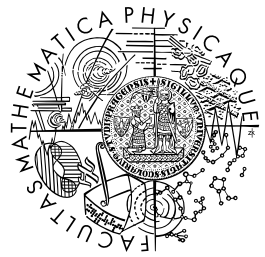


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Anagnoste Marius-Alexandru

# Procedural Generation Of Skill Trees In Video Games Using Graph Grammer

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2020-2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  .....

Author's signature

I would like to give my deepest gratitude to my supervisor Jakub Gemrot who has been a wonderful, patient, helpful and understanding teacher and has placed a lot of trust in me; to my parents, who, despite dealing with hardships of their own, motivated me to finish my studies and my thesis and found the time to check on me every day; to my teachers who were always helpful and have given me a lot of insight in the many fields I have studied while guided by them; to my colleagues who were very helpful, and my closest friends who were always a fountain of motivation, fun times and have been available for drinking beer together. I would also like to thank Solange Petracchi and Kristýna Kysilková, the student coordinators, for helping me with information about the faculty and courses. Additionally, I would like to thank Vojtěch Černý for his course on Procedural Content Generation, and for allowing me to use some files for exporting my results and also Alexis Jacomy, Guillaume Plique and the other contributors for their amazing graph display library SigmaJS, and the authors and contributors to the Gson Java library.

Title: Procedural Generation Of Skill Trees In Video Games Using Graph Grammer

Author: Anagnoste Marius-Alexandru

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: This study investigated the possibility of procedural generation of skill trees which are similar to skill trees in contemporary video games. A set of randomly-selected skill trees from contemporary video games, from different game genres, was compiled, and an analysis was performed to extract relevant observations from the set. Using the observations, models for skill tree generation, and for skill tree comparison were proposed, and they were followed for the generation and analysis of the results. It was found that the method of Graph Grammars provided satisfying results compared to the set of skill trees from video games. Additionally, the other methods researched, L-Systems and Naive Randomized Graph Generation, while both may still require improvements discussed in the thesis in order to provide more satisfying results, they may still be used for particular needs by game designers as they are.

Keywords: game development, game design, procedural content generation, skill trees, graph grammars

# Contents

# Introduction

Procedural generation in video games is a technique of generating elements of video games by using logic and rules, and is becoming more popular for many of the aspects of video games, while the techniques used for content generation are becoming more widely spread and complex. This thesis aims to expand research in the procedural generation of content in video games by spreading the technique to the skill trees element of video games.

Procedural generation can be used for many aspects of games, and also newer elements of video games, and one such feature is skill trees. Skill trees are a video game system frequently used in the Role Playing Games genres, which layout a set of checkpoints to empower the player or to show a hierarchical progress of growth.

While procedural content generation is very popular, using it for skill trees did not meet much research. Procedurally generating skill trees can be achieved through various methods, and one such method is by using graph rewriting on the grounds that skill trees have the form of graphs, often oriented and predominantly acyclic. However the method of graph rewriting is an interesting, challenging and researched subject, consequently it can provide the base knowledge for the generation of skill trees, and this thesis will focus on graph grammars as a method for generating skill trees and also at the best practices for when this method is used. Additionally the thesis will make a comparison of results with other methods, such as a naive randomized generation, and L-Systems. Two simple skill trees which may be able to be generated are displayed in fig. 1


**(a)** Batman Arkham Knight (source: youtube.com)


**(b)** Dead Island Riptide (source: deadisland.fandom.com)

**Figure 1** Two examples of skill trees

# The importance of skill trees

Skill trees are present in an increasing number of game genres and under different forms, however, their resemblance to graphs as well as a few other attributes are maintained, and moreover they have the same goal: to empower the player.

Frequently, skill trees are portrayed as interconnected icons, which the player will unlock sequentially. The player can be seen as a set of attributes that define him, and also a set of actions that are available for him to perform. Unlocking nodes in the skill tree will usually either modify the number of the player's attributes, or give him a new action that is available to be performed or even both of them at the same time. A skill is able to be unlocked only after a requirement is met, such as having unlocked another skill which it is connected to. However, the process of unlocking may still require spending a game specific currency in order for the skill to be unlocked and able to be used. Generally, the term "ability point" is used for this currency.

In the most common skill trees, choosing the empowerment path is not linear. This nonlinearity gives the players limited freedom to choose how they will be empowered. The requirements that need to be met for being granted a specific powerup are usually easily understood from the layout of the skill tree.

Empowering the player incrementally using skill trees will have different advantages over giving them all the powers from the beginning of the game.

The first advantage is given by the fact that, when the player first starts the game, they will have to discover everything the game has to offer. If the game were to give them all the powers and mechanics from the very start, it can become too overwhelming for the player to understand why they would use a spe-

cific skill and they will not become used to everything offered. This may not have been a problem in older games, as their complexity was rather limited, but with the fast growth of the video games medium, the elements which need to be discovered and understood in each game has increased as well. Having the powers unlocked sequentially, the player will already have built a good enough understanding of the game up to that point, that being given the new power will not confuse them, but rather will seem like a natural occurrence. That relates to the second advantage, which is character development.

From a narrative viewpoint, character development is important for the player, as they will bear witness to the growth of their characters over the course of the game. While under the disguise of finishing a part of the story of the game, the character can receive a powerup from the skill tree that will help the story move further. This event can only be seen as natural by the player given the experiences seen through the eyes of the character.

Skill trees add an extra level of replayability to games through the different choices allowed for the players. A skill tree may offer the player more solutions to their in-game encounters. These can add or remove extra mechanics in the game, while also offering trades in advantages and disadvantages for the choices.

As an example, shown in fig. 2, the first "Dishonored" game added a "Blink" teleportation ability which could be upgraded to a level which would give the player access to different paths throughout the game usually not accessible without that ability. While upgrading that ability, the player will have to use their hard-earned ability points in the game to unlock it, but doing so would also mean not unlocking other abilities which could help the player in combat situations. As such, the player would trade combat efficiency for a better level navigation, which will even impose onto the player the sense of need for stealth instead of going guns-blazing into a combat disadvantageous situation.

**(a)** Looking at the rock

**(b)** Looking from next to the rock towards the target area

**(c)** Preparing to use Blink

**(d)** Reached the top of the rock and using the wire to walk to the target area

**Figure 2** Dishonored, using ability Blink to access the target area in a different way

## Why generate skill trees procedurally?

The reasons for procedurally generating skill trees in video games are mostly the same as for the general reasons why procedural generation is used. The main concern is to reduce the time spent on designing a skill tree during the development of a video game. The method of procedurally generating content does not need to reduce the time spent to exactly zero, but it should nonetheless be more time efficient than manual design for it to be practical.

Another important factor is replayability. Given a numbering system that can provide a relationship between skills and the nodes in a generated graph, a game can introduce online procedural generation of skill trees, and generate different skill trees for each new start of a game. This facilitates the replayability of a video game by using the randomness of the generation as the driving factor for playing the game multiple times given the unpredictable skill tree made available. An example of a skill tree which pushes the player towards making decisions based on their desired style of play is present in fig. 3.

A very uncommon use of procedurally generated skill trees would be to have a nearly infinite skill tree. By using online generation while the character grows in their attributes, new nodes of the skill trees will contain additional powerups for the player to choose from. This could provide, in theory, a nearly endless


**Figure 3** Assassin's Creed: Origins; Skill tree contains skills for 3 types of gameplay: Hunter - stealthy and long-range combat, Warrior - close combat fighting, Seer - higher effectiveness of tools, throwables, etc

gameplay experience if foes are also becoming stronger with the passage of time. However this approach may be suitable only for a very few genres of games. The limitations are the practicability of this approach given the methods for expanding a skill tree, as well as the increasing hardware requirements for doing such a task.

# Related Works

A closely related scientific thesis is "“Methods for Procedural Generation of Skill Trees for Computer Games”" [Jar19]. In the thesis mentioned, the subject is focused on a few methods of generating skill trees using evolutionary algorithms and the results are validated using a card game and artificial intelligence to play the game while using the generated skill tree.

# Goals

This thesis aims to achieve the following goals:

1. analyze several skill trees in games, and propose a data structure for generating skill trees,

2. present some of the methods of procedurally generating skill trees, side by side with the research of the practicability of the methods,
3. propose a comparison method for the skill trees; and
4. illustrate the differences in the results.

The main point of focus for the generation methods will be graph grammars.

## Thesis outline

This thesis is structured as follows:

- **Chapter 1** will provide a non-exhaustive list of video games which feature at least one skill tree in their gameplay, each game placed in their primary genre of games, and a few observations made regarding the complexity and layout of the skill trees provided in the list.
- **Chapter 2** will provide a proposal for integrating skill trees in a system of rules, and will present the methodology used for three types of generation, and will present the theory used in the generation steps, as well as possible post processing steps. Additionally, the methodology for the choice of comparison is explained.
- **Chapter 3** will contain the results of the comparisons and a discussion on the results of procedural generation, as well as possible work that can be conducted in the future for expanding the research topic.

# Chapter 1

# Fundamentals

## 1.1 Skill Trees in video games

Games have many mechanics that make the playtime more interesting, and sometimes even rewarding. One such mechanic is skill trees. Skill trees are an important element of many games from many genres, which might be present inside the games under different shapes and names, but they can still be placed under the label of skill trees. The most common are passive trees, talent trees and technology trees. All of these are a progression system which empower the player. Passive trees are practically numerical advantages of already existent abilities of the player's kit, skill trees are the extension of passive trees adding unlockable skills as well, and technology trees are moving the discussion from "What you are" towards "What you have" instead. That means that a technology tree is an external system from the playable character, while the skill tree is a defining element of the playable character.

Following is a non-exhaustive list of video games which feature skill trees or some variant, either a talent tree, hierarchical upgrade system or technology tree. The game genres have been selected from [Mat18] and verified with how popular game shops group their games in genres. The selection of video games in this list is mostly random, each genre was searched and the game that was selected was the first game mentioned in search results which featured skill trees. The games have then been verified to belong the game genre by how they were marketed as by their developers and publishers.

- **Platform games**: Trine 2
- **Shooter games**: Dead Island
- **Fighting games**: Punch Club
- **Beat-em-up games**: Batman Arkham Knight

- **Stealth games**: Assassin's Creed Origins
- **Survival games**: Rimworld
- **Survival Horror games**: How to survive
- **Metroidvania games**: Monster Sanctuary
- **Text adventure games**: Avalon The Legend Lives
- **Action RPG**: Grim Dawn, Final Fantasy X, Salt And Sanctuary, Path Of Exile, Wolcen
- **MMORPG**: Black Desert Online
- **Roguelike**: Rogue Legacy
- **Tactical RPG**: The Dungeon Of Naheulbeuk: The Amulet Of Chaos
- **Sandbox RPG**: Project Zomboid
- **JRPG**: Dragon Star Varnir
- **First-person RPG**: Dark Messiah of Might and Magic
- **IDLE/RPG**: Grim Clicker
- **Life simulation games**: My time at Portia
- **4X games**: Distant Worlds: Universe
- **Artillery games**: ShellShock Live
- **MOBA games**: League Of Legends
- **RTS games**: Spellforce 2
- **Turn Based Strategy Games**: Sid Meyer's Civilization IV
- **Turn Based Tactics games**: XCOM 2
- **Wargame games: Mechs & Mercs**: Black Talons
- **Racing games**: F1 2017
- **Sports games**: NBA 2k20
- **Competitive games**: World of Tanks

Each of the specified video games will have attached an image displaying the skill tree which was considered to be observed for this thesis.

