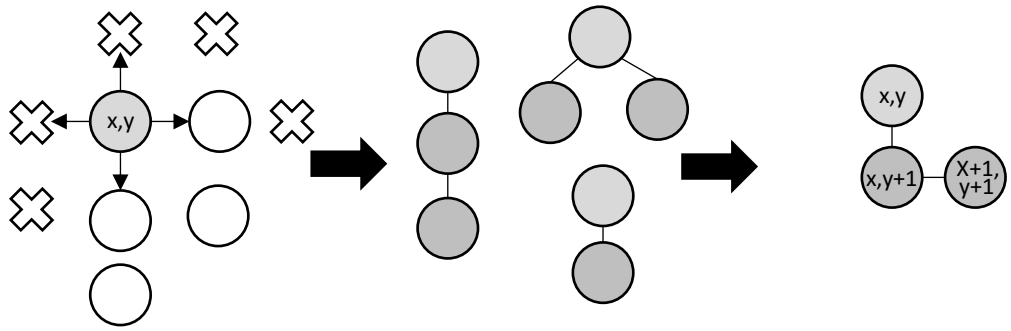


Figure 22. The process of neighbourhood checking, list narrowing, decision, and placement

The first action after the system determines the starting node is generating the path from start to finish, then all of the dead ends, which will be referred to as general paths. A simple 3x3 example can be of the maze model can be seen in Figure 23., where the orange node signifies the start node, the dark green node the finish node, the lighter green nodes the path to the finish node and the grey nodes the general paths:

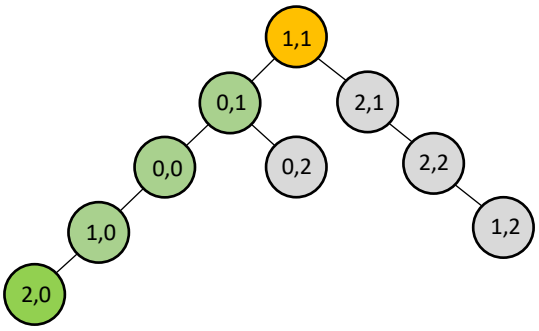


Figure 23. Generated maze in graph form

After the graph model is done, the grid is generated, and the values of the graph are mapped onto it creating the final maze. Taking the example from figure 23. an arbitrary maze generated by the method can be seen in Figure 24. using the same color-coding:

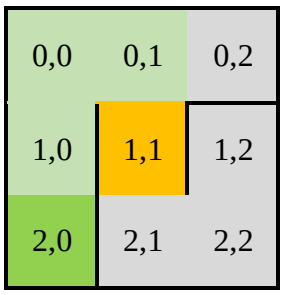


Figure 24. Mapped maze

For this solution I decided to separate the rewriting into two methods, one dealing with generating the path from start to finish and the other generating everything else. The reason behind this decision is that there are enough differences in these functionalities to make their own methods, even though the Start-to-Finish method does build on the General Rewrite.

The General Rewrite method will take up most of the calculation time of the generation process. The method's behaviour I've mostly described earlier: it checks the neighbourhood of the current node, decides if it can be replaced, and if so, determines what it will be replaced by. This method adds every node newly added to the candidate list and removes it if no neighbour is found. There is also some randomization involved: every time a replacement happens the method decides if it will pick from the path generated or if it will leave the method altogether based on percentages either hardcoded or determined by the situation. Figure 25. shows the procedure described in a flow chart:

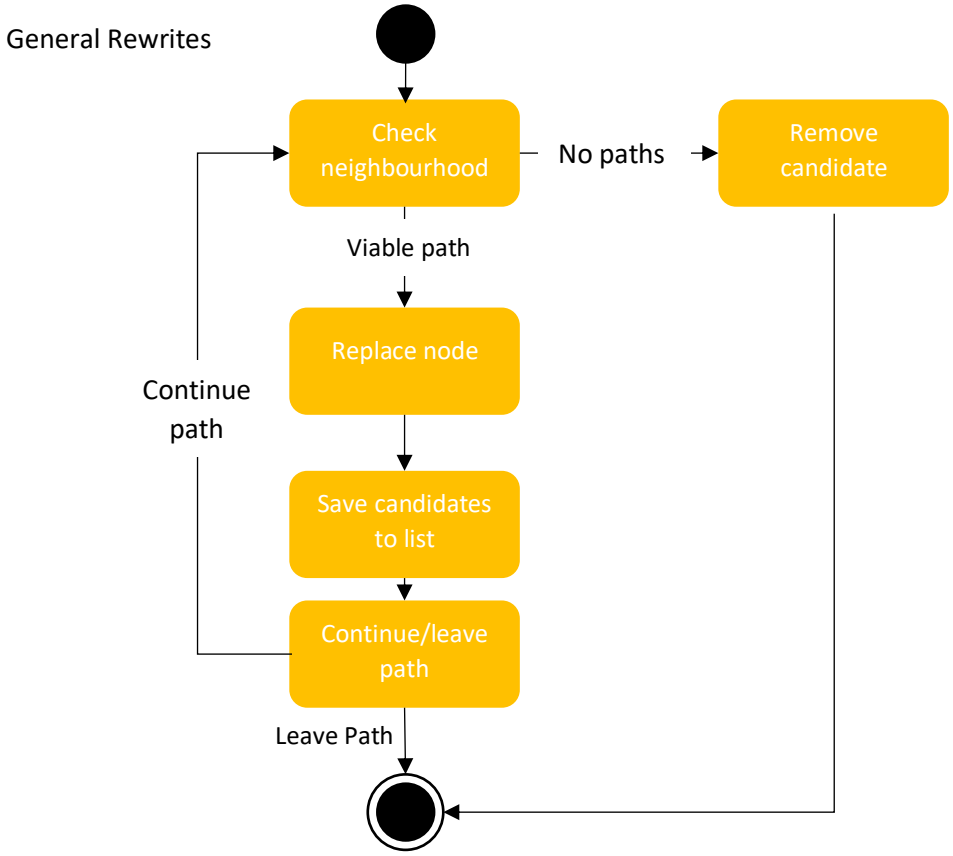


Figure 25. The structure of the General Rewrite method

The Start-to-Finish Rewrite is a method is the most essential step in the generation process and thus no errors should be allowed. If the system is given a start and a finish point the method needs to find a path that connects them and since it is embedded in RNGs (random number generators) the possibility of the winning path getting stuck somewhere is there. To ensure no failure happens at the start of the program, the method restarts itself in the case this happens.

To make an interesting, fun, and potentially difficult challenge the Start-to-Finish Rewrite method consists of two separate rewrites: at the start for a given number of steps it generates completely randomly and if those steps run out, the method switches to a more directed rewrite to make sure that the two ends are connected. Naturally this function also includes the search for neighbourhood and candidate saving elements of the General Rewrite.

Another rule could be added to this method to make the distance between start and end settable to make difficulty even more controllable. In the case that the path doesn't match

the requirements, the system would just revert to beginning and deleting the progress. This, in theory, would only be efficient to add if the proper optimization is done, since the system already seems quite heavy.

