FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Jaroschy

Methods for Procedural Generation of Skill Trees for Computer Games

Department of Software and Computer Science Education

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

Study programme: Computer Science

Specialization: General Computer Science

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., 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 paragraph 1 of the Copyright Act.

In…...... date...........
signature

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D. for all the motivation he has given me, for all the constructive criticism and for all guidance he has given me during the writing of the thesis as well as the development of the software. He has given me precious ideas to consider and develop.

Title: Methods for Procedural Generation of Skill Trees for Computer Games

Author: Petr Jaroschy

Department: Department of Software and Computer Science Education

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

Abstract:

Game developers often face the issue of having well balanced game in terms of difficulty, especially in role-playing games. Skill tree is a game element which contributes to solve this issue by giving the player more power in-game. Many game elements can be procedurally generated to save time and money to developers, but can skill trees be procedurally generated too? And how do we validate, that generated skill trees are well suited for the game? That is the goal of this work. We have made a simple turn-based game. Then we made several variations of generators of skill trees for the game. We took the best trees and validated their performance using artificial players based on data collected during their gameplay. Then we compared the trees and concluded that skill trees can be generated by our suggested method and their variations.

Keywords: Procedural content generation, skill tree, role play games, evolutionary algorithms

Contents

1. Introduction ... 1
   - 1.1. Skill trees ... 1
   - 1.2. PCG in games ... 3
   - 1.3. PCG validation ... 4
   - 1.4. Goal of this work ... 5
   - 1.5. Text structure ... 5
2. Analysis ... 7
   - 2.1. Flow in games ... 10
   - 2.2. RPGs and difficulty ... 11
   - 2.3. Skill trees ... 12
   - 2.4. Problems with skill trees and difficulty ... 17
   - 2.5. Skill tree space ... 18
   - 2.6. Generating skill trees ... 18
   - 2.7. Generative process ... 20
   - 2.8. Multi-objective optimisation ... 21
   - 2.9. Validation ... 22
   - 2.10. Choosing appropriate game ... 22
3. Our game ... 24
   - 3.1. Scenario ... 25
   - 3.2. Skill tree - node pick ... 26
   - 3.3. Encounter ... 26
   - 3.4. Turn ... 27
4. Proposing solution algorithm – Selection methods ... 28
   - 4.1. Individual ... 29
   - 4.2. Hill climbing algorithm ... 29
   - 4.3. Randomized hill climbing algorithm ... 30
   - 4.4. Evolutionary algorithm ... 30
   - 4.5. Weighted evolutionary algorithm ... 31
   - 4.6. Non-dominated Sorting Genetic Algorithm Revisited (NSGA-II) ... 31
5. Creating new generation – mutation and crossover ... 33
   - 5.1. Relevant input ... 33
   - 5.2. Operators ... 34
   - 5.3. Crossover - merging two trees ... 37

6. Fitness function 40
6.1. Relevant input 43
6.2. Reference decks 43
6.3. Choices 43
6.4. Approximating simulation of encounter 45
6.5. Rating types 45
6.6. Aggregating results 46
7. Implementation 48
7.1. Relevant input model and data storage 48
7.2. Whole algorithm 49
7.3. Mutation + crossover step 49
7.4. Fitness step 50
7.5. Thread safety 51
7.6. Post processing 51
8. Validation 52
8.1. Hypothesis, expectations 52
8.2. Design of the validation 54
8.3. Result model 57
8.4. Result interpretation + results for hypothesis 57
9. Discussion 64
9.1. Conclusions of validation 65
10. Conclusion 66
10.1. Remarks 66
10.2. Future work 67
Bibliography 70
List of Figures 72
List of Tables 75
List of Abbreviations 76
Appendix A – Reference Decks 77

1. Introduction

The goal of this work is to find out whether it is possible and useful to use PCG for creating skill trees for a game. There is plenty of works done on the subject of either PCG or skill trees, but as far as we know, there is none, which would combine both of the subjects. It is therefore unknown, if such use of PCG is possible and, more importantly, if it is as useful as using PCG for content like textures, dungeons or race tracks.

1.1. Skill trees

Figure 1: Example of skill tree in Incredible Adventures of Van Helsing (NeocoreGames, 2013). Multiple tree roots can be seen on the top of the skill tree window; these nodes do not need a predecessor to be obtained before they can be obtained themselves.

Skill trees have become one of the basic game elements. They are widely used and have multiple good properties, which allow us to make a game better. Their form is usually some adaptation of a weighted graph, where the nodes stand for new

abilities or stat boosts (both seen in Figure 1) and are given to the player once he acquires them in-game (forms of acquisition may vary).

The primary use of skill trees is to manage difficulty in games. Skill trees are a great tool to improve player's strength and therefore make future encounters with enemies and obstacles more manageable. The reason, why game developers want to do this, is to make the experience of most players better since it is not hindered by the game being too difficult to play or simply boring because everything in the game is effortless.

Additionally, they might also serve as additional content in the game and make a player interested in seeing how different combinations of choices in the skill tree impact the game. Some skill trees are made so that different combinations, chosen by the player, result in a different gaming experience. A great example of this is Path of Exile (Path of Exile review, 2018).

Difficulty is a property of each game, and it tries to quantify how good (skilled) the player needs to be to beat the game or its parts. Different games vary in their difficulties. Some games have greater difficulty than others, which brings in different groups of players since somebody is skilled and therefore enjoys harder games. Difficulty as a property is relative to the player's power in the game.

Power of the player is an abstract value, which represents all the game elements, which are given to the player in order to reduce his perceptive difficulty. Power of the player negatively correlates with relative difficulty for the player. Therefore the higher the player's power is, lower his perceived difficulty is. Game elements, which commonly give power to the player, are abilities, equipment, companions, skill trees and many more.

The simple process of progression is closely tied with difficulty. Since the skill tree provides power to the player, making the future gameplay easier, it can control at what state of the game is the player able to beat the next part of the game and progress further.

Progression is another aspect, which makes games enjoyable for plenty of players. Progression usually gives the player a satisfying feeling, and therefore, it is a wanted addition to many games by developers. For example, completing a part of skill tree can feel very nice in term of overall progression in the game (Figure 2), especially if you need the skill tree for reaching the endgame, like in Path of Exile (Brown, 2018).

Figure 2: A finished part of a skill tree, which is focused on damage dealt with bows. This part of the skill tree gives the player a sense of progression because of the finished node cluster. This cluster of nodes also gives a lot of power to the player, making future fights easier. (Grinding Gear Games, 2013)

1.2. PCG in games

Over time, content production has grown so much that nowadays, it is the main bottleneck of game development (Kelly & McCabe, 2007). Procedural content generation has been already surveyed (Hendrikx, Meijer, van der Velde, & Iosup, 2013), and therefore, I am going to give just a brief overview related to this work. PCG is a development approach, which uses algorithmic generation of game content with little to no human contribution (Togelius, et al., 2013). There are many advantages to this approach, the main one being saving resources to developers, namely time and money. Another advantage is the scalability of the content, which is able to be created by PCG rather than by manual creation. This is important for large scale Massively Multiplayer Online Role-Playing Games (MMORPGs). According to Hendrikx et al. (2013) "Main disadvantages of PCG are computational overhead and the requirement of good judgement of cultural and technical values of generated instances". This can be trivial for some game content like loot in RPGs, but can be quite complex when we take into an account how many skill trees can be created from given number of nodes, which we will discuss in Analysis chapter (2), and all the possible ways they impact the difficulty of the game (also discussed in Analysis), which makes the skill tree generation problematic to evaluate.

PCG can be split into two major categories by their moment of happening. Offline (or build) PCG is when the content generation is being done on or before the actual build of the game. Examples of this kind of PCG is generating textures and

maps during development (Carli, Bevilacqua, Pozzer, & Cordeiro d'Ornellas, 2011), usually together with adjusting the generating algorithm according to the generated content or manual adjustment of the generated content after its generation. Online (or runtime) PCG is when the content generation is happening in the released version of the game, generally multiple times. This kind of PCG helps the replayability of the game and its variance during the different playthroughs. Generating loot from monsters or generating replayable dungeon maps both fall under the online category, since they happen after the release of the finished product. In this thesis, we are going to utilise offline PCG, mostly because of computational complexity and lesser added value for doing the generation during runtime.

1.3. PCG validation

Generating any objects via PCG for games is one thing, making sure they are correct and will be useful once added to the game. Both of these parts are not trivial to perform, and we will talk about that in Analysis. Validation of PCG varies heavily on the type of PCG, but most PCGs can be tested by manual review (i.e. player testing, designer). Sometimes, the manual review can be replaced by approximating simulation of the game or other forms of computational approximation. Both of these are tightly fitted for each game-specifically. Only after validation is successful, the object is eligible to be added to the game.

In order to perform validation of our PCG application, i.e., skill tree generation, we will need a suitable game and the validating algorithm designed for such a game. This game's system must enable such validation and must use generated skill trees from our algorithm. Implementation of this game is also part of this thesis. The validating algorithm will consist of several bots, designed to play the implemented game. We will need to represent multiple types of players, and therefore, there will be several different types of bots. These bots will be used to play the game and estimate the difficulty of the game and suitability of the skill tree for a given game scenario.

Data will be collected from the bots in order to find out if the skill tree is suitable for the scenario used for its generation. Data will need to confirm several hypotheses before we can accept the skill tree as suitable.

1.4. Goal of this work

The main goal of this work is to find out whether it is possible and useful to generate skill trees, as they are an unusual structure, which affects game's difficulty. In order to do this, we will need to do the following:

1. Analyse the requirements of a skill tree and its impact on the game.
2. Create generators for skill trees using different methods of generation.
3. Validate the generated trees by a simulation using multiple different bots

1.5. Text structure

After this introduction, this work will be split into the following chapters:

1. The Analysis (2) will discuss the requirements of a skill tree, what is Flow, and how does it apply to skill trees and games. Then we will talk about role-playing games and difficulty. We will also discuss what method of generation we will be using and why.
2. Then, the chapter about our choice of game (3) will describe the game and give and give reasoning on why we chose to make this kind of game in particular. We will highlight the choices of a player in this game.
3. Then we follow up by proposing the solution algorithm (4). In this chapter, we will discuss different methods for generating a skill tree and point out their differences.
4. Then we talk about parts of the algorithm, starting with the basis of our solution (5). We will describe different mutation operators and their effect on every entity. We will also highlight the specific purpose of an operator if there is any. We will also explain how we are merging two different skill trees to create a new one.
5. Then we talk about the other important part, the fitness function (6). This chapter will explain how we evaluate trees during generation and why I chose to implement it the way I did. This chapter will also describe the whole process of taking an entity and giving it ratings based on performance in different objectives.
6. Then we talk about the implementation details (7). This chapter will mostly explain the reasoning for some choices, some specific decisions made during the programming of the game and the generator.

7. Finally, we talk about validation (8) and have a discussion about our results. We will talk about the design of the validation, reasoning behind it, and we will interpret the results of the simulation.

2. Analysis

The problem of generating skill trees starts with role-playing games (RPG) as that is the domain we are addressing in this thesis. Every RPG has a certain difficulty. Difficulty of an RPG is usually formed by obstacles or enemies the player needs to overcome or defeat respectively in order to progress through the game. But to defeat enemies or overcome obstacles, the player needs some kind of powers or abilities, which he can use to do so. We want to let the player obtain the power using the skill tree we generate for him.

Common tools, which let player obtain power, are gear such as weapons or armour, stats like strength or dexterity given with level ups or similar other means, such as giving the player points in a skill tree, which he can spend to gain available bonuses of his choice. Giving player power through gear and level-ups has a significant advantage of being almost deterministic when being designed. Only nondeterministic part of these tools worth mentioning, except skill trees, is whether the player decides to use the gear or the abilities given to him.

The situation with skill trees is not as trivial since the bonuses given are determined only by player choice and amount of points the player can spend in the skill tree. Additionally, obtaining nodes in skill tree is to some point permanent (some games offer 'reset' of a skill tree, either at a price or at certain points of the game), which is very different from other means of giving the power to the player, as gear can be swapped at will of the player. This means that the player's choice is much more important here.

However, this raises a question. How can we know how much power can we give to the player? This question can be answered, thanks to the theory of Flow (Cziksentmihalyi, 1990). The idea of Flow, put to context with games (Chen, 2007), gives us a very good baseline on how to handle just how much power should be given to the player. Flow in games tells us that the game should neither be too easy nor too difficult. Otherwise, boredom or anxiety may appear. However, Flow does not tell us exactly how much power should we give to the player; it only gives us the general idea, how we should treat power in comparison to challenge. Shortly, the idea is to have power rise with challenge, so the player does not feel bored or anxious.

Therefore, creating a good skill tree is not a trivial task. The number of possible combinations of nodes that form a skill tree or a tree, in general, is very high (see 2.6), more so if we account for different types of nodes and scale of nodes' effect values. But could we generate skill trees, which deliver just enough power to the player, via procedural generation, so that the player does not feel bored or frustrated?

This is not an easy question to answer. Let us take a look at two examples of skill trees (Figure 3, Figure 4). Let *P* be the power given by the skill tree. And let the power needed to beat the next enemy between 10 and 20. Now consider following trees with 2 available points to be taken.

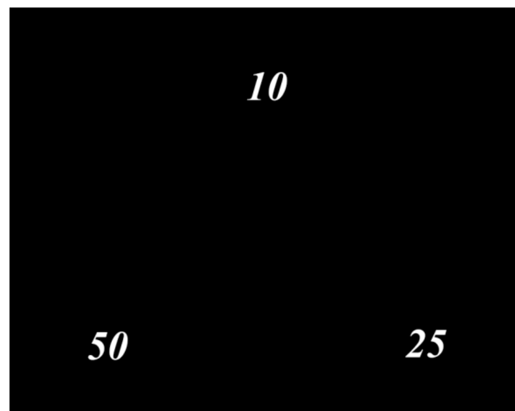

Figure 3: Example of a skill tree that is giving the player too much power; with two nodes, a player can take either 35 or 60 power.

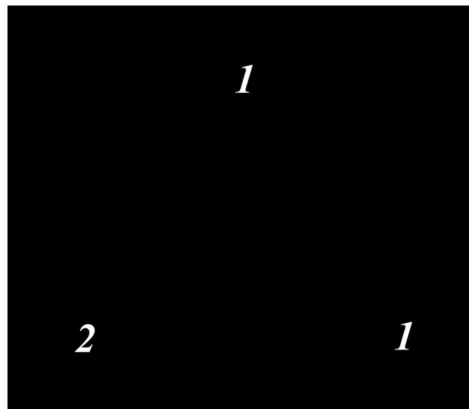

Figure 4: Example of a skill tree that is giving the player too little power; with two nodes, a player can get either 2 or 3 power.

In the first example tree (Figure 3), regardless of player choice, his final power *P* will be at least 35, which is way over the power needed to beat the encounter. This results in effortless gameplay, which often tends to be very boring and uninteresting. Given the second example (Figure 4), power *P* can get only up to 3, which is, on the other hand, way lower than the player needs. This would lead into impossible-to-beat

part of the game, which would lead to frustration and ultimately into the player not playing the game. So how can we tell, what is a good skill tree?

Skill trees do not have only one metric, which tells us if the tree is good or not. Through observations of existing skill trees in existing successful games, like in Figure 1, Figure 2 or Figure 6, we can see that there is more than one point of view regarding how good a skill tree is. One point of view is how well the skill tree manages the difficulty of the game. In other words, it is if the tree delivers an appropriate amount of power to the player. Another point of view is how the skill tree looks. The skill tree should not look like a path of nodes, neither like a hedgehog of nodes with the root in the centre. The goal is to have the appropriate number of nodes compared to the maximum distance from the root (or roots). Last significant point of view is how interesting the skill tree is. A player finds a skill tree interesting if he is given a lot of interesting choices out of available nodes while picking new nodes. The fact that we have multiple points of view of a skill tree makes us focus on multiple things at once. Therefore, our algorithm will have multiple objectives to achieve while generating a skill tree. This will be described in multi-objective optimisation (2.8).

In order to properly generate a good skill tree for a game, we need a couple of things. First, we need to be able to tell what kind of functions skill trees should have. This includes difficulty management, the appearance of the tree and its non-redundancy. A player must also find the tree interesting, which can be done by giving player interesting choices in the tree. Secondly, we need an algorithm to generate it. Thirdly, we need a way to validate our results.

In this analysis, we will first talk about Flow and Flow in games, as I believe it is a great stepping stone to be able to tell, what do we expect from a skill tree. Then we will talk about skill trees in general and explain their connection with difficulty and Flow. Then we will talk about skill tree space. We will explain what it is and follow up with a section about generating skill trees. Then we will talk about the generative process, how we want to generate skill trees and about multi-objective optimisation. Lastly, we will talk about what games are suitable for the generation of skill trees.

2.1. Flow in games

Flow is described in the book: Flow: The Psychology of Optimal Experience (Cziksentmihalyi, 1990). According to Cziksentmihalyi (1990), “The state in which people are so involved in an activity that nothing else seems to matter; the experience itself is so enjoyable that people will do it even at great cost, for the sheer sake of doing it.“ (p. 4).

Then Flow was used in the context of games (Chen, 2007). Flow in games gives us a decent baseline for what kind of difficulty management should we be aiming for. Difficulty of the game should not be too hard, because it induces anxiety. On the other hand having difficulty too low induces boredom. Therefore, for every difficulty, there should be some abstract range of values, representing the power of the player that is suitable for the current state of game. In order to combat high difficulty, we will need to give power to the player. We will call value, which represents the power of the player, **Player-power**. Player-power is an abstract value, which represents the power of the player and the obstacles and enemies he can overcome. Lower Player-power means tougher difficulty than higher Player-power while encountering the same obstacles/enemies. As the player progresses further, the challenge tends to increase in RPGs. In order to keep the game at a similar difficulty for the player all the time, the player needs to obtain Player-power.

This Player-power will help the player stay in the **Flow Zone** (Chen, 2007), which, as seen on (Figure 5), is a hypothetical area around very optimal experience curve (or path). *Flow* in the figure stands for the experience the player has while playing the game. This curve is impacted by the design of the game and the player’s choices during his playtime. *Abilities* axis in the chart represents multiple values. It represents the skill of the player and his Player-power.

Most, if not all, game elements have an impact on how Flow Zone is going to look like, and many even impact the optimal path itself, usually through the difficulty of the challenges or how much Player-power they give to the player.

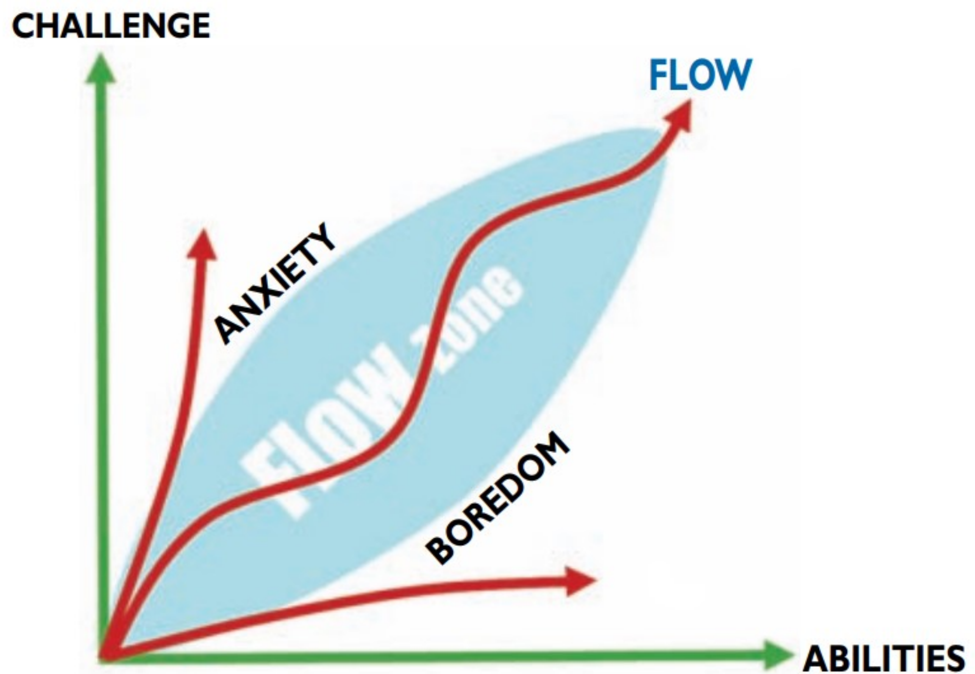

Figure 5 (Chen, 2007): Simple chart of Flow Zone in relation to challenge and abilities of a player. When the player does not have strong enough abilities, he experiences anxiety. When the player has too strong abilities for the current challenge, he experiences boredom.