A few genres for which the research did not return results are Visual novels games, Interactive Movie Games, Real-time 3D adventure games, Rhythm games, Autobattler games, and Battle Royale games.

## 1.2 The complexity of skill trees

Skill trees differ largely from one game to another. Under normal observation, it is difficult to find similarities. However it can be observed that there exists a tendency with the complexity of skill trees in video games.

The complexity can be defined as the difficulty with which a player creates the best available set of unlocked skills to maximize one or many of his attributes.

Skill trees can be seen as deterministic problems that need to be solved. Given the deterministic nature of the problem, we know that finding the perfect strategy is possible. One very good example is the multiplayer game Path of Exile, where, despite having huge skill trees, there are players who calculate which sets of unlocked skills offer the most amount of "Damage Per Second" (DPS) to the player. Detailed information for this particular game and its skill tree can be found at [mem17, poe.ninja]. This already indicates that complexity will not make skill trees non-deterministic. However, games are not usually decided by just one attribute of the player. Consequently, solving for multiple attributes at once can be a non-deterministic problem, when we do not have a clear relationship between the attributes. In Path Of Exile, the player can determine the best way to maximize DPS, but they can not also maximize the character‘s life points with the same set of unlocked skills. Players do not know which of the two attributes is the best choice for them. Maybe none is, and a balance must be found. This is actually better determined by the proficiency with which the player plays the game. If the player is good at dodging enemy attacks, life points can be traded for DPS, and the opposite is true as well. It can be assumed that with higher complexity of the skill tree, the deterministic nature of the skill tree is partly avoided.

By closer inspection into the games mentioned in the list prior, it is possible to make an assumption on how the complexity changes for each game. The complexity of a skill tree is directly proportional to the number of mechanics in the game, types of enemies, and the game genre.

Out of all genres of video games, games under the Role Playing Game genre (RPG) and its many variations, are the main beneficiary for the inclusion of a skill tree, followed closely behind by the genres which usually feature a base-building system or element upgrades system, such as Real Time Strategy (RTS), Turn Based Strategy(TBS), and Survival genres. A perfect ranking may not be conducted as games can belong to multiple genres at the same time, and moreover, researching every game ever created is not feasible.

Mechanics in the game allow the player to perform a different number of actions. These actions will have some characteristics assigned to them. However, since these characteristics are subject to change, we can assume that a skill tree can involve these characteristics. One example is jumping in a platformer game.

The height of the jump could be changed through the skill tree so that the player can navigate vertical areas faster. Each addition of an additional mechanic in the video, may add the desire of improving that mechanic, just like instead of only jumping higher, the player may desire to be able to jump twice, the second jump while being in the air after the first jump.

The number of different enemies or their types in a game can also increase the complexity of a skill tree. Each enemy can bring different inconsistencies to the damage output of the player through resistances to types of damages done by the player. This could also be a sub-category of the mechanics in the game, but usually is to be treated separately by video game communities trying to solve the deterministic skill trees. An example can be found in the game "Spiral Knights", where a new level of the dungeon may shelter enemies which are not affected by swords with an elemental effect such as fire, but may instead take additional damage if hit with a sword that has a freezing effect. An example is in fig. 1.1 where a room in a level contains enemies with many different effects.


**Figure 1.1** Spiral Knights; Enemies and traps have different effects when damaging the player, such as Stun, Fire, Shocked, Poisoned

These are definitely not the only factors for changes in complexity of skill trees, but they definitely play an important role. To classify the complexity of skill trees, an analysis of skill trees is required and a method for classification needs to be proposed.

# Chapter 2

# Methodology

## 2.1 Skill trees as graphs

As all the other content that can be procedurally generated, skill trees require some structure for them to be procedurally generated. Having detailed the informal definition of skill trees, it is necessary to provide a formal definition as well to include the skill trees in the content that can be procedurally generated.

Skill trees do not have a visible set of rules that bind them all together, but this does not imply that there can not exist a proposal for a set of rules that will include most of them. Given the list of games that feature skill trees in Chapter 2, we will observe possible rules and make the rules based on the observations.

From the observations of the games in the list, we extract the following:

1. The drawing layout of the skill trees is frequently planar when the skill tree is not a technology tree;
2. Technology trees are frequently non-planar;
3. Some skill trees are wrapped in some particular shapes to be given meaning or to display what the skills are relevant to;
4. Some skill trees have a layout which looks like a result of a mirroring half of the skill tree;
5. The skill tree itself may look like it is composed by many smaller and simpler skill trees;
6. Unlocking a skill can be made possible either by unlocking a neighbouring skill, or by events related to other elements of the game environment;
7. Unlocking a skill does not come after unlocking an effect for that skill;

A detailed list of observations can be found in the the table of results attached to this thesis, in the "Games_Classification" Sheet.

To provide this definition, it is mandatory to define the events of the game environment first. The game environment represents the space of all the elements available in the game. And an event from the game environment is a specific state of an element at a given time.

Most often, skill trees in video games allow the player to choose in a sequential style gameplay elements that are connected by an edge with another node which was unlocked. The skill desired to be unlocked can either require the previous unlocking of one or more other nodes which it is connected to (hierarchical order), or requiring only one of them to be unlocked, or even have an external requirement not immediately related to the skill tree. The method of unlocking skill in an order can be viewed as creating a path through the skill tree such that the player will unlock their desired skills.

From the general layout of skill trees, in addition to the observations made, we can immediately observe a similarity with the definition of graphs. A graph is a 2-tuple (V, E) where V=set of nodes, and E=set of edges $\{(a, b)|a, b \in V\}$.

Therefore, I make the proposal that a skill tree is an extension of a graph with the addition of a set of rules for visiting certain nodes, while starting from one or more specified nodes. The nodes are elements of gameplay which require to be visited to be able to be used during the game.

The proposed definition is as follows:

A skill tree can be defined as a 5-tuple $K = (S, E, R, \phi, T)$, with

- S = set of nodes each usually representing an unlockable element of gameplay or attribute gain;
- E = set of edges representing either a hierarchical order of the nodes or a relation of neighbourhood between two nodes, $\{(a, b)|a, b \in S\}$;
- R=set of requirements, $\{r|r \in R\}$,
- $\phi$=set of relations between nodes and requirements; for each node a, $\phi_a$ is the set of requirements for node a, $\phi_a = \{(a, \omega), a \in S, \omega \in R\}$
- T=set of starting nodes.

In this scope, we may split the procedural generation of a skill tree in the following steps: Generation of the graph-like structure of the skill tree, the 2-tuple (S, E); Assignment of skills to the generated graph-like structure; Definition and assignment of additional requirements to the skills, generation of R and $\phi$; Selection of the set of starting nodes, T.

This thesis will focus exclusively on the first step.

To achieve the first step, the generation of particular types of graphs is necessary. Not just any graph should be seen as suitable to be the basis of a skill

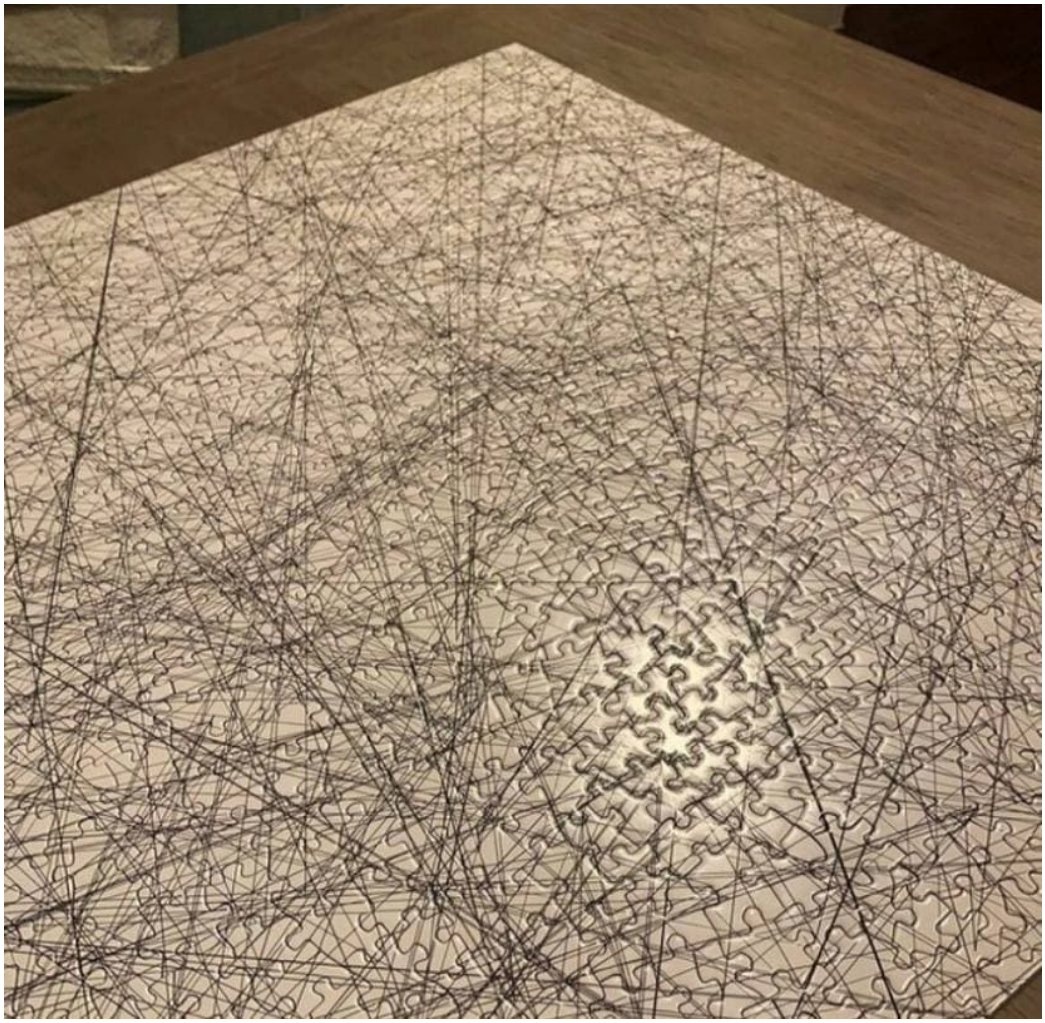
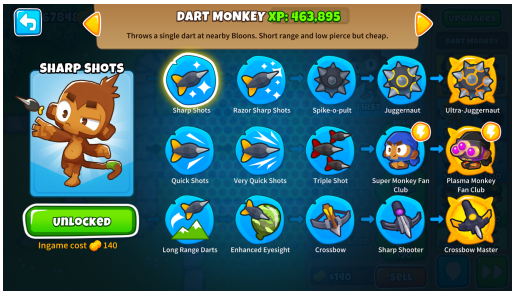