Figure 26. shows the planned workflow between the elements I described (without control over the length of the path):

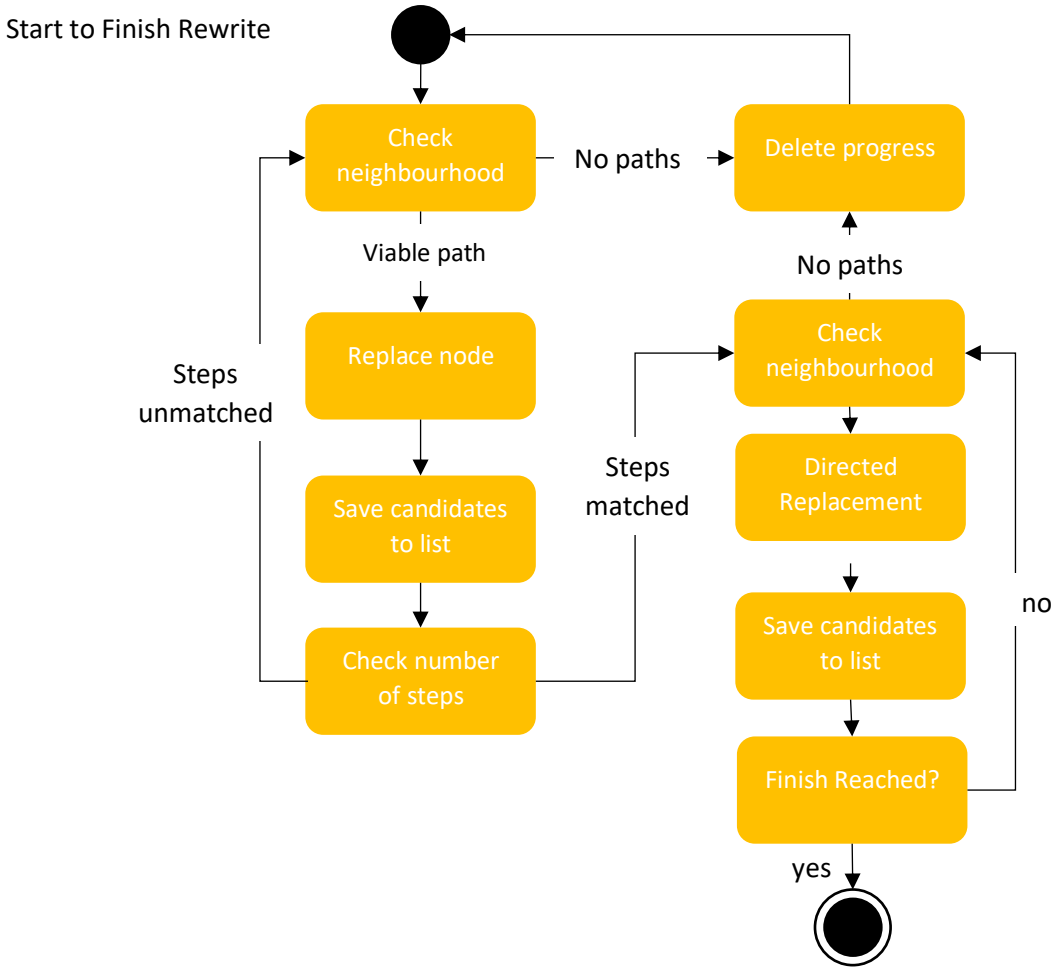


Figure 26. The structure of the Start to Finish Rewrite method

With the General and Start-to-Finish Rewrite methods written, the structure of the entire maze generator can be planned. The main method takes in the parameters given by the developer to determine the size of the maze and the starting node accordingly. After the winning path is decided the system randomly selects a node calling the General Rewrite method. If the method either decides to abandon that path or the path can't be continued the system jumps back to the random selection. Naturally the node with no viable neighbours gets removed from the advocate list. If this list becomes empty that signals to the system that the graph generation is done and the grid to house the layout is generated. After this each node is mapped to its location with walls being generated every direction an edge is absent. For testing purposes, I included an A* algorithm to check completability, even though it should be guaranteed. This will come in handy when

checking through the generation process step-by-step. Figure 27. visualizes the functions and relationships between them according to my description:

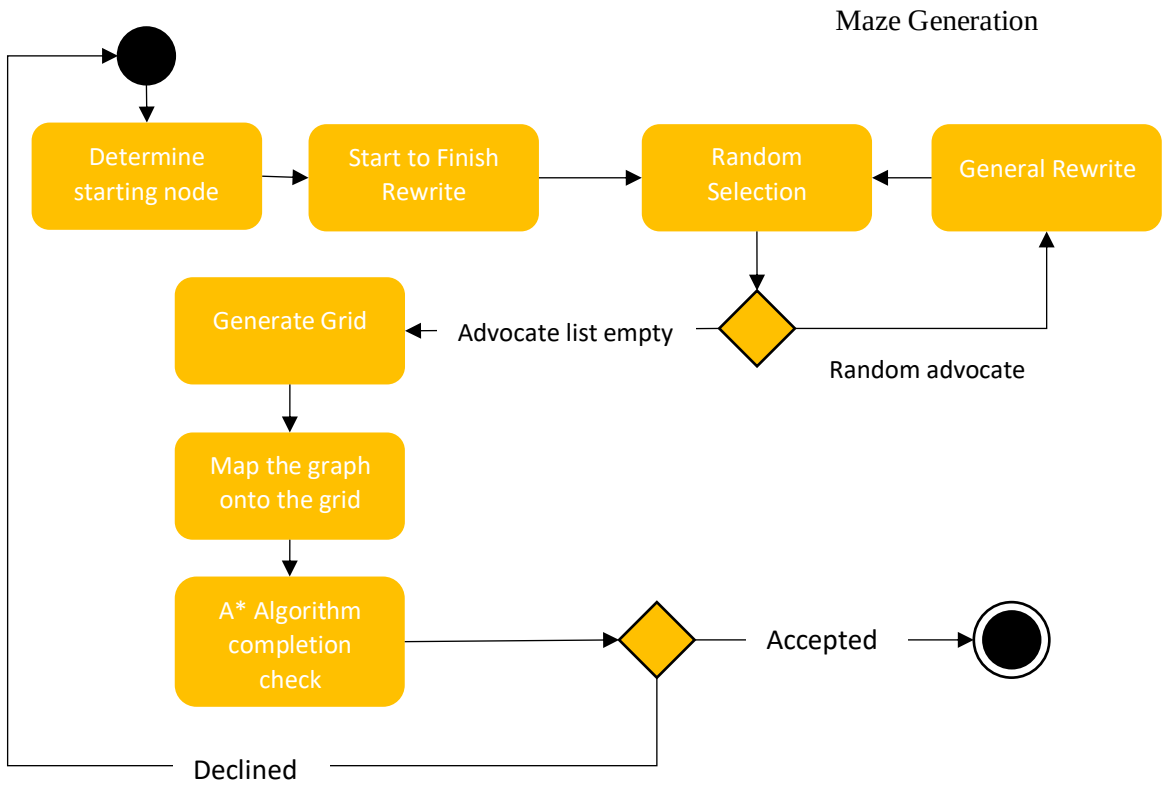


Figure 27. Flow chart of the maze generator

As I alluded to earlier, this system most likely won't be the most efficient for general maze generation, however its adaptability, flexibility and controllability aspects make up for it. My plans include a lot of optimizing after a beta version is done to get as much efficiency as possible.

4.2. Technical Requirements

The program will be written in Unity, due to multiple factors. For one, it uses C# as scripting language, which is the language I'm most familiar with. It is also a beginner friendly engine in terms of game development, having tools that could shorten the development time, since I've never developed a larger scale game like this. Third, the materials I took inspiration from are based in Unity as well.

4.3. Deployment Plan

At this point this maze generator is merely a skeleton that can be used as an asset for game development. I want to make the three ways that the system can be deployed. The first would be a method that would serialize the generated object so it can be used for games built in different languages and engines. The other two would show the capabilities of this system and to create something that can visualize its outputs.

One would be a simple program slowing the step-by-step process by taking inputs from the keyboard. This would serve as a testing and a demonstration tool.

The last way that I'm planning to deploy the generator is to build at least a beta version of a game that utilizes the asset. This would be an extension of the pre-existent code in Unity. The game will be a 2D top-down maze-solving game. The game will have the look an indie game with pixel graphics with the sprites created by me. I'll add a difficulty slider and game modes making use of the input parameters. The slider will control the size of the grid, the number of steps the Start-to-Finish rewrite will take before switching to directed rewrites and the chance of abandoning paths in the general rewrite, more difficult setting causing potentially more dead-ends. The game mode setting will essentially determine the placement of the starting and finish cells: regular, where the starting cell is in the middle of the level and finish at one of the corners, traditional, where the starting and finish nodes are placed in opposite corners of the grid, and chaos, where both nodes are placed at random. The "chaos" mode would require tempering with the Start-to-Finish Rewrite method, since my planned course of action is to take out the directed rewrite entirely, creating an even more randomized playing experience. In theory, this could be done by just making the directed rewrite part of the method optional, so it's not called mandatorily.

Another mechanism I would like to add is the ability to see already explored routes. This could be possible since a single cell has a 1:1 ratio and the screen is wider (16:9 aspect ratio at full screen). The game would black out areas not visited before and light up the areas that have. A restart button would be added to the gameplay to instantly regenerate the maze and a quit button to exit the game.