2.2. RPGs and difficulty

Role-playing games (RPGs) are a genre of games which let the player embody a character in the game and play as the said character. What is important about this genre of games is the difficulty and difficulty management of these games. The general aim of any game is, besides other things, to make players play it and enjoy it. Enjoyment of the game is based on many things, but we are interested in how enjoyment is related to the game's difficulty.

Difficulty of any game is usually somehow quantifiable. However, it is game-specific. Generally, difficulty has a high correlation with the requirement of a player's individual skill in a specific game needed to enjoy or even play the game. So the greater the difficulty of the game, the higher the skill is needed to play and enjoy the game. If this difficulty is too low, the player can become bored. And similarly vice versa, the player can become anxious if the difficulty is too great for his skill. Both of these states may lead to the player not playing or enjoying the game, which is malicious for any publisher or developer, as it influences reviews and opinion of the player base negatively.

Determining difficulty of the game tends to be very difficult for real-time games. Even turn-based games usually have a significant branching factor. This is

the reason why most developers either turn to approximations or manual testing. Manual testing, however, is very expensive to do properly, since it needs many human players and their feedback, which is not affordable by smaller developer groups. On the other hand, approximating difficulty has computational complexity and is game-specific. This means that the author of the approximating algorithm needs to be very familiar with the game mechanics, concepts and the target player base.

Difficulty management is a routine that is supposed to eliminate the problem mentioned above as much as possible. Some developers even introduced dynamic difficulty management in their games, which allows games to adjust their difficulty during playtime (Fallout 3, 2008) depending on player's performance (i.e. dynamic enemy levels, gear requirements). Another example is the dynamic levels of mercenaries in Assassin's Creed: Odyssey (Ubisoft Entertainment, 2017). Difficulty management usually consists of adjusting the strength of the enemies or the toughness of the obstacle in relation to Player-power.

In this fashion, it is very similar to increasing challenge or abilities in the Flow chart (Figure 5). However, the Flow zone, in this case, would be the spectrum of player skill that is acceptable to play the game comfortably. Because of this, it is in developers' interest, to make Flow zone as wide as possible, either through dynamic difficulty management or through other game elements.

2.3. Skill trees

Figure 6: Example of an undirected skill tree. Nodes highlighted are considered active and give power to the player. All edges have the same weight of 1, and this price is paid by points, which player gets by levelling up. (Path of Exile, 2013)

Skill trees[1] (or technology trees in strategy games) are structures, which usually take the form of weighted graphs, where nodes represent skills (stat boosts, technologies) and edges represent their availability, example on (Figure 6). Every node has its type, which represents, what kind of effect the node will have once it is picked. Most common effects of nodes are unlocking ability or spell and increasing some stat, effectively making abilities the player already has better.

Nodes generally have two different states, active and non-active. Active nodes then have some kind of effect on the gameplay, which usually is empowering the player or giving him additional abilities, as shown in Figure 7. Every skill tree is specific in terms of management, when and at what price can a player obtain nodes in the skill tree. For example, Path of Exile (2013) has all nodes at the same price of one skill point, which is given every level up and for some quests. It starts with two nodes available for picking and only constraint on picking nodes is to have them connected to the current set of obtained nodes.

The general approach, however, is to let the player obtain nodes at some point in the game. This time can be a specific level, point in story or location. Every skill tree has a set of available nodes (nodes that can be obtained next). This set tends to grow over time in a game and can be managed by different game mechanics. Usually, however, when the player obtains a node in a skill tree, every node, which is connected to the obtained node, is added to available nodes. Some nodes may be unlocked and added to the set of available nodes at different points than just obtaining another node, but those events are external to the skill tree and depend purely on the game design.

[1] Historically, skill trees were only in form of trees with one root and were fully connected. Nowadays, skill trees have adapted into simple graphs. Term skill tree is used due to historical reasons.

Figure 7: This screenshot from Assassin's Creed: Origins (Ubisoft Entertainment, 2017) is an example of a node and its effect. This node gives the player additional ability after putting an animal to sleep. The player can tame the animal afterwards to gain a strong companion.

One of the many things that skill trees let us do is manage the difficulty of the game. By acquiring nodes in the skill tree, the player gets Player-power. This acquisition usually results in the ability to acquire more nodes in the future. Every obtained node results in an increase of Player-power (Figure 8), which increases abilities in the Flow graph to push the player from anxiety to Flow zone (Figure 9). This supports decent Flow between anxiety and frustration from challenging gameplay and boredom from overly easy to overcome obstacles through different gameplay and discovery of different possibilities of viable skill tree choices.

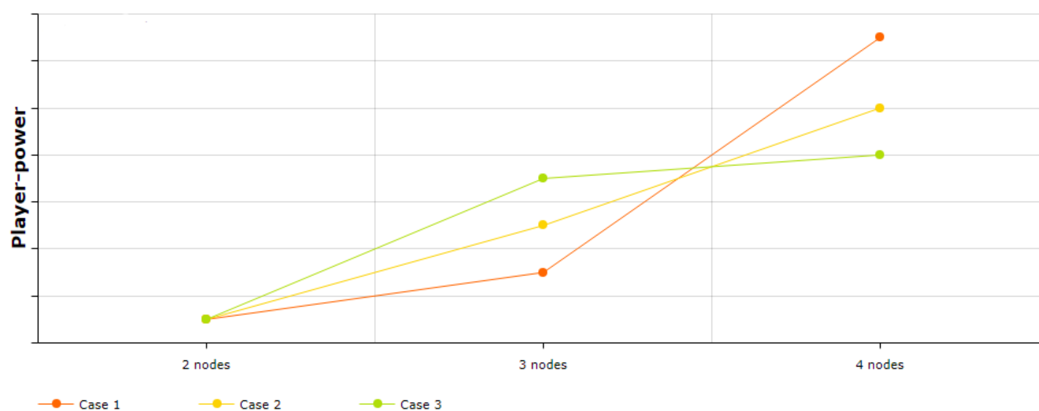

Figure 8: Effects of choices in a skill tree. At every state of a skill tree, there is a set of nodes we can choose from to obtain a new node. Depending on the choice of the player, the Player-power rises differently.

In Path of Exile (2013), you need to acquire enough power to beat late-game bosses or to simply be able to progress further. Even though most games have other mechanisms that prevent the player from progressing too quickly (see nad), skill trees are often part of that mechanism.

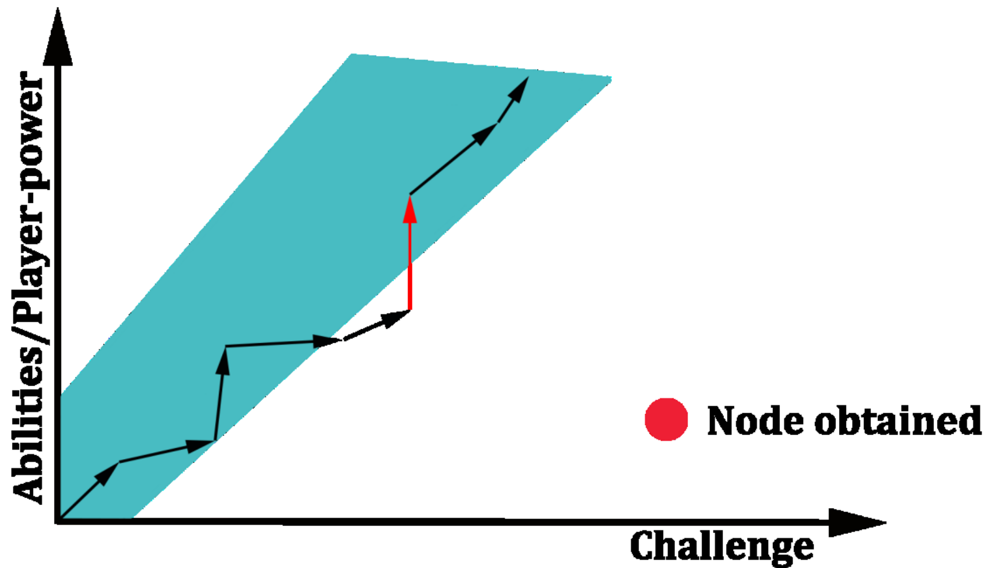

Figure 9: Effect of skill tree in regards to Player-power and Flow.

There are a couple of requirements we can have for skill trees. One of them is to manage the difficulty, but we will talk about that in the next part of the Analysis. Another one is for the skill trees to be interesting and appealing. This is meant in a sense that every choice from available nodes should not be obvious, but also not redundant.

For a skill tree to appear interesting, it needs to provide plenty of interesting choices for the player. Those interesting choices must make the player think twice when choosing from available nodes, both by their diversity and by what they add to available nodes after the player obtains them. This makes the skill tree more interesting to use in games.

Redundancy can be caused by two main causes. The first cause can be lack of possible choices. When the amount of available nodes is only 1 at a time, any user input is redundant, and the idea of skill trees is voided. Another cause is the homogeneity of available choices (both illustrated in Figure 10). When all available choices have the same or very similar effect, the choice may appear redundant. However, if there are significant differences in further choices, this redundancy may become harmless.

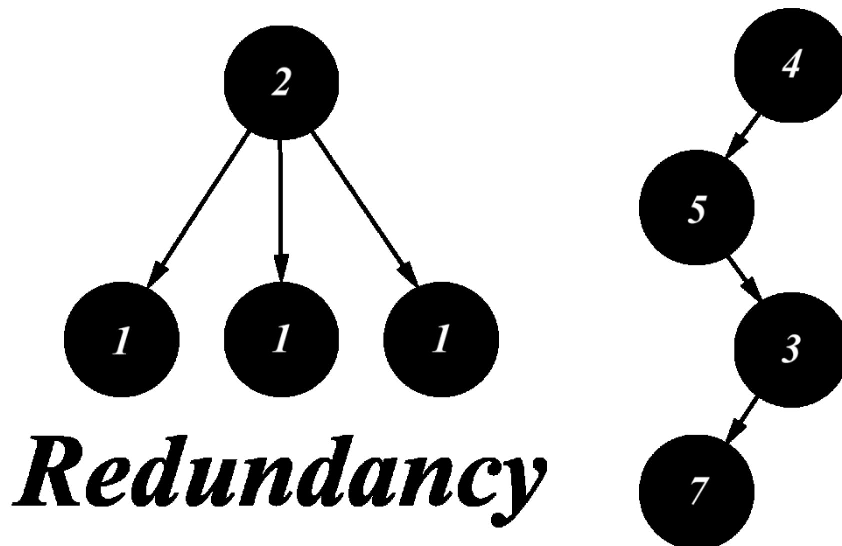

Figure 10: Example of redundant skill trees.

What makes choices in skill tree obvious is a big difference in the effect of the available nodes. In a case, where one choice is vastly superior to the others, player's choice is a mere formality, since the best choice is obvious and the others will never beat the best one (Figure 11). This problem is not that visible and intrusive if the differences in choices are small, even though there is an objectively best node. What mostly eliminates this problem is, having weaker nodes as an investment to get something better in future picks.

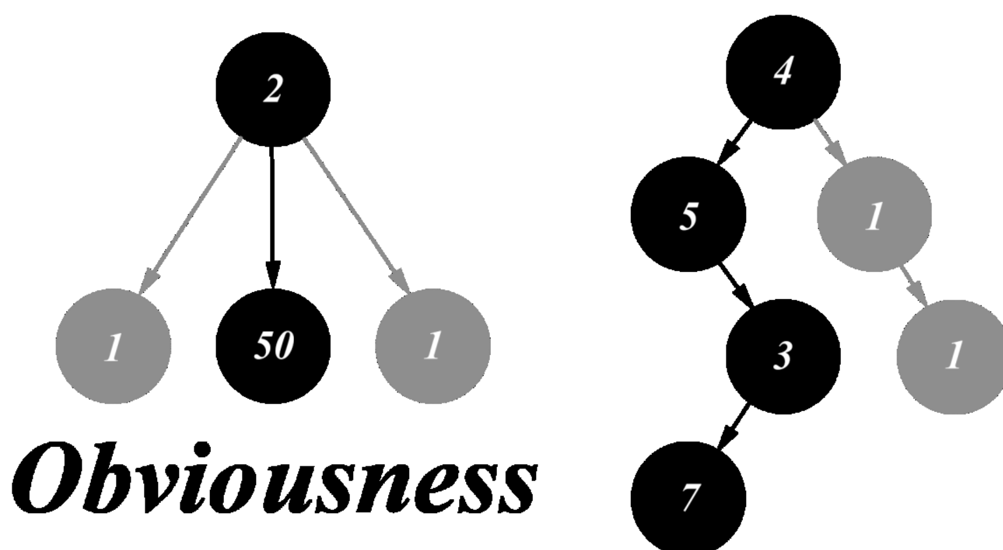

Figure 11: Example of obvious choices in skill trees.

2.4. Problems with skill trees and difficulty

We have shown the effects of a skill tree on the Flow chart; specifically, the effect of picking one node in the skill tree. However, if we would like to show the effect of picking two nodes, the number of different increases to Player-power rises. Let us have a very simple small 10 node skill tree in Figure 12.

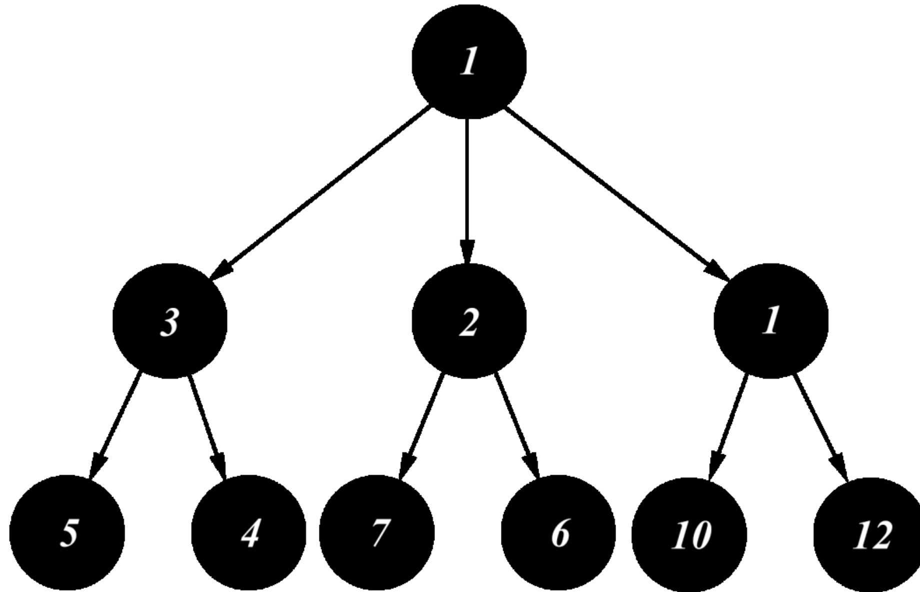

Figure 12: Example of a simple tree of 10 nodes. Numbers at each node represent their increase of Player-power when picked.

If we would already have the root of this tree obtained and we were to pick two other nodes, the range of possible Player-power increase is pretty wide. Depending on player choice, we can get an increase of Player-power anywhere between 3 and 13. This means that the increase of Player-power is not deterministic, and every increase does not necessarily need to push the player into the Flow zone. This makes generating skill tree complicated and opens a question if the player should be punished for making bad decisions.

This question has been answered by many modern titles, but differently every time. In Path of Exile (Grinding Gear Games, 2013), one can easily get next to no Player-power even if spending over 100 points in the skill tree. On the other hand, The Incredible Adventures of Van Helsing (NeocoreGames, 2013) makes you get a new skill or skill upgrade for each skill point spent, making you noticeably more powerful every time. The most deciding factor of skill tree node choices in this game is the preference of which abilities a player wants to use.

Therefore, every skill tree should give the opportunity to get the player into the Flow zone. However, not necessarily every combination of nodes needs to push the player into the Flow zone.

On top of calculating or approximating Player-power, we need to account for all the different combinations that can be taken at each point of the game. The number of different combinations of nodes rises with possible points and is highly dependent on the tree's structure. In order to generate skill trees, we will need to take in account all possible combinations of nodes to be taken in a tree for every state of the game (and its respective number of skill points gained).

2.5. Skill tree space

Skill trees have a certain space they occupy. This space must be defined in a way that all possible skill trees fall into it. We cannot place a skill tree in a normal vector space since that would require a variable amount of dimensions, based on the number of total nodes. However, what we can do is to confine the skill tree to a certain amount of nodes *N*. This allows us to have different combinations of up to *N* nodes, but we also need a way to represent the parental relations. These parental relations can also be represented in the number without sacrificing space to invalid skill trees by adding information about the parent of each node (i.e. at the end of the number representing node).

What we would need to represent for each node is its type, value (if any), its parent (in the form of id or something like this). There will be invalid entities using this method (disconnected trees, cycles), but we can eliminate those.

Finally, we can tell how many (or at least upper bound) trees can be created using *N* nodes, T types and V being the max value of node effect. According to Cayley's formula, there are N^{N-2} trees over *N* labelled nodes. If every node has T possible types and value of 0 up to *V*, then we can have $(V \cdot T)^N$ possible combinations of *N* labelled nodes in the skill tree. This gives us the following number of skill trees.

$$((V \cdot T)^N)^{(V \cdot T)^N - 2}$$

2.6. Generating skill trees

Therefore, we can see that the space of all possible skill trees is very massive, so using any kind of enumeration to try which tree rates the best would be

unreasonably slow. There is also going to be a non-trivial percentage of trees, which will be insufficient, since plenty of trees will be too powerful in some stages and vice versa. Let us look at the definition of a **node**, which will be used in this thesis:

Property	Note
Category	Category defines the effect of a node when it is obtained
Effect value (or just value)	Specifies the strength of the effect. Usually how big boost is given to the player.
Children	Which nodes become available when this node is obtained
State	State of the node. Can be non-active, available or active.

Table 1: Definition of node's properties

We can combat this complexity by following:

Reducing maximum node value or number of categories usually results in way less optimal solution, since there are wider gaps between different tree instances. Therefore, the desired Player-power is harder to achieve accurately. Reducing the number of categories is also undesired, because it may be either impossible or very game impacting in a negative way by reducing content and also widening the gaps between achievable Player-powers.

Reducing the number of nodes is very unwanted. It limits the space we are exploring in a naïve way, since we are reducing the interesting aspect of the tree, increasing redundancy and therefore reducing the amount of content in the game.

The number of nodes in games ranges from just a couple of nodes (Hamilton, 2018) to over a thousand nodes (Path of Exile review, 2018). However, removing any of the nodes hurts the quality of the skill tree for said reasons. This can cause the best part of the skill tree space to be unexplored, which results in a possibility to severely impact the output tree in a generation.

A way of combating complexity, which does not impact the result too severely, is using some dynamic approach (local search, evolutionary/genetic algorithm). By dynamically searching only parts of the skill tree space, we can achieve the desired performance of generating skill trees. One can also see that if we change a good tree slightly, we should not get a completely terrible tree. A slight change is considered a small adjustment of value, adding a new node with small value or deleting a leaf node with a small value.

There are many algorithms that use a dynamic approach, but we are going to focus on algorithms with an evolutionary approach. These algorithms use fitness functions to evaluate fitness (goodness) of entities and apply changes to their entities to achieve better fitness. In this thesis, we will be using an adaptation of the Simple Genetic algorithm (SGA).

2.7. Generative process

SGA uses an iterative process, where it takes a **generation** of entities, evaluates every entity using a fitness function, selects the best individuals and performs mutation and crossover to create new entities with small changes compared to the ones from the last generation, creating a new generation (Snippet 1). Usually, SGA uses encoding for its individuals and applies standard operators to them to mutate them and then performs standard crossover. However, our algorithm will have a few changes.

1: Initialize population
2: Evaluate individual fitness of initial population
While (not stopping criteria) do
3: Select best individuals for mutation and crossover based on fitness
4: Perform crossover and mutation to create new generation
5: Evaluate new individuals' fitness
6: Return best fit individual

Snippet 1: The pseudo-code of classic Simple Genetic Algorithm

Our algorithm will not have encoding into a string, because it is very unintuitive to do for trees and the operators would be cryptic. Therefore, we will have to use custom operators to mutate our entities, and the crossover will also be custom. Then we also have to choose our initial population and decide on the selection process.

A very important part of any evolutionary algorithm is having a good fitness function, which is both accurate in terms of telling us, how good an entity is and fast, so we can evaluate as many entities as possible. We will talk about the fitness function of this thesis in chapter 6.