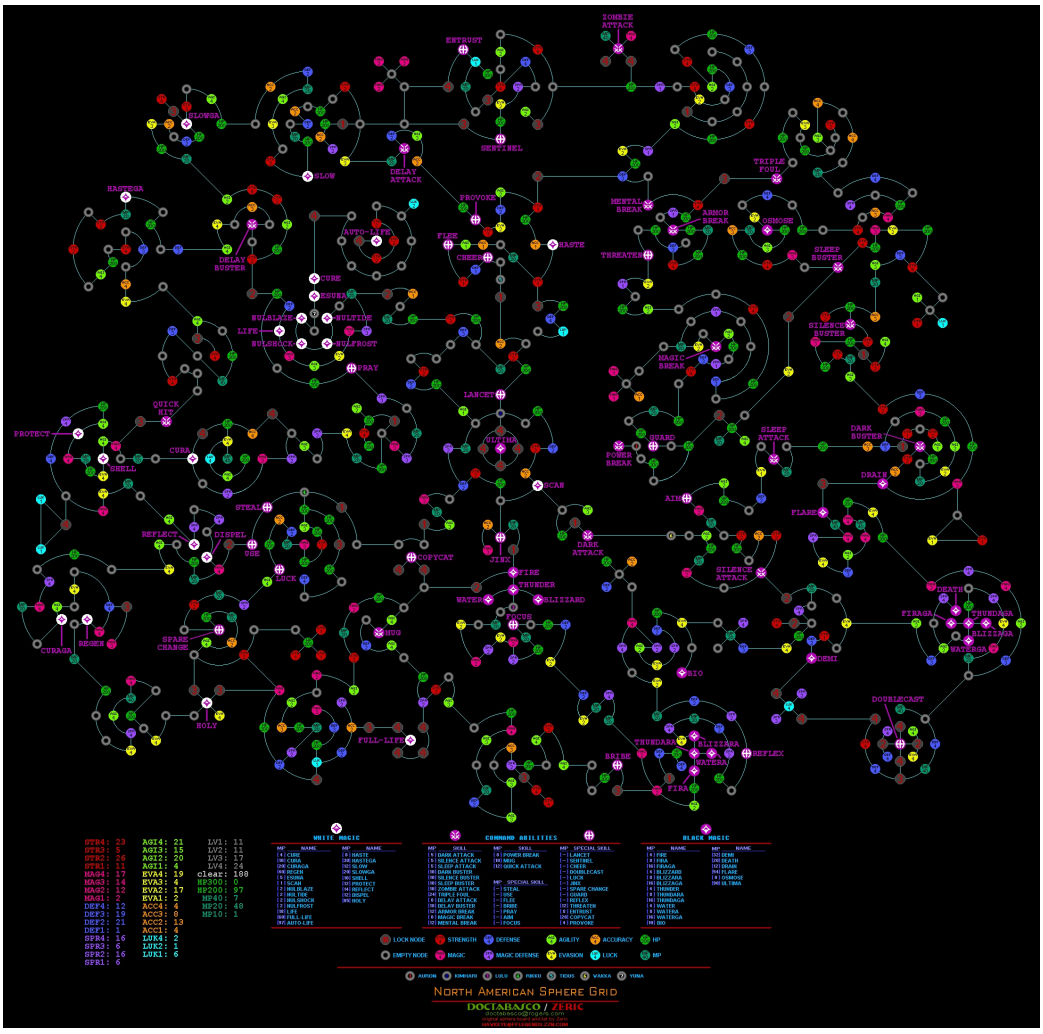
**Figure 2.1** Exceptionally complicated puzzle;

tree. Skill trees usually keep a smaller amount of intricacy, and should not be as difficult to solve as a puzzle with 1.000 lines drawn randomly across it, like in figure 2.1. One such example is a complete graph; its property of having all nodes interconnected with each other creates a redundancy of the connections, since the graph could be reduced to a graph with no edges, and in the resulting skill tree, all nodes will be contained in the set of starting nodes. The generated graphs must have properties similar to the graph-like element of skill trees from the video games in the list provided prior.




**(a)** Bloons TowerDefense; Simple skill tree, with linear skill progression


**(b)** NBA 2k20; A medium complexity skill tree, small amount of skills, but many interconnections




**(c)** Final Fantasy X; A complex and intricate skill tree

For the procedural generation, the following methods will be presented:

1. Naive Randomized Generation of Graphs;
2. L-Systems;
3. Graph Grammars.

## 2.2 Naive randomized generation

Randomized graph generation can be seen as an iterative choice between two simple operations:

- adding a node,
- and adding an edge that connects two nodes.

It is a certainty that, when an undirected graph which contains no multi edges and has less than 2 nodes or is already a complete graph, adding another edge to this graph is impossible. As a consequence, an ordered set of these two operations must have a ratio between their appearances in the set so that a correct graph is generated.

A very naive method of proposing a relationship between the two operations is by expressing a ratio of probability of choosing one operation over the other. For further simplicity, we will represent these probabilities in percentages.

Let:

- $X_i$ = chance for the "add node" operation at iteration i
- $Y_i$ = chance for the "add edge" operation at iteration i = 100% - $X_i$
- n = maximum number of iterations

$X_0$ and $X_n$ will be the maximum and minimum percentage chance for the "add node" operation. We will express the in-between elements based on the number of the current iteration i, the total number of iterations n and the percentages $X_0$ and $X_n$. A simple relationship between the variables can be expressed easily by using mathematical functions. The relationship can be seen by drawing $X_0$ and $X_n$ on a coordinate system, with their respective values on the Y axis, and the iteration number on the X axis. For providing the result, the two points need now to be connected by a line or by a curve. As such we can propose the first degree and the second degree polynomials as the choice of functions. Other functions could be used as well.

First degree polynomial is the function $f(x) = a \times x + b$. To solve for x, we will look at the percentages $X_0$, $X_n$, the current iteration i and the . We know

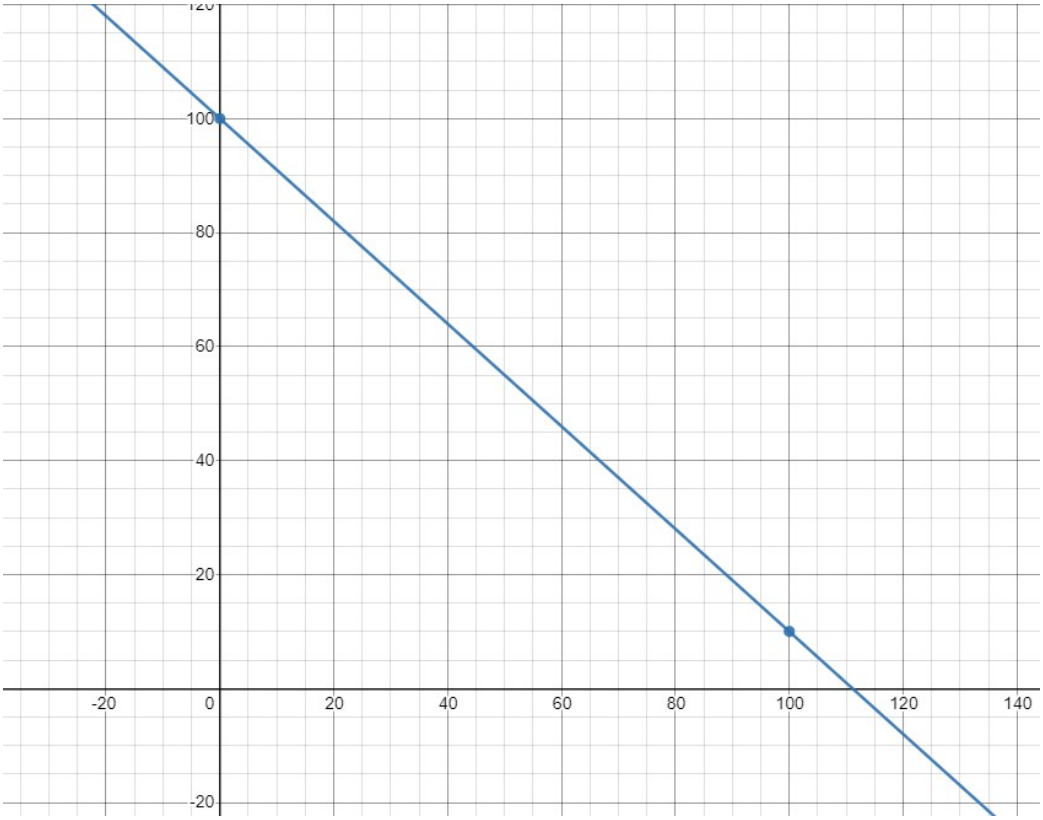



**Figure 2.3** Linear Easing Function for 100 to 10

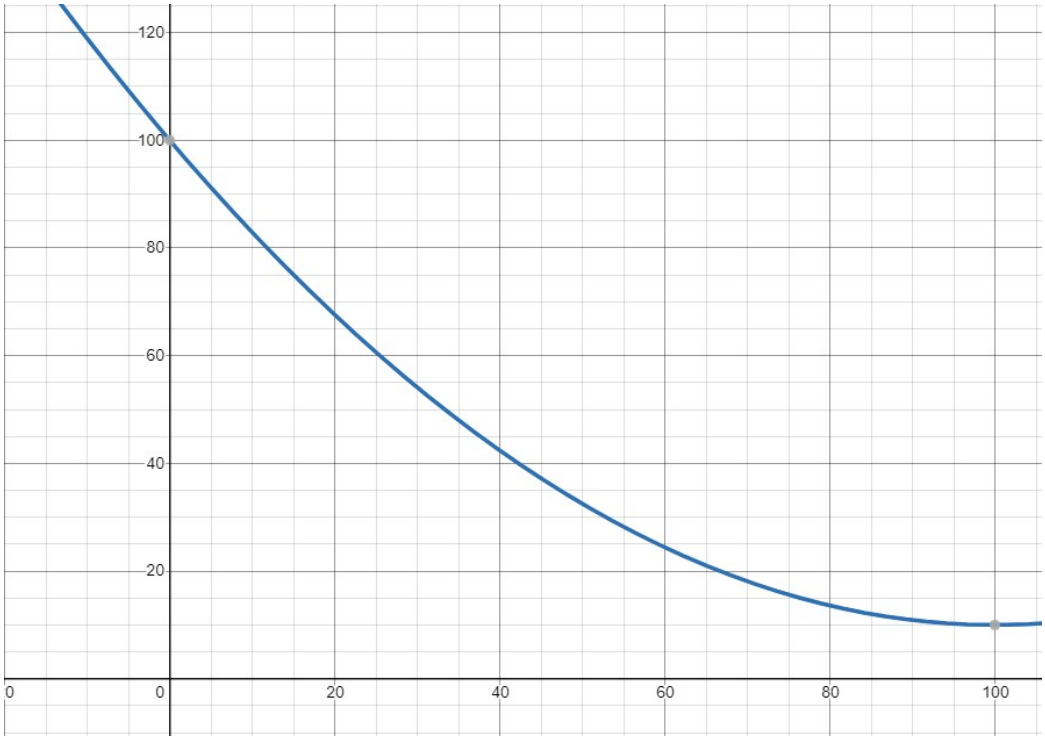



**Figure 2.4** Quadratic Easing Function for 100 to 10

that points (n, $X_n$) and (0, $X_0$) are on the graph. Making a system for the two points, it will give us $a = \frac{X_n - X_0}{n}$, and $b = X_n$.

And the second degree polynomial is given by $g(x) = a \times x^2 + b \times x + c$. To simplify the operations, we will write the function in the vertex form $g(x) = a \times (x - h)^2 + k$. This can easily tell us that the vertex point, which is the minimum (or maximum) point of the function, is (h, k). Using our variables, (h, k) is actually (n, $X_n$). We solve for $a$ when expressing the other point, and we get $a = \frac{X_0 - X_n}{(-n)^2}$.

Using such a function, and a random number generator, we can now generate graphs by having the random number generator make the choice between the two operations, and also by choosing the two nodes which an edge will connect. For correctness, we will have some pre-verification if the edge added already has its two nodes connected by another edge, which would require another choice of nodes, which situation can be repeated until it occurs a specified maximum number of times. We will also have a pre-verification for the completeness of the graph. If any of these events occur, the "add node" operation will be chosen instead.