A possible addition to the game would be the key and lock system mentioned in multiple sections. By creating another special method based on Generic Rewrites that would trigger before it to create a path that ends with a cell containing a key. The Start-to-Finish Rewrite method could be tempered as well to randomly place a lock along its path. Due to time constraints this feature might not be added, but it would make a great addition to show the adaptability of the system.

4.4. Function List

This section goes over the already mentioned functions with the aim of creating a logical relevance to each other and describing them in simple terms.

4.4.1. Maze Generator Functions

Graph grammars/rewrites are used to overwrite a node with another node with different data or with a graph structure. These rewrites will be used in two larger functions: General Rewrites and Start-to-Finish Rewrites. The purpose of these functions is to create a maze layout in graph form to eventually create the final maze. A graph list will be used as a database containing the elements that will be used to rewrite nodes with the rules to accompany this process. A neighbourhood checker function will be used to evaluate the surroundings of a node to aid the rewriting decision. A candidate list will be used to keep track of possible continuations of paths. Each rewrite piece's nodes are added to the list and every node with no neighbours is removed. Random selection chooses a candidate node from to list to checked and possibly replaced. After the graph layout is created a grid is generated with respect to the size input and the input graph is mapped onto it using the Mapper function. The grid would generate with 4 walls surrounding each cell, and each edge of the graph would delete these walls connecting adjacent cells as the mapping would happen.

4.4.2. Additional Functions for the Planned Game

The difficulty slider function will be used as input and will have hardcoded spectrums of values determining the generation process. The game mode selection function will decide the starting coordinates of the start and finish nodes with some randomization as well. A texture generator function is also needed to create the UI of the game based on the input. A basic function of controlling the players movements need to be added at lest in the four cardinal directions and another to stop the player when hitting an object, in this case a wall. The function that serves as a tracking tool will memorize the cells traversed before and will light them up if the player is in an adjacent path, while keeping every other path darkened. This would have to happen seamlessly, my plan being, if the player is within one cell length proximity to a cell not separated by a wall it lights up. Figure 28. shows the relations between the functions listed in a system plan:

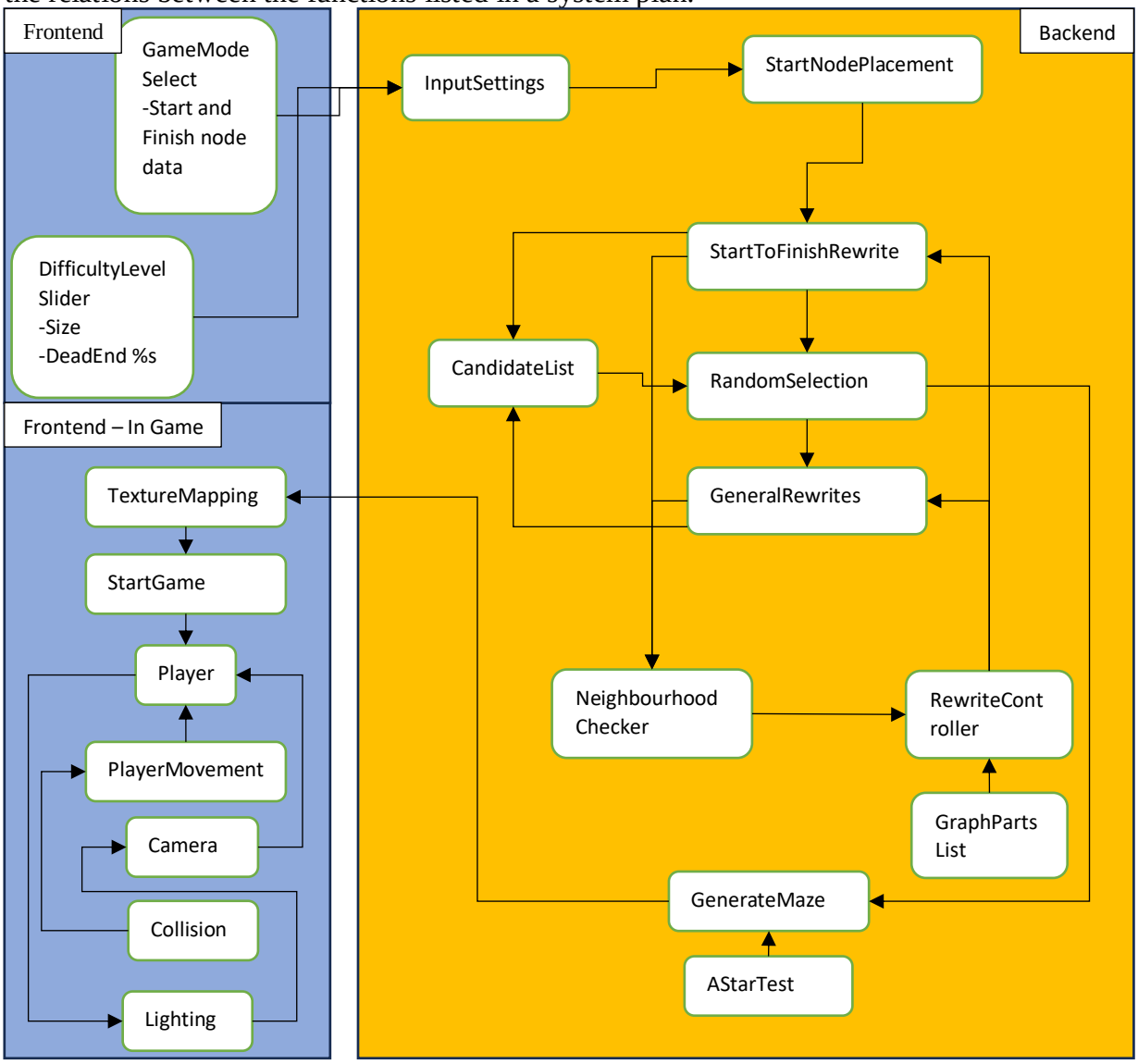


Figure 28. System Plan

4.5. Planned Course of Development

I will start the development of this project by setting up basic graph grammars on fixed size, first with just the Start-to-Finish Rewrite, and if that's done, move onto the General Rewrites. This will also require creating the list of graph parts, but at first the list will only contain two node graphs and I'll expand it with time. A testing phase trying different inputs will also take place to make sure the program runs correctly. After this I'll transfer the solution into Unity and start testing the graph mapping onto a grid. I'll try pre-written graphs at first and then move onto generated ones.

If the system functions correctly, I'll start creating the sprites necessary for the game, while also developing the in-game logic. Around this time the development of the tool for serialization and for the step-by-step simulation should start. If the game settings function properly the logic behind player and camera movement will be written. Only after these steps were tested and the sprites are done, I will write the method for texture mapping. The last step before system testing optimalization will be to develop the lighting system. Figure 29. Shows the planned course of development:

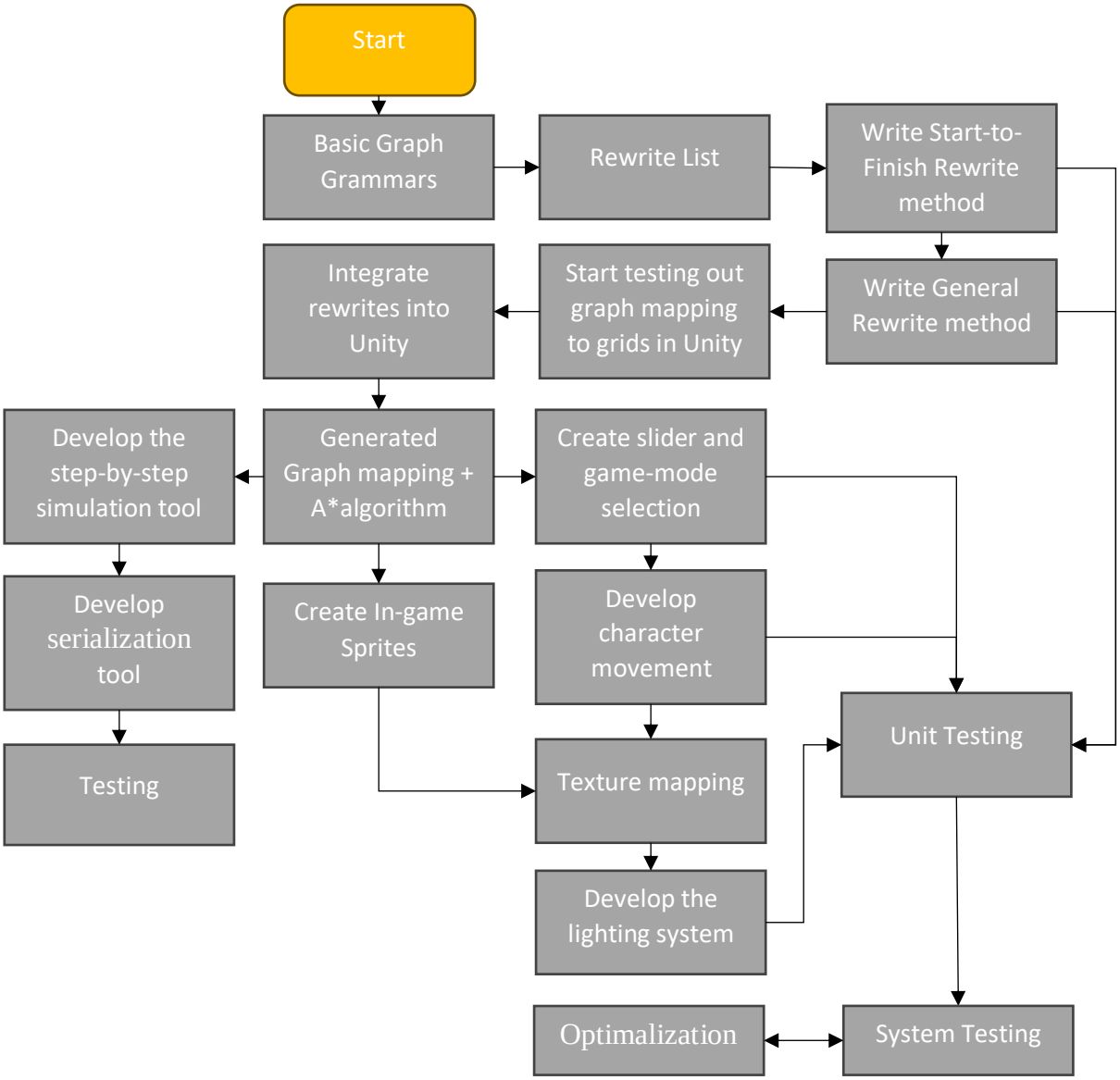


Figure 29. Planned Course of Development

5. Development

In this section I will detail the progress and trials of development of the Maze Generator program following the workplan explained in Section 4. This is to give insight into the thinking process behind the coding choices and the potential challenges and solutions to these challenges during development. The current state of the work is still in deep development; however, I will highlight what has been implemented so far, what is planned to be changed, and how I plan to continue.

5.1. Grid-Graph Structure

For the data structure of the maze generator, a standard graph class was modified to support the modular and constraint-heavy design requirements of this program. These modifications introduce a fixed 2D layout bordered by limits, with an InBound() method, for example, to indicate if a node can be added to the graph. This approach ensures that the generated maze fits within a predefined spatial boundary, allowing for control and predictability.

5.1.1. Node Functionality

In this Grid-Graph, nodes represent rooms, with edges signifying possible passageways between rooms. In this implementation, nodes are divided by types, all inheriting a common INode interface, utilizing Object Oriented Programming and polymorphism also. These types include the generic Node, which will populate most of the maze, and special nodes: a StartNode and an EndNode. Originally, I planned to use Boolean flags to distinguish special nodes by purpose. However, separating nodes into specific types supports extensibility, enhancing readability and allowing for simpler, more intuitive future expansions, like a key-and-lock system or specialized paths. This approach would also simplify management for path-specific rewriting processes. Type specific operations are mostly reserved for future extensions, since the special types currently in use only have one instance per generation, and these operations are truly useful when effecting multiple nodes, for example, if a combat system or a randomized texturization was implemented to the solution.

For the positioning of nodes, I used a Tuple data structure holding the X and Y values of nodes, and for the Dimensions of the Graph. Other important functions of nodes include ManhattanDistance() and VectorDifference(), functions for determining absolute (int value) and vector (int x, int y) distances between nodes. ManhattanDistance() is used extensively in determining regulation and process of the generation (examples of this will be shown in 5.2.). VectorDifference() was included, because it can provide useful information about the directional difference of a node and it would have been used in Directed Rewriting, however since the direction of the project changed in development it’s currently not utilized, this change will discussed in deeper detail in 5.2.

Figure 31. displays these distance determining methods:

```
public int ManhattanDistance((int x, int y) to)
{
    return Math.Abs(Position.X - to.x) + Math.Abs(Position.Y - to.y);
}
public (int x, int y) VectorDifference((int x, int y) to)
{
    return (Position.X - to.x, Position.Y - to.y);
}
```

Figure 31. Manhattan Distance (top) and Vector Difference (bottom) methods

The Connect() and Disconnect() methods are affect the connection of nodes. The Edge object used for this is a simple class using from and to node variables. The Connect() method makes use of Manhattan distance, since the distance between two nodes of a Grid-Graph can only be one unit. Disconnect() was used in the solution for end of generation cleanup, replacing nodes if required. Also, since the aim is undirected graphs, the goal node gets connected back to the origin node. Figure 32. displays these methods in code:

```
public void Connect(INode node)
{
    Edge edge = new Edge(this, node);
    if (ManhattanDistance(node.Position) == 1 && !Edges.Any(x => x.to.Position == node.Position))
    {
        Edges.Add(edge);
        node.Connect(this);
    }
}
public void Disconnect(INode node)
{
    if (Edges.Any(x => x.to.Position == node.Position))
    {
        Edges.Remove(Edges.Find(x => x.to.Position == node.Position));
    }
}
```

Figure 32. Connect and Disconnect methods

5.1.2. Graph Functionality

The Graph class functions as the database for all nodes, and for this reason all CRUD (Create, Read, Update, Delete) operation methods were included in its functionality. GridGraph can be instantiated in two ways: with a single tuple variable for determining X and Y dimensions and with two additional INode variables determining the Start and End nodes. Having two constructors aided the development of the generator, since the planned difficulty and game mode variables handle dimensions and the special nodes separately, this will be expanded upon in section 5.3. GridGraph also has a field for potential nodes, meaning nodes that have not been deemed dead ends. Nodes are removed from this list when the direct neighbourhood search returns null during the rewriting process. The GetRandom() method takes advantage of potential nodes, by only returning nodes from the list.

The most important function of graphs is the NeighbourhoodSearch() method. NeighbourhoodSearch() serves as a key method connecting the current state of the graph with the rewriting functions. It defines neighbours as unoccupied positions in the cardinal directions of a selected node inbound the confines of the graph (bool Inbound()). Neighbours located at two units are considered viable only if a direct neighbour exists between the target node and the selected position. This function is designed to return only empty spaces at the specified distance, without including any spaces in between. These design choices were made to improve efficiency and compatibility with rewriting functions. Figure 33. shows the main code of this method:

```
foreach(var side in sides)
{
    if (InBound(side) && GetNode(side) == null)
    {
        Node newNode = new Node(side);
        if (!emptyPositions.Contains(newNode))
        {
            emptyPositions.Add(newNode);
        }
        if (n > 0)
        {
            foreach (var side2 in NeighbourhoodSearch(side, n - 1))
            {
                if (!emptyPositions.Any(x=>x.Position==side2.Position))
                {
                    emptyPositions.Add(side2);
                }
            }
            emptyPositions.Remove(newNode);
        }
    }
```

Figure 33. NeighbourhoodSearch() method

It is important to mention that, as can be seen, the method uses recursion to get nodes at distances greater than one. This isn't the most efficient approach for larger searches, but it was chosen for due to the scale of the project only requiring search at most distance 2. If the final application will be expanded the approach for neighbourhood searching needs to be changed also.

5.2. Rewriting Function(s)

The first iteration of the rewriter was a simple program utilizing only direct neighbours without special node types, naturally for testing purposes. Since then, the rewriter class has gone through multiple approaches and stages to get to a final design, due to issues mainly regarding the specialized and separated Rewrites discussed in Section 4. This section will go through the main troubles and milestones in the process.