We will talk about the operators later in detail in section 5.1. Here I want to talk about the initial population and then describe the selection methods used in this thesis.

The **initial population** is very crucial for every genetic algorithm (GA), and it must correspond to the design of the operators. Meaning, if the operators are designed to grow the individuals, it is a better idea to go for an initial population consisting of 'small' individuals to get less biased results (bias, in this case, depends on the creation of the individual, as it is not developed by our operators). Therefore, we went with the strategy of starting with 'small' individuals (skill trees of few nodes), and then we let them grow through the operators. This also follows the idea of Flow, since the small and weak tree will usually leave the player below the Flow zone and growing the tree will gradually push him upwards.

The most basic selection method is taken from **hill climbing**. This local search based selection method only takes the best individual from a generation and then tries to look in its 'local' proximity. This local proximity is determined by the small changes performed in by operators. This method relies heavily on the initial population (in this case, only one individual). Therefore, additionally, we adapt this method to run it multiple different times with a random small tree as the initial population.

Another selection method used in this thesis will be more like SGA, as it uses the crossover. In this thesis, we will call this algorithm the *Evolutionary algorithm*. This algorithm increases the size of the generation from 1 to what will be specified as a parameter in the input. Additionally, an adaptation of this algorithm is created to weight different properties of every individual differently, based on input parameters. This adaptation will be called the *Weighted Evolutionary algorithm* in this thesis.

Finally, the last selection method is an adaptation of the *Non-dominated sorted genetic algorithm* (NSGA). This method is a form of multi-objective optimisation, which means we do not aggregate all ratings of the tree into one final rating and then sort them by this value. Rather, we form *ranks* of trees and calculate their *crowding distance* (more in 4.6).

2.8. Multi-objective optimisation

Multi-objective optimisation was mentioned in the previous section, and I would like to state what it is and why we need it in our algorithm.

Multi-objective optimisation is a form of optimisation, which tries to focus on multiple properties or ratings at the same time. We can use this behaviour in our thesis because skill trees have several different ratings that can be given to it. NSGA

is a great example of an algorithm, which performs multi-objective optimisation and therefore we can use it for our purpose.

Multiple objectives for skill trees were introduced because having only one objective would make the algorithm have a tendency to optimize one aspect of the skill tree while sacrificing the others, which would make a terrible tree.

2.9. Validation

An important part of every offline PCG is the validation of generated content. This can be done in various ways and depends heavily on the type of generated content. In our case, however, few options come to consideration. One is manual validation through player testing. Another way is to perform a much stronger evaluation than is being done during the generation, perhaps using bots instead of people to play the game with given a skill tree.

This validation will be described and done after generation. We will need to represent multiple different player groups depending on skill, and therefore, multiple differently deciding bots will be made. This will represent different groups of players to more accurately reflect the actual gameplay as if we were manually testing. Bots will collect data, which we will use to evaluate whether the skill tree is appropriate to use.

This data will mostly consist of their performance and success in the game. The most important information is if the bot won, how well it won, and what was the chosen combination of nodes in the skill tree it picked. The most important information we want to extract from this data is how important the skill tree is, whether the skill tree can make the bots win and lose based on bot's decisions in-game.

2.10. Choosing an appropriate game

Not every game is suitable for PCG of skill trees. PCG in general needs to be used to generate either large scale elements or many small elements. Otherwise, it would be simply easier to design them manually. There are a few games, which make use of multiple skill trees. Example of this would be Dead by Daylight (Behaviour Interactive, 2016), which generates pseudo skill trees each level, where the player can obtain items and new perks by purchasing nodes for blood points

earned in matches. Using PCG to generate large scale skill tree is possible, but PCG should serve as a baseline, which should be manually reviewed and balanced.

A game, which would be suitable for PCG of skill trees, requires a couple of properties, so the PCG is beneficial. Following features of the game are required or beneficial for the PCG.

Multiple skill trees must be required either for the development of the game or the gameplay itself. Without this, the skill tree generating algorithm would be almost redundant, as only a single skill tree can be created by an experienced game designer. Therefore, creating a generating algorithm would very likely create an overhead, which would not save money or time. Additionally, it may require learning more about skill trees for a developer, who does not necessarily need to know, how should the skill tree look.

Large scale skill tree is another feature, which enables PCG of skill trees to be beneficial. Designing large scale game elements, including skill trees, may be very time consuming for manual creation and therefore it is helpful to create at least a baseline using PCG.

It is required that the game, for which is the validation done, has a system, which allows for the creation of a fitness function. This includes a variety of possible solutions. One is the ability for the game to be (at least approximately) **simulated**. Therefore, as I mentioned above, turn-based games have an advantage as they are usually easier to be simulated and therefore, the difficulty approximated. Another solution might be a **simple game system**. The simple game system would allow computing of the exact ranges of Player-power needed at different stages of the game.

3. Our game

For the use of this thesis, I needed a turn-based game that involved skill trees as a part or the entirety of its difficulty management. I was directly inspired by Slay the Spire (Mega Crit Games, 2017) (Figure 13). I created a card game, that has a set of enemies in every encounter, and the cards stand for actions the player can do. Player's goal is to defeat all enemies in the encounter, and then he is taken to a skill tree screen when he can choose one node from the tree, after which he is going to face a new, more difficult encounter.

Figure 13: Screenshot from the game Slay the Spire (Mega Crit Games, 2017). Cards in this game stand for actions and spend the player's energy. The goal is to face encounters and progress further while upgrading and expanding one's deck of cards.

Since we needed a simple game system to benefit better from PCG, I stuck to having a combination of damaging cards, which target all enemies, and cards, which target only a single enemy. I needed to split the cards into different categories, and then every node in the skill tree would buff a category-specific subset of player's cards.

In order to implement the need for multiple skill trees, the game is played in scenarios, and there will be a different skill tree for each scenario. This simulates the need for multiple skill trees while saving time of game designers. This game system also allows us to group any deck, any encounter and any skill tree into a scenario.

In this game, Player-power is increased only through picking nodes in the skill tree, and the difficulty is defined only by the set of enemies.

Every playthrough of the game, the player gets to play one scenario. Player has 50 hit points, 20 points of energy and 10 points of mana. Energy and mana refresh to their full values every turn. The whole state diagram is shown in Figure 16.

Figure 14: Screenshot of an encounter of our game. Cards are used as actions to defeat enemies. Some are targeted, and some are area.

3.1. Scenario

Scenario is formed by a sequence of encounters, a deck and a skill tree. Encounters are defined as sets of enemies, which will the player be facing during the encounter. Deck is a set of 15 cards and skill tree is a set of nodes with defined connections, category and effect value.

Card is defined by the following properties:

Property	Note
ResourceType	Type of resource to be spent when this card is played, so either Mana or Energy.
Cost	Amount of resource spent when this card is played.
Category	Type of card effect, in our case, only area or targeted damage.
EffectValue	The value of the effect representing how much damage is dealt.

Table 2: Card’s definition

Scenario starts with the first encounter with no upgrades from the skill tree. After each encounter, the player is taken to purchase one node from the skill tree to

buff his cards until the end of the scenario. The scenario ends when the last encounter is beaten or when the player dies in any scenario.

3.2. Skill tree - node pick

After every encounter, the player is taken to the skill tree view, where he will pick one node from the available ones in the skill tree (Figure 15). Player has visual assistance to see what node will upgrade what cards, and how much more damage will the cards do. After the player chooses a node he wants, he is taken to the next encounter, which is usually harder than the first one. Nodes definition is described in Table 1: Definition of node's properties.

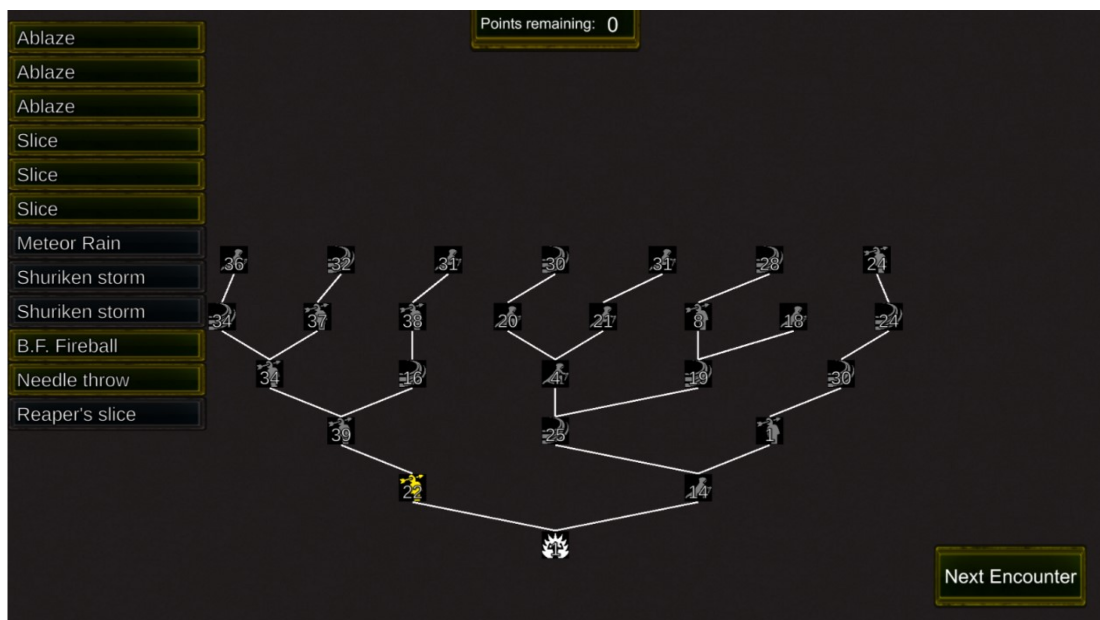

Figure 15: Screenshot of skill tree view in the game. One point per encounter is picked, and the cards that are upgraded are highlighted.

3.3. Encounter

Every encounter starts with enemies and player at full hit points and the player having his starting 5 cards and full mana and energy. Player's deck gets shuffled at the start of every encounter too. Encounter is split into turns for the player. Encounter ends with all enemies being dead or player's hit points going equal or below 0. If the player's deck runs out, cards in his discard pile are shuffled and moved to his deck (effectively cycling cards). We can see how the game looks in Figure 14.

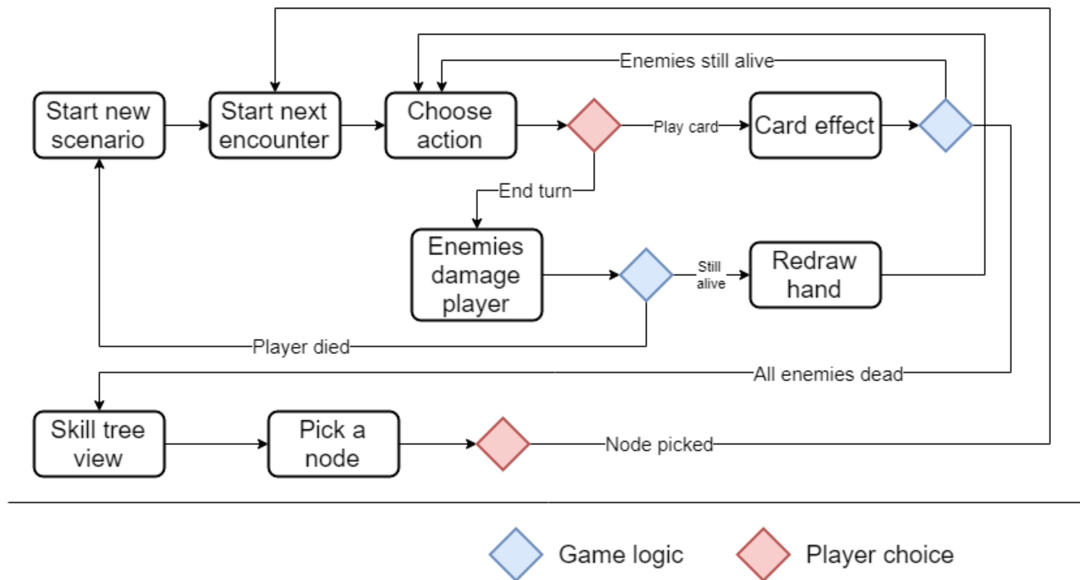

Figure 16: State diagram of our game

3.4. Turn

Every turn, player starts with 5 cards that he draws from the deck. His energy and mana are replenished to 20 and 10 respectively. After that, the player can play any cards he wants as long as he has resources to play them. Cards damage enemies and kill them, once their hit points reach 0 or less. After the player is done with playing cards, he presses the end turn button. Upon ending turn, all living enemies damage the player for their attack value, and then the player discards remaining cards in his hand into the discard pile and redraws 5 cards for the start of next turn.

4. Proposing solution algorithm – Selection methods

Since we have chosen to implement a GA, we are going to follow the usual GA steps, according to our adaptation in Figure 17. Starting with the initial population, we mutate the current population and create a crossover of the original population. After that, we run all the entities through our fitness function, and we select only top fitting trees, depending on the method used and input parameter. For now, the fitness function is going to be a method, which will take a skill tree and evaluate its properties into multiple ratings. Fitness function will be explained in detail in chapter 6.

After that, we check, if we hit the stopping criteria (which will be just the number of generations in our case) and either stop the iteration or start with mutation again.

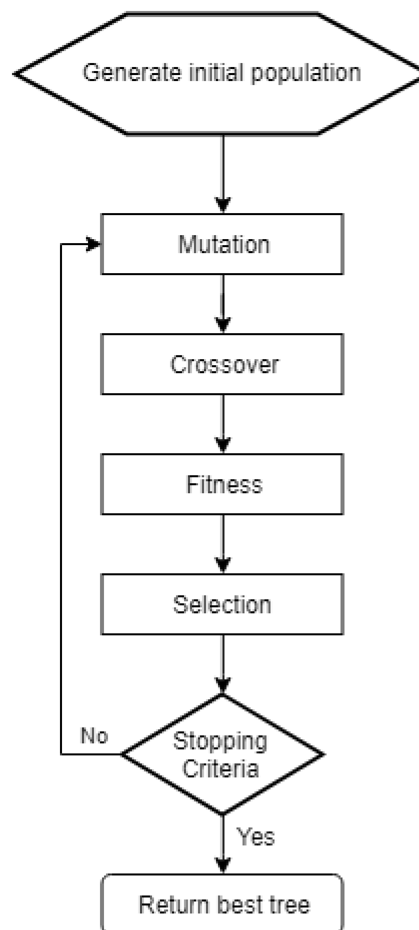

Figure 17: Our adaptation of SGA for this thesis.

In this chapter, we will discuss the methods of selection, and initial populations we used in all different implementations. We will describe the structure of the individual in our algorithm, and then we will start from the simple methods and get to the difficult ones by explaining extensions and changes of the previous methods.

4.1. Individual

An individual of our algorithm will be a skill tree. Since trees have a non-linear structure, we cannot describe the whole structure of the tree, since it will be dynamic. We can first describe the structure of the node used in the implementation.

Property	Note
Depth	How far is the node from the root
EffectValue	Amount of increase
Targets	What cards will the node affect
Children	What nodes become available once this is picked

Table 3: Node's structure in the algorithm implementation

So our skill tree object will be defined by the following properties.

Property	Note
Root	Root of the tree
NodeList	List of all nodes, mostly for easier manipulation

Table 4: Tree's properties in the algorithm implementation

4.2. Hill climbing algorithm

The idea of the hill climbing algorithm is having the main entity, which we mutate every generation and look for a better one. Due to this approach, we do not do any crossover of trees during the generation of a new generation.

The **initial population** of this algorithm is a pseudo **zero-point**, which is only a root node, which cannot be altered. It is repeatedly mutated, and the best tree out of the mutated is taken as the new best tree, and the process continues iterating. **Selection** is done simply by taking the best tree of the entire population.

In the first implementation, the **stopping criteria** used to be only to check, whether we got a better tree in the generative step. However, the second implementation, which included the probability-based operators with random effects, used a new stopping criterion to be consistent with other selection methods. The stopping criterion is the fixed number of generations, which is part of the input, and therefore is also consistent with other algorithms implemented.

4.3. Randomized hill climbing algorithm

This algorithm is a direct extension of the Hill climbing algorithm. The difference between this and its predecessor is that Randomized hill climbing algorithm runs several times with **randomized starting points**, instead of once with a zero-point start. After all of the runs ended, the best-rated tree out of the best trees is returned as the result.

So technically speaking, the **initial population** is a randomized small tree. A randomized small tree is a tree of 5 to 10 nodes with random targets and structure. The number of nodes in the initial tree may vary depending on the total size of the tree. The **selection** method does not change since the previous one, and the algorithm is executed multiple times.

Randomized starting points are created by taking the root node, choosing a small random number (in respect to the node count in input) and creating that amount of random nodes (random type, random effect value). After that, we sequentially assign these nodes one by one to the random node already in the tree as its child.

4.4. Evolutionary algorithm

In addition to the previous ones, this algorithm introduces multiple entities to carry over from the old generation, which allows us to introduce crossover (more in 5.3) in the step that creates a new generation. The **initial population** of this algorithm is X randomized small trees of up to 10 nodes (their amount is equal to population size, which is part of input parameters) population, where X is specified in the input. After that, at the end of every iteration, a number of trees specified in the input are taken to be mutated and randomly merged in order to create a new generation. At the end of this algorithm, the best-rated tree from the last generation is returned.

Therefore, the **selection** is altered that it takes the best X trees from the population, where X is specified in the input (same as the number of trees in the initial population). The **crossover** step is not omitted anymore. The crossover step is explained thoroughly in section 5.3.

During the implementation of this algorithm, the decision was made to split different rating categories instead of aggregating all of them into one number during the fitness function (detailed in chapter 6). This was made in order to prepare for implementation of the next algorithms.

4.5. Weighted evolutionary algorithm

This algorithm is a simple direct extension of the evolutionary algorithm, and the only difference is that the ratings are now weighted (more about ratings in 6.5) instead of having fixed weights. The weights are given as input parameters. This allows us to focus on a certain part of the rating more than on others, which can yield much more different trees than before.

So, everything besides the **selection** step is unchanged compared to the previous method. The **selection** step now takes a number of entities based on their order after multiplying their different ratings by their respective weights given in input.

4.6. Non-dominated Sorting Genetic Algorithm Revisited (NSGA-II)

This is another extension of previous algorithms, but this is by far the most complicated extension. In this method, we are performing the multi-objective optimisation, which was mentioned in section 2.8. Instead of sorting all trees by aggregating their rating either by summing or weighted summing, we create a Pareto-optimal front by a non-dominated sort and then sort the ranks based on their distance, in order to promote diversity across all different objectives. This algorithm was proposed (Deb, Pratap, Agarwal, & Meyarivan, 2002) as an improved version of the Non-dominated Sorting Genetic Algorithm (NSGA) (Srinivas & Deb, 1994).

According to NSGA-II, all entities have two properties, which provide complete sorting information to sort our algorithm. One is Rank; other is Crowding distance.

This method changes the preference for selecting individuals. Similarly, a specified number of individuals are taken from the sorted list according to fitness, but the sorting is done differently. Sorting of the entities is done primarily by *Rank* ascending and then by *Crowding distance* descending.

Definition (Domination): Let u and v be real number vectors of the same dimensions. Vector u dominates $v \Leftrightarrow \forall i: u_i \geq v_i \wedge \exists j: u_j > v_j$.

Rank is given to all entities based on their mutual domination. Starting the whole list at *Rank* 1, every entity is going to increase *Rank* of all entities, which it dominates. Then we do the same for the entities with higher *Rank*. We iterate this process until there are no remaining entities in the current *Rank*.

Crowding distance is computed sequentially depending on how close are the values of the previous and next entity for each rating. Process repeats for each rating. All entities are sorted by a rating. Entities with extreme values (min, max) are assigned positive infinity, then all entities with non-extreme values have their crowding distance increased based on the range of previous and next entity's values.

The crowding distance represents each entity's diversity compared to others. For example, if there is a couple of entities with similar values for most properties, then their crowding distance is going to be very low, and therefore they are less likely to be picked if the whole *Rank* is not being picked.

5. Proposing solution algorithm – Mutation and crossover

In this chapter, we will talk about the mutation and crossover step of the algorithm. All methods share these two steps in common. **Mutation** is performed via operators, which take an entity as input and output a mutated entity. Every operator has some form of random choice and random change of a given entity. Operators' random choice is designed in a way that would support the creation of interesting choices. For example, an operator which increases EffectValue will be biased to pick a node with lower EffectValue rather than a node with a higher one.