**Algorithm 1** Generate randomized graph

$initialization$
$addNode()$
**for** $iterationStep = 0$ to $n$ **do**
    ratio = $easingFunction($ $iterationStep$, $initialChanceForAddNode$, $finalChanceForAddNode$ )
    choice = $random()$
    **if** choice $>$ ratio AND $graphNotComplete()$ **then**
        $addEdge()$
    **else**
        $addNode()$
    **end if**
**end for**

Skill trees generated using this method were exported to JSON files using the following variables separated by an underscore:

- adjective = a randomly chosen adjective from a file containing a list of adjectives
- noun = a randomly chosen noun from a file containing a list of adjectives
- startingAddNodePercentage = initial chance for the add_node operation
- minimumAddNodePercentage = final chance for add_node operation
- iterations = number of iterations
- easing = easing function chosen
- seed = random number generator's seed

## 2.3 Generation using L-Systems

L-Systems are a formal grammar rewriting system. Starting from an initial "axiom", a set of production rules are applied to the symbols of the formula. The definition is given in the scientific book [PL12, Let V denote an alphabet, $V^*$ the set of all words over V, and $V^+$ the set of all nonempty words over V. A string OL-system is an ordered triplet $G = (V, \omega, P)$ where V is the alphabet of the system, $\omega \in V^+$ is a nonempty word called the axiom and $P \subset V \times V^*$ is a finite set of productions.].

The book [PL12] additionally provides a technique for drawing the result onto a coordinate system. The method proposes that replaceable symbols represent drawings of lines, and non-replaceable symbols represent changes to the angle of the next drawing (e.g. “+”, “-“), or the location from which the drawing is resumed (e.g. “[“, “]”).

Expanding from this method to draw a graph, we will represent the replaceable symbols by lines with nodes at the end of the lines, with an initial node at the origin of the drawing.

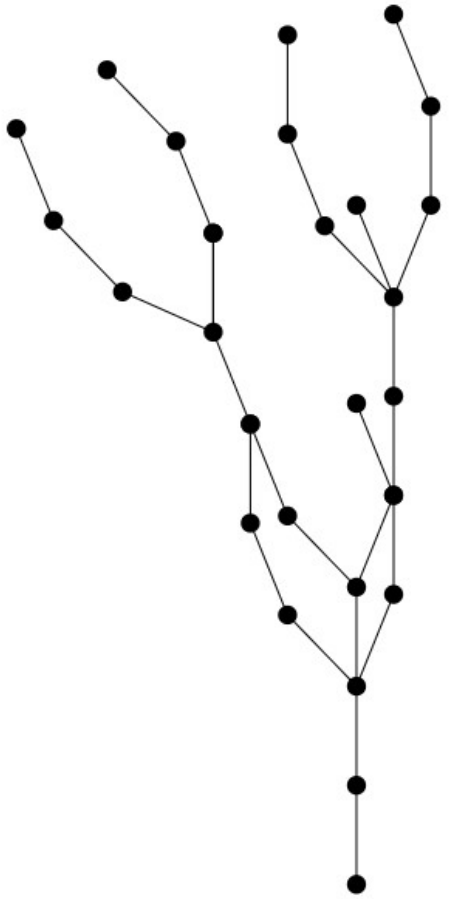


**Figure 2.5** An L-System drawn as a graph

Depending on our approach on rewriting, this thesis will employ two different types of L-Systems:

1. Parallel Rewriting L-Systems,
2. Stochastic L-Systems.

## 2.4 Parallel Rewriting L-Systems

Parallel Rewriting L-Systems are the main type of L-Systems, and the most common. For this type of L-System, there is only one production rule for each replaceable symbol, and the rewriting of symbols in one iteration happens for all replaceable symbols. This approach tends to produce fractals, but many sets of productions can produce structures resembling trees. Extensive research on this resemblance was conducted by Lindenmeyer in his book[PL12] . Additionally in his book, a few particular L-Systems are proposed and drawn. Using these L-Systems and additional ones proposed, graphs will be generated and analyzed.

This method has the drawback of creating graphs that have a high amount of repetitive subgraphs. Consequently, an additional variant of L-Systems is explored.

## 2.5 Stochastic L-Systems

Stochastic L-Systems differ from their Parallel counterpart by introducing elements of randomness during the rewriting. There can be more than one production rule for each replaceable symbol, and the production used will be randomly chosen.

To choose from the available productions, at each symbol a random production will be chosen from the available ones by a method similar to throwing a dice with the same number of faces as the sum of weights of all productions for that specific symbol.

Given the set of productions P = {(S, W) | S = production, W = weight}, the dice which chooses the production will have $\sum W_i$ faces with $(S_i, W_i) \in P$ .

Stochastic L-Systems will remove some of the similarity of the parts of the structure, but not entirely. Although it can be reduced even further by the addition of an ample set of symbols.

For the generation, 8 L-Systems from three sources, [Roa12], [Jen17], [PL12], have been used, in addition to a random L-System rule generator, which generates a rule using the symbols "F", "+", "-" and "[" in random positions, while the last symbol requires the additional symbol "]" to be placed in a random position after the previous symbol's position.

Files containing L-Systems generated using these methods were included the seed for the random number generator, and minimum and maximum iterations, which, together with the random number generator will give the iteration number.

## 2.6 Generation using graph grammars

Graph Grammars, known also as graph rewriting, is a method of modifying an original graph using an algorithmic approach. [] Similarly to L-Systems, it is an application of grammars from the discipline Formal language theory. This particular application manages graphs instead of strings. The productions of grammars are replaced by graph rewriting rules. Graph grammars have a sizeable list of fields and subjects in which they are found useful, a few of these discussed in [Nag79].

[Dor+95, Graph Rewriting rule is the 3-tuple $r = (g_l, g_r, M)$. The graph $g_l$ is the left-hand side, $g_r$ is the right-hand side, and M is the set of embedding descriptions of the rule.]

Additional definitions and algorithms for the graph rewriting subject can be found in [GK20], and [SHW19].

This thesis will propose an non-exhaustive list of graph rewriting rules, which, used together, will provide a very good chance at generating skill trees similar to the ones in contemporary video games.

Dorr et al. explained in their paper how graph rewriting is reliant on the sub-graph isomorphism problem. The sub-graph isomorphism problem is an NP-complete problem. A proof for NP-completeness is included in the paper Cook [Coo71]. [Dor+95, "Consequently, applications of graph rewriting systems are very rare", Chapter 1.2].

Graph Grammar is a set of graph rewriting rules R, applied on a graph G.

In this thesis, graph rewriting for graph G and a set of graph rewriting rules R will be solved by following the attached procedure:

1. Graph Rewriting rules are grouped by the number of nodes N of $g_l$, the groups and the rules contained are then shuffled;
2. For each group, extract all connected subgraphs of N nodes from the graph;
3. For each extracted subgraph, check for an isomorphism H between the subgraph and the $g_l$ of each graph rewriting rule from the group;
4. When a isomorphism H is found, use the isomorphism and the set of embedding descriptions M to rewrite the graph, and stop the procedure.

The complexity of this procedure is exponential. This emphasizes the need for graph rewriting rules that have a small amount of nodes in the left-hand side $g_l$.

In order to propose a list of graph rewriting rules that could be used together to generate skills trees very similar to the ones featured in contemporary video

games, the generation should follow these guidelines compiled by using the information presented thus far in this thesis:

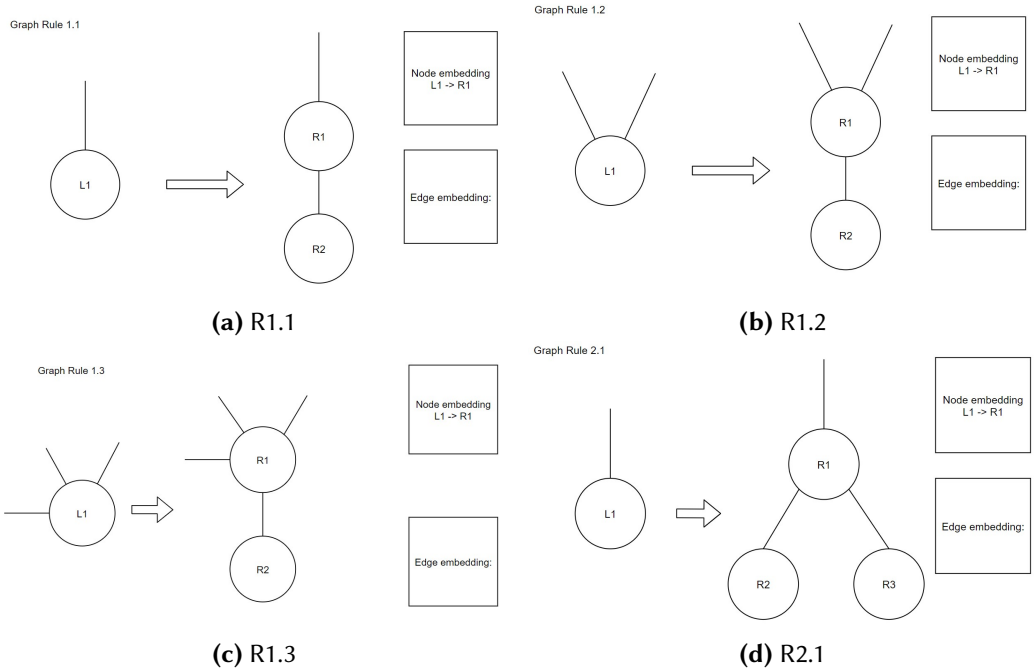
1. $g_l$ of graph rewriting rules must be small, required by the exponential complexity of the algorithm;

2. continuous application of graph rewriting rules must be possible, as it is a pattern of procedural generation;

3. original graph should be kept simple, required by the need of time saving by the game designer;

4. the graph should be kept planar, required by the observations of skill trees in the list of video games provided;

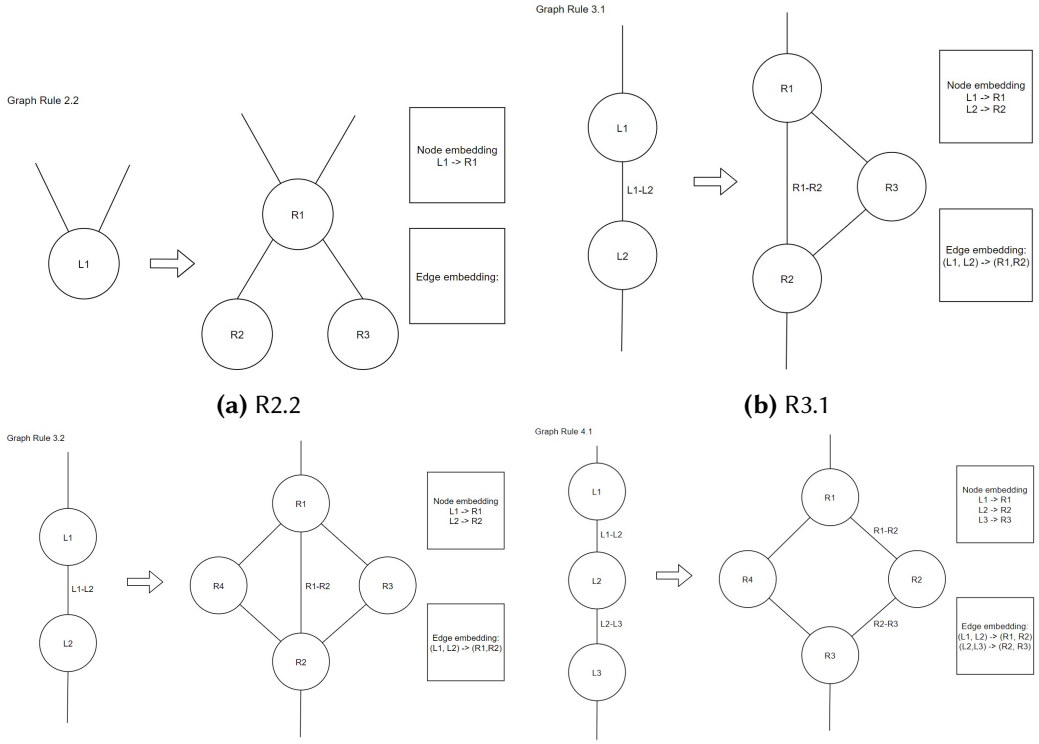
5. creation of a complete graph should be avoided, required to oppose redundancy of skill trees.