5.2.1. Issues with Separate Rewrites

The first encountered issue with implementing a separate rewriter for start-to-end connection was constant cornering. Since the method would have to take several steps randomized steps before trying to connect with the exit of the maze it, in most cases, either chose a node with no viable connection points or the shape generator created an additional node blocking the path.

Figure 34. provides some clarity by visualizing these issues:

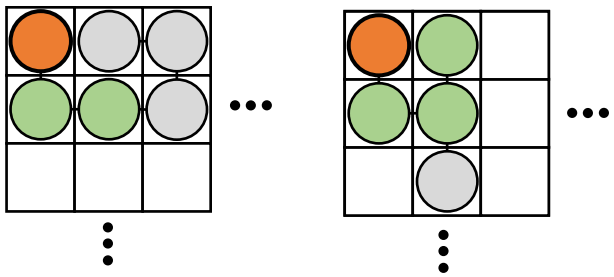


Figure 34. Self-Cornering before (Left) and after (Right) EndNode Assignment

The fix for the issue seemed simple, by adding rules to inhibit EndNode assignment to near out-of-bounds “cells”. This approach, while improving results, still didn’t guarantee maze completion.

In the next approach directed and undirected rewrites would be launched in the function, the method choosing one of the rewrites randomly. This was my original plan when starting the project for the directed rewrites, to disguise the obvious path patterns they would create. Consistency improved; however, the output still wasn’t optimal.

The last approach was to restart the process each time the minimal steps requirement wasn’t met. This approach, while made the outcomes consistent, introduced two new issues. As expected, when scaling the method up to generate multiple mazes after each other, the generation time increased massively when compared to the successful generations of previous iterations, but the biggest issue was recurring patterns, making the generated mazes easily solvable, and decreased variety.

5.2.2. Final Design for Rewrites

From the earlier iterations I concluded that creating a Start-to-Finish Rewrite function was redundant and inefficient. While, in theory, it would guarantee the completability of the maze, the efficiency that is lost still makes it a wrong approach for the purposes of this work and relegates the validation tool of the solution to just an added feature. By simplifying the system to an all-purpose rewriter is both efficient and, if given specific enough rules, also fault-tolerant, and provides consistent results. The downside of the approach is the loss of some control over the output by small margins.

The Rewriter class’s actions are governed by an Adjacency ruleset, which is a dictionary for determining path continuation of a node. The key of the collection represents how many edges a node has and the value field represents what the possibility of adding an additional edge to node is based on the key. For an instance of a Rewriter to be created the percentage chance of adding a 3rd and a 4th edge and chance to add nodes at distance two from the node need to be specified. In earlier versions these percentages worked on a case-to-case basis, generating too many crossways and dead-ends. In its current state, the rewriter prioritizes dead ends to be continued, making a more natural looking maze.

The core of rewrites is the GenerateShape() function. The purpose this shape generator is to create a smaller path system based on a node of the graph, using the previously mentioned neighbourhood search. This generation needed to be specific, due to some

encountered issues. Generating the shape blindly caused the exit/finish node to continue, creating redundant paths. Two mistakes caused this error: the EndNode not being removed from the potential nodes list and later distance-two rewrites connecting distance-two nodes to it. For the latter, if statements were added to stop distance-two rewrites when the rewritten node is at distance one from the goal and to stop the rewriting completely one the goal is found.

The final code first attempts to create a node at distance one, based on the parameters discussed earlier:

```
while (neighbours.Count() > 0 && numberOfEdges < AdjacencyLimit && AdjRuleSet[numberOfEdges] > rnd.Next(1, 101))
{
    if (ChanceForDist2Rewrite > rnd.Next(1, 101) && neighbours_2.Count() > 0 && node.ManhattanDistance(Graph.goal) > 2)
    {
        INode newNode = neighbours_2[rnd.Next(neighbours_2.Count)];

        List<INode> connectionPoints = returnList.FindAll(x => x.ManhattanDistance(newNode.Position) == 1);
        connectionPoints.AddRange(neighbours.FindAll(x => x.ManhattanDistance(newNode.Position) == 1));

        INode connectionPoint = connectionPoints[rnd.Next(connectionPoints.Count())];
        if (!returnList.Any(x => x.Position == connectionPoint.Position) && numberOfEdges < AdjacencyLimit)
        {
            neighbours.Remove(connectionPoint);
            returnList.Add(connectionPoint);
            connectionPoint.Connect(node);
            numberOfEdges++;
        }
        newNode.Connect(connectionPoint);
        returnList.Add(newNode);
        neighbours_2.Remove(newNode);

        if (newNode.Position == Graph.goal)
        {
            break;
        }
    }
```

Figure 35. Shape Generation at distance two

If rewriting at distance two doesn't meet its requirements, the method continues with distance one:

```
else
{
    INode newNode = neighbours[rnd.Next(neighbours.Count)];
    returnList.Add(newNode);
    neighbours.Remove(newNode);
    node.Connect(newNode);
    numberOfEdges++;
    if (newNode.Position == Graph.goal)
    {
        break;
    }
}
```

Figure 36. Shape Generation at distance one

An important mention for the Rewriter is that, like the GridGraph class, it can also be instantiated multiple ways. Only a graph variable is essential for instantiation; by leaving out the percentage chances for adjacencies, the Rewriter creates an Adjacency ruleset and the adjacency limit according to the input. An example of this is if instantiate Rewriter without the percentage value of generating a fourth adjacency, the adjacency limit will decrement to three.

A note for the distance two section of the shape generation is that the code is way too bloated, and I've considered only using the distance one part of the code, because it could

potentially increase efficiency, but due to the number of changes already made and this approach taking away some extendibility, the modification wasn't made. In the future I plan to compact the code for readability.

The Rewrite function (Figure 37.) is responsible for making the actual change in the input graph. It takes in a node, forwards it to the Shape Generator, and based on the response either removes the node from the potential nodes list or rewrites the node with the given path structure.

```
public List<INode> Rewrite(INode node)
{
    if (Graph.NeighbourhoodSearch(node.Position, 0).Count()==0 || node.Edges.Count()==AdjacencyLimit)
    {
        Graph.PotentialNodes.Remove(node);
        return null;
    }
    else
    {
        List<INode> replacement = GenerateShape(node);
        foreach (INode n in replacement)
        {
            Graph.AddNode(n);
            if (n.Position == Graph.goal)
            {
                Graph.PotentialNodes.Remove(Graph.GetNode(n.Position));
            }
        }
        return replacement;
    }
}
```

Figure 37. Rewrite function

5.3. Generator

The Generator class is the central component to the entire application encompassing and making use of every class and method mentioned so far. A Generator object can be initialized two ways for separate purposes: using pre-made instances of GridGraph and Rewriter for testing purposes and using integers for difficulty and mode, that are then used to generate the mentioned objects in the class itself. The latter uses existing setups with some randomization, this is for the practical use of the deployment application. Section 5.3.1. will discuss the details of the class-contained graph and rewriter generation.

5.3.1. Generate Graph, Layout

Three values were created for both the difficulty and mode variable with dictionary structures defining the correlating values to the numbered keys. A unique GenerateGraph() function was defined, which makes use of the multi-constructor property of the GridGraph class, since difficulty affects the size of the structure while mode affects the placement of the special nodes within the structure. The values assigned to difficulty level in the graph generator are not straightforward, instead they use ranges, where a randomizer chooses their values (for example: difficulty value: 1 => size between 5*5 and 10*10). At this point only an empty graph is created, so the nodes can make use of the Graph class's functions. After the size is decided, the start and end nodes are assigned based on the game mode value:

- Mode Value 1: Special nodes are assigned to corners
- Mode Value 2: Start node assigned to the middle, end node to a random corner
- Mode Value 3: Both nodes assigned randomly, with at least a set distance between them (Graph dimension value divided by 2)

After node assignment, the function returns a graph with the defined start and end points.