Crossover step will be explained, as it is not trivial to perform for skill trees. This step is also based on probability.

First, we will look at mutation and describe all operators and how they mutate the trees. We will describe the process of every operator and its resulting effect on our game. After that, we will describe the way we perform crossover of skill trees and how we solve different structures of entities.

The most basic operators are *Strengthen* and *Weaken*. These provide the basic increases and decreases in nodes' effect values. The *Split* and *Merge* operators mutate the skill tree in a way that gets rid of strong obvious choices and weak redundant ones. Finally, the *AddNode* operator lets the tree grow another random leaf node. This operator takes care of structure growth.

The **mutation** step takes all entities selected from the last generation and applies every operator to every entity from the selection with probability given as an input parameter for each operator. Each operator creates a modified (mutated) copy of an entity given to it. This means all entities from the selection of the last generation are preserved.

5.1. Relevant input

Relevant input for this part of the algorithm is the probabilities of operators modifying the entity.

Parameter	Type	Note
`StrengthenOpProb`	`Integer`	Probability of using Strengthen operator
`WeakenOpProb`	`Real number`	Probability of using Weaken operator
`MergeOpProb`	`Real number`	Probability of using Merge operator

SplitOpProb	Real number	Probability of using Split operator
AddNodeOpProb	Real number	Probability of using AddNode operator

Table 5: Relevant input for mutation

5.2. Operators

Strengthen node operator is a very simple operator, which increases the EffectValue of one node of the tree. This choice of the node is biased to choose nodes with lower EffectValue over nodes with a higher one. The choice is made following way. Let *MaxValue* be the maximum value of all nodes. Root (n_0) cannot be picked. Node's (n_i) chance to be picked is represented by the following equation.

$$p_{n_i} = \frac{MaxValue + 1 - Value_{n_i}}{\sum_{i=1}^{|N|-1} MaxValue + 1 - Value_{n_i}}$$

This probability model supports the creation of interesting choices, as it directly tackles obvious choices by creating alternatives.

The increasing amount is random, and the range depends on the depth of the node from the root. This results in a bigger increase of Player-power when the player picks the strengthened node. In our game, it means a bigger buff for the cards the node would have buffed originally. Therefore, there is a bigger incentive to take this node when it is available.

Weaken node operator is also a simple operator, and it is very similar to strengthen, but it is biased to stronger nodes instead. This node reduces the EffectValue of the node by a random amount. This amount also ranges higher, the higher the depth of the chosen node is. The chance for (n_i) to be picked is:

$$p_{n_i} = \frac{Value_{n_i}}{\sum_{i=1}^{|N|-1} Value_{n_i}}$$

This operator also helps create interesting choices by directly affecting obvious choices. However, in this case, it is done by reducing the EffectValue of a high EffectValue node in most cases.

Merge node operator is way more interesting. First, it chooses a node randomly, more biased towards choosing the weaker nodes. Probability for (n_i) to be picked is again the same.

$$p_{n_i} = \frac{Value_{n_i}}{\sum_{i=1}^{|N|-1} Value_{n_i}}$$

Then it finds its weakest sibling and merges them together. If the chosen node has no siblings, no modification is performed. Merging process creates a new node (Figure 18). New node's type will match the type of the stronger of the original nodes and the effect value will be the sum of the originals'. Children of original nodes will be concatenated together and added as children of the new node.

Effect of this operator in-game is to reduce the redundant choices during the picking of a skill tree node. Since the player is likely to choose the strongest node, merging two weakest together may appear as a more interesting choice.

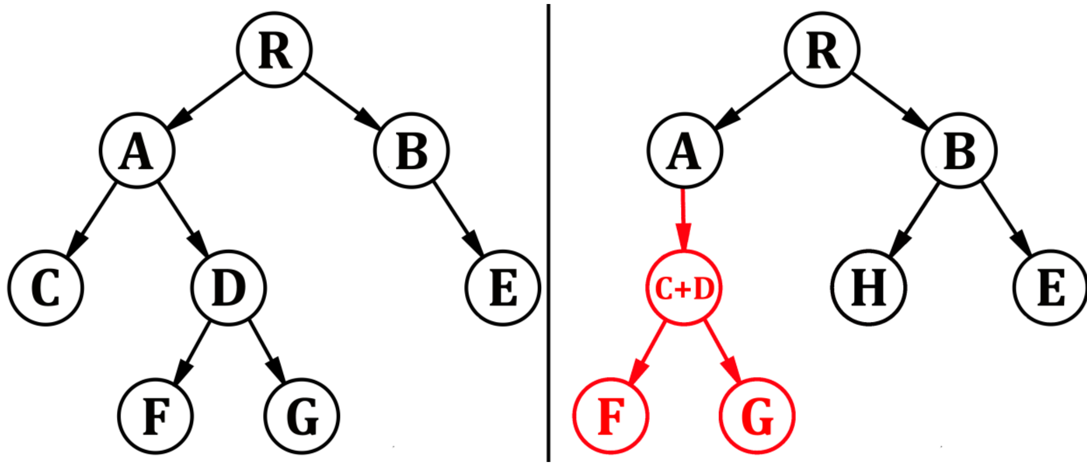

Figure 18: Example of Merge node operator. Children of both selected nodes are concatenated together and added to the merged node. The new node type that is matching the stronger node of the two selected and the power of the nodes is summed up.

Split node operator works in a similar fashion as the merge node operator. It chooses a node randomly, more biased towards stronger nodes. The chance to pick (n_i) is the same as for Strengthen operator.

$$p_{n_i} = \frac{MaxValue + 1 - Value_{n_i}}{\sum_{i=1}^{|N|-1} MaxValue + 1 - Value_{n_i}}$$

Then it creates two new nodes (Figure 19), which will have the same parent as the original. Both of new nodes will have the same type. Let X be a random real number between 0 and 1 (inclusive), then One node's effect value will be the original node's value times X, and the other's value will be the original's times 1-X. Children of the original node will be randomly distributed between new nodes.

This operator directly tackles both redundancy and obviousness in the skill tree, since the player would likely pick the strongest node. We can provide new and more interesting choice by splitting this strong node.

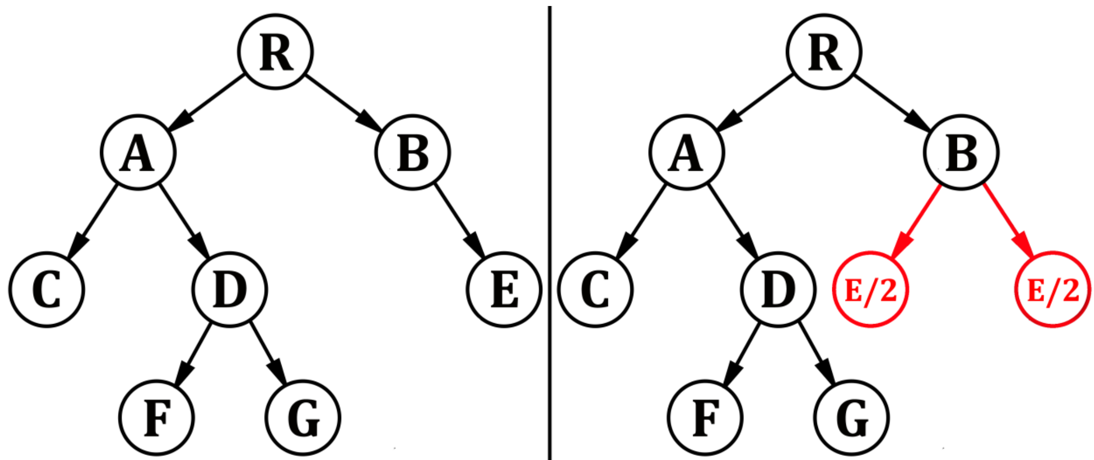

Figure 19: Example of Split node operator. Node chosen for split is split into two new. Both new nodes have the same type, and their power is split with a random ratio. Original node's children are assigned randomly between the new nodes.

Add node operator is a simple operator, which adds a new leaf node as a child of an existing one (Figure 20). The choice of a parent is heavily biased towards nodes with less than 2 children. There is a 50% probability to pick node with no children, a 40% probability of picking node a node with 1 child and a 10% probability of picking node one of the others. The choice within these three groups is random. The type of the new node is chosen at random.

This operator directly reduces redundancy of a skill tree, as it gives more choices to the player. In combination with the strengthen operator, this operator creates viable alternative choices to already viable choices.

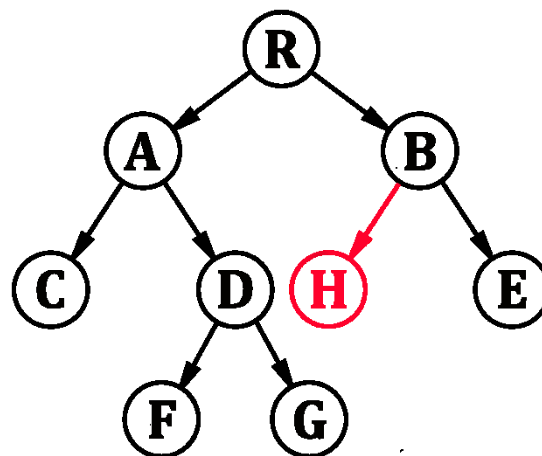

Figure 20: Tree with an added node (red). The new node has a random type, random value, and it will always be a leaf node.

All these operators are used for creating mutation of a tree, which is one of the parts of creating new generations. Another part of creating new generation is merging two trees together to try to achieve the good properties of both.

5.3. Crossover - merging two trees

The selection of entities to merge is made randomly. Let X be the number of trees selected from a generation in the **selection** step. Then we are going to take X random pairs of entities from the said selection of old generation and perform the crossover step using these pairs. All entities created by this step are added as copies to the next generation (original trees are preserved).

Usually, the crossover of two entities in an evolutionary algorithm is a matter of averaging their numeric values or randomizing them to have a similar effect. This is not that simple to do with trees since we do not have a fixed tree structure. Merging two nodes is as simple as averaging or randomly combining their values, but combining their children is not trivial at all. Simply combining the nodes' children is out of the question, since it would introduce just a denser tree with plenty of redundant choices and twice the node count. Therefore, we need a method to combine two sets of nodes' children in a way, that takes a child from each node and then combines is, which suggests a recursive method. The recursive method would be solving the current node pair, and then their children would be paired and solved the same way as current node pair if there is the same number of children. Extra children, which could not have been paired, would be randomly either discarded or added to the solved node.

Even with using recursion, we need a way to handle nodes, which do not have the same amount of children. The naïve but decent way is to have a 50% probability of adding the extra child to the merged node and a 50% probability to discard it (and its subtree). I chose to go this way since it accomplishes everything I need. Let us describe the crossover process step by step.

The algorithm is recursive, which means we will handle one node pair at a time starting from the root. For every node ***pair A***, we first create a new node with type randomly chosen between the types of the nodes in the ***pair A***. Then we randomly choose between effect value of the first node (25%), effect value of the second node (25%) or their average (50%). After that, we create as many pairs of nodes, where first of the ***new pair B*** is going to be a child of first of the nodes in the ***pair A*** and the other of the ***new pair B*** is going to be from the other node's children. The nodes which remain unpaired are the *extra* children (Figure 21). These *extra* children will be all the remaining children of only one of the nodes from ***pair A***. For all ***new pairs B***, there will be a recursive call of this algorithm, and each result of this call is added

as a child of the new node. Then the *extra* children are randomly discarded or assigned to the new node (both with 50% probability). Then the algorithm returns the new node.

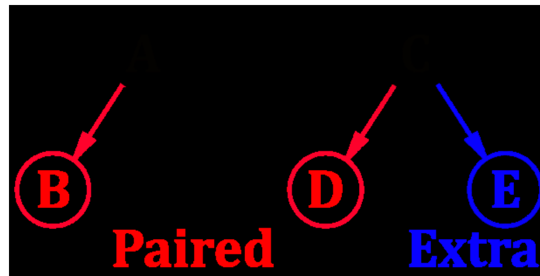

Figure 21: Example of pairing nodes during merging. Here we are merging A and C, and we are pairing their children. B and D get paired, as they are the first children of their parents. E is left as extra since there is no child of A left to be paired.

```
Node MergeNode(Node one, Node two)
{
	var solvedNode = new Node
	{
		Children = EmptyList,
		NodeTargets = PickRandom(one.NodeTargets,       //type of node
					two.NodeTargets),
		NodePower = PickRandom(                         //effect value
				(one.NodePower + two.NodePower) / 2,
				one.NodePower,
				two.NodePower)
	};

	var maxChildren = Max(one.Children.Count, two.Children.Count);
	var minChildren = Min(one.Children.Count, two.Children.Count);

	//recursively merge common children
	for (var i = 0; i < minChildren; i++)
	{
		//recursive call for pair of children
		var mergeNode = MergeNode(one.Children[i], two.Children[i]);

		solvedNode.Children.Add(mergeNode);
		mergeNode.Parent = solvedNode;
	}

	//randomly discard or assign extra children
	for (var i = minChildren; i < maxChildren; i++)
	{
		if (one.Children.Count <= i)
		{
			if (Random.NextDouble() > 0.5)
				solvedNode.Children.Add(two.Children[i]);
		}
		else
		{
			if (Random.NextDouble() > 0.5)
				solvedNode.Children.Add(one.Children[i]);
		}
	}

	return solvedNode;
}
```

Snippet 2: recursive node merging method, which averages current conflicting node and merges their children by calling itself recursively if both nodes exist, else it randomly adds or discards the extra nodes.

6. Proposing solution algorithm – Fitness function

In this chapter, we will talk about the fitness function, which is the function used in the **fitness step** of the algorithm. The fitness function should return ratings for the tree for us to compare it with others. This rating will be split into different values based on the specific property of the tree.

Definition: We will define our **fitness function** as a function $f: T \rightarrow R^n$ where T is space of all skill trees with structure defined in section 4.1. The returning value of this function is an n dimensional vector of real numbers, which represent all different ratings of a skill tree in our thesis.

The perfect fitness function for our algorithm would require complete simulation of all cases that might happen in the game. From this information, we would be able to determine all the needed ratings.

Given our game system, explained in Figure 16, we would have multiple branching factors. We would have the branching factor of the skill tree choice, which would depend on the skill tree, but it would be limited by the number of nodes in a tree. The worst case is to have a root with only leaf children. In that case, with N nodes in the tree and M being the number of obtained nodes per scenario, we would have $\binom{N}{M}$ possible choices. However, this needs to be multiplied by $M!$ to account for the possible order of how the nodes were obtained.

Then we need to approximate the complexity of the encounter in the simulation. The complexity is highly dependent on how many cards are targeted and how many enemies there are. Every turn we get 5 cards, and in every encounter, we face up to 7 enemies. This means that in the worst case, we have 7^5 possible plays in our hand. We can omit the order of the cards, as their order can be changed for the same effect. We still, however, need to put this number to the power of turns player in the encounter, since the outcomes of every turn might differ. Therefore we end up with encounter simulation complexity of $(7^5)^T$ where T is the number of turns (this number usually ranges between 1 and 10 because of our game design.

Additionally, we need to account for the order of the cards in the deck, and there will be $\frac{15!}{5! \cdot 5! \cdot 5!}$ possible orderings. Therefore with E being the number of encounters in a scenario, we end up with the following formula of complexity for complete simulation.

$$\binom{N}{M} \cdot M! \cdot E \cdot (7^5)^T \cdot \frac{15!}{5! \cdot 5! \cdot 5!}$$

This is the worst-case scenario, and the complexity for average scenario would be looking much better since there are many area cards (which do not require target) and there are not many encounters with 7 enemies, but the average would be between 4 and 5. Also, the skill tree's structure will be much different in the average case, and the complexity of choice would be lower. However, the complexity of such simulation is still too large to be used in the fitness function, especially because it covers only one skill tree and only one deck. Therefore, we have to make compromises and rely on approximations.

Given our changing environment (different scenario every generator run) and the variety of combinations, best-proposed fitness function was approximating simulation on all possible choices in the tree and aggregating them to a single value (fixed amount of values).

To simplify the simulation, I still decided to run all possible tree combinations, but in terms of simulating the encounter, we have simplified the simulation greatly. Instead of searching all possible cases and tracking every choice and variable element, we average the targeted and area damage of the deck every turn. Doing this while running a simple greedy heuristics, that puts the average targeted damage on the minion, which has the highest attack to health ratio, makes the simulation performance much better and enables its use in repeated skill tree evaluation.

Definition: Approximating simulation is a function $f: (D, E) \rightarrow Z$ where D is a set of cards (Table 2: Card's definition), which are already buffed by a combination of nodes, E is a set of enemies where enemy is defined by attack value and health value. E effectively means $(Z, R)^n$, where *n* is a number of enemies. Returning value of this function would be the turn number if the simulation succeeded in beating the encounter. Otherwise, it will return the sum of remaining hit points of the enemies, which are still alive (hit points above 0).

On top of that, we can use reference decks instead of all possible decks and also average the turns instead of having a huge branching factor during the encounter because of the number of possible decks. Both of these increase the algorithm performance greatly. We have chosen this kind of function because it is close to the perfect simulation, but it is not as complex. Therefore, it can be used in the fitness function.

We will need to select a few representative ratings, which will make the skill tree viable to use and which will make it interesting to the player. In order to make a skill tree viable to use, we will need to control the win ratio of all combinations in every given state of the game. This raises the need for *WinRateRating* which will represent, how well does the skill tree do against the specific encounter over all of the possible combinations of nodes achievable in that state of the game. This will only make sure not to fall under the Flow zone into anxiety, but will not give us upper bounds. That is why we introduce a few ratings, which control the length of the encounter since the length of the encounter represents its challenge well. These will be looking for average, median, and most frequent case (MFC) of the winning turn. Their matching ratings are called *AverageWinTurnRating*, *MFCWinTurnRating* and *MedianWinTurnRating*. In order to control the amount of redundancy in the tree, we introduce the *BalanceRating*, which will check the sizes of subtrees of children of nodes to rate more balanced trees better than unbalanced ones. Finally, we introduce the *GraciousLossRating* to combat obviousness in the tree. This rating will rate trees, which will have closer losses higher than a tree with total losses, which were not close to winning at all. Ratings will be explained in detail in section 6.5.

The approximate simulation gives us the result for each deck, each encounter of a scenario and each combination of nodes for a given encounter. This gives us a three-dimensional array of results from the approximate simulation. For every combination of nodes for given deck and encounter, we will convert the one-dimensional array into chosen ratings. Then we will be left with a two-dimensional array of ratings (per deck and encounter), and these ratings will need to be aggregated into a single set of ratings, which we will finally return as a result of the fitness function.

First, we will need some reference decks to cover most of the likely deck types that can be in the game. We will also need to represent the choices a player can make in the skill tree for proper fitting and how we address the combinations of choices any player can make. After addressing that, we will describe the approximating simulation. Then, we will talk about extracting the results and splitting them into different ratings and how these results will be aggregated over all decks and choice combinations.

6.1. Relevant input

For this part of the algorithm, the relevant input of fitness function is the data about encounters and weights given to our ratings.

Property	Type	Note
Encounters	Set of pairs (Integer; Encounter)	What encounters will the game contain at what numbers of nodes available
WinRateWeight	Real number	Weight of the WinRate rating
AverageWinTurnWeight	Real number	Weight of the AverageWinTurn rating
MedianWinTurnWeight	Real number	Weight of the MedianWinTurn rating
BalanceWeight	Real number	Weight of the Balance rating
MfcWeight	Real number	Weight of the MFCWinTurn rating
GraciousLossWeight	Real number	Weight of the GraciousLoss rating

Table 6: Relevant input for the fitness function

Encounters are an ordered set of pairs of a set of enemies and the number of available nodes for that encounter.

6.2. Reference decks

Since the deck of cards is not in our input, we have to account for different deck types, which would reflect most of the possible decks that could play the encounter. Therefore, I created a set of reference decks, some determined, some random based on the possible cards that might appear in the deck (all of them listed at Appendix A – Reference Decks). There will be few decks filled with only one card type, few decks with 5 non-targeted cards and 10 single target cards and few completely randomly made up of 15 cards from 8 possible card types. The evaluation will be done for each deck and then aggregated (more on rating aggregation of data pod).

6.3. Choices

There are two types of choices to reflect in our simulation. One type of choice is picking a node in the skill tree. Another type is the choice of cards to play every turn. First, I will describe the choices of the tree and the important details that relate to it. After that, I will describe how the choices of playing cards are handled during the encounter.