**(a)** R1.1 **(b)** R1.2

**(c)** R1.3 **(d)** R2.1

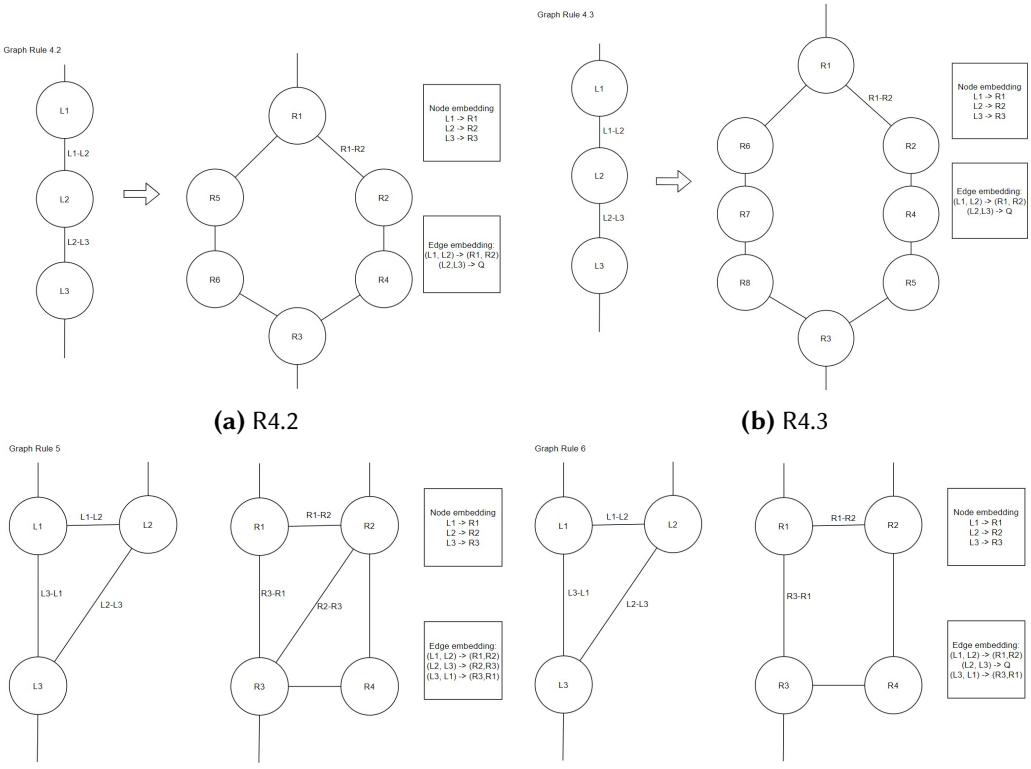
**Figure 2.6** R1.1, R1.2, R1.3, R2.1




**(a)** R2.2 **(b)** R3.1

**(c)** R3.2 **(d)** R4.1

**Figure 2.7** R2.2, R3.1, R3.2, R4.1




**(a)** R4.2 **(b)** R4.3

**(c)** R5 **(d)** R6

**Figure 2.8** R4.2, R4.3, R5, R6

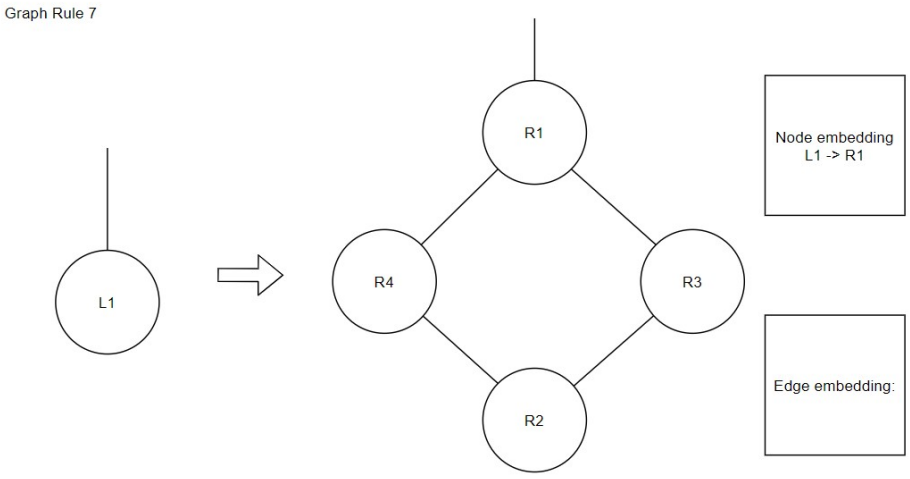



**Figure 2.9** R7

A number of 13 graph rewriting rules are proposed to meet the guidelines.

They are presented in fig. 2.6, fig. 2.7, fig. 2.8, and fig. 2.9.

The rewriting rules have been compiled with the goal of meeting the previous observations, in addition to simulating a few subgraph structures met in the graphs of the list of skill trees from the video games provided in Chapter 2.

1. To satisfy the first guideline, the left-hand graph has been kept below 4 (a very small number) nodes;

2. For the second guideline, all rules add or create a structure which is isomorphism with the left-side graph of another rule. Additionally, no rules that remove nodes have been proposed, for the reason that, given a sequence of applications of such a rule may produce a graph for which no rule proposed can be applied;

3. The third guideline is achieved by starting only from two possible simple graphs of three nodes each;

4. The fourth guideline is carried out by having the initial graph and the proposed rules not allow the creation of non-planar structures;

5. The final guideline is achieved by the rules for the reason that applying any of the applicable rules to any graph with at least 3 nodes would not produce a complete graph.

## 2.7 Post-Processing step

In procedural content generation, it is not uncommon to include pre-processing and/or post-processing steps to simulate the creation of content which has a higher resemblance to the already existing content.

For skill tree generation, by observing the layout of some of the skill trees in the video games list provided in Chapter 2, we can extract some important piece of information:

- Skill trees are usually included in some figure or pattern;

- Skill trees may be formed out of smaller skill trees connected between them by a few edges;

- Skill trees may have a mirrored layout.

From these observations, we can consider some possible post-processing steps:

1. The rearrangement of the generated skill tree in a figure;
2. The generation of other small skill trees that will be connected;
3. The mirroring of the generated skill tree.

This thesis does not include the post-processing step in its results section, and only employs the method of mirroring as a showcase for a higher resemblance to skill trees in contemporary video games after the post-processing step. The mirroring is performed by random selection two consecutive nodes from the convex hull after a convex hull algorithm [Far12] is used to find the nodes on the convex hull. All other nodes are mirrored using the two selected nodes to form the line around which the mirroring is executed.

**Figure 2.10** Wolcen; A skill tree which contains several interconnected graph-like structures, each seemingly mirrored

## 2.8 Comparison methodology

The method of comparison will differ from the one used in ""Methods for Procedural Generation of Skill Trees for Computer Games"" [Jar19], instead of the in-game validation, this thesis will use a validation that will pay close attention to the complexity and the possible layout of the generated skill tree, while the second term of comparison will be the skill trees which are being used in some popular contemporary games.

Comparison will be performed by using the complexity of skill trees. The complexity of the skill tree will be given by two main factors. First will be the size of the skill tree and the second will be the difficulty of solving the skill tree, given by the number of possible paths that can be taken to reach a node. These two will be expressed by the number of nodes and cycles in the graph-like structure of the skill tree.

Additionally, the layout of the skill tree will be taken into account for the comparison. However, because of the complete subjectivity of such a comparison, the planarity of the graph-like structure will be the main point of focus for the comparison of layout.

For simplicity, the following points awarding system has been proposed for the classification of the skill trees based on their size, cycles and planarity.

- 1 Node = 1 point;
- 1 Cycle of any length = 5 points;
- Non-Planarity = 20 points;

For classification, the following criteria has been proposed.

- under 35 points regards the skill tree as possessing "Simple" complexity;
- over 35 and below 80 points regards the skill tree as possessing "Medium" complexity;
- over 80 points will regard the skill tree as "Complex".

This numerical system has been chosen for the reason that it spreads the skill trees in the video games from the list provided in Chapter 2 into three even groups. The random choice of the video games in the list should allow for an even distribution into the three categories. This system may provide an objective classification of the generated skill trees just as well.

| | |
|---|---|
| No. of Simple Skill Trees | 11 |
| No. Of Medium Skill Trees | 10 |
| No. of Complex Skill Trees | 12 |
| | |
| No. of Simple Planar Skill Trees | 11 |
| No. of Medium Planar Skill Trees | 8 |
| No. of Complex Planar Skill Trees | 10 |

**Figure 2.11** The summary of classification for the skill trees of games provided in Chapter 2

# Chapter 3

# Results and discussion

The generated results were grouped by the method of generation used. The files in each group were added to an external web tool, made in Javascript and PHP with the help of the graph drawing library SigmaJS [JP17] for ease of access. Then, the results were written down in a table that associates each file with the properties of the generated graph. The properties were then quantified as suggested in the proposed comparison method to obtain a score for the graph and the graph was classified based on the score. Tables with the summary of classification of results were created and completed for each generation method.

A detailed list of each result can be found in the the table of results attached to this thesis, in the method's own Sheet.

## 3.1 Results of Naive Randomized Generation

| | | |
|---|---|---|
| No. of Simple Complexity generations | 2 | 4% |
| No. of Medium Complexity generations | 38 | 76% |
| No. of High Complexity generations | 10 | 20% |
| | | |
| No. of Planar generations | 3 | 6% |
| No. of Non-Planar generation | 47 | 94% |
| | | |
| No. of Simple Linear generations | 2 | |
| No. of Medium Linear generations | 14 | |
| No. of Complex Linear Generations | 7 | |
| | | |
| No. of Simple Quadratic generations | 0 | |
| No. of Medium Quadratic Generations | 24 | |
| No. of Complex Quadratic Generations | 1 | |

**Figure 3.1** Summary of generated graphs using Randomized Graph Generation

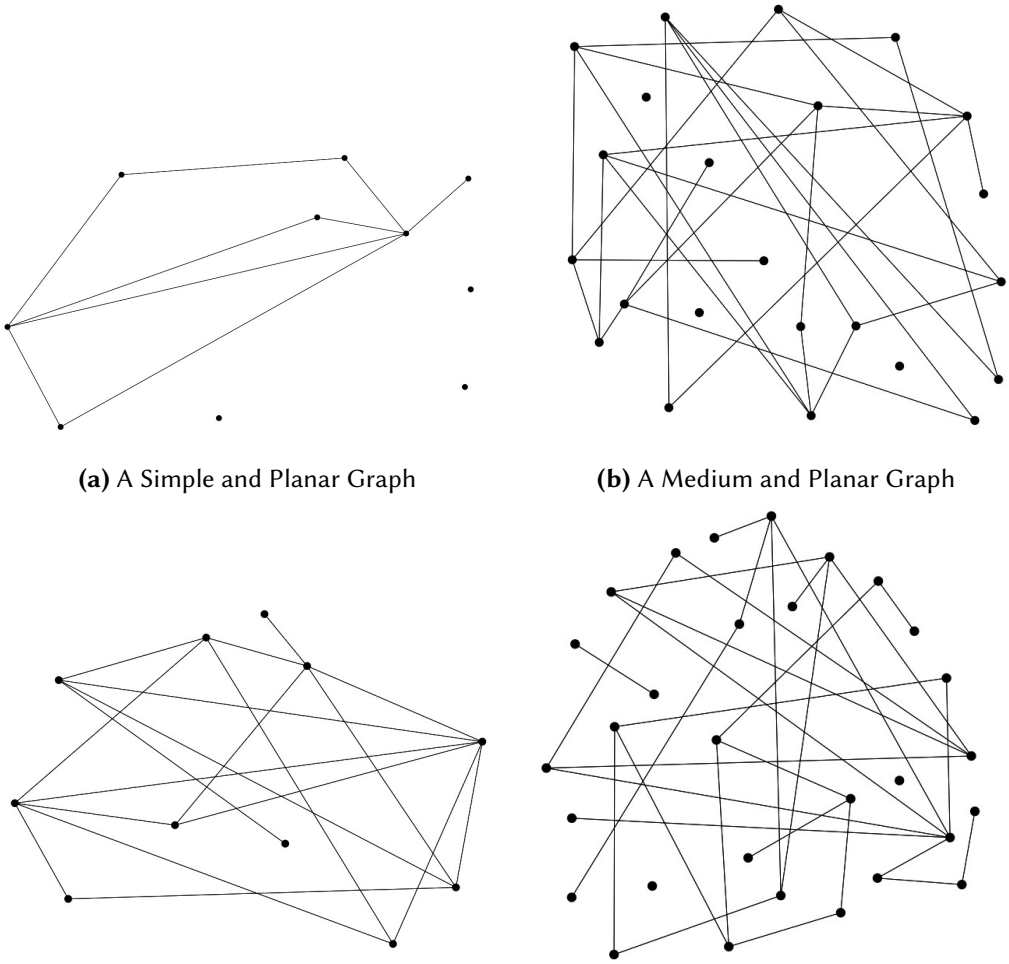
The current method generated results with varying complexities very fast. However, the graphs generated were predominantly non-planar, and have a tendency to have higher complexity, summarized in fig. 3.1, which is the opposite

of the desired property, according to the observations made on the skill trees of the video games.

This method provides very little control over the results. The easing function is the only input, except for the seed, the game designer can modify in order to get a result. Finding a desired result will depend much more on the seed provided to the random number generator.

The time complexity of this method is linear, which can be deduced from the provided pseudocode in Chapter 2.

This method may still serve a purpose for game designers who want an initial template generated very fast, which they will model themselves afterwards. Outside of this use-case, the method provides no further benefits, since, with few exceptions, they are very different from the skill trees in the video games list provided in Chapter 1



**(a)** A Simple and Planar Graph

**(b)** A Medium and Planar Graph

**(c)** A Medium and Non-Planar graph

**(d)** A Complex and Non-Planar Graph

**Figure 3.2** Graphs generated using RGG

## 3.2 Results of L-Systems

| | |
|---|---|
| No. of Simple Generations | 24 |
| No. of Medium Generations | 6 |
| No. of Complex Generations | 21 |

**Figure 3.3** Summary of generated graphs using L-Systems

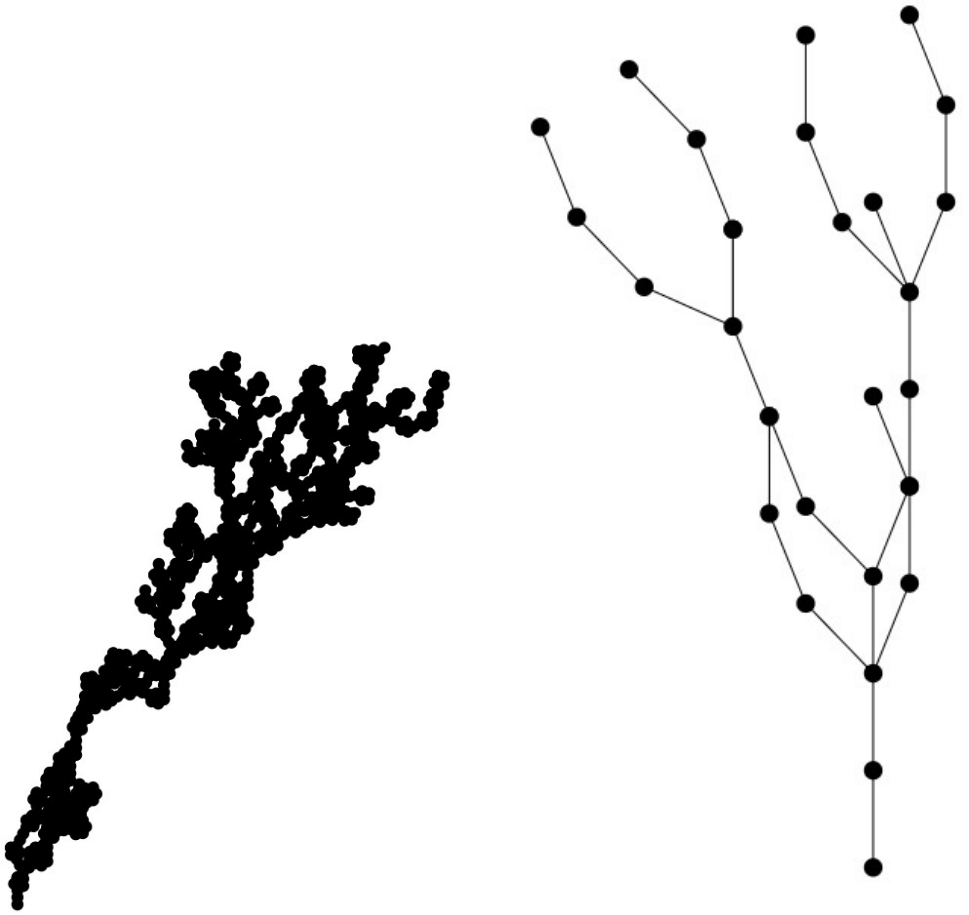
The results, summarized in fig. 3.3, drawn using the very simple extension provided to the drawing method proposed in *The algorithmic beauty of*

*plants*[PL12] , provides planar graph structures with no cycles.

The results are not evenly distributed, but for this method of generation, each iterative step has the chance of doubling the complexity. As such, a strong bias is created towards simple and complex graph generations. This bias is reduced slightly by the stochastic L-Systems, but it still has the same exponential increase potential of generation in the worst cases.

The repetitive nature of the L-Systems make the results look unoriginal and banal. This is not necessarily a drawback, but is instead an interesting attribute.

The planarity and no-cycles attributes, in addition to the repetitive nature and banal layout are found predominantly in technology trees, but seldom in the other types of skill trees. This already makes L-Systems a good generation method for the technology tree variant of skill trees. Furthermore, given an application for drawing and modifying graphs, the time spent by the game designer modeling the graph in order to turn it into a desired final product is reduced greatly.



**(a)** A very big L-System generated graph **(b)** A Simple L-System generated Graph

**Figure 3.4** Graphs generated using RGG

## 3.3 Results of Graph Grammars

| | |
|---|---|
| No. of Simple graph generations | 11 |
| No. of Medium graph generations | 19 |
| No. of Complex graph generations | 21 |

**Figure 3.5** Summary of generated graphs using Graph Grammars

Graph Grammars managed to generate graphs with a classification distribution function that is evenly inclined towards medium and high complexity graphs, as can be observed from fig. 3.5. These results may reveal either the

general tendency of the method itself, or a bias created by the choice of graph rewriting rules. In case a bias was created due to the choice of the rules, the specific origins for bias must be found. And to find these origins of bias, the choice of graph rewriting rules and their average contribution to the graph rewriting method must be analyzed.

- Firstly, none of the proposed rules removes nodes. This proves the tendency of the graph's complexity to either rise or stay the same after each graph rewriting rule application. A rule that removes nodes was avoided due to the guidelines proposed in Chapter 2;

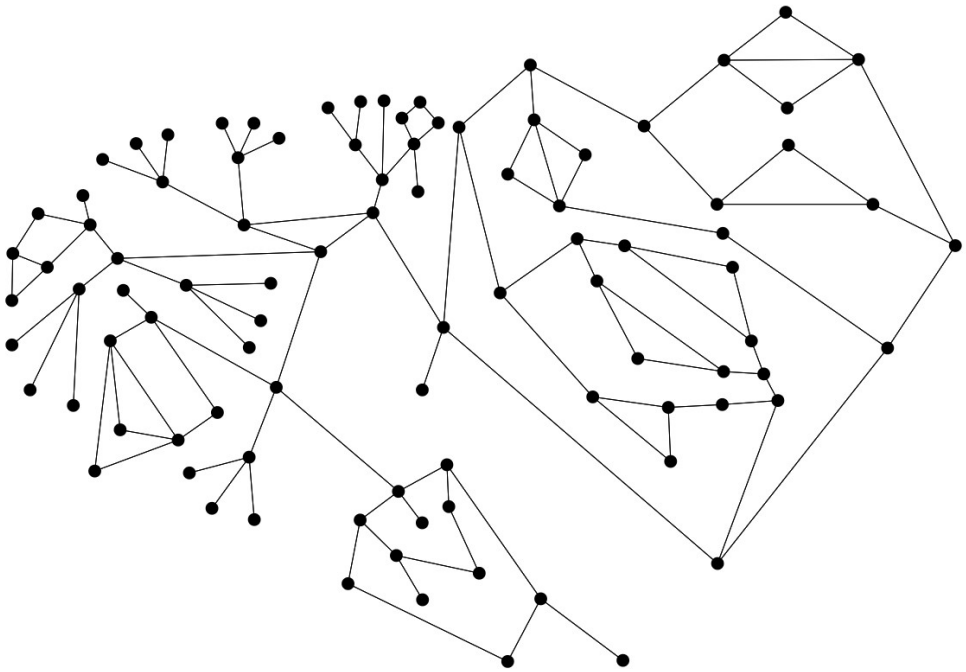
- Secondly, all rules add at least one node. Together with the first observation, it proves that there is a bias towards expanding the complexity at each step. The choice of not including rules that do not add nodes and instead add edges would not comply with the guideline for keeping the generated graph planar and incomplete.

- Finally, only a quarter of the rules do not add cycles, which explains the tendency of creating graphs with higher complexity.

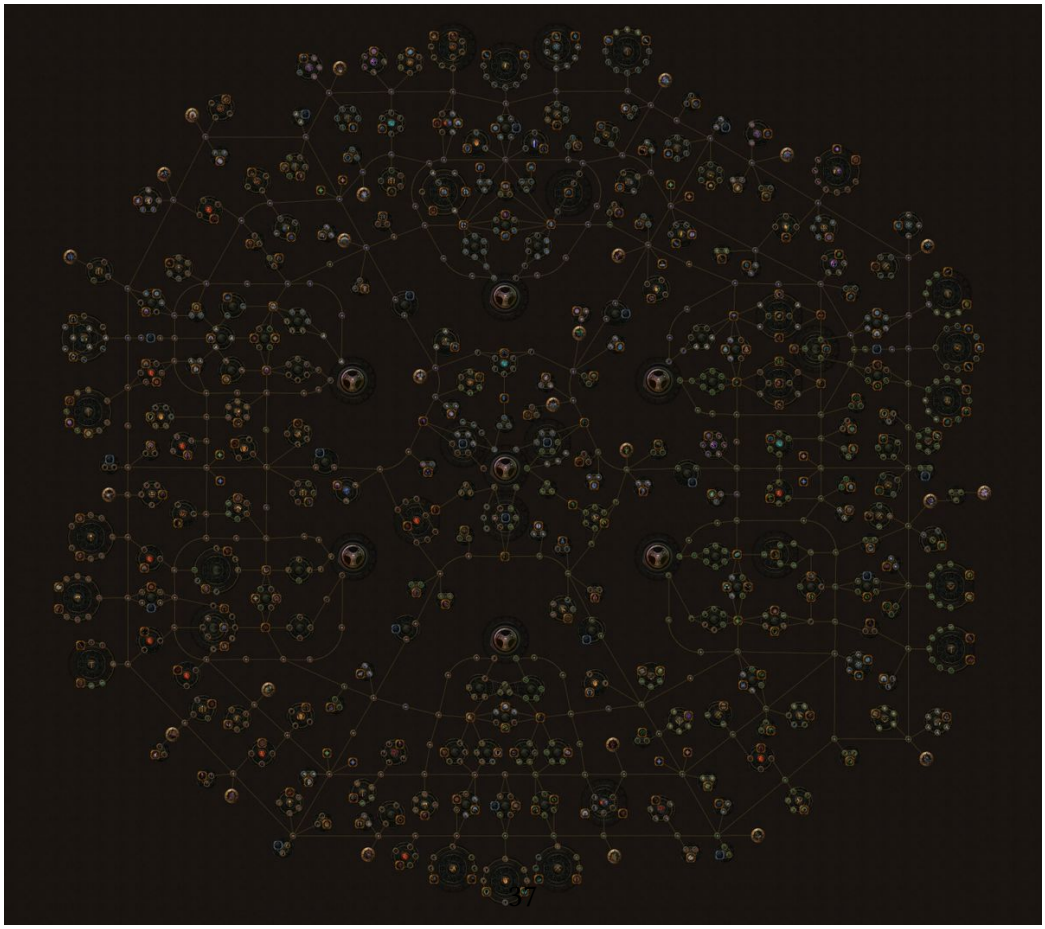
Correcting for these origins of bias, will make the generated results able to be classified more evenly. To correct for these origins of bias, future generation of graphs using the proposed graph rewriting rules could extend the set of graph rewriting rules, or set limitations to the variables of the generation.

This generation type is a bit slower than the other methods of generation, but it heavily relies on the choice of graph rewriting rules. By setting a limit L to the number of nodes in the left-hand side of graph rewriting rules, we would have a time complexity of O(exp(L));

This method of generation produces graphs that can be used satisfyingly in most variants of skill trees, as a result of the guidelines proposed in Chapter 2.



**(a)** A big, Complex generated graph, after being modelled manually to look like two hearts



**(b)** Path Of Exile Skill Tree

**Figure 3.6** A visual comparison of a big generated graph, and a big graph from a contemporary video game

## 3.4 Discussion

The results generated by the three methods provided a good understanding on the direction the procedural content generation of skill trees should take.

The first method, the naive randomized graph generation, provided disorganized graphs, many lacking the property of planarity observed in the vast majority of the skill trees from the video games in the list compiled for this thesis. The time spent by the game designer trying to model a graph generated using this method may outweigh the time saved by the speed of this method of generation and the lack of input variables specification from the game designer.

The second method, the generation using L-Systems, proved to be a good upgrade from the first method, by providing planar graphs, and additionally, a method of partially drawing the result of the generation. The L-Systems have the great benefit of providing increased control over the result, by way of specification of rules, and additionally by means of using stochastic L-Systems.

The drawback of this method may be found in the similarity that can be observed from elements of the structure of the results. This is reduced with the stochastic L-Systems, but it still persists. One solution can be found in the creation of a much bigger list of symbols, and extending the drawing of the L-System as a graph further by associating a different arrangement of nodes for each symbol. However, in some cases, this minor drawback can actually be beneficial, given that the game designer wants to use a graph with many subgraph similarities in it. In the same manner that Lindenmeyer [PL12] wants the L-System to look like a flower, a game designer may want to have the same flower pattern added as a skill tree.

Creating visually pleasing L-Systems that can be added to a game as its skill tree is a time consuming task. For this method to be desired by the game designer, he should either use one of the more common ones, or spend the time designing one suitable for their needs.

The final method, the use of graph grammars, adds additional improvements over the L-Systems method while keeping nearly all of the other advantages of L-Systems. With carefully chosen graph rewriting rules and the blessing of the RNJesus, graph grammars are able to generate the same graphs an L-System could, while sacrificing the advantage of the self drawing method. The level of control over the generation is also high, similarly to L-Systems, but the time spent designing graph rewriting rules would usually be lower than the L-System counterpart.

## 3.5 Future Work

Additional research can be conducted to extend the field of procedural generation of skill trees. Following are some note-worthy proposals for research.

### 3.5.1 Naive Graph Generation Improvements

By observing the results of this generation method, we can recognize the need for improvements for this method. The simplicity and speed of this method has a few advantages over its counterparts, but it is outweighed by the other generation methods due to the very distant similarities to skill trees in contemporary video games.

Further research may include analysis of different easing functions that will provide the ratios of the two operations performed at each iterative step. Research can be conducted by using mathematical curves instead of the polynomial functions provided in this thesis. The research may discover that some specific curves provide better results for the generation.

Another possible improvement can be performed during the step of choosing which nodes will be chosen during an "add edge" operation. The choice can be conditioned by rules that may improve the overall look of the graph, such as verifying if adding an edge between nodes A and B will lead to creating a non-planar graph, or rules that keep graphs simplicity, such as forbidding nodes from going over a specified degree.

### 3.5.2 L-System Graph Generation Improvements

L-Systems provided satisfying results but mostly for one specific type of skill trees present in video games, technology trees. Further research may look into the variants of L-Systems not researched in this thesis. The field of L-Systems still has a big list of open problems for study, and a few of them may be relevant to the generation of skill trees. One of the open problems that is directly relevant is the finding of a L-System that can produce a given structure. Finding a solution to this problem can partially solve the need for the post-processing step for the generated graphs to look more similar to skill trees in contemporary games.

Other subjects that may be beneficial to be researched are the other variants of L-Systems and how they influence the results in relation to the skill trees in contemporary games. The context sensitive grammars in L-System variation may be a huge improvement to the generation as it may provide results applicable to the other types of skill trees.

### 3.5.3 Graph Rewriting Efficiency Improvements

The algorithm chosen for the graph isomorphism step of the graph rewriting method is not the most time efficient. Ullman's 1978 algorithm [Ull76] is in exponential time and a variant of this algorithm was used for the results of this thesis. The current algorithm finds difficulty in using bigger production rules during the generation because the time spent for the generation may be higher than if a game designer would do it without the generation. This leaves room for a big improvement for this step, and there are a few options to improve this step. First one is by using the improved version of Ullman's algorithm [Ull11], or one of the more advanced algorithms which increases the performance from an exponential time complexity to a quasi-exponential time complexity, [Bab16], [Dor+95]. The improved time efficiency of this step will allow for larger production rules to be used so that the result is provided in a convenient amount of time to the game designer.

Another important improvement can be found in parallelization of the tasks. Methods for parallelization can be proposed and used to increase the time spent during this step. Given the increasing parallel processing power of hardware that is widely used today, better parallelization methods will provide different time results. One simple method would be to parallelize both the subgraph finding step and the subgraph isomorphism step so they can have an early-stopping condition.

Another improvement can be found in caching. Caching results for earlier iterative steps will allow for the exclusion of certain subgraph isomorphism analysis if the subgraph was checked in a previous iteration step and returned a negative result. Given the continuous development of the graph after each step, finding a good caching improvement is a considerably challenging problem. One possible proposal would be to cache all conducted subgraph analysis at each iterative step and at the end of the step invalidate all subgraphs containing nodes related to the nodes included in the subgraph used for the graph rewriting. However, including the extra step of invalidation of subgraphs may be detrimental to the overall efficiency of the iterative step. The problem arises from the exponential number of subgraphs that may be cached that require to be checked and invalidated. Further research into this problem can be conducted.

### 3.5.4 Post-Processing step improvements

The use of extra steps may ensure that the generated skill tree is closer in similarity to skill trees from contemporary video games. There are many methods that may be used during the post-processing, and each can be researched to verify their practicability. One such method can be the inclusion of the generated

graph in a shape. One particular way to achieve this is by having the shape triangulated, the triangles placed in bounding boxes, and the nodes of the graph snapped inside one of the triangles of the closest bounding box. Including some extra rules for not overlapping nodes and edges, this method would layout most generated skill trees into shapes nearly ready to be used in the game, with minimal work from the game designer to rearrange the nodes.

One particular method for improving the overall look of the generated skill trees, is that, if a planar graph is generated, a planar embedding drawing is associated with it. This method requires the inclusion of the NP-Hard problem of untangling a planar graph [Goa+09]. L-Systems and Graph Grammars would both benefit heavily from this method, as these two generation methods can much more efficiently generate planar graphs.

### 3.5.5 The remaining steps of the skill tree procedural generation

The current thesis does not research the remaining steps proposed for the skill tree procedural generation proposed in the beginning of Chapter 2. The remaining steps may imply that they are too subjective to the particular game that a game designer wants to create, but a systematic approach may be researched and a general theory could still be proposed for at least a subproblem of the remaining steps.

# Conclusion

Procedural generation of skill trees in video games did not meet much interest for research purposes. The work of Jaroschy[Jar19] has touched upon the subject to present how a procedurally generated skill tree can find its way into at least one video game. Consequently, it is important to answer the question whether procedurally generated skill trees are possible to be used in a larger set of video games of different genres.

After the analysis of a compiled set of skill trees from video games belonging to a varied set of game genres, the observations extracted are used for a formal definition of skill trees, and additionally the proposal of an algorithmic approach to generate skill trees. With the steps for the generation of skill trees laid out, the first step is attempted in this thesis - the generation of the graph-like structure of skill trees. Extracting further observations about the graph-like structure of skill trees, a method for classification is suggested, and three different methods of procedural generations of graphs are tested - naive randomized graph generation, L-Systems, and Graph Grammars. Additionally, for graph grammars, a set of guidelines are made based on the observations of skill trees in video games, and the guidelines are used for the proposal of an initial set of graph rewriting rules that would allow graph grammars to generate graphs that have a very high resemblance to the graph-like structure of skill trees in video games.

With the results of the three methods, it is discovered that graph grammars is a method that has some clear advantages over the other methods researched, but it is not the perfect method of generating graphs for skill trees alone, but requires several future improvements to be considered clearly superior to the other methods performed. From the guidelines, by improving the algorithm for the graph grammars, the limitation of size for the graph rewriting rules may be mitigated further.

# Bibliography


[Bab16] László Babai. “Graph Isomorphism in Quasipolynomial Time [Extended Abstract]”. In: *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*. STOC ’16. Cambridge, MA, USA: Association for Computing Machinery, 2016, 684–697. ISBN: 9781450341325. DOI: 10.1145/2897518.2897542. URL: https://doi.org/10.1145/2897518.2897542.

[Coo71] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[Dor+95] Heiko Dorr et al. “Efficient Graph Rewriting and Its Implementation”. In: 1995.

[Far12] Charbel Fares. “Convex envelope generation using a mix of gift wrap and quickhull algorithms”. In: 2012. URL: https://dspace5.zcu.cz/handle/11025/760.

[GK20] Fabio Gadducci and Timo Kehrer. *Graph Transformation: 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*. Vol. 12150. Springer Nature, 2020.

[Goa+09] Xavier Goaoc et al. “Untangling a planar graph”. In: *Discrete & Computational Geometry* 42.4 (2009), pp. 542–569.

[Jar19] Petr Jaroschy. “Methods for Procedural Generation of Skill Trees for Computer Games”. In: (2019).

[Jen17] Christopher G. Jennings. “Lindenmayer systems”. In: (2017). URL: https://www.cgjennings.ca/articles/l-systems/ (visited on 07/22/2021).

[JP17] Alexis Jacomy and Guillaume Plique. “SigmaJS”. In: (2017). URL: http://sigmajs.org/ (visited on 07/22/2021).

[Mat18] Vince Matthews. “The Many Different Types of Video Games & Their Subgenres”. In: (2018). URL: https://www.idtech.com/blog/different-types-of-video-game-genres (visited on 07/22/2021).

[mem17] POE gaming community members. “POENinja”. In: (2017). URL: https://poe.ninja/challenge/builds (visited on 07/22/2021).

[Nag79] Manfred Nagl. “A tutorial and bibliographical survey on graph grammars”. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 70–126. ISBN: 978-3-540-35091-0.

[PL12] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[Roa12] Kevin Roast. “L-Systems”. In: (2012). URL: http://www.kevs3d.co.uk/dev/lsystems/# (visited on 07/22/2021).

[SHW19] Satyaki Sikdar, Justus Hibshman, and Tim Weninger. “Modeling Graphs with Vertex Replacement Grammars”. In: *2019 IEEE International Conference on Data Mining (ICDM)*. 2019, pp. 558–567. DOI: 10.1109/ICDM.2019.00066.

[Ull11] Julian R. Ullmann. “Bit-Vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism”. In: *ACM J. Exp. Algorithmics* 15 (Feb. 2011). ISSN: 1084-6654. DOI: 10.1145/1671970.1921702. URL: https://doi.org/10.1145/1671970.1921702.

[Ull76] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *J. ACM* 23.1 (Jan. 1976), 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925. URL: https://doi.org/10.1145/321921.321925.

# Appendix A

# Electronic attachments

Attached to this thesis will be the following:

- An archive "VideoGamesSkillTreeImages.zip" containing images with skill trees from each of the video games chosen for the set in Chapter 1;
- An archive "ProposedGraphRewritingRules.zip" containing images with the 13 graph rewriting rules proposed for the graph grammar method of procedural generation;
- A spreadsheet file "ResultsTable.ods" containing sheets with observations on the skill trees from the video games in Chapter 1, and the results of each method of procedural generation of skill tree explored by this thesis;
- An archive "ProjectFiles" containing the Java 1.8+ project which was compiled and run to generate the results for each of the procedural methods, and the PHP7.0+ web application that was used to display the results.

# Appendix B

# Setup of the application

For the complete setup of the application, Java SDK 1.8+ with Maven, and PHP5.6 or newer are required.

The project is separated into a generation application written in JAVA, and a results drawing web application written in PHP and using the SigmaJS library.

For the generation part of the project, a config file for IntelliJ IDEA IDE exists in the project's root directory so that the project can be imported with ease into the mentioned IDE.

To unpack and set up the application, proceed as follows:

- unpack "ProjectFiles.zip";
- import the JAVA project found in directory "generation" into the JAVA IDE of preference, and setup Maven to download the required library "Gson";
- use directory "display" for deployment of a website on a server that uses PHP5.6 or newer.

Using the Java Project to run the procedural generation of graphs using any of the three methods is presented in the $GeneratorRun$ class inside the function $main$. The method is structured so that the generation process for each method is easily visible and independent from the others.

Using the PHP application to display the generated results is done by accessing the "graphReader.php" page and sending the following two GET parameters:

- "file" to specify which file to load;
- "type" to specify which generation method was used to generate the file out of "rgg", "lsystem", "graphgrammars".

Examples:

- /graphReader.php?file=FILE.json&type=rgg
- /graphReader.php?file=FILE.json&type=lsystem
- /graphReader.php?file=FILE.json&type=graphgrammars

The page will have a window with information on the top right, and a menu with all the files from that generation method will appear if the key $M$ is pressed. Additionally, a few other actions are available through the pressing of some keys. Each available action and its key is displayed in the right window.

- "C" will mirror the graph using 2 random points from the graph's convex hull;
- "B" will round the node's coordinates for every node to their nearest integer;
- "R" will reload the graph from the file;
- "N" will start the Noverlap plugin, which will force nodes apart from each other if they are colliding;
- "P" will attempt to save the modifications on the graph in a new JSON file.

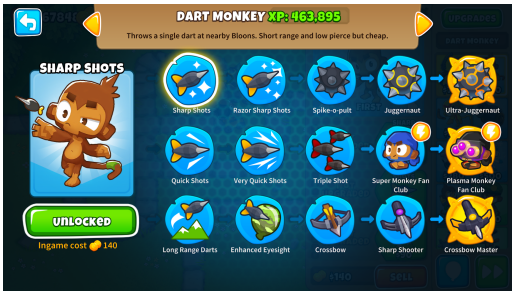
# Appendix C

# Video Games Skill Trees Images

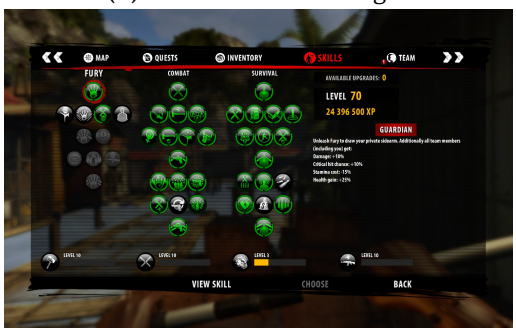
Included here are a few of the skill trees from the games in the list compiled in Chapter 1.


**(a)** Assassin's Creed Origins


**(b)** Batman Arkham Knight




**(c)** Bloons Tower Defense




**(d)** Dead Island Riptide


**Figure C.2** Dark Messiah Of Might And Magic

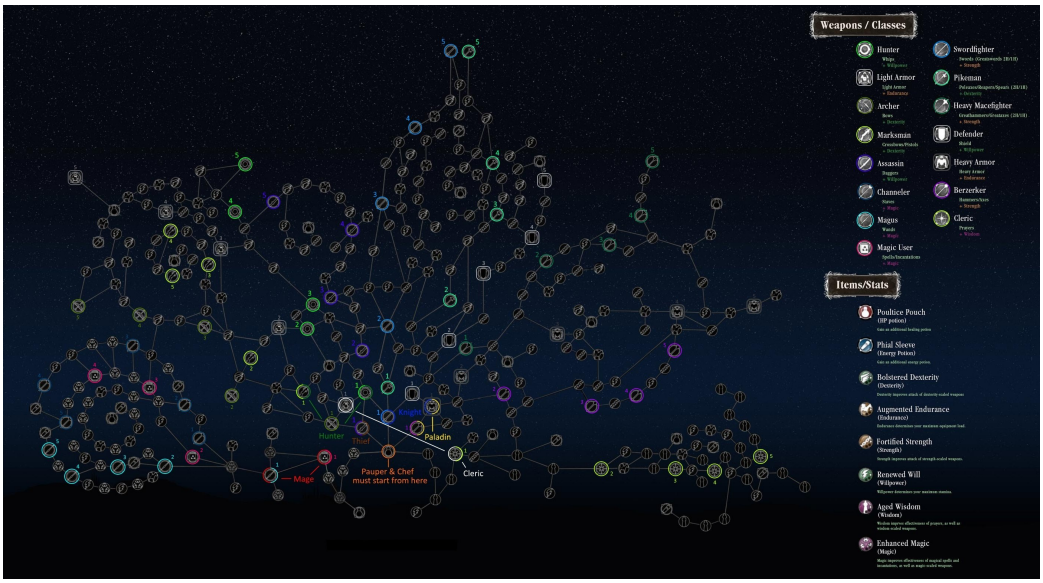



**Figure C.3** Salt And Sanctuary

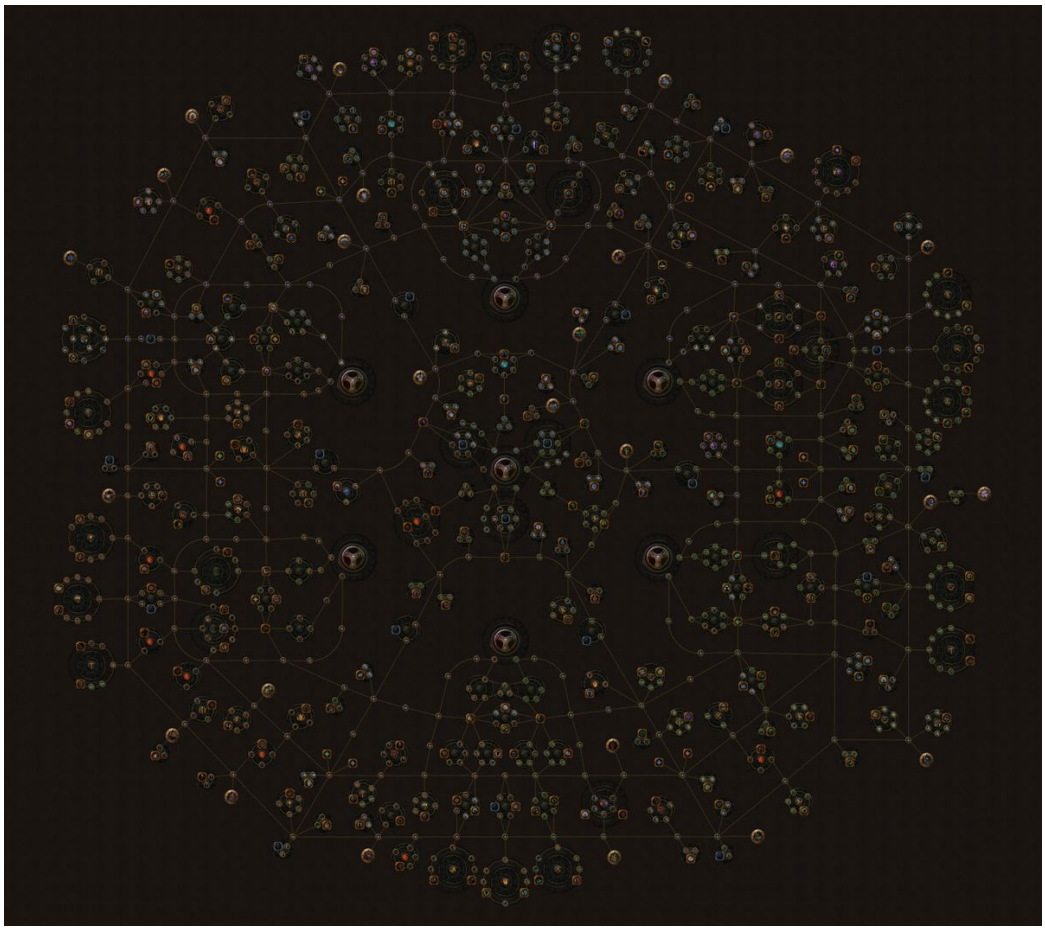


**Figure C.4** Path of Exile