GenerateLayout() produces the final maze before validation, and uses the functions provided by the rewriter. The main functionality of this generator is to pick a node from the Potential nodes list and rewrite it, continuing this process until the maze is completely populated. An extra feature was added, that on a 25% basis picks a path ending (a node with only 1 connection) from the rewritten structure and attempts to continue that path, instead of random decision. This also helps prevent cases, where the maze gets populated by short, dead-end paths. When the layout is generated, the end node gets assigned. Figure 38. shows the code for the GenerateLayout() method:

```
public void GenerateLayout(INode node)
{
    while (Graph.PotentialNodes.Count() != 0)
    {
        Random rnd = new Random();
        List<INode> nodes = Rewriter.Rewrite(node);
        while (nodes != null && 0 == rnd.Next(4))
        {
            List<INode> continuations = nodes.FindAll(x => x.Edges.Count() == 1 && x.Position != Graph.goal);
            if (continuations.Count() > 0)
            {
                INode continuation = continuations[rnd.Next(continuations.Count())];
                node = continuation;
                nodes = Rewriter.Rewrite(node);
            }
        }
        node = Graph.GetRandom();
    }
    EndNode end = new EndNode(Graph.goal);
    Graph.Replace(end);
}
```

Figure 38. Layout Generator Method

Figure 39. shows the culmination of the individual parts discussed in the practical instantiation logic of the Generator class

```
public Generator(int difficulty, int mode)
{
    Graph = GraphGenerator(difficulty, mode);
    Dictionary<int, Rewriter> DifficultyManager = new Dictionary<int, Rewriter>()
    {
        {1,  new Rewriter(Graph, 15, 5, 75)},
        {2,  new Rewriter(Graph, 25, 10, 60)},
        {3,  new Rewriter(Graph, 30, 15, 50)},
    };
    Rewriter = DifficultyManager[difficulty];
    GenerateLayout(Graph.StartNode);
}
```

Figure 39. Generator Class logic

5.3.2. Validation

For the validation, Depth-First Search (DFS) was chosen as the traversal algorithm instead of the planned A* algorithm. This is due to the further research into maze validation best practices pointing to redundancy in using the A* algorithm. This redundancy comes from the fact that the A* algorithm is designed to detect obstacles in more open environments with interconnected paths. A closed structure like a maze, with every two points being connected by a single path doesn't capitalize on the functionalities of the A* algorithm and makes the approach overkill. Also, in this scenario, not much, if any efficiency is lost when using DFS. DFS is perfect for the purposes of the project, because it suits a closed environment well and also guarantees to find the path to the exit. In this solution a tuple

was used to validate if a neighbouring node leads to the maze exit while also storing the winning path into a list once validated. The algorithm uses recursion as DFS usually does, the function calling itself each time it travels to the next node. The DFS used in the solution can be seen in Figure 39:

```
public (List<INode> path, bool valid) DFS(INode start, INode goal, HashSet<INode> visited)
{
    visited.Add(start);
    List<INode> path = new List<INode>();
    bool valid = false;
    if (start.Position == goal.Position)
    {
        path.Add(start);
        valid = true;
    }
    else
    {
        foreach (Edge e in start.Edges)
        {
            if (!visited.Contains(e.to))
            {
                (List<INode> path, bool valid) i = DFS(e.to, goal, visited);
                if (i.valid)
                {
                    valid = true;
                    path.AddRange(i.path);
                    path.Add(start);
                    break;
                }
            }
        }
    }
    return (path, valid);
}
```

Figure 40. Depth-First Search algorithm

DFS was used both for the Validate() and GetWinningPath() functions. GetWinningPath() returns the path from the start to the end node, first returning the starting node.

5.4. Serialization

Serialization to JSON format was added to the GridGraph structure for use of the generated structures outside of C#. For the objects of the maze to be serializable some changes were made, and fields were added to the existing classes. ID fields were added to all node classes to be more readable once converted, and to make adjacencies clearer. All nodes are assigned IDs once they are added to the graph, the ID being calculated the following way: Position X * Position Y + Position X. There were issues with cycles, because the graph is undirected, adjacent nodes would cycle between each other. The solve for this was to introduce a new field NeighbourId that would track every new adjacency neighbouring node, and the edges field was set to be ignored by serialization. The other fields converted in the serialization process are X and Y separately (because JSON doesn't comprehend tuples), and type which is a specific string for each type of node.

5.5. Unity Application

The other two deployments, being the test game and the visualization tool, were both implemented in Unity as part of the same application. This section will discuss both parts of this solution, the usefulness of Unity tools in development and touch on the graphics (sprites and animation) that were created for the visuals of the program.

5.5.1. Unity

Asset types, important to the project, in a Unity file are scripts, sprites (visual assets), animations (making use of sprites), game objects (structures using the aforementioned assets alongside Unity native functions) and scenes (containers for environments). Every script created in Unity includes a start function, responsible for executing code within it at initialization, and an update function, that executes code every frame after start. Unity allows for the creation of 2D and 3D environments, and, while for this work only 2D was used, there are no restrictions in using the maze generator in 3D solutions. The platform was chosen because it provides many native assets to make development easier, and since the test game and the visualization tool are both for demonstration purposes, it's the best option to show the results of the application.

5.5.2. Menu

For the menu scene of the application, I used native UI elements that Unity provides for the buttons, slider and dropdown list. All these native elements allow for action triggering and saving data for later use. A static Game Settings class was created and linked to the game mode selection dropdown and difficulty level slider to save these fields when generating the maze. They also uniformly affect both the game and the visualizer.

It's important to mention that each menu component is handled by a canvas object which creates an immovable overlay on the screen. A "Main Menu" canvas item was also added to the other scenes, making sure a user can return any time. A Menu Logic script was created to handle all these interactions.

5.5.3. Game Development

The first step in developing the test game was to create all the game objects that will be procedurally placed and copied as part of the generation.

The Player game object uses two Unity native components: Rigidbody and CircleCollider. Rigidbody allows on key displacement for a game object. Player movement was implemented using this component, and the platform's in-built cardinal key push recognition. The collider component's functions serve as the interaction between game objects. It can be set to stop player movement when colliding with other game objects, or to trigger actions. The main camera was added to the player tracking its movement.

Room objects are representations of each node of the generated graph. To implement the edges to act as connections between rooms, every wall acts as a separate object. The ConfigureWalls() method acts when generation happens, tit searches for an adjecncy each cardinal direction and destroys the wall object in that direction. The script also assigns X, Y, Width and Height values which will be necessary in generating the rooms at their proper place.

The exit game object has a single function, which is to activate the game over overlay when the player object reaches it. This is done by adding a Collider component to it and setting it to trigger the moment the player's Collider component touches it. This sends a message to the Logic class to stop player movements and activates the "Game Over" layer.

After creation each object was saved as Prefabricated Objects (prefab for short) for reuse in the main Maze Generator object. Figure 40. shows how the Maze Generator script is constructed:

```
public GameObject PlayerPrefab;

public GameObject RoomPrefab;

public GameObject ExitPrefab;

Generator gen = new Generator((int)GameSettings.difficulty, GameSettings.mode);
3 references
GridGraph graph => gen.Graph;
```

Figure 40. Maze Generator Script input values

The generator object runs graph generator, saves the layout and constructs the Maze. For each of the prefabs a spawn method was written using an INode parameter as input. For an example Figure 41. Displays the SpawnRoom() method (the float values signify the scale of the rooms):

```
public GameObject SpawnRoom(INode node)
{
    int x = node.X;
    int y = -(node.Y);
    Vector3 pos = new Vector3(x * 7.5f, y * 7.5f, 0);
    GameObject newRoom = Instantiate(RoomPrefab, pos, Quaternion.identity);
    Room roomScript = newRoom.GetComponent<Room>();
    roomScript.ConfigureWalls(node);

    return newRoom;
}
```

Figure 41. Room Spawning method

At the start of the scene the rooms load first for each of the nodes, then the player and exit objects generated based on the input value of mode.

5.5.4. Generation Visualizer

The visualizer uses many of the same assets as the game with additional functions. The purpose of this app is to follow the generation process. This is achieved by following every button press the next room generated is shown in order. Since in the main application nodes are already saved in order of creation no further methods were needed to organize the order. The main camera of this scene is always adjusted to the size of the input graph. After every room is placed, all rooms are destroyed and replaced with the path connecting the start and exit rooms on the next button press.