The tree's evaluation must represent the choices that the player can make (i.e. in the form of having a method to return picked nodes). Therefore, **all the possible choices and combinations of choices should be evaluated** to make an accurate evaluation of a tree. This is because if the player chooses a different combination of nodes than we evaluated, he might end up with experience significantly worse because the Player-power given to him is too low or too high to contrast the challenge properly. Additionally, we might miss some cases of redundancy or obviousness in the skill tree, if we do not evaluate all combinations.

Another aspect of the tree we must reflect is the continuity of choices. If there is a node, that makes the player very strong, he needs to have a way to reach it in order to pick it without losing the game beforehand. Therefore, only winning choice combinations can be developed further, since losing combinations would not progress further in the game. This means we can cut these choices from our calculations. This prevents good rating of all combinations with an incredibly powerful node, which has no way of reaching it because the player fails before he can.

Reflecting player choice of cards played during an encounter can be tricky since you need to approximate the behaviour of the player. Great players will beat significantly harder encounter than a bad player. However, as we have discussed in section 2.2, the difficulty is relative to the player. Therefore we can choose what kind of players we want to have the most optimal experience. We have gone with targeting the decent players to have the optimal experience and therefore, we have chosen to simulate the player choices in encounters with greedy strategy heuristics.

We might end up in a situation, where the results of choices are succeeding only in part of the cases (not all combinations of nodes beat the encounter). In that case, it would be a good idea to ensure that some amount of combinations succeeds. This amount depends on whether we have multiple tries or whether the failure is permanent. If the loss is permanent, then we should aim for higher odds of success in an encounter so that most decent or better players succeed. Otherwise, we can get away with a lower chance of success, as the player can try repeatedly. Since our loss is permanent and the player would have to start the scenario over if he failed during any of its encounters, we will aim for a higher success percentage.

6.4. Approximating simulation of encounter

For our evaluation, we went with an approximating simulation, as it seemed the best compromise between accuracy and speed. Before the simulation, we take the deck and upgrade it based on the combination of nodes that was chosen.

Since the deck gets shuffled every start of an encounter, it is better to, instead of running multiple times for different card orders, average the damage player can deal every turn. Then we iterate player damage during turns and enemy damage at the end of every turn until player died or defeated the encounter. Average area damage is done to enemies every turn, while average targeted damage is done to the enemy with the highest attack to health left ratio. Reason for the targeted damage heuristics is described in the previous section. If the simulation failed, negative of the sum of remaining health of enemies was returned to represent, how close, was the lost encounter. If the simulation succeeded in beating the encounter, the number of the turn, in which the simulation was able to defeat said encounter, was returned.

Costs of the cards were taken in account and if the cost would surpass the resources of the player, effect of all cards of that resource type would be reduced to compensate the inability to play all of them in one turn.

6.5. Rating types

There are a couple of different ratings, which I have defined for each tree. These ratings represent the performance of a tree well because what we need from a tree is to provide enough power to beat the encounter, not to have redundant choices, and not to give too much power.

All ratings are designed to range between negative infinity and 1 with negative values representing very likely unusable trees. Such values can be achieved by having a tree that fails at all encounters. The way these ratings are calculated is described in the next section.

Average, **Median** and **MFC** win turn ratings all contribute to a reasonable game length. The number of turns in a winning game was estimated to be between 3-5 to be reasonable in length and difficulty. The rating is then given based on following formula for each of said ratings.

$$f(x) = \begin{cases} -x, & x > 5 \\ x, & x \leq 5 \land x \geq 3 \\ x, & x < 3 \end{cases}$$

WinRateRating is by far the most important rating of a tree. It represents, how much is the desired win ratio of the combinations (for each deck and encounter) of the actual win ratio. This rating should always be at a high weight, as it ensures a baseline performance of our tree.

Let W be the win ratio of all possible combinations of nodes in a skill tree for given deck and encounter and let R_i be the total remaining hit points of all enemies where I is the set of indexes of combinations and $i \in I$ is the index of combinations. Finally, let T be the target win ratio. Then *WinRateRating* is computed with the following formula.

$$f(x) = \begin{cases} \frac{1-W}{T}, & W > T \\ \frac{W}{T}, & W \leq T \bigwedge W > 0 \\ \frac{1}{\sum R_i}, & Otherwise \end{cases}$$

The target of this function will be 70% in this thesis. This was decided after manual development testing and will be validated in chapter 8.

BalanceRating was introduced after a couple of unwanted artefacts were discovered in generated trees. Trees that were generated were significantly unbalanced, with cases that contained nodes with children that had subtrees of sizes 1 and 7 and similar cases. This rating is based on sizes of subtrees of nodes' children and weighted more if the node is closer to the root.

GraciousLossRating was introduced to balance the outcomes of the different encounters. Before introducing this rating, part of the tree was basically designed to be 'losing' and therefore had very little to no power. To change this, I have introduced *GraciousLossRating* to make the losses as close as possible. The closer the loss was, better the rating. Results of the generation were surprisingly much better than I originally expected, especially in combination with a higher target of win ratio.

6.6. Aggregating results

Another secondary goal is to minimize outliers. This means to value trees with more consistent performance better than a tree, which has a perfect performance in most cases but fails completely at few. We can tackle this problem by choosing a

proper aggregation method, which would highlight these outliers and reflect them in the aggregated rating more than just a plain averaging.

Therefore, for aggregation of ratings of all encounters for each deck, each encounter and each choice combination, I chose root mean square distance from number 1. The choice fit with the range of different ratings and the aggregation method is biased against outliers more than just an averaging the results.

I have experimented with different combinations of aggregations, like median values, maximums, averages and their use to determine the performance of the skill tree. However, the results were only marginally better and only in specific cases. Because of this, I have chosen to stick with root mean square as the only aggregation function.

7. Implementation

This chapter will talk about the implementation of the game and the generator for it. I will explain the reasons behind my choices of implementation and give examples and snippets of the code.

The game was done using Unity with C# as its scripting language. This choice was made out of simplicity and licensing. The generator was also implemented in C# using .NET Framework 4.6.1. This was more of a pragmatic choice.

Generator used following NuGet packages:

Name	Note
morelinq	More advanced LINQ methods
Newtonsoft.Json	Json serialization and deserialization
System.ValueTuple	Dependency of morelinq

Table 7: List of NuGet packages used in the solution

The goal was to make an implementation, which would be able to work in parallel, in order to maximize performance, while preserving sequential approach concerning generations. Therefore I wanted to parallelize as much as possible without too much overhead.

Since the first implementations were consistently returning artefacts, which were fixable very quickly and obviously, therefore I decided to implement one last part of the generation, the post processing. In this post processing method, I would delete and replace the obvious redundancies. I will talk about this method later in this chapter.

First, I will talk about the generator. I will explain how I formed the input model and how did I decide to store data (both input and results). Then I will take a look at the whole algorithm, and after that, I will explain parts of it in detail. After that, I will address the thread safety across the implementation. Finally, I will mention the post processing of a tree.

7.1. Relevant input model and data storage

The rest of the relevant input for our algorithm is the number of generations to be run in addition to how many nodes the final tree can have. Reference decks, which

we talked about in the previous chapter, are a fixed part of the algorithm and thus are not part of the input.

Parameter	Type	Note
`Generations`	`int`	Number of generations to run
`NodeCount`	`int`	Maximum number of nodes

Table 8: InputModel's properties.

To address storing input and result data, we have chosen the JSON format. This allows us to simply deserialize data into our objects directly and makes loading and saving data very simple.

7.2. Whole algorithm

Before I go into details about parts of the algorithm, I wanted to have a quick overview, how does my algorithm look like. As seen in (Snippet 3), the core of my algorithm follows the Flow diagram of our GA introduced in chapter 4. The stopping criterion, as mentioned before, is the only number of generations, which is why the main part of the algorithm is in *for* cycle.

```
NodeTree Generate(InputModel inputModel)
{
        var currentGeneration = GetRandomTrees();
        // perform number of generations specified
        for (var i = 0; i < inputModel.Generations; i++)
        {
                var newGeneration = EvolveTree(currentGeneration);
                // perform fitting
                var rated = newGeneration.Rate();
                // adds old generation to new one for selection
                rated.Add(currentGeneration);
                // sorts trees by fitness descending
                rated = new List<RatedTree>(NonDominatedSorter.NonDominatedSort(rated));
                // take only specified number of trees
                currentGeneration = rated.Take(inputModel.GenerationSize);
        }
        // choose final entity to return
        var final = WeightedMax(currentGeneration);
        // post process final entity before returning
        PostProcess(final, inputModel.PowerMarks.Keys.Max());
        return final;
}
```

Snippet 3: Overview of the algorithm in pseudo code.

7.3. Mutation + crossover step

In the mutation and crossover steps of the algorithm, we take the list of entities, which were selected from the last generation, and apply all operators to all entities and then performs crossover on random pairs of the entities from the last generation.

All operators are under the `IOperator` interface, which has one method. This method is called mutate and takes an entity and input model as its parameters. All the operators always create a copy of the entity given to it and then perform mutation with probability given in the InputModel. If there is no mutation to be performed (mutation does not happen due to probability, or no other suitable part of the tree was found to be modified), null is returned and removed from the list of new entities after all operators are applied.

During the crossover, the amount of random pairs of entities is taken from the selected generation. This amount is equal to generation size after selection. After pairs are selected, copies of entities in every pair are given as arguments to the merge method.

```
NodeTree Crossover(NodeTree one, NodeTree two)
{
       // merge roots of two trees
       var mergeNode = MergeNode(one.Clone().Root, two.Clone().Root);
       // create a new tree object and initialize it
       var newTree = new NodeTree(mergeNode);
       newTree.InitTree();
       return newTree;
}
```

Snippet 4: Method for crossover. The main process is done in the MergeNode method, which recursively merges nodes together. After the merging is done, a new tree object is created and initialized.

Crossover method calls the `MergeNode` method, which recursively merges nodes together starting from the root. This method was described in subchapter 5.3.

7.4. Fitness step

The fitness step is being done using a `RateTree` method. This method is split into two main parts. The first part consists of performing the approximating simulation of the game and saving results into a three-dimensional array. One dimension is for different reference decks, one is for all different game encounters, and the last one is for all possible combinations, which can be picked in the skill tree with a specified number of available points for each encounter. The second part is focusing on extracting the results from the array and rating the tree based on those results.

For every deck in the reference decks and every encounter in InputModel all possible combinations of chosen nodes are found. Since the encounters are sorted in their chronological order, only combinations that succeeded in all previous

encounters are developed to be simulated in following encounters. Then, for every combination, the deck is upgraded by the chosen combination of nodes and the approximating simulation with the upgraded deck and the corresponding set of enemies is started.

In order to keep the simulation as fast as possible, we calculate average damage per turn for both single target cards and non-targeted cards before the simulation begins. We account for the player not having enough resource by reducing the average damage by the ratio of used resource and resource available (if above 1). Then the turn iteration begins. Simulation damages all the monsters by average non-targeted damage and damages a living enemy with the highest attack to health ratio. This heuristic provides a balanced approach when simulating the result of an encounter.

The second part of the fitness step is to extract results from the three-dimensional array. For every deck and encounter, we have an array of combinations' performance in the simulation. After manual review of the game, we have decided to aim for 3-5 turn encounters, as shorter felt very simple and longer felt tedious.

7.5. Thread safety

To improve the performance of the algorithm, I decided to do the most performance heavy task in parallel to greatly improve the execution time. This is the reason why most of the objects that are used and changed in each generation have a definition of a Clone method, which creates a deep copy of itself. This method is used during every potential place that could cause inconsistency and during modifications like operations and crossover (since we are creating new objects).

7.6. Post processing

After the best entity is chosen, post processing is done to prevent easy to spot and easy to fix artefacts, which are unwanted. Such easy to spot artefacts are nodes with 0 value or cases with obvious choices, which, as a result, will never get picked. Every node gets its children searched for subtrees consisting only of one type of node, and if there is a subtree, which cannot be fully selected, it is converted into a path which is cut off at the maximum depth reachable in-game.

8. Validation

In this chapter, we will talk about the validation of our skill trees. In order to validate generated skill trees, we need to choose a way to validate them. As mentioned before in Analysis, validation can be done either automatically or manually. We have chosen to go with automatic validation for the sake of time and resources.

That means we will need an algorithmic solution to trying the game and to see if a player would be able to play it comfortably. To do just that, we have chosen to implement bots to play our game. However, we needed our bots to represent the players that would be playing the game, which raised a need to have multiple different types of bots that would play our game to cover most of the spectrum of players.

We needed bots that would be objectively good and bad at the game, but we needed to give them the same environment as the players would have. Therefore, the only information for the bot to decide would be cards in hand and enemies against him. However, there will be one exception with a bot, which is called *Lucky.* This bot will have information of his next hand and greedily decide based on next hand's best play as if a player would be able to correctly guess what cards he will draw, hence the name.

First, we will talk about our expectations of the results and form a hypothesis. Then we talk about the design of the validation. We will describe all different bot types and their purpose in the validation. We will also describe how different bots reflect player's choices in both encounters and skill tree picks. After that, we will address collecting and storing data from the validation. This includes talking about what data we collect and what format did we choose for the data manipulation and interpretation.

8.1. Hypothesis, expectations

In order to reflect different players playing the game, we needed multiple different bots to play the game. Some need to be better and reflect players who think about every turn. Some bots need to be dumb to reflect players who either cannot play the game or are just simply bad at it. Some bots need to be somewhere in the middle to reflect a player who thinks about what he is doing but not too deeply.

Therefore, the first and the simplest kind of bot will be the ***RandomBot***. This bot, as its name suggests, will play cards randomly and will randomly end the turn, sometimes even before it played all the cards in its hand, if less than 3 cards are playable in his hand. Targets of targeted cards will also be chosen randomly.

Another bot will represent the decent player, who thinks about what he is doing but is not looking for all possible outcomes and combinations in the game. This bot is called ***GreedyBot***. *GreedyBot* always looks for the card, which will have the biggest impact on the game, killing most enemies or damaging enemies the most out of all cards in its hand. This bot represents the players, who read every card and find the one that does the most and play it.

Even this kind of decent player may not succeed in finding the best card. That is why we introduce ***RouletteBot***. This bot will play the best card with 50% probability, second best with 25% probability, third with 12.5% probability and both fourth and fifth with 6.25% probability. A similar approach is when there are fewer cards in its hand. Best card will be played with 50% probability, reducing this chance by a factor of 2 every worse choice.

To represent the good players, ***BestTurnBot*** was implemented. As the name suggests, this bot will look for the best possible outcome of the current turn. The search of best play is done via depth first search (DFS) of game states. Additionally, another similar bot was implemented called the ***LuckyBot***. This bot searches game states similarly, but this bot will also look at the hand, which it would draw next turn, and greedily selects the best card to play and choose the best outcome after this one card is played.

Next, we needed a way to reflect player's choices in the skill tree, but given our size of a skill tree, we did not need to make specific bots for choosing the skill tree, but rather we covered all possible choices during the scenario.

There are few expectations we have from the validation of the skill tree. First and foremost, it needs to allow players (bots) to beat the game. The better bots should almost always beat the encounter, and generally, they should have lower turns to win than worse bots. Also, the win ratio of all bots should not be 100% to prove there is a possibility to lose. The average turns to win should fall between 2 and 6, preferably between 3 and 5, since that is what we aimed for during the generation.

Therefore, the hypothesis consists of the following statements:

- Bots will be able to lose the game

- Bots will be able to beat the game
- Average turns to win will be between 2 and 6
- Win ratio will be higher for ‘better’ bots
- ‘Better’ bots will average lower turns to win than worse bots

These statements all speak of the viability of the skill tree, the fact that bots will be able to beat the game means the skill tree can give enough Player-power to the player to beat the game and not feel frustrated and anxious. The fact that bot can lose a game means that the skill tree is not redundant and the choices in the skill tree matter (at least to some extent).

If the bots were able to beat the encounter in 1 turn, it would mean that the game is too easy and will tend to be boring. Therefore another statement will be that the average turns to win will be between 2 and 6. **Win ratio will be higher for ‘better’ bots** means that we have implemented the bots correctly and that the game will reward the skill level of the player. The same reason goes for the last statement as well.

8.2. Design of the validation

We will take one of each deterministic bots and 10 of each non-deterministic bots and let them play each encounter for all possible choice combinations in the skill tree.

For each bot run, copy of the deck is made and then buffed by the node combination. Then set of corresponding enemies from the encounter is taken and put up for the bot to play the encounter using a simplified game interface.

When each bot is done playing the encounter (lost, won), it returns a result row, which gives us all the information about the result of the encounter. This includes how many cards were played, starting and ending attack and health of all enemies, seed if relevant, corresponding node combination and bot’s type.

After all the bots are done playing all cases, all result rows are written in a csv file to be manually validated later.

This validation algorithm will be run for skill trees from all variations of our selection method to compare differences between them. The skill trees used in the validation are shown in Figure 22, Figure 23, Figure 24, Figure 25, and Figure 26.

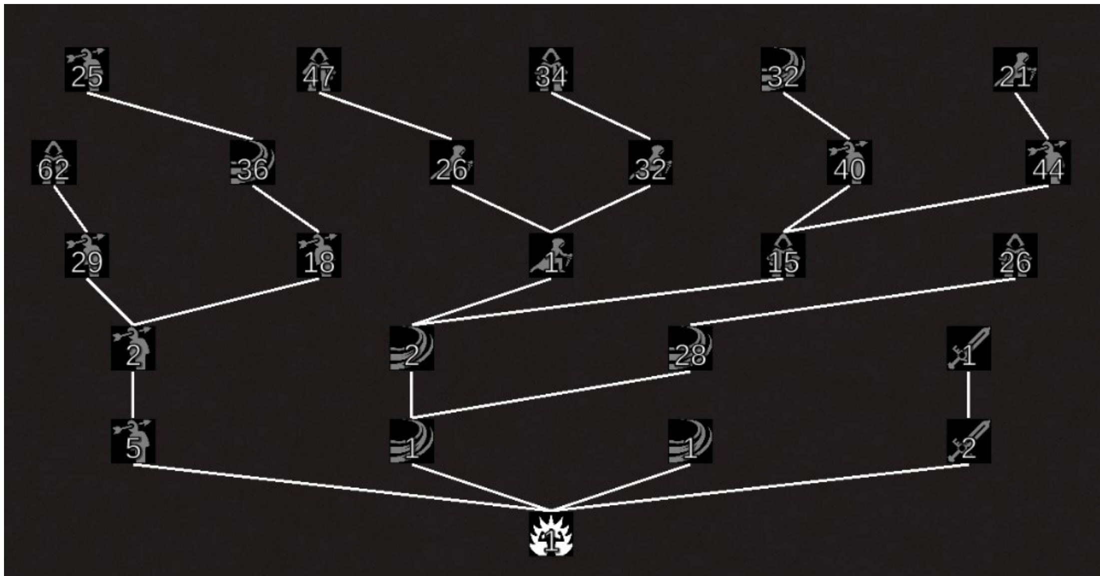

Figure 22: NSGA skill tree

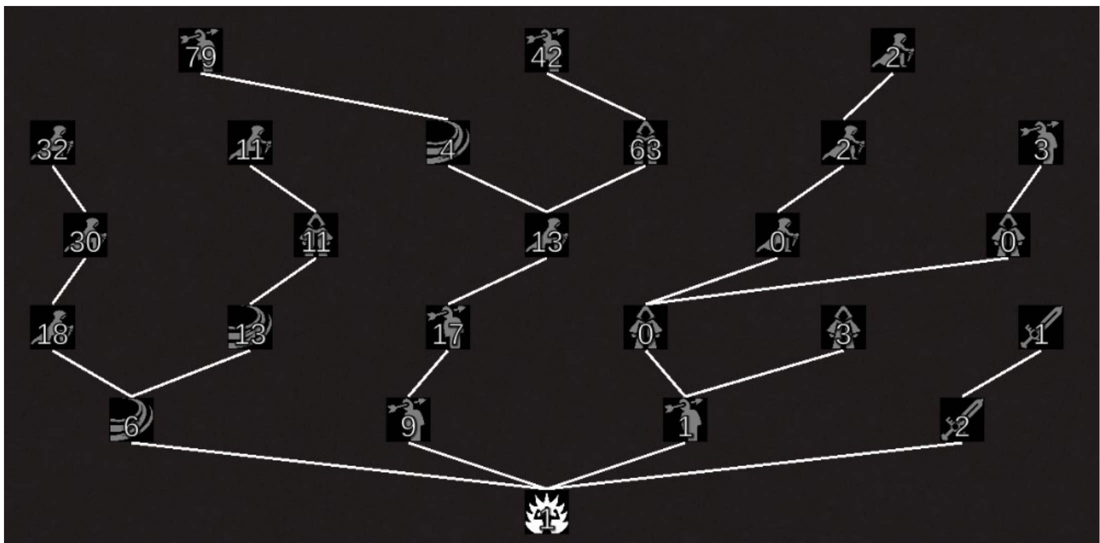

Figure 23: Weighted Evolutionary skill tree

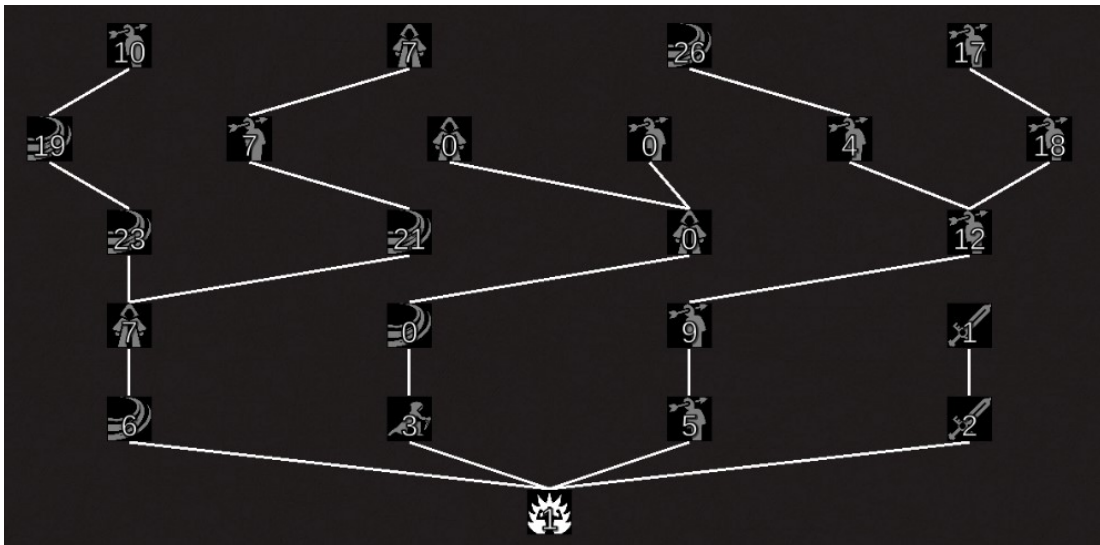

Figure 24: Evolutionary skill tree

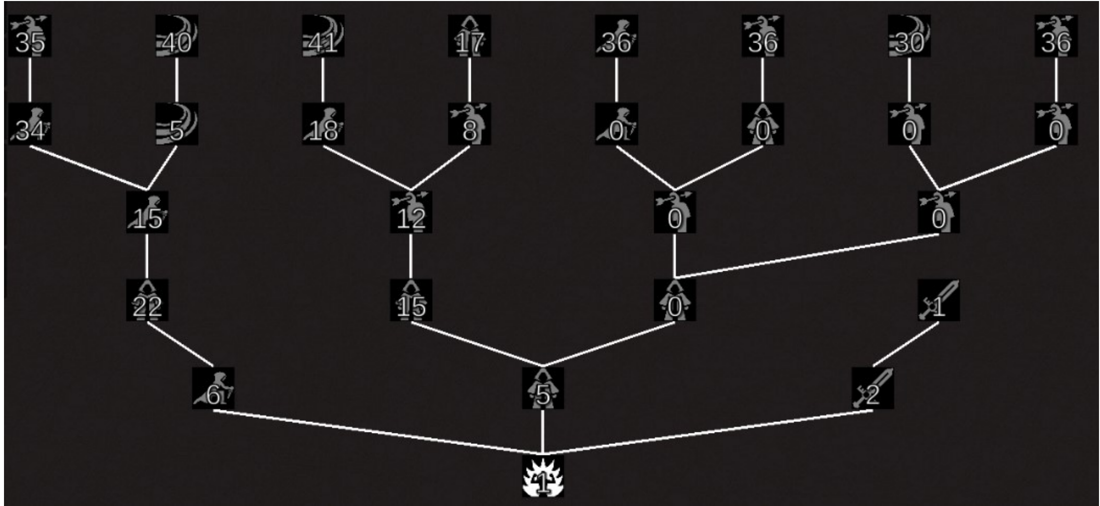

Figure 25: Randomized Hill Climbing skill tree

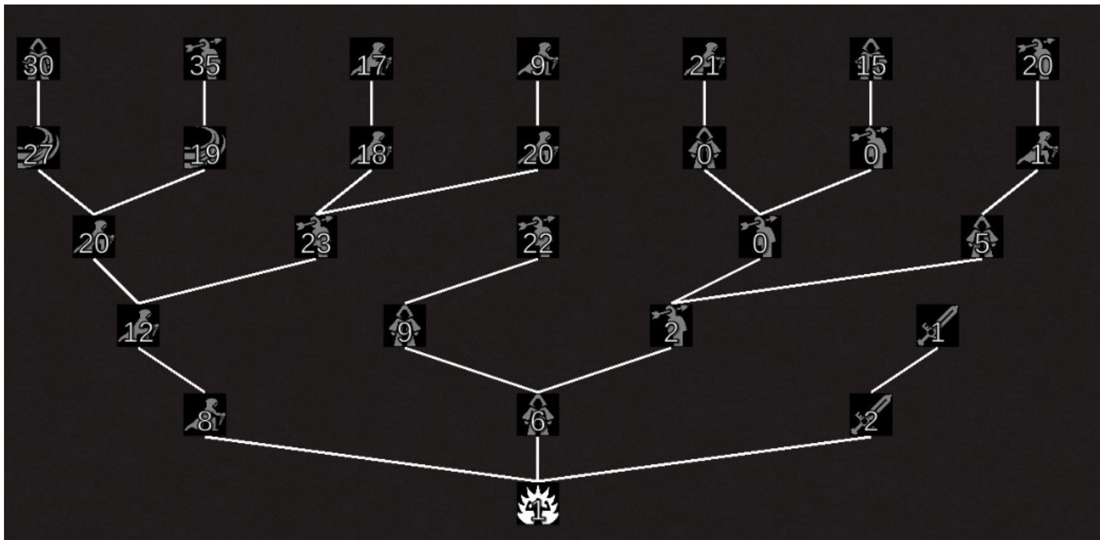

Figure 26: Hill Climbing skill tree

8.3. Result model

Data from bots will be gathered per encounter. Every encounter will create one record of data. In one record of data, multiple values are included.

Name	Note
ScenarioID	Identification of a scenario
EncounterNumber	Number of an encounter in sequence
BotType	Type of bot, which played the encounter
AISeed	Seed of AI choices for reproduction
CardOrderSeed	Seed of card order for reproduction
TreeId	Identification of tree used for validation
NodeIds	Ids of nodes chosen at the time of playing encounter
Won	0 if loss 1 if win
RemainingHp	Remaining hit points at the end of the encounter
Turns	During which turn did the encounter end
CardsPlayed	Number of played cards in total
EncounterEnemies	Attack and Health of enemies before and after the encounter

Table 9: Format of the data record of validation

All played encounters are stored in a CSV format for easy data manipulation since the size allows us to have redundant data. This data format also lets us interpret all results of validations we need as well as confirm or deny statements from the hypothesis.

8.4. Result interpretation + results for hypothesis

Results of the simulation turned out as expected in hypotheses. The bots' win ratio looks like about as we would expect compared with each other, and bots achieved following win ratios across all variations (Figure 27, Figure 28, Figure 29, Figure 30, and Figure 31). As seen on the plots, the win ratio of the bots was very similar between skill tree variations, with all following the same order.

1. LuckyBot
2. BestTurnBot

3. GreedyBot
4. RouletteBot
5. RandomBot

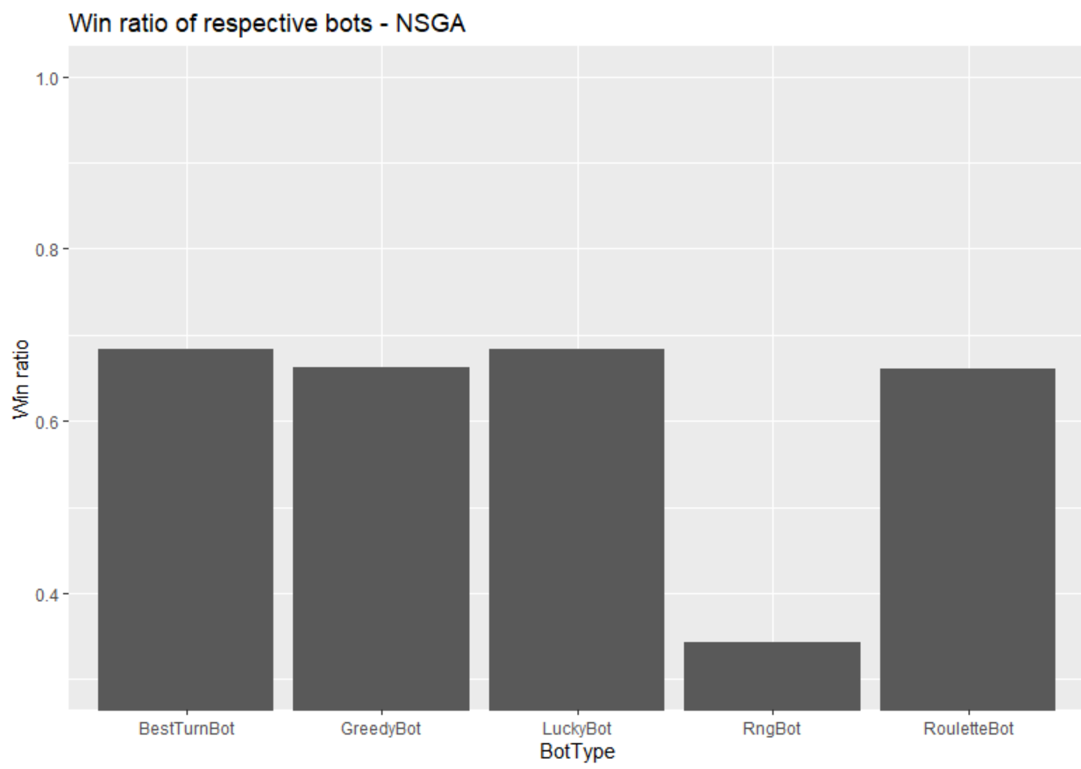

Figure 27: Win ratios of bots with NSGA skill tree

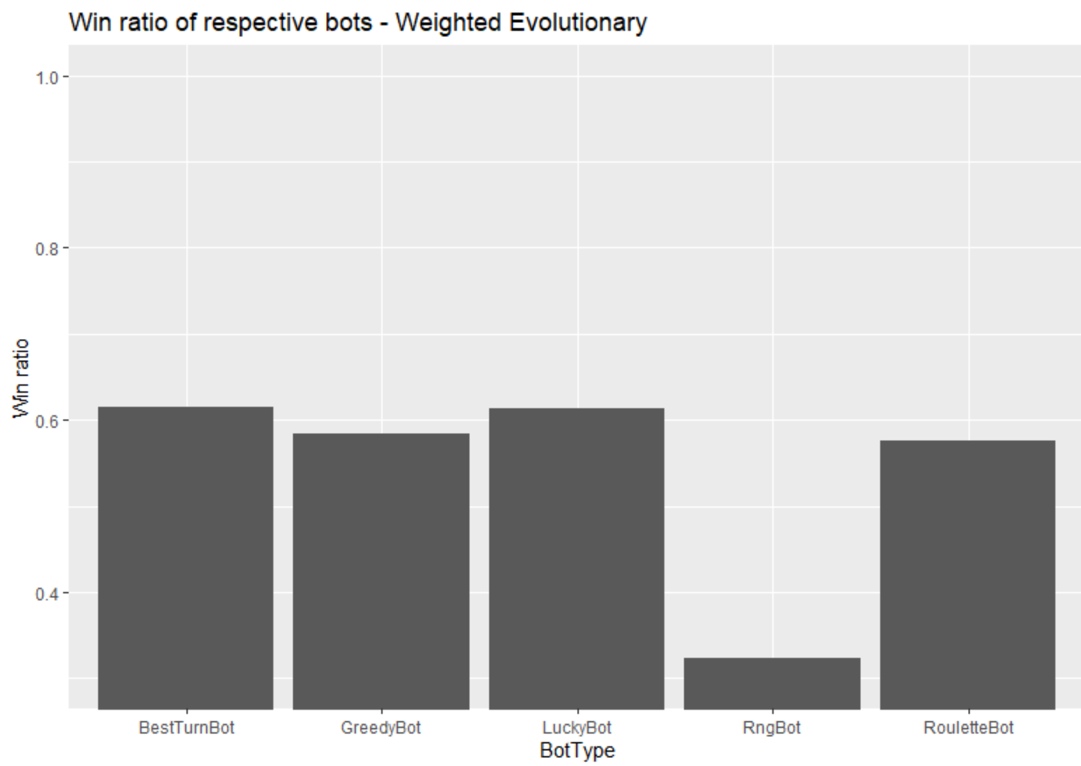

Figure 28: Win ratios of bots with Weighted Evolutionary skill tree

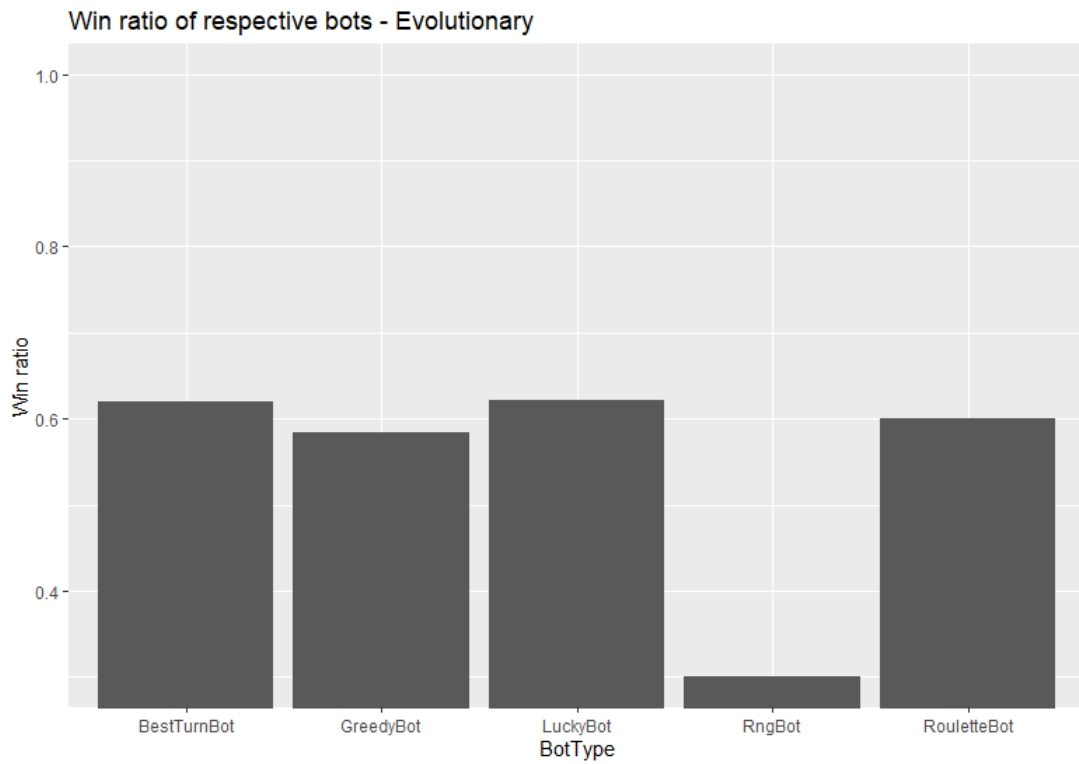

Figure 29: Win ratios of bots with Evolutionary skill tree

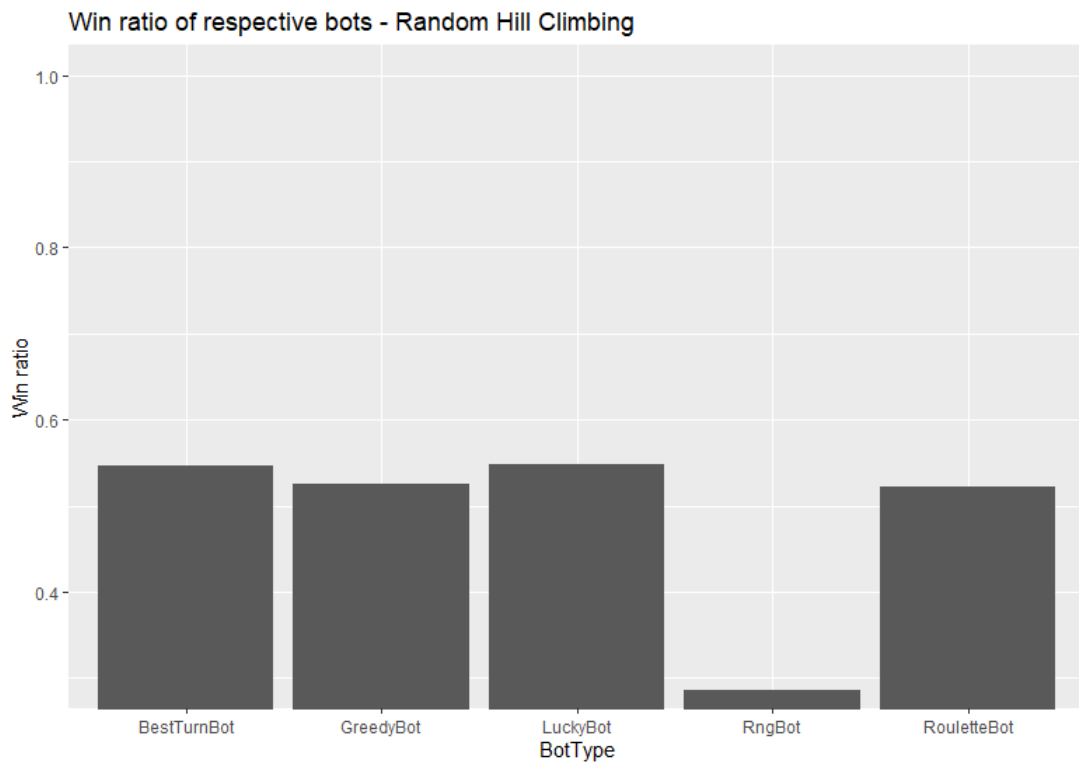

Figure 30: Win ratios of bots with Random Hill Climbing skill tree

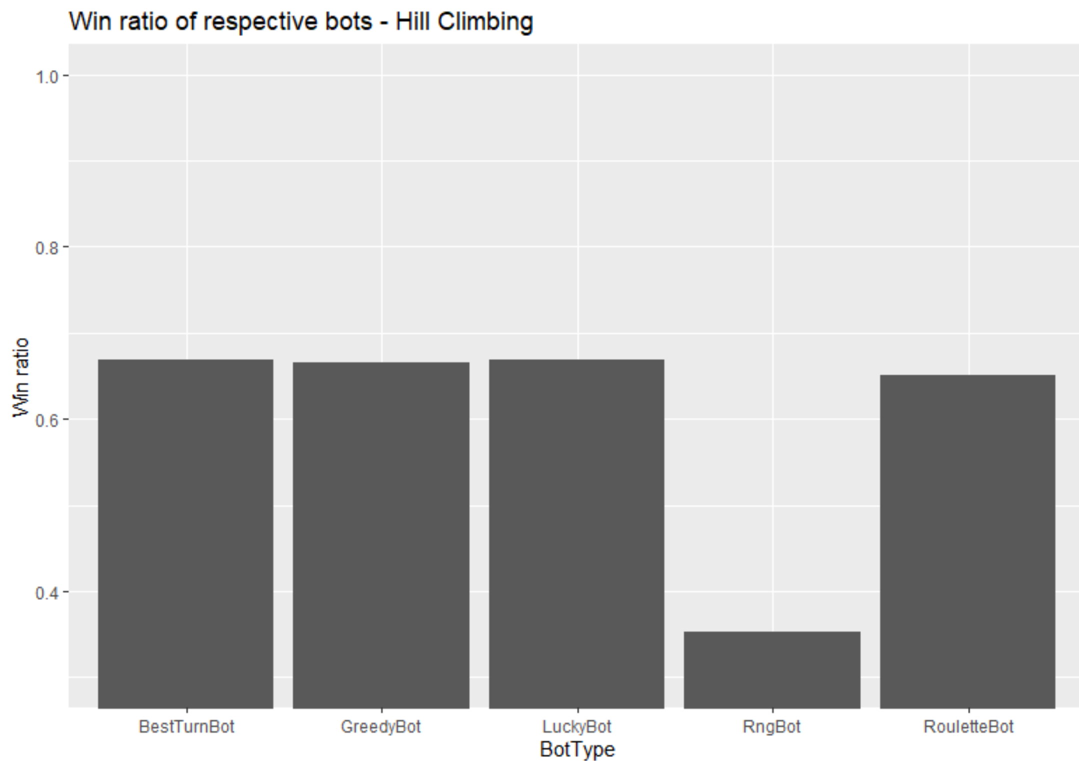

Figure 31: Win ratios of bots with Hill Climbing skill tree

The only interesting observations are NSGA and HillClimbing having higher win ratio for all bots and HillClimbing having very little difference between GreedyBot and BestTurnBot. Another thing was that RandomBot was significantly lower than other bots in terms of win ratio in all cases.

Another thing to look at was the turns to win in won encounters (Figure 32, Figure 33, Figure 34, Figure 35, and Figure 36). The hypothesis stated that the average turns of won encounter should be between 2 and 6. Let us take a look at the average turns to win for all bots. The turn to win was lower than expected, but for all the bots within the range of hypothesis. Additionally, an average turn to win was lower for 'better' bots than for the 'worse' ones.

Sample standard deviations were 0.681; 0.596; 0.631; 0.576; 0.615 for NSGA, Weighted Evolutionary, Evolutionary, Random Hill Climbing and Hill Climbing respectively.

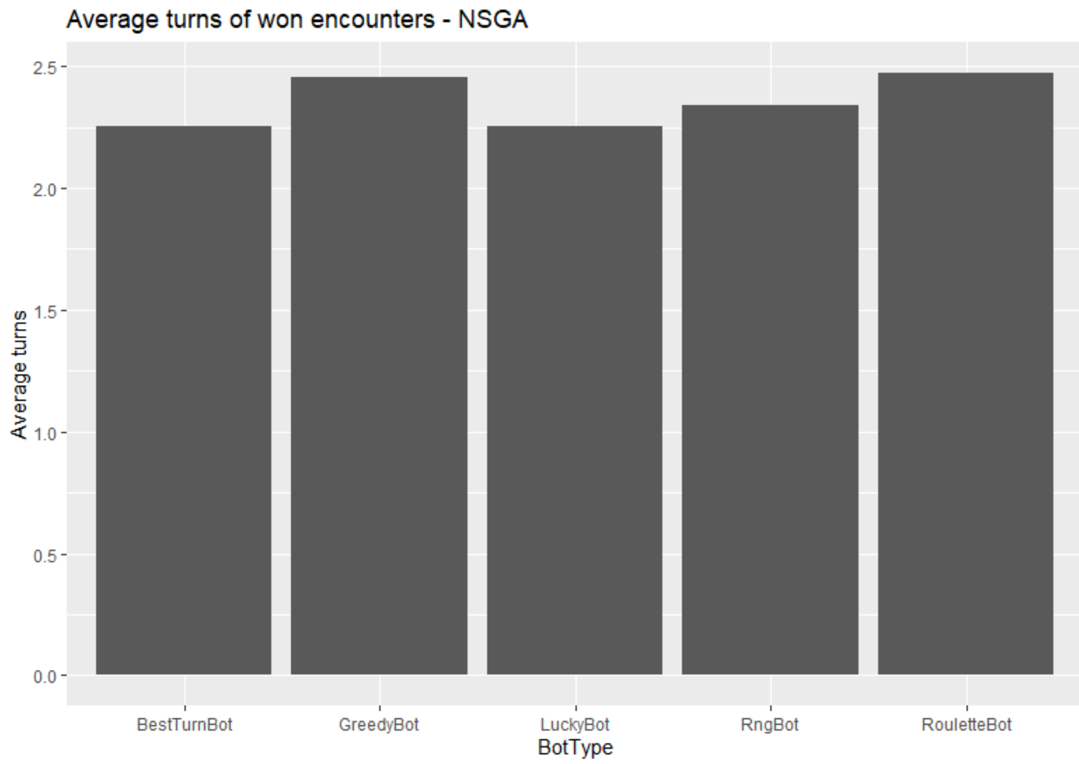

Figure 32: Average turns to win for different bots for NSGA trees

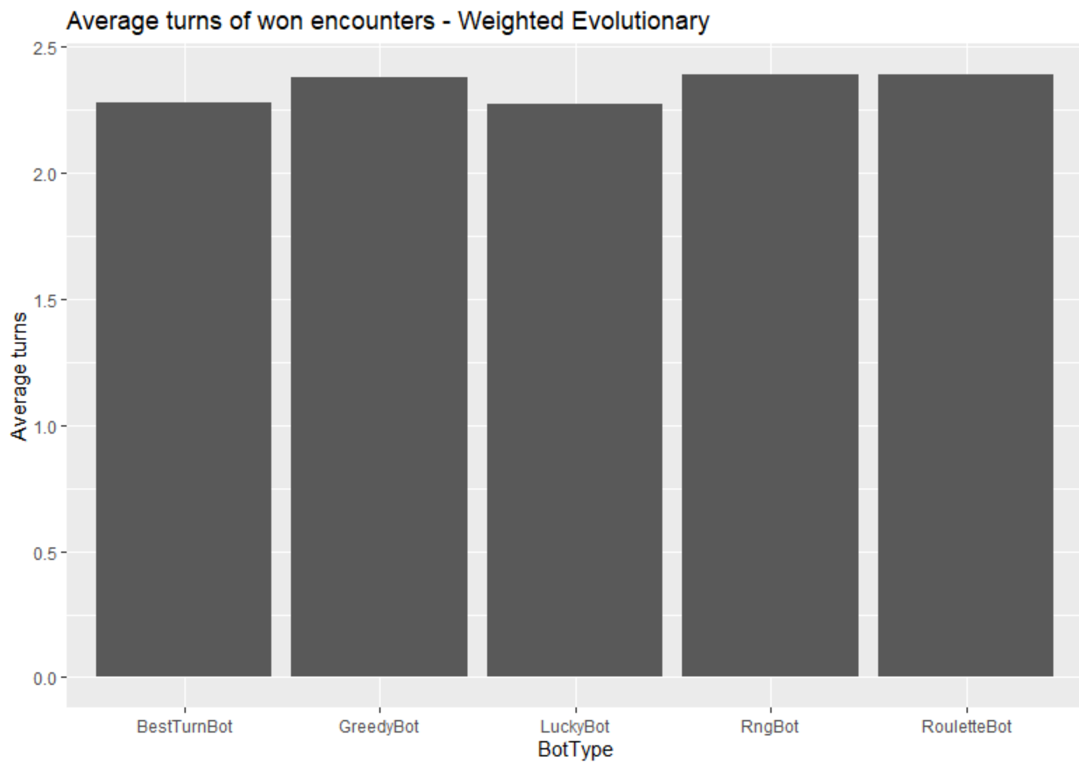

Figure 33: Average turns to win for different bots for Weighted Evolutionary trees

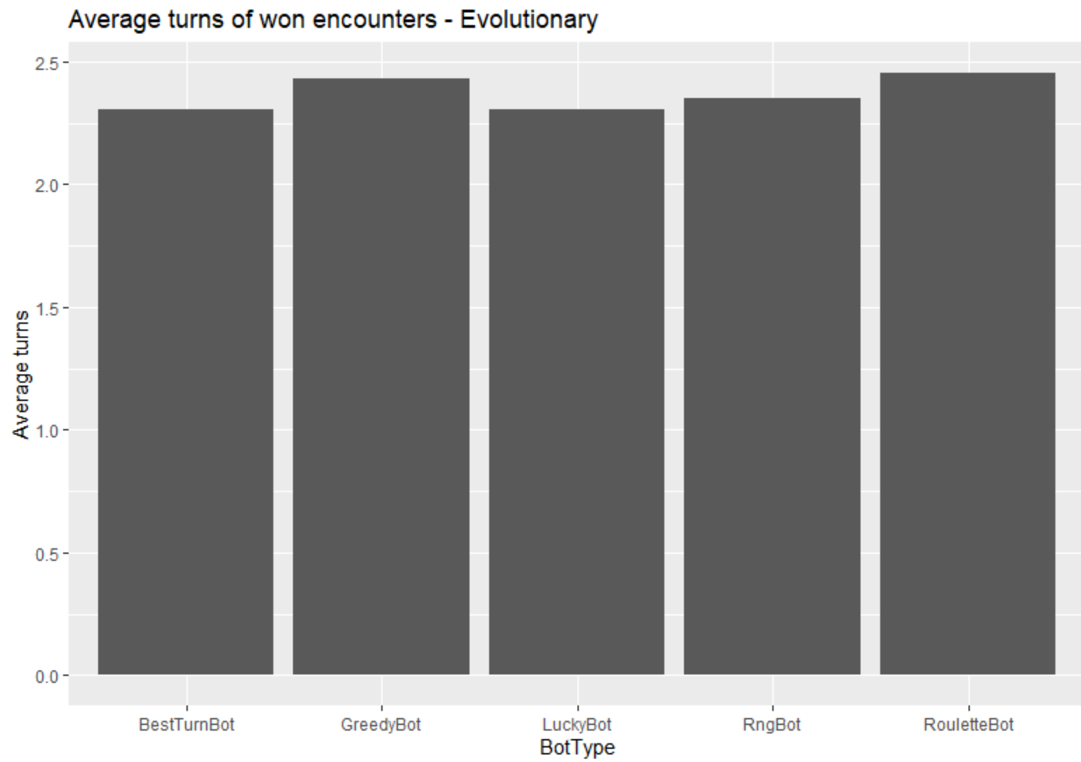

Figure 34: Average turns to win for different bots for Evolutionary trees

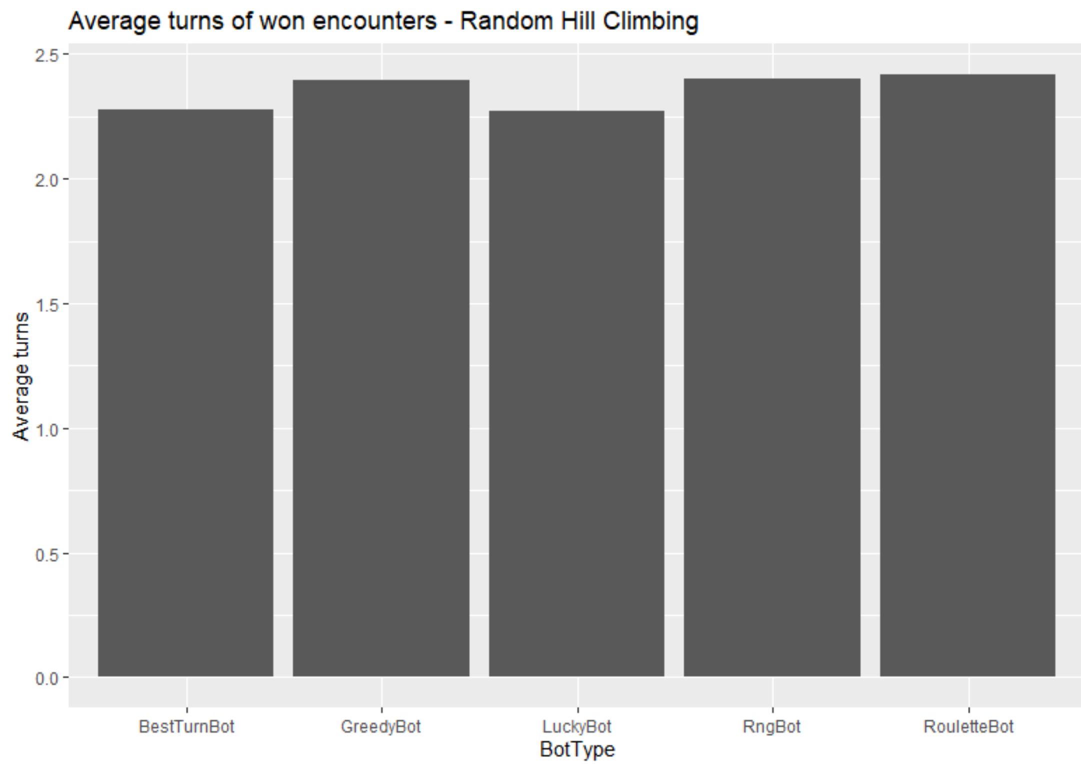

Figure 35: Average turns to win for different bots for Random Hill Climbing trees

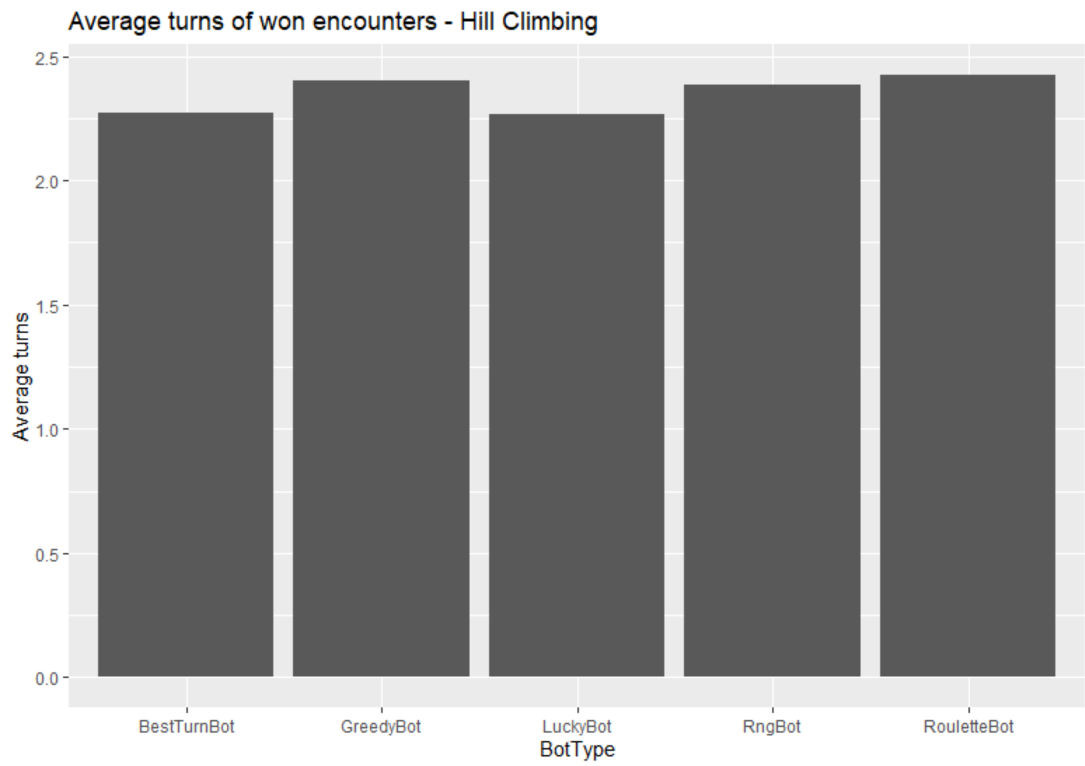

Figure 36: Average turns to win for different bots for Hill Climbing trees

9. Discussion

The data that we collected fully confirm the hypotheses for all variations of the skill tree generation algorithm. In all cases, all the bots were able to both win and lose the game. Bots had different win rates, which were better for the better playing bots like the BestTurnBot and LuckyBot. The RandomBot had significantly worse performance, which is good because it was designed to be a bad player, but the game environment still let it win. This means that the skill tree and game environment reflect the skill of the player, letting him win in fewer cases than the better players.

Both Evolutionary and Weighted Evolutionary had the same artefacts of obviousness (redundant branch of the skill tree). This could be solved by adding another rating to the algorithm in the future that would be ranking trees, which used all nodes at least a few times to win the approximate simulation, higher than others.

The win ratios of Lucky, BestTurn, Greedy and Roulette bot were very close to each other. This was probably caused by the simplicity of the encounters since it is obvious the superior strategy won in very few extra encounters over a very simple but effective one. It is also likely that the difficulty of the encounters was not very hard and therefore was not rewarding the player for the higher skill in the game. The encounters were designed manually, and it is possible that they are not optimal for the validation.

The standard deviation of win ratio of choice combinations in the skill tree is decently big ($0.379;\ 0.395; 0.367; 0.386; 0.348$), which suggests that the outcome of every encounter is highly influenced by the choices in the skill tree. This is good because the redundancy of the skill tree, in that case, is very minimal. This also means that being able to pick the correct combination in the skill tree is as important as having high skill in the rest of the game.

Additionally, this means that a skill tree can be used as the only means of difficulty management for some game designs. A direction of future work could be to define at what conditions the game can be difficulty managed only via a skill tree.

Data, I was lacking and which would be important, is how many trees that underwent crossover operation were selected afterwards and if the crossover function contributed in any way to the final skill tree creation. Data from the validation suggest that the use of the crossover function in Evolutionary and higher algorithms had little impact, but we cannot say that for sure.

9.1. Conclusions of validation

Given the results of the validation process, we can conclude that PCG of skill trees is possible and viable to do using GAs since all of the selection method variants performed rather well during the validation. This also means that the impact of the crossover function, which was used during the Evolutionary, Weighted Evolutionary and NSGA variants, was not that impactful to search the skill tree space.

10. Conclusion

The goal of this thesis was to explore methods of PCG of skill trees in games, first by analysing the requirements of skill trees and their impact on the game, secondly by using the conclusion of analysis to create a generator, which would generate skill trees for our game and finally by validating the generated skill trees.

In this thesis, we have described what a skill tree is and how does it impact games. We have talked about the connection of Flow with difficulty and skill trees with difficulty and Flow. We have analysed, what problems does generating a skill tree bring and what are the requirements we have for a skill tree. We have also analysed multiple bad artefacts that can come up in a skill tree, like redundancy and obviousness.

We have created a couple of variations of generators based on what we have learned during analysis. These generators are based on an adaptation of SGA. There are 5 variations of selection steps introduced in this thesis, which, sorted from simplest to the most complex, are hill climbing, randomised hill climbing, classic evolutionary, weighted evolutionary and finally the NSGA-II.

We have described all mutation operators and the specific way we did crossover with skill trees. After that, we have described the most important part of the generating algorithm, which is its fitness function. Validation of trees during the generation is hard to design because we need both as much accuracy and as much performance to try as many generations as possible in a given time.

After that, we validated our generated skill trees using multiple bots that represent different skill level in our game. We had the bots play the game for all possible combinations in the skill tree and report the result for all encounters, multiple times for non-deterministic bots. This validation returned data, which confirmed our hypotheses for the validation.

10.1. Remarks

At first, during the implementation of the generating algorithm, the approach was very simplistic. I started with only the Hill Climbing method with only the Strengthen and AddNode operators. This proved to be very unwanted as it offered very little control during the generation. The probability of artefacts appearing in the skill tree was also very high. Reason for this was that the skill tree always chose the

best out of the current generation and therefore threw away all the potential skill trees.

That is when I implemented the randomised hill climbing and evolutionary approach. This offered a better result, but I felt that the fitness function should be refined, and I split the rating to different sub ratings and introduced weights to these sub ratings. This improved the playability of the skill tree, but the redundancy and obviousness were still a large factor. At the same, time I decided to refine the operators to include larger variability to the mutation and created the crossover function.

At that moment, I introduced the gracious loss rating, which significantly reduced the obviousness of the tree and balance rating, which made the skill tree look way more appealing. These two ratings heavily improved the skill tree.

After that, we decided to focus more on multi-objective approach and decided to implement the NSGA variant, which did not offer more control, but it offered more consistent results because of the way it selected the individuals for next generation.

10.2. Future work

From this thesis, we have concluded that GAs are a viable way to perform PCG of skill trees for games. However, the game should meet some specific requirements, which were stated in section 2.10, and the performance will go down with the size of the skill tree and its branching factor greatly.

Ideas for future work could be, for example, defining the games, which can use PCG of skill trees formally, based on what we talked about in Analysis. This could bring more insight into what is the PCG of skill trees capable of covering and how it is able to help the development of games.

It is also needed to define a good crossover operation for skill trees since a skill tree is not a binary tree and swapping subtrees of different skill trees has a very uncontrolled effect on the skill tree performance. The crossover function should take advantage of what skill trees do well and try to combine it in a non-trivial way. Maybe it would be a good idea to consider following good combinations on the skill tree and base the crossover on preserving these combinations.

Another aspect of skill trees is the correct presentation to the player. This consists of using patterns and pictures to show the player immediately what the

nodes in the skill tree are doing. A great example of such presentation is seen well in Path of Exile (Grinding Gear Games, 2013) in Figure 2. There is a couple of interesting patterns, which can be done in a skill tree, like node 'clusters' or subtree specialisations. These patterns and other forms of visualisation could be explored to decide, which patterns are good and under which conditions are they to be used. Examples of these patterns can be seen in **Figure** 37 and Figure **38**.

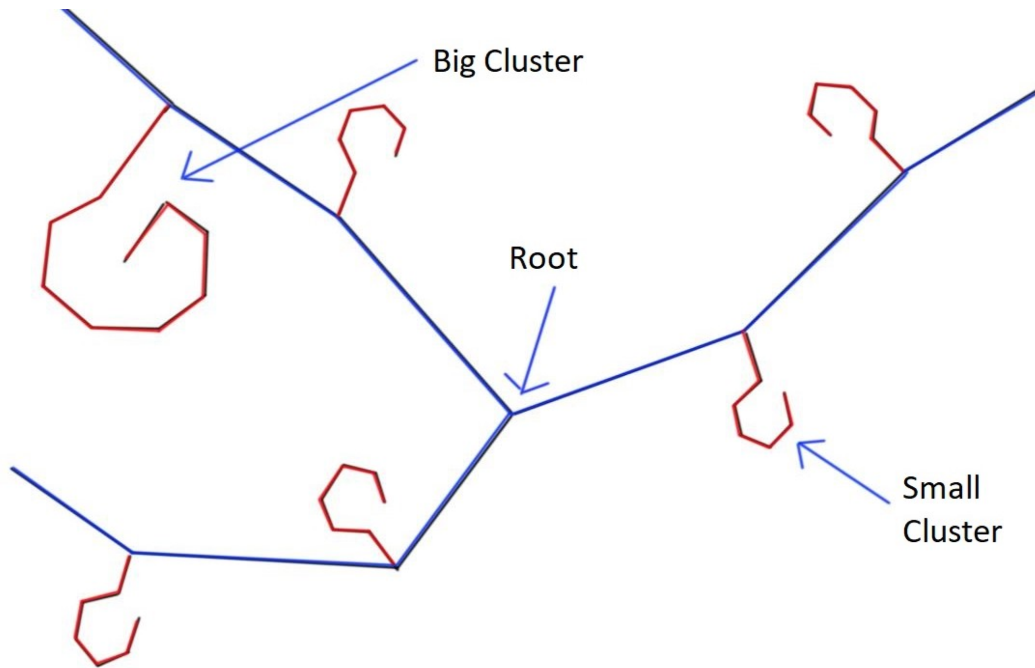

Figure 37: Cluster pattern

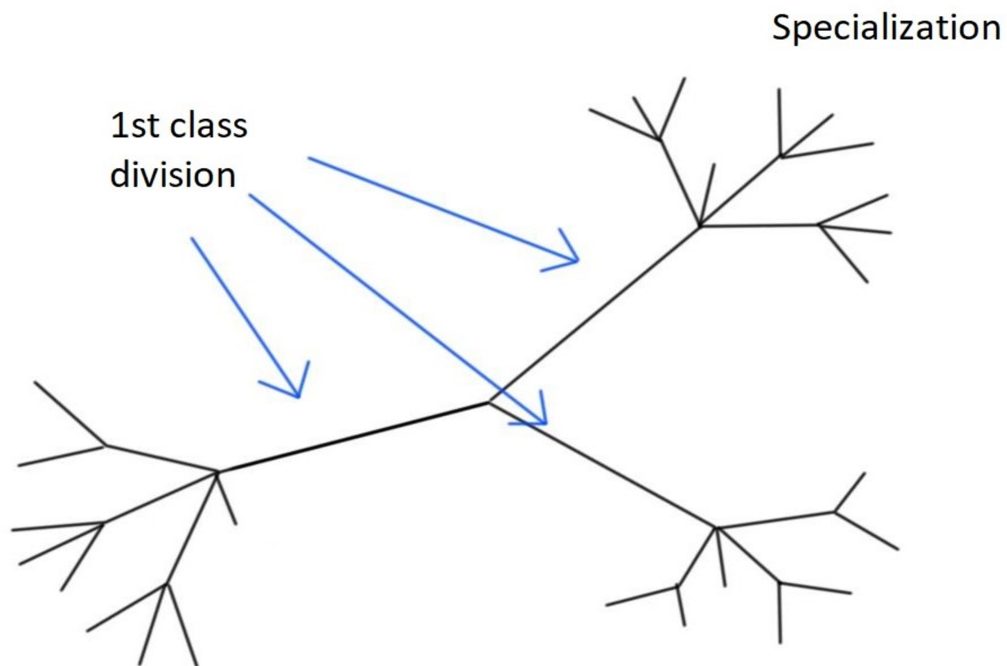

Figure 38: Subtree specialisation

Another topic that would be interesting to look at would be exploring the possibilities, methods and usefulness of online PCG of skill trees. This likely has very narrow use in games, but it is already used in Dead by Daylight (Behaviour Interactive, 2016), which means that it has its place in the gaming environment.

Bibliography

Behaviour Interactive. (2016, June 14). *Dead by Daylight*. Retrieved June 21, 2019, from http://www.deadbydaylight.com/en

Bethesda Softworks. (2008, October 28). *Fallout 3*. Retrieved June 19, 2019, from https://fallout.bethesda.net/en/games/fallout-3

Brown, F. (2018, May 25). *Path of Exile review*. Retrieved April 4, 2019, from PC Gamer: https://www.pcgamer.com/path-of-exile-review/

Carli, D. M., Bevilacqua, F., Pozzer, C. T., & Cordeiro d'Ornellas, M. (2011). A survey of procedural content generation techniques suitable to game development. *2011 Brazilian Symposium on Games and Digital Entertainment.* Salvador: IEEE.

Chen, J. (2007, April). Flow in Games. New York, NY, USA: ACM.

Cziksentmihalyi, M. (1990). *Flow – The Psychology of optimal experience.* Harper.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002, April). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*.

Grinding Gear Games. (2013, October 23). *Path of Exile*. Retrieved from https://www.pathofexile.com

Hamilton, K. (2018, October 16). *Assassin's Creed Odyssey Smartly Refines The Origins Skill Tree*. Retrieved April 25, 2019, from Kotaku: https://kotaku.com/assassin-s-creed-odyssey-smartly-refines-the-origins-sk-1829791194

Hendrikx, M., Meijer, S., van der Velde, J., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *Multimedia Computing, Communications, and Applications* .

Kelly, G., & McCabe, H. (2007). Citygen: An interactive system for procedural city generation. *Proceedings of the 5th International Conference on Game Design and Technology*, (pp. 8-16).

Mega Crit Games. (2017, November 15). *Slay the Spire*. Retrieved June 21, 2019, from https://www.megacrit.com/

NeocoreGames. (2013, May 22). *The Incredible Adventures of Van Helsing*. Retrieved June 19, 2019, from https://neocoregames.com/en/games/the-incredible-adventures-of-van-helsing/overview

Srinivas, N., & Deb, K. (1994). Muiltiobjective Optimisation Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation*.

Togelius, J., Champandard, A. J., Luca, P. L., Mateas, M., Paiva, A., Preuss, M., et al. (2013). Procedural Content Generation: Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games.* Wadern: Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-62-0.

Ubisoft Entertainment. (2017). Retrieved from https://www.ubisoft.com/en-gb/game/assassins-creed-origins/

List of Figures

Figure 1: Example of skill tree in Incredible Adventures of Van Helsing (NeocoreGames, 2013). Multiple tree roots can be seen on the top of the skill tree window; these nodes do not need a predecessor to be obtained before they can be obtained themselves. 1

Figure 2: Part of finished skill tree, which is focused on damage dealt with bows. This part of skill tree gives player sense of progression because of the finished node cluster. This cluster of nodes also gives a lot of power to the player, making future fights easier. (Grinding Gear Games, 2013) 3

Figure 3: Example of a skill tree that is giving the player too much power; with two nodes, player can take either 35 or 60 power. 8

Figure 4: Example of a skill tree that is giving the player too little power; with two nodes, player can get either 2 or 3 power. 8

Figure 5 (Chen, 2007): Simple chart of Flow Zone in relation to challenge and abilities of a player. When the player does not have strong enough abilities, he experiences anxiety. When the player has too strong abilities for current challenge, he experiences boredom. 11

Figure 6: Example of an undirected skill tree. Nodes highlighted are considered active and give power to the player. All edges have the same weight of 1 and this price is paid by points, which a player gets by levelling up. (Path of Exile, 2013) 12

Figure 7: This screenshot from Assassin's Creed: Origins (Ubisoft Entertainment, 2017) is an example of node and its effect. This node gives the player additional ability on putting an animal to sleep. Player can tame the animal after to gain a strong companion. 14

Figure 8: Effects of choices in skill tree. At every state of skill tree, there is a set of nodes we can choose from to obtain a new node. Depending on the choice of the player, the Player-power rises differently. 14

Figure 9: Effect of skill tree in regards to Player-power and flow. 15

Figure 10: Example of redundant skill trees. 16

Figure 11: Example of obvious choices in skill trees. 16

Figure 12: Example of a simple tree of 10 nodes. Numbers at each node represent their increase of Player-power when picked. *17*
Figure 13: Screenshot from the game Slay the Spire (Mega Crit Games, 2017). Cards in this game stand for actions and spend player's energy. Goal is to face encounters and progress further, while upgrading and expanding one's deck of cards. *24*
Figure 14: Screenshot of encounter of our game. Cards are used as actions to defeat enemies. Some are targeted and some are area. *25*
Figure 15: Screenshot of skill tree view in the game. One point per encounter is picked and the cards that are upgraded are highlighted. *26*
Figure 16: State diagram of our game *27*
Figure 17: Our adaptation of SGA for this thesis. *28*
Figure 18: Example of Merge node operator. Children of both selected nodes are concatenated together and added to the merged node. The new node type that is matching the stronger node of the two selected and the power of the nodes is summed up. *35*
Figure 19: Example of Split node operator. Node chosen for split is split into two new. Both new nodes have the same type and their power is split with random ratio. Original node's children are assigned randomly between the new nodes. *36*
Figure 20: Tree with added node (red). The new node has a random type, random value and it will always be a leaf node. *36*
Figure 21: Example of pairing nodes during merging. Here we are merging A and C and we are pairing their children. B and D get paired, as they are the first children of their parents. E is left as extra, since there is no child of A left to be paired. *38*
Figure 22: NSGA skill tree *55*
Figure 23: Weighted Evolutionary skill tree *55*
Figure 24: Evolutionary skill tree *55*
Figure 25: Randomized Hill Climbing skill tree *56*
Figure 26: Hill Climbing skill tree *56*
Figure 27: Win ratios of bots with NSGA skill tree *58*
Figure 28: Win ratios of bots with Weighted Evolutionary skill tree *58*
Figure 29: Win ratios of bots with Evolutionary skill tree *59*

Figure 30: Win ratios of bots with Random Hill Climbing skill tree......................... 59
Figure 31: Win ratios of bots with Hill Climbing skill tree...................................... 60
Figure 32: Average turns to win for different bots for NSGA trees........................... 61
Figure 33: Average turns to win for different bots for Weighted Evolutionary trees 61
Figure 34: Average turns to win for different bots for Evolutionary trees................ 62
Figure 35: Average turns to win for different bots for Random Hill Climbing trees 62
Figure 36: Average turns to win for different bots for Hill Climbing trees............... 63
Figure 37: Cluster pattern 68
Figure 38: Subtree specialization 68

List of Tables

Table 1: Definition of node's properties 19

Table 2: Card's definition 25

Table 3: Node's structure in the algorithm implementation 29

Table 4: Tree's properties in the algorithm implementation 29

Table 5: Relevant input for mutation 34

Table 6: Relevant input for fitness function 43

Table 7: List of NuGet packages used in the solution 48

Table 8: InputModel's properties. 49

Table 9: Format of data record of validation 57

List of Abbreviations

Abbreviation	**Explanation**
RPG	Role-playing games, game genre, which focuses on having the player play as a character in-game.
NSGA	Non-dominated Sorting Genetic Algorithm
PCG	Procedural Content Generation
MFC	Most Frequent Case
SGA	Simple Genetic Algorithm
GA	Genetic algorithm

Appendix A – Reference Decks

Deck number: 1
  15x - Type: Area; Energy; Damage: 2
Deck number: 2
  15x - Type: Area; Energy; Damage: 6
Deck number: 3
  15x - Type: Area; Mana; Damage: 4
Deck number: 4
  15x - Type: Area; Mana; Damage: 4
Deck number: 5
  15x - Type: Targeted; Energy; Damage: 6
Deck number: 6
  15x - Type: Targeted; Energy; Damage: 9
Deck number: 7
  15x - Type: Targeted; Mana; Damage: 5
Deck number: 8
  15x - Type: Targeted; Mana; Damage: 20
Deck number: 9
  5x - Type: Targeted; Mana; Damage: 20
  5x - Type: Targeted; Mana; Damage: 5
  5x - Type: Area; Mana; Damage: 4
Deck number: 10
  5x - Type: Targeted; Mana; Damage: 5
  5x - Type: Targeted; Mana; Damage: 20
  5x - Type: Area; Energy; Damage: 6
Deck number: 11
  5x - Type: Targeted; Energy; Damage: 9
  5x - Type: Targeted; Energy; Damage: 6
  5x - Type: Area; Energy; Damage: 2
Deck number: 12
  5x - Type: Targeted; Mana; Damage: 5
  5x - Type: Targeted; Mana; Damage: 20
  5x - Type: Area; Mana; Damage: 4
Deck number: 13
  5x - Type: Targeted; Energy; Damage: 9
  5x - Type: Targeted; Energy; Damage: 6
  5x - Type: Area; Mana; Damage: 4
Deck number: 14
  1x - Type: Targeted; Mana; Damage: 20
  1x - Type: Targeted; Energy; Damage: 9
  2x - Type: Area; Energy; Damage: 6
  1x - Type: Targeted; Energy; Damage: 6
  2x - Type: Targeted; Mana; Damage: 5
  8x - Type: Area; Mana; Damage: 4
Deck number: 15
  3x - Type: Area; Energy; Damage: 2
  2x - Type: Area; Mana; Damage: 4

1x - Type: Targeted; Mana; Damage: 20
1x - Type: Targeted; Energy; Damage: 6
2x - Type: Area; Energy; Damage: 6
4x - Type: Targeted; Mana; Damage: 5
2x - Type: Targeted; Energy; Damage: 9

Deck number: 16
1x - Type: Area; Energy; Damage: 2
5x - Type: Area; Mana; Damage: 4
5x - Type: Targeted; Mana; Damage: 5
4x - Type: Targeted; Energy; Damage: 9

Deck number: 17
1x - Type: Area; Energy; Damage: 6
3x - Type: Targeted; Energy; Damage: 9
1x - Type: Area; Energy; Damage: 2
2x - Type: Targeted; Mana; Damage: 5
2x - Type: Area; Mana; Damage: 4
6x - Type: Targeted; Energy; Damage: 6

Deck number: 18
1x - Type: Targeted; Mana; Damage: 5
1x - Type: Area; Energy; Damage: 6
1x - Type: Area; Energy; Damage: 2
3x - Type: Targeted; Energy; Damage: 6
1x - Type: Targeted; Mana; Damage: 20
4x - Type: Targeted; Energy; Damage: 9
4x - Type: Area; Mana; Damage: 4

Deck number: 19
1x - Type: Targeted; Mana; Damage: 5
2x - Type: Targeted; Mana; Damage: 20
2x - Type: Targeted; Energy; Damage: 6
5x - Type: Area; Mana; Damage: 4
2x - Type: Area; Energy; Damage: 2
1x - Type: Area; Energy; Damage: 6
2x - Type: Targeted; Energy; Damage: 9

Deck number: 20
1x - Type: Targeted; Energy; Damage: 9
2x - Type: Targeted; Mana; Damage: 20
3x - Type: Area; Energy; Damage: 6
4x - Type: Targeted; Energy; Damage: 6
5x - Type: Area; Mana; Damage: 4

Deck number: 21
1x - Type: Area; Energy; Damage: 2
2x - Type: Targeted; Mana; Damage: 20
4x - Type: Targeted; Energy; Damage: 9
2x - Type: Area; Energy; Damage: 6
2x - Type: Targeted; Mana; Damage: 5
4x - Type: Area; Mana; Damage: 4

Deck number: 22
2x - Type: Targeted; Energy; Damage: 9
2x - Type: Targeted; Mana; Damage: 20
4x - Type: Area; Mana; Damage: 4

2x - Type: Targeted; Mana; Damage: 5
2x - Type: Targeted; Energy; Damage: 6
3x - Type: Area; Energy; Damage: 2

Deck number: 23

2x - Type: Area; Mana; Damage: 4
3x - Type: Area; Energy; Damage: 2
2x - Type: Area; Energy; Damage: 6
1x - Type: Targeted; Mana; Damage: 5
2x - Type: Targeted; Mana; Damage: 20
1x - Type: Targeted; Energy; Damage: 6
4x - Type: Targeted; Energy; Damage: 9

Deck number: 24

2x - Type: Targeted; Energy; Damage: 9
1x - Type: Area; Mana; Damage: 4
4x - Type: Area; Energy; Damage: 2
1x - Type: Targeted; Energy; Damage: 6
4x - Type: Targeted; Mana; Damage: 20
3x - Type: Area; Energy; Damage: 6

Deck number: 25

4x - Type: Targeted; Mana; Damage: 5
1x - Type: Targeted; Energy; Damage: 6
3x - Type: Targeted; Energy; Damage: 9
2x - Type: Area; Energy; Damage: 6
4x - Type: Area; Mana; Damage: 4
1x - Type: Targeted; Mana; Damage: 20

Deck number: 26

1x - Type: Targeted; Energy; Damage: 9
1x - Type: Targeted; Energy; Damage: 6
3x - Type: Targeted; Mana; Damage: 5
1x - Type: Targeted; Mana; Damage: 20
2x - Type: Area; Energy; Damage: 6
4x - Type: Area; Mana; Damage: 4
3x - Type: Area; Energy; Damage: 2

Deck number: 27

1x - Type: Targeted; Energy; Damage: 9
1x - Type: Targeted; Mana; Damage: 5
3x - Type: Area; Mana; Damage: 4
1x - Type: Targeted; Energy; Damage: 6
2x - Type: Targeted; Mana; Damage: 20
1x - Type: Area; Energy; Damage: 2
6x - Type: Area; Energy; Damage: 6

Deck number: 28

1x - Type: Area; Energy; Damage: 2
1x - Type: Targeted; Energy; Damage: 6
2x - Type: Targeted; Energy; Damage: 9
1x - Type: Targeted; Mana; Damage: 5
2x - Type: Targeted; Mana; Damage: 20
4x - Type: Area; Mana; Damage: 4
4x - Type: Area; Energy; Damage: 6

Deck number: 29

1x - Type: Targeted; Energy; Damage: 9
2x - Type: Area; Energy; Damage: 6
3x - Type: Area; Mana; Damage: 4
3x - Type: Area; Energy; Damage: 2
1x - Type: Targeted; Mana; Damage: 20
3x - Type: Targeted; Energy; Damage: 6
2x - Type: Targeted; Mana; Damage: 5

Deck number: 30
2x - Type: Targeted; Energy; Damage: 6
4x - Type: Area; Mana; Damage: 4
2x - Type: Area; Energy; Damage: 2
3x - Type: Targeted; Energy; Damage: 9
1x - Type: Area; Energy; Damage: 6
3x - Type: Targeted; Mana; Damage: 5

Deck number: 31
2x - Type: Targeted; Energy; Damage: 9
1x - Type: Area; Energy; Damage: 6
1x - Type: Targeted; Mana; Damage: 5
1x - Type: Targeted; Energy; Damage: 6
3x - Type: Area; Energy; Damage: 2
2x - Type: Targeted; Mana; Damage: 20
5x - Type: Area; Mana; Damage: 4

Deck number: 32
1x - Type: Area; Mana; Damage: 4
2x - Type: Targeted; Mana; Damage: 20
3x - Type: Targeted; Mana; Damage: 5
3x - Type: Targeted; Energy; Damage: 6
2x - Type: Area; Energy; Damage: 2
2x - Type: Targeted; Energy; Damage: 9
2x - Type: Area; Energy; Damage: 6

Deck number: 33
2x - Type: Targeted; Energy; Damage: 6
1x - Type: Targeted; Mana; Damage: 20
3x - Type: Targeted; Mana; Damage: 5
7x - Type: Area; Mana; Damage: 4
2x - Type: Area; Energy; Damage: 6