Some of these paths in different game modes can be seen in Figure 42:

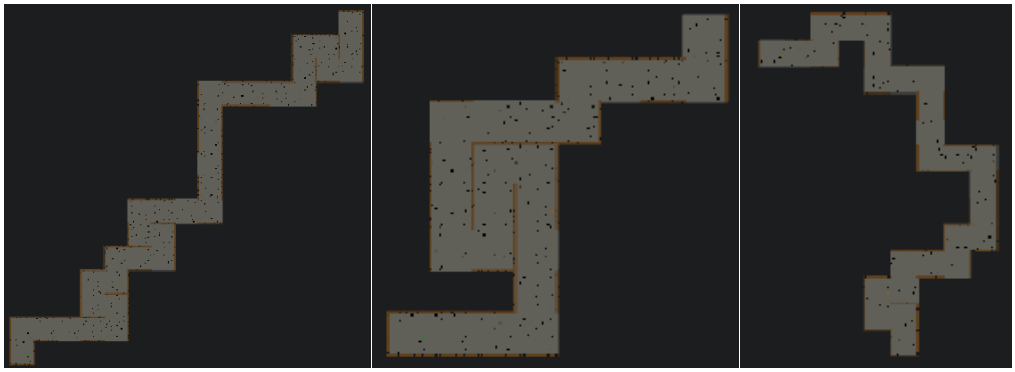


Figure 42. Generated paths between start and exit

6. Testing

This section aims to provide insight into the limits and performance of each component of the project. Testing will be split into four categories: manual, which will include every test conducted during development and the playtesting of the test game, unit testing, for more concrete larger scale testing of the components, integration, to ensure connectivity between the modules, both in the generator and the Unity application, and performance testing, to assess and compare speeds of execution.

6.1. Manual Testing

Manual testing was integral for each step of development, using multiple test cases to see if the individual methods function correctly.

The first test conducted was to see if creating different types of nodes, connecting them and adding the to a graph works properly. Multiple aspects were tested at this stage, like:

- Are the nodes set up properly, so that different types inheriting the sane interface can be connected without issue?
- Is the ManhattanDistance() method working properly, and is returning the proper values?
- Can two nodes with Manhattan distance greater than one be connected to each other?
- Are the nodes integrated properly to the graph? Does each one get added regardless of type? Can they be added if their positions point outside the boundaries of the graph, or simply, is the InBound() function working?
- Does node connectivity work both ways? While not integral to this test, does the Disconnect() method work?

After several tests, it was concluded that each module works as intended, the data structure of the project is ready to be used.

NeighbourhoodSearch() was tested heavily, due to the reliance of the rewriting process on the proper working of it. As mentioned, this method was reworked multiple times to get optimal results. The points of requirements are as follows:

- Does NeighbourhoodSearch() return node placeholders at the required distance, and only at the required distance?
- Does it account for connectivity to the origin node, i.e. if a neighbour position is already filled, does it stop searching for nodes connected to the filled neighbour position?
- Does it account for out of bounds node positions?
- (optional) Can it be scaled to larger than the required distance two?

The final method covers all necessary aspects considered and returns optimal results. From the testing, while in rare cases, when searching for nodes greater distances away than three the method tends to return either duplicates, or nodes inaccessible with a path

from the origin node. Also, since it is a recursive function, execution time for input distances larger than five can increase exponentially. For the work this isn't a big issue, however it does need reworking if extensions are planned to be added.

The Rewriter and Generator classes were tested simultaneously, because they are essential in each other's functioning. A multitude of tests were performed, due these functions defining the entire program, and need to work precisely. Contrary to the schema used so far, each point of testing will be observed and reflected upon separately.

- When generating a shape to rewrite a node, does the GenerateShape() function account for EndNode proximity and avoids making it a connection point between two nodes, and also removes the possibility of it potentially being rewritten in the future?
  - ✓ After testing for hundreds of scenarios, making a variable to track EndNode number of edges in each iteration, it was determined that in each case the node only has a single connection, as designed.
- Do all nodes have connections, i.e. was there no inaccessible area generated?
  - ✓ In all cases, no issues were found.
- How do adjacency percentages affect the output? How does changing these values affect the outcome and difficulty? Do all instantiation cases return optimal results?
  - ✓ When researching mazes, it was found that the difficulty of a maze is based on balanced percentages. If the probability of adding additional adjacencies is below 10%, not too many paths are generated, making the maze easier to solve, as expected; however, if the probability is high (beyond 50%), the maze is flooded with very short paths, a couple of long paths, seemingly causing the same result as before. The fine balance used in the solution is between 5-30%, which covers all difficulty settings.
  - ⊗ Some issues were still found, when the adjacency limit was set to two. With this input the generator tries to create a labyrinth instead of a maze, however, due to the approach, it either makes exceptions in cases, generating nodes with 3 adjacencies, or returns null, signifying failure. This is a non-issue in terms of the goal of the project, but it does show the limitations of the algorithm.
- Does validation work properly? Does the GetWinningPath() method return the required path?
  - ✓ When running the 100 test case method with random input values, each time the maze was validated, and from the reviewed examples, the path and thus the assessment was correct as well. In playtesting the same result can be said.

During playtesting a single issue was found at the beginning of development, which was the maze being generated upside down. This was due to the generator accounting for the down direction being associated with increasing the Y coordinate and up being with decreasing it, while in Unity it's vice versa. This was a quick fix by negating the Y values of position in nodes and dimension in graphs. No other issues were found, the program runs perfectly as intended, aside from very minor visual glitches.

6.2. Unit Testing

Unit testing ensures individual components of a system function correctly in isolation, helping to identify and fix issues early in development. For this project, the NUnit framework was used, offering a powerful and flexible toolset for writing and executing tests in .NET applications. Its rich assertions and test utilities enable efficient validation of key features.

This section covers tests implemented for functionalities, most of which I covered in Manual testing, to guarantee everything works properly. These tests aim to cover multiple fields at once; Table 3. displays the information gathered:

#	Test Description	Goal	Result
1	Test connectivity rules between nodes	Any two node types should be able to be connected Only nodes in Manhattan Distance = 1 can be connected	True
2	Test graph boundary rule	Only nodes with positions within the boundaries of a given graph can be added to it	True
3	Test start to end connectivity (100 randomized cases)	When the layout is generated a path between start and exit must be generated, regardless node placement	True
4	Test if graph was filled (100 randomized cases)	As established, the type of maze required from the program must fill out an entire “grid”	True

Table 3. Maze Generator test cases

6.3. Performance Testing

Performance testing evaluates the efficiency and responsiveness of the maze generator under various conditions. Using NUnit, custom tests were designed to measure execution times and ensure the generator consistently meets performance expectations.

This section highlights how the generator was tested to handle different maze sizes and complexities, focusing on runtime and success rates. NUnit's capabilities allowed for clear performance benchmarks and provided insights into potential optimizations. Table 4. Shows my findings in the tests conducted:

#	Test Case	Result
1	Graph.based instantiation: 10x10	18 ms
2	Graph.based instantiation: 30x30	153 ms
3	Graph.based instantiation: 60x60	2.2 s
4	Graph.based instantiation: 100x100	16.5 s
5	Difficulty-based instantiation: Lowest Difficulty	~18-20 ms
6	Difficulty-based instantiation: Medium Difficulty	~31-76 ms
7	Difficulty-based instantiation: Highest Difficulty	~447- 1500 ms

Table 4. Performance tests

The performance statistics clearly show that optimalization is needed for larger generations. This isn’t an issue, when using the pre-established difficulty/mode parameter approach; during playtesting the wait time is unnoticeable. Still, if this project gets extended, some changes need to be made to the code to optimize larger scale generation.

7. Assessment

What is clear after comparing the assigned task paper and final product I created it is clear lots of changes have been made along the way. In this section I will assess the results of my work, from the decisions made in the planning stages to the development phase, judging the products implications and performance, while also reiterating the thought process behind the necessary changes.

The Start-to-Finish Rewrite was the original plan for connecting and guaranteeing validation even before that step. Compared to that plan by using a straightforward all-purpose rewriter approach greatly increased efficiency and consistency. The change in course regarding the validation of the maze from the A* algorithm to a much simpler Depth-First search algorithm I thought was necessary. While the A* algorithm is the superior approach for finding the shortest paths between points, the structure of a maze dictates a single path in all cases, making the A* algorithm redundant. Through research I found that the difference between the algorithms in terms of speed in this scenario is marginal.

After conducting tests, every functionality that was asked was developed and working with great results. The generator is modular and flexible allowing for all kinds of game development, be that in a 2- or 3-Dimensions. The system can be integrated into an environment, with little effort needed to build a game around it. I have proven this by developing my own using the functionalities the generator provides, with the layout integration taking the least amount of time of the whole project.

Besides easy integration, the modularity provided makes extensions to the program easier, so that it can be shaped to whatever a new project requires it to do.

Every deployment planned was realized. The only thing left out of the final program was the lighting system for higher difficulty and for game aesthetics, for time constraints. This feature is planned out and will be added in the future. The maze solving game works perfectly, generating the maze and allowing for the player movement discussed in Planning. The same can be said about the visualizer, providing deeper insight into exactly how the generation process happens. This program could also be used in the future as feedback to see every change future development causes.

Testing shows that for the purpose of what the program was intended to be used for the performance of the generator is sufficient. Still, under heavier loads the generation time increases to less-than-ideal metrics. This is a point that can be improved upon greatly.

Overall, the task was finished, with each component implemented, with the slight change of the validation algorithm. The final program created in Unity can be downloaded and viewed through the link in the attachment, where the created sprites (visual assets) are displayed as well.

8. Future

This sections focus is on everything that hasn't been developed yet, encompassing both the features that were left out due to time constraints, fixing current bugs, and potential new features that the application can be extended by.

8.1. Fixes, Missing Features

The most glaring feature left out was the lighting system detailed in the Planning section (Section 4.), which was not integrated due to a lack of time. A simple way of including in the future is to create a varied Room game object, functioning the same way the parent object does, but with an all-black texture. All rooms would spawn as this variant, and with a script used for triggering a change in texture once the player's collider interacts with either the rooms or its neighbours' collider. Unity native shaders and lighting could also be used, but as of writing, these components have not been studied enough yet.

Visual bug fixes, the shortenings of some code and reworking certain generation components all need to be addressed in the future. These elements were discussed in more detail in previous sections.

8.2. Possible Expansions

Due to the modular nature of the code many expansions can make for unique solutions using this application. The plan was for an even more flexible program, however, with the tools given and some slight adjustments to the code, many possibilities are still open.

With adjustments to the rewrite method, or creating separate rewrite for this purpose, a layout generator for more of a dungeon-like map creation can be made, by the end goal being changed from filling the entire grid, to completing a certain number of steps. Another method could be added to generate rooms instead of paths, connected by doors. Each shape generated could become an interconnected room, the outcome becoming an interesting take on the mentioned rogue-like genre of maps.

Extension into 3-Dimensional Maze generation as discussed in the Literature Study is also a possibility. This approach would require a great deal of extending established fields, but the functions, once extended with a Z dimensional value, should be able functionally work the same way, producing optimal results. Since this field of study is very thin, this would be the first step in further developing this solution after the bug fixes and optimalization.

9. Summary

This body of work was a passion project of mine, entailing one of the topics I find most in IT, Procedural Generation. To conclude, this thesis set out to give a description of the landscape and history of PCG/PLG, reviewing solutions of past and current solutions in the field. The findings show how vast this area is in programming, and how many ways a single problem can be solved. Validation approaches and other useful technologies were also discussed, that can be used to expand my solution in the future.

In my solution I used a less used approach in Level Generation, Graph Rewriting, which, from my experience, is a very underutilized approach, with some great advantages. These advantages being the amount of control the approach provides, while also being highly modular. I am satisfied with how the project came together, both with the generator itself and with the demo game. The flexibility of the generator, along with its performance across different platforms, demonstrates its utility in game development.

The outline of this work is not by any means the extent of my plans with this approach. I want to enhance, polish and experiment with the current code further, to make something truly unique. The potential of Procedural Generation is endless, and this work was to contribute to the advancement of it, offering a foundation for further innovation in videogame map generation.

I am truly grateful, that I could work on a project like this, even with all the trials and tribulations that have come with it. A special thank you is necessary to my consultant, Miklós Sipos, for the guidance and patience for the last two semesters, and also to everyone I talked to that have shown the same interest in the subject as me.

References

[1] N. Brewer, “Computerized Dungeons and Randomly Generated Worlds: From Rogue to Minecraft,” *Proceedings of the IEEE Editorial Office: Scanning Our Past,* pp. 970-977, 2017.

[2] T. Bacher, “Procedural Level Generation Algorithms,” 2018.

[3] R. C. e. Silva, “Procedural Game Level Generation by Joining Geometry with Hand-Placed Connectors,” 2020.

[4] F. Himsl, “YouTube.com - Florian Games,” 19 04 2020. [Online]. Available: https://www.youtube.com/watch?v=1-HIA6-LBJc.

[5] BorisTheBrave, “boristhebrave.com,” 12 09 2020. [Online]. Available: https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/.

[6] A. Zucconi, “alanzucconi.com,” 05 06 2022. [Online]. Available: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/.

[7] C. Salge, “Impressions of the GDMC AI Settlement Generation Challenge in Minecraft,” in *Association for Computing Machinery*, New York, NY, USA, 2022.

[8] J. Doran, “Controlled Procedural Terrain Generation,” *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,* vol. 2, no. 2, pp. 111-119, 2010.

[9] A. Esko and J. Fritiofsson, *Multi-Agent Based Settlement Generation,* 2021.

[10] M. Awiszus, “World-GAN: a Generative Model for Minecraft,” in *2021 IEEE Conference on Games (CoG)*, Copenhagen, Denmark, IEEE, 2021, pp. 1-8.

[11] P. Gabrovsek, “Analysis of Maze Generating Algorithms,” *IPSI Transactions on Internet Research,* vol. 15, no. 1, pp. 23-30, 2019.

[12] R. v. d. Linden, “Procedural Generation of Dungeons,” *IEEE Transactions on Computational Intelligence and AI in Games,* pp. 78-89, 2014.

[13] H. S. Nwana, “Software Agents: An Overview,” *Knowledge Engineering Review,* vol. 11, no. 3, pp. 1-42, 1996.

[14] D. Adams, *Automatic Generation of Dungeons B. Sc. thesis,* Scheffild, UK, 2002.

[15] D. Ashlock, C. Lee and C. McGuinness, “Search-Based Procedural Generation of,” *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,* vol. 3, no. 3, pp. 260-271, 2011.

[16] T. Roden and I. Parberry, “From Artistry to Automation: A Structured Methodology for Procedural Content Creation,” in *Entertainment Computing - ICEC 2004*, Eidenhoven, The Netherlands, 2004, pp. 151-156.

[17] A. Chad and S. Louis, “Procedural maze level generation with evolutionary cellular automata,” in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, Honolulu, HI, USA, IEEE, 2017, pp. 1-8.

[18] A. Gellel and P. Sweetser, “A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels,” in *Proceedings of the 15th International Conference on the Foundations of Digital Games*, New York, NY, USA, Association for Computing Machinery, 2020, pp. 1-10.

[19] N. Kumar and S. Kaur, “A Review of Various Maze Solving Algorithms Based on Graph Theory,” *International Journal for Scientific Research & Development,* vol. 6, no. 12, pp. 431-434, 2019.

[20] P. C. Iloh, “A COMPREHENSIVE AND COMPARATIVE STUDY OF DFS, BFS, AND A* SEARCH ALGORITHMS IN A SOLVING THE MAZE TRANSVERSAL PROBLEM,” *International Journal of Social Sciences and Scientific Studies,* vol. 2, no. 2, pp. 482-490, 2022.

[21] S. Xue, M. Wu, J. Kolen , N. Aghadie and K. A. Zaman, “Dynamic Difficulty Adjustment for Maximized Engagement in Digital Games,” in *International World Wide Web Conferences Steering Committee*, Republic and Canton of Geneva, CHE, 2017.

[22] M. Zohaib, “Dynamic Difficulty Adjustment (DDA) in,” *Advances in Human-Computer Interaction,* vol. 2018, no. 5681652, p. 12, 2018.

[23] C. Pedersen, J. Togelius and G. N. Yannakakis, “Modeling player experience in Super Mario Bros,” in *2009 IEEE Symposium on Computational Intelligence and Games*, Milan, Italy, IEEE, 2009, pp. 132-139.

List of Figures: