Generating Gloomhaven Dungeons through Evolutionary Game Design

Bachelor Thesis of Kristian Tijben
Supervisor: Marcus Gerhold
Critical Observer: Faizan Ahmed
University of Twente

Abstract

In this thesis, evolutionary algorithms were used to generate unique and enjoyable dungeons of the board game Gloomhaven. Using Python, existing Gloomhaven dungeons were stored as objects, and went through an evolutionary cycle of selection, crossover and mutation to generate new dungeons. These dungeons were checked for playability and judged on several factors based on previous research on game enjoyability. The highest-scoring dungeons that came out of this process were then tested in a user test, returning mainly positive results.

The code used in this thesis can be found on the following GitHub page:
https://github.com/kristianwillem/GloomDungeons.git

Table of Contents

Abstract.....2
1. Introduction.....4
2. Related Works.....5
    2.1 Procedural Content Generation.....5
    2.2 The Fitness Function.....5
        Player Perspective.....5
        Designer’s Perspective.....5
        Measurable Factors.....6
        Factor Groups.....6
3. Methods.....7
    3.1 Evolutionary Algorithms.....7
    3.2 Gloomhaven Dungeon.....7
    3.3 Evolutionary Process for Gloomhaven.....9
        Initialization.....9
        The Evolutionary Cycle.....9
4. Implementation.....12
    4.1 Coordinates.....12
    4.2 Algorithm Overview.....13
        Parent selection & crossover.....14
        Mutation.....14
        Monster Mutation.....14
        Map Mutation.....15
        Fix.....16
        Validity.....16
        Fitness.....19
5. Experimentation.....20
    5.1 Experiment Setup.....20
    5.2 Results.....21
    5.3 Conclusion.....21
        Dungeon Enjoyability.....22
        Dungeon Weights.....22
6. Discussion.....22
    6.1 Necessary Improvements.....23
    6.2 Additional Improvements.....23
    6.3 User Test Validity.....24
    6.4 Future Research.....24
References.....26
Appendix A: User Test Questionnaire.....27
Appendix B: User Test Dungeons.....28

1. Introduction

The creation of games is an expensive and time-consuming process. One point of concern is the creation of unique content, especially if the game requires a lot of content that is very similar in nature, yet also needs to be distinct from each other, such as plants, levels or dungeons. If the creation of such content could be automated, it would save on precious resources. This process, called procedural content generation (PCG), is already commonly used in the video game industry. However, in the case of board games, it is still rarely used.

PCG is an incredibly useful tool in board game design. It could be especially helpful in creating components that require both a large amount of variety and a lot of effort to create manually. An example of this is a dungeon in a board game dungeon delver. In such a game, the same basic rules are used to play the game on a different board each time. If the dungeon creation for such a game is not done well, it will either not have enough content, thus losing replayability, or it will quickly become repetitive, which will often result in the game being boring. While it is possible to create a hundred or so dungeons for a game manually, it would be much more efficient if such dungeons could be procedurally generated.

While there are numerous tools for random dungeon generation already out there, most of them generate dungeons with no regard for the game system that the dungeon will be used for. While this generality has its benefits, it also means that they are impractical to use with a system that has many constraints. This is especially problematic when designing new dungeons for an already existing board game system, since these systems each have their own rules, enemies and components that these existing dungeon generators cannot take into account.

Since the aim is to create randomized dungeons which align as much as possible with the game system it is being used for, the generator needs to keep the constraints of the game system into consideration. A possible way to do this is through the use of evolutionary game design, which will take existing dungeons of a specific system, before combining them and mutating the result to get a new dungeon that has a high likelihood of fitting in the system.
Thus the research question becomes:

How can evolutionary game design be used to generate random dungeons of a given game system?

Because the aim of the thesis is to generate dungeons with a given game system in mind, there is a need of a specific game system to test the generator. In order to do this, the game *Gloomhaven* [1] will be used. This choice was made because Gloomhaven is a game highly ranked in the board game community, being ranked as the number 1 overall board game on boardgamegeek at the time of writing [2], and each time playing the game exists of going through a premade dungeon. In addition to that, Gloomhaven is a complex game with many components and variables that all need to fit together in a specific way to create a playable and thematically consistent, which means that more general tools for dungeon design could not be used for it. Finally, while Gloomhaven does have a way to generate random dungeon, the existing system is both repetitive and lacks the narrative consistency that the other dungeons in the game do have.

In addition to that, the evolutionary algorithm needs a *fitness function* to work properly. A fitness function will judge the quality of the generated dungeon through a number of factors, but as important as the nature of the factors is the weight that is assigned to each. Therefore, a two subquestions of this thesis become:

What are the factors of the fitness function needed to properly judge generated Gloomhaven dungeons?
What are the optimal weights in the fitness function in order to generate the best Gloomhaven dungeons?

2. Related Works

2.1 Procedural Content Generation

Procedural content generation (PCG) is defined by Togelius et al. [3] as “the algorithmic creation of game content with limited or indirect user input”. In this thesis, the content that is algorithmically generated is a dungeon for the board game Gloomhaven, which also fits their definition of content.

Most sources on PCG focus on generating content for digital games. However, Brown and Scirea [4] have analyzed non-digital methods for PCG and thus show that using PCG for board games is a valid option.

On the other hand, PCG for dungeon generation is a far more common subject. Most important for this thesis are Chapters 5 and 6 in the book Procedural Content Generation in Games [5] and the paper Evolving Dungeon Crawler Levels With Relative Placement by Valtchanov and Brown [6]. Chapter 5 [5] introduces the use of grammars to create content, which has significant benefits over other methods of generating dungeons. Most importantly, it allows for control over the content generation, ensuring that the created content is consistent with a given set of conditions that every dungeon should adhere to. Valtchanov and Brown [6] introduces the concept of fitness, which measures how much a given piece of generated content matches the desired outcome on factors that are more lenient than the stricter conditions. This fitness score can be used to evolutionary generate more dungeons, in a similar way as done through evolutionary game design by Browne and Maire [7].

2.2 The Fitness Function

Part of an evolutionary algorithm is testing the new creating using a so called fitness function. In order to make a suitable fitness function for generating a good Gloomhaven dungeon, it is relevant to know what makes a good board game in the first place. The result of that can be translated into different factors that can be used for the fitness function.

Player Perspective

Before considering the designer’s perspective of how to improve the player experience, it is important to know what players look for in board games. Hunicke et al. [8] introduces eight aesthetic components, which represent different ways in which people enjoy games. These components are sensation, fantasy, narrative, challenges, fellowship, discovery, expression and submission. These aesthetic components are explored in depth by Costikyan [9].

Sensation is the sensory pleasure of a game. While for video games this is usually limited to visuals and audio, for board games it includes the feel of the components. Fantasy is the fictional context of the game. Narrative is not necessarily the story of the game, but instead its tension arc. Challenge is the fact that winning is not assured, the player will have to put effort into the game. Fellowship is the social aspect of the game, which is usually the other people at the table in the case of board games, and the competitive or cooperative relationship within the game. Discovery is the exploration aspect of the game, whether it is in-game exploration or exploration of the game’s mechanics. Expression is the way a game allows you to express yourself. In board games, this is more rare than in tabletop roleplaying games or in video games, but it might be present in minor parts such as naming your character, playing a specific style or even choosing a color. Finally, submission is the desire for a player to submit to the rules of the game. This might mostly become clear when the game does not allow for this submission, such as when the rules are too vague, conflicting or ambiguous.

Designer’s Perspective

While the perspective of the player is important to keep in mind during game design, it does not provide factors that can easily be implemented. From the game designer’s perspective, the factors that can be used to improve player experience can be approached in two different ways. The first approach, as shown by Hunicke et al. [8], is to consider the desires of the player, and then choose

factors that satisfy those desires. An example they give is that challenge, one of the aesthetic components, can be improved by including a time limit.

The second approach is to select factors that need to be included in a game in order to make it a good game. The idea behind this approach is not that the game is guaranteed to be enjoyable if these factors are included, but rather that the game will certainly not be enjoyable if these factors are left out. Costikyan [9] states that a game has to include goals, struggle, structure and endogenous meaning. These are the bare minimum requirements for a game. Should a game not include all of these factors, it would not be a game at all. He does state that games do not necessarily need an explicit goal, but this seems to be less applicable to board games than either video games or roleplaying games.

Thompson [10] suggests four different factors. These factors are given with combinatorial games in mind, but they also be applied to many other types of games. The factors are depth, the game should not be solvable; clarity, the players of the game should be able to deduce the best course of action in a given situation; drama, the game should allow for a player to come back from a bad position; and decisiveness, the game should eventually end in a win for one of the parties involved.

Kramer [11] gives a longer list of objective criteria that are necessary for a good game. Several criteria are self-explanatory. Those criteria are originality, freshness and replayability, surprise, equal opportunity, winning chances, no early elimination, reasonable waiting times, uniformity, quality of components, target groups and consistency of rules, tension, and learning and mastering a game. Kramer includes three more criteria that are less self-explanatory. First of all, he states that no game should have a kingmaker effect. This means that a player who no longer has any chance to win should not be able to decide the winner. Then there is the criteria of creative control, which means that it should be possible for the players to influence the game. The final criteria he gives is complexity and influence. While complex rules are not necessarily bad for a game, this criteria does mean that games with more complex rules also need to give players more influence over the game.

Measurable Factors

In addition to using such factors while designing a game, the design process can also be improved by predicting player experience. While this is usually done through extensive playtesting, it can also be done by measuring specific variables that relate to the discussed factors through testing the game with AI players. Such a system was developed by Browne [12], who used 57 different criteria in order to predict the quality of a game. A later research by Browne and Maire [7] found that six of these criteria have more impact on game quality than the others. Those criteria are uncertainty, lead change, permanence, killer moves, completion, and duration. Each of these criteria is explained in more depth by Browne [12].

Factor Groups

There are many different factors that all contribute to the design of an enjoyable game. The challenge of a game seems to be especially important, since many factors connect to it. Apart from challenge and struggle, which describe the concept, those factors are drama, equal opportunity, winning chances, no early elimination, tension, kingmaker effect, uncertainty, lead change, permanence, and killer moves. Combined with the statement of Costikyan [9] that a game cannot exist without challenge, the high amount of terms related to challenge show that it is significant to the player’s enjoyment of the game, and thus should be thoroughly evaluated before starting the extensive playtesting process.

Some of the factors are very practical in nature. These factors rely mainly on the game rules or components, and can thus probably be tested even without playing the game. These factors are goals, structure, endogenous meaning, originality, uniformity, quality of components, target groups and consistency of the rules, creative control, and complexity and influence. Submission is also a practical factor, since it mainly relies on the game being consistent and complete. A few factors,

namely depth and clarity, are mainly practical, but testing is still required to ensure that there are no unexpected results for these factors. For depth that would be a winning strategy, and for clarity that would be specific situations that have illogical or unintuitive outcomes. Some of the remaining factors do require testing, but are easy enough to test that a simulation would probably suffice. Those factors are decisiveness, reasonable waiting times, completion and duration.

This leaves a total of nine factors that are significantly more complicated, and are very hard, if not impossible, to test through simulations. There are a few different problems with these factors. Some of these factors are subjective, such as sensation, fantasy, narrative, discovery and expression. Some of them only become clear after several tests with players that learn from their previous experiences, namely freshness and replayability, surprise, and learning and mastering a game. Finally, there is one factor, fellowship, that relies for a large part on influences from outside the game. While there are ways to improve and promote fellowship through game design, a great game can still be unenjoyable when played with the wrong people.

3. Methods

The process of generating new Gloomhaven dungeons exists out of two main components; an evolutionary algorithm and a dungeon description, which is then combined into a specific evolutionary process for Gloomhaven.

3.1 Evolutionary Algorithms

Evolutionary algorithms are based on the theory of evolution, but instead of organisms, it is now data that goes through the evolutionary cycle. While there are multiple variants of evolutionary algorithms, the process used in this thesis is based on the evolutionary cycle used by Browne and Maire [7], since it is already shown to work in the context of board games.
This evolutionary algorithm starts with an initial population, which exists of entries that the evolutionary cycle will create variants of and, ideally, improve upon. The evolutionary cycle goes through the process of selection, crossover, mutation, validation and evaluation. If the outcome of the cycle is not deemed invalid, it will return to the population. This cycle will repeat until a set goal has been reached, which can be a limited number of cycles or a preferred score.

Selection is the process of choosing the *parents* of a new entry, or *child*, which means that the child will initially be based on both parents. This is done through *crossover*, where each attribute of the child is randomly chosen from one of the two parents. After this, the child will be *mutated*, meaning that one or more randomly chosen attributes of the child will be changed into another possible value. The child will then be *validated*, which means that it will go through several checks to make sure that it adheres to the given boundary conditions. Finally, the child will be *evaluated* through the fitness function, which will give the child a score that represents how well the child compares to the other entries of the population. The child will then become a new entry in the population, and thus it can be chosen as a parent for a future cycle.

3.2 Gloomhaven Dungeon

A Gloomhaven dungeon has many different components. The program will change these components, either through crossover or through mutation. To make the replacement of components easier, each type of component will be its own class, so that all attributes of the component will be replaced together. The dungeon class itself will have attributes for each class that is a collection of objects of these classes. The only exceptions to this are the “Main theme” and “Coins” attributes, since those can be represented by only a single entry, and do not have any further attributes. Finally, the dungeon class contains a list of “Placement” objects, which describe where each component is to be placed. This brings us to the following classes, which describe an entire Gloomhaven dungeon when put together:

Dungeon Class:
Attribute: Goal (list of *Goal* objects)
Attribute: Special_rules (list of *Rule* objects)
Attribute: Rooms (list of *Room* objects)
Attribute: Connections (list of *Connection* objects)
Attribute: Dungeon_monsters (list of entries of the form [*Monster* object, string (“normal” or “elite”), integer (amount)])
Attribute: Obstacles (list of *Obstacle* objects)
Attribute: Traps (list of *Trap* objects)
Attribute: Loot (list of *Chest* objects)
Attribute: Terrain (list of *Terrain* objects)
Attribute: Coins (integer)
Attribute: Main_theme (string)
Attribute: Placements (list of *Placement* objects)

Goal Class:
Attribute: Win_condition (string)
Attribute: Ties (string); Describes what is necessary for the dungeon to work, such as that there needs to be a treasure tile if the goal is “loot the treasure tile”, or that the dungeon needs a designated obstacle and special rules regarding that obstacle if the goal is “destroy a specific obstacle”.

Rules Class:
Attribute: Rule (string)
Attribute: Ties (string); Describes what is necessary for the rule to work.
Attribute: Difficulty (int); Special rules can make the dungeon easier or harder. This attribute is a modifier to the difficulty that is used in the fitness function.

Room Class:
Attribute: Name (string)
Attribute: Theme (string)
Attribute: Tiles (list of coordinates, which are lists of three integers each)
Attribute: Exits (list of coordinates, which are lists of three integers each)
Attribute: Entries (list of coordinates, which are lists of three integers each)

Connection Class:
Attribute: Entry_room (string)
Attribute: Entry_coordinate (list of three integers)
Attribute: Exit_room (string)
Attribute: Exit_coordinate (list of three integers)
Attribute: Connector_type (string); door or corridor

Monster Class:
Attribute: Name (string)
Attribute: Max_amount (integer)
Attribute: Difficulty (integer); double the given rating, since it allows using integers instead of using floats.

Obstacle Class:
Attribute: Name (string)
Attribute: Theme (string)
Attribute: Size (integer)

Attribute: Max_amount (integer)

Chest Class:
Attribute: Content (string)

Placement Class:
Attribute: Type (string); whether this placement refers to a monster, obstacle, etc.
Attribute: Type_nr (integer); which entry of the specific type this placement refers to.
Attribute: Location ([string, integer, integer, integer]); which room and coordinates of that room this placement refers to.

3.3 Evolutionary Process for Gloomhaven

The evolutionary process will create new Gloomhaven dungeons from a selection of original dungeons using an evolutionary algorithm. The program running the evolutionary process can be split into two parts; initialization, and the evolutionary cycle.

Initialization

Before the evolutionary cycle can begin, the initial population needs to be determined. In this case, the initial population exists of Gloomhaven dungeons from the original dungeon book. While the book contains 95 different dungeons, not all of them are usable for the generation of random dungeons. For example, boss dungeons often have very specific interactions between the boss and the rest of the dungeon, such as interaction with specific monsters, opening specific doors, or attacking in specific patterns. Combined with the fact that many bosses are unique monsters with attached lore, the boss dungeons will not be used for the initial population. Several other dungeons will also not be used, whether they require specific items, have rules that are very difficult to implement and verify, or are so fundamentally different from other dungeons that they make a poor parent.

After the selection, 56 dungeons are left that could be suitable parents for the evolutionary cycle. These dungeons are turned into dungeon-class objects, and are used as the initial population for the evolutionary cycle. As a final step before the evolutionary cycle can begin, the fitness function is applied to the original dungeons, which is needed since their fitness score is used in the evolutionary cycle.

The Evolutionary Cycle

The evolutionary cycle will create new dungeons from the initial population. It will do so by repeating several steps a certain number of times. The steps will be explained in further detail below, and are visualized in Figure 1.

1. Select parents
2. Creating a new dungeon by crossing over the parents;
3. Mutate the new dungeon;
4. Fix initial problems;
5. Check the dungeon for validity;
6. Apply fitness function and get the dungeon's fitness score.

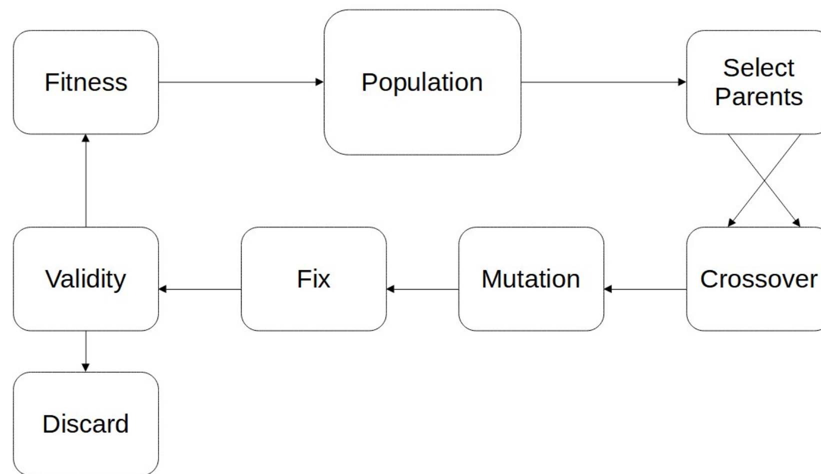

Figure 1: Evolutionary Cycle

1. Selecting Parents
First of all, two parent dungeons are selected from the population, with a bias to dungeons with a higher fitness score.

2. Crossover Parents
A new dungeon is created, with its initial attributes taken from the parent dungeons. Since Gloomhaven dungeons are a complicated system that quickly loses its playability or thematic consistency if something small changes, the different attributes are combined into groups. For each group, one of the parent dungeons is randomly chosen, and the child dungeon gains the attributes of that dungeon that belong to that group. The groups are;

- Rules, containing the Goal and Special_rules attributes
- Map, containing the Rooms, Connections, Main_theme and Placement attributes
- Monsters, containing the Dungeon_monsters attribute
- Environment, containing the Obstacles, Traps, and Terrain attributes.
- Treasure, containing the Loot and Coins attributes.

This specific granularity is chosen because all the attributes in a single group either depend on each other, such as rooms and connections, or they fulfill a similar purpose in the dungeon, such as obstacles and traps.

3. Mutate Dungeon
Now that the initial dungeon is created, it will be mutated so that it becomes a unique dungeon rather than just the combination of two existing ones. The problem that small changes will first make the game worse before making it better is once again relevant. In this case, while combining the attributes into groups is possible, it does not solve the entire problem. Grouping the attributes and then exchanging an entire group with another one works for the crossover, but for mutation it would require every possible combination of a group to be predefined, which is practically impossible. Instead, the concept of groups will still be used, but instead of putting a new group in right away, the group will be emptied. Then, new data for that group will be randomly generated.

4. Fix Initial Problems
After crossover and mutation, it is very likely that there are still several problems that need to be resolved before the dungeon can be tested for validity. These problems come in two forms.
First of all, since dungeons differ in the amount of monster, obstacles and other components, it is likely that the placement attribute has a different number of entries for a given type than the amount actually used in the dungeon. If the placement attribute has a higher number of entries for a given type than there actually are, the unused entries should be removed. If the placement attribute has less entries of a given type, new placements should be randomly generated for those entries. Only

the placement for the entries that do not yet have a placement will be randomly generated, so the original placement attribute of the parent dungeon is used as much as possible. This is to keep the possible coherency of the original dungeon as much as possible. Coherency might be important, but it is very hard to measure or to find an optimal value for it, so it will not be used in the fitness function.

Besides that, certain goals and special rules interact with monsters or obstacles. In changing the dungeon, these monsters or obstacles might have changed or disappeared. Therefore, each of those rules should be checked if it still refers to components that it should interact with, and refer to new, randomly selected components if it does not. This is done to prevent an unfair bias to simpler goals, such as “kill all enemies”, since those do not interact with any specific components and will thus never have this issue.

5. Check Validity

While the fitness score is used to assess the quality of a dungeon, there are certain boundary conditions that every dungeon should conform to, to prevent the program from accidentally creating an impossible dungeon.

First of all, every dungeon should be playable using only basic move and attack actions. Therefore, it should be checked whether the goal of the dungeon is achievable with only movement and attacks targeting adjacent tiles. What exactly should be checked depends on the goal. For example, if the goal is to defeat all enemies, each enemy should be reachable without crossing obstacles or walls. On the other hand, if the goal is to escape, the exit should be reachable without crossing obstacles or walls, but not all monsters have to be reachable in that way.

Besides the playability of the dungeon, it should also be checked whether the dungeon is physically possible with the components contained in the base game. This means that rooms should not overlap with each other, and that obstacles, terrain and other tiles cannot exceed their maximum, which is the amount available in the game box. This limit also applies to monsters, although there the limit is decided per monster type.

6. Apply Fitness Function

As a final step in the evolutionary cycle, a fitness function will be applied to the dungeon, giving it a score that indicates how good the dungeon is. The ideal randomly generated dungeon would be different from any dungeon in the book, but have a similar difficulty and thematic coherence.
The score is calculated using a weighted sum considering the following measurements.

- Difficulty: The base difficulty is calculated from the monsters in the dungeon. The creator of the game has given an indication of difficulty of all monster types, which will be added together for all monsters in the dungeon [13]. After that, dungeon effects will be added, usually increasing the difficulty score slightly. If the dungeon goal is to kill all enemies, the predicted difficulty will now be compared to the ideal difficulty, and give a decreasing output the further away the predicted difficulty is from the ideal one. If the dungeon has a different goal, additional difficulty score will be added based on the specific goal, such as how tough the obstacles are that need to be destroyed, or how far away the chests are if they need to be looted. The ideal difficulty is also increased in these situations, since not all monsters need to be killed, and thus the monster difficulty has less impact.
- Size: The dungeon should not be too small or too big. Most dungeons use three or four rooms. However, many dungeons that have more rooms make use of several smaller rooms, so the number of tiles seems to be a better measurement than the amount of rooms. The amount of tiles in the new dungeon will be compared to the average number of tiles in the original dungeons, with only slightly decreasing returns the further away the new dungeon’s size is from the ideal, since only large deviations from this ideal should be filtered out.
- Complexity: To make sure that the new dungeons do not become unnecessarily complicated, there will be a negative score for using too many goals or too many special rules. For every goal or rule after the first two of each, a significant penalty will be given. Special rules that

are necessary, such as those that are required by specific goals, are excluded from this calculation.
- Theme: To make sure that the new dungeons keep the thematic consistency present in the original game, a penalty will be given for using monsters of different themes. A similar penalty will be given for using rooms of different themes. It is worth noting that this measurement lowers the score of several existing dungeons, which do use monsters of different themes. This is not a problem since those dungeons usually come with an explanation of why these monsters are in a dungeon together, which cannot be done for randomly generated dungeons.

For the factors of difficulty, size and complexity, the score will be given based on how close the generated dungeon is to the optimal value. This optimal value will be determined using the official scenarios, since it can be assumed that those are well made and close to an optimal value.

After the fitness function is applied, the generated dungeon will go into the population with the outcome of its fitness function as score. It is worth noting that even dungeons with low fitness scores go into the population, since they might become better in a next generation. Instead, the lower score makes that the dungeon is less likely to be chosen as a parent, still maintaining the idea of evolutionary design.

4. Implementation

The algorithm described above has been implemented using Python. While most of the implementation is in line with the methods, there are some differences. One of the main changes is the addition of another factor in the fitness function named “clutter”. This factor is based on the amount of tiles of a dungeon that are occupied by monsters, obstacles, etc., as compared to the total amount of tiles in the dungeon. Should this factor be too low, the dungeon will feel empty and with that less interesting. Should this factor be too large, it will be hard to keep overview, and with that the dungeon becomes more confusing and perhaps even frustrating.

In addition to that, some changes were made to prioritize a program that works well over one that tries to be able to generate any possible Gloomhaven dungeon, but fails. This includes changes such as removing all rules except for “kill all monsters”, not including the possibility to merge two smaller rooms into a single larger one, and reducing the amount of dungeons that are part of the initial population to 9.

Finally, several classes that were suggested in Section 3.2 have been turned into attributes of the Dungeon class instead. This includes the goal, rules, terrain, obstacles, traps and terrain classes. Having these categories as classes would not add significant benefits to the process, and having them as attributes made it easier to keep track of them and change them.

4.1 Coordinates

Since Gloomhaven uses hex-based maps, a coordinate system needs to be chosen. The choice was made to use the so called cube coordinates, which are described by Red Blob Games [14] and shown in Figure 2. This coordinate system uses three numbers to denote coordinates, and while two numbers are technically enough to give each hex a unique coordinate, the cube coordinates have the benefit that it is easy to determine whether two coordinates are adjacent to each other, which is used several times in the algorithm.

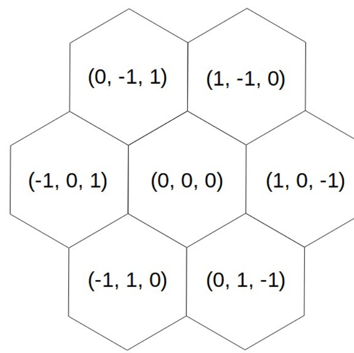

Figure 2: Cube Coordinates

The coordinates will be used both to form a graph that can be used for a search function when checking for reachability, as described below, and as a way to describe the placement of all components of the dungeon. While the coordinates of a dungeon are not given by the dungeon itself, they can be determined using the dungeon's rooms and connections. In order to do so, the program takes the first room in the dungeon's room list and adds its coordinates to the dungeon's coordinates. Which rooms is used first does not matter, as long as it is consistent, which it is in this case because the order of the rooms in the list never gets changed. It will then add the coordinates of the other rooms using the connections. This can be done because each connection has two sets of coordinates, one for each room that it connects. By determining the difference between those coordinates, and adding that difference to the coordinates of the second room, the resulting coordinates all describe their unique location from the first room's origin, as seen in Figure 3. By repeating this process for every connection, the resulting coordinates describe the entire dungeon with the origin of the first room as the dungeon's origin.

Figure 3: Coordinate Calculations.
Made using the Virtual Gloomhaven Board [15]

4.2 Algorithm Overview

The "main" file contains the "generate" function, which is ran to generate new dungeons. Before it actually starts the evolutionary cycle, however, some preparation needs to be done.
The rooms, special rules, monsters, chest content, and initial dungeons that are available to the program to use are stored in separate files, which first need to be loaded before they can be used by the evolutionary algorithm. This content is in separate files to make it easier for someone using the program to later add or remove such content. This content is loaded and put into lists. While special rules and chest content are strings, the rooms, monsters and dungeons are stored as an object of their respective classes, since the program later needs certain attributes of these to work properly. The initial dungeons then receive a score from the fitness function, as described below, and make up the population for the evolutionary algorithm at the start of the process.

The program then goes through the evolutionary cycle a predetermined amount of times. The evolutionary cycle consists of the parts that have been described before: selecting parents,

crossover, mutation, fixing, validity check and applying the fitness function. If the dungeon is valid, it is then put into the population, no matter its score. After the cycle is done, the program removes all the initial dungeons from the population, since the output should be a newly generated dungeon. It then selects the top three dungeons, judged by overall score, and outputs them.

Parent selection & crossover

The implementation of selecting parents and crossover is fairly straightforward. Two parents are selected from the population, with a bias for the dungeons with a higher score. The crossover selects one of the parents for each of the categories of “rules”, “map”, “monsters”, “environment” and “treasure”, and then creates a new dungeon object with for each attributes the value of the chosen parent.

Mutation

The implementation of the mutation part of the evolutionary cycle is one of the largest parts of the program. This part of the program goes through the same categories as the crossover section, namely “rules”, “map”, “monsters”, “environment” and “treasure”, and has a chance for each category to randomize its entry in the dungeon object to entirely new values. One of the important parts of this is the generation of new component amounts, so that the generated dungeons can, for example, have a different number of obstacles than any of the dungeons that have been used as the initial population. Without context, randomizing this value would be difficult, especially since most components do not have a direct impact on the fitness function, and thus do not have an ideal value. Therefore, the program instead generates a new amount based on what the amount was before the dungeon mutated. The program uses a Gaussian distribution that gives roughly a 95% chance that the new value is between 0.5 and 1.5 times the original amount. The “mutate_environment” part of the code shows best how this is done, since it has no other changes than the amounts.

```
def mutate_environment(self, dungeon):
    # using gaussian to create a 95% chance that the new number will be between 0.5 and 1.5 times the original.
    environment_list = [dungeon.obstacles, dungeon.traps, dungeon.h_terrain, dungeon.d_terrain]
    for i in range(len(environment_list)):
        mu = environment_list[i]
        if mu == 0:
            sigma = 1
        else:
            sigma = mu/4
        new_entry = round(random.normalvariate(mu, sigma))
        environment_list[i] = abs(new_entry)
    dungeon.obstacles = environment_list[0]
    dungeon.traps = environment_list[1]
    dungeon.h_terrain = environment_list[2]
    dungeon.d_terrain = environment_list[3]
```

Figure 4: mutate_environment

The mutations of the “rules”, “environment” and “treasure” sections are relatively simple, using only the generation of new amounts and the random choice of strings for the new rules or treasure chests. The mutation sections of the “monster” and “map” categories are significantly more elaborate.

Monster Mutation

The mutation of the “monster” category is more involved than the categories described above since there are various amounts that depend on each other. There is the amount of monsters in a dungeon, but also their types, how many of them are normal or elite, and what their difficulty is. Since the difficulty of the monsters is very important for the score of the dungeon, it is chosen as one of the factors to determine through the same process as above. Then, the amount of different monster types is randomized, with a minimum of two. That many monster types are randomly chosen from the list of all possible monsters. The first selection has no bias, but all following have a bias for monsters of the same type as the first monster that was chosen. For each chosen monster, one normal version of that monster is added to the dungeon, so that each monster appears at least once.

```python
# grab new monsters, add one normal of each.
current_difficulty = 0
new_dungeon_monsters = []
monster_type_list = []
for i in range(new_number):
    if i == 0:
        # choose first monster
        new_monster_type = random.choice(self.all_monsters)
        dungeon_monster_theme = new_monster_type.theme
    else:
        # choose additional monsters
        theme_weight = []
        for entry in self.all_monsters:
            if entry.theme == dungeon_monster_theme:
                theme_weight.append(self.monster_theme_bias)
            else:
                theme_weight.append(1)
        new_monster_type = 0
        while new_monster_type in monster_type_list or new_monster_type == 0:
            new_monster_type = random.choices(self.all_monsters, theme_weight)[0]
    # add a normal monster and its difficulty
    current_difficulty += new_monster_type.difficulty
    new_monster = [new_monster_type, 1, 0]
    monster_type_list.append(new_monster_type)
    new_dungeon_monsters.append(new_monster)
```

Figure 5: Mutating Monsters: choosing types

After the monster types have been chosen, monsters of those types need to be added until the dungeon has reached its desired difficulty. To do so, a random monster type is chosen. If there are normal monsters of that type present, there is a 50% chance that it removes one normal monster of that type and adds an elite monster of that type. If that does not happen, or if there are no normal monsters of that type, a new normal monster is added instead. This is done because it makes sure that both elite and normal monsters of each type can be present, and because elite monsters are considered to be twice as difficult as normal monsters of the same type, this will always increase the difficulty by the same amount for each monster type.

```python
while current_difficulty < new_difficulty:
    # choose a random monster
    # 50/50 chance of; adding a normal or changing a normal into an elite (if any normal are present)
    added_monster = random.choice(new_dungeon_monsters)
    add_elite = random.randint(0, 1)
    if add_elite == 1 and added_monster[1] > 0:
        added_monster[1] = added_monster[1] - 1
        added_monster[2] = added_monster[2] + 1
    else:
        added_monster[1] = added_monster[1] + 1
    current_difficulty += added_monster[0].difficulty
```

Figure 6: Mutating Monsters: adding monsters

Map Mutation

The mutation of the “map” changes the layout of the room tiles and the placement of components within the rooms. It is more intricate than the other mutations because the map is built up from different room tiles that only fit together in a specific way. The rooms are chosen in a similar way to the monster types, where the first chosen room sets the theme, and all other choices have a bias for rooms of that theme. Once a room is chosen, unless it is the first room, it needs to be connected to the rest of the dungeon. This is done through ‘links’, which are the jigsaw-like parts of the room tiles that are either a gap or an extension at the edge of a room tile, called ‘entries’ and ‘exits’ respectively by the program. A list is kept with all links of the existing dungeon that a new room can attach to. After a new room is chosen, the program compares all the links of the new room with the links of the already existing dungeon and checks if they can be connected that way. There are two conditions that need to be fulfilled in order for the connection to be valid. First of all, they need to be physically connectable, meaning that if one link is an entry, the other needs to be an exit and vice versa. The other condition has to do with the hex formed by the connection. Every link is half of a hex tile, so that two connected links form a new hex. However, hexes can be cut in half in two different ways; either from corner to corner, or from the middle of one edge to the middle of another. A valid connection needs both links to be cut in half in the same way. In the program, this is resolved by their orientation. Every link also has a value for its direction. There are twelve

possible directions for a link, which have been numbered so that they coincide with the hours on a clock. When doing this, all the links that cut a hex from edge to edge have an even number as its direction, and all links that cut a hex from corner to corner have an odd number. So, in order for the connection to be valid, both links should either have an odd or an even direction.

After all possible connections have been collected, a random one is chosen to be the way that the room connects to the dungeon. Should there be no possible connections, then the room cannot be connected to the rest of the dungeon, and thus the program should look for a different room to connect.

```
def connect(self, open_links, new_room):
    possible_connections = []
    # check possible connections:
    # for each room, check for all its links with this room's links whether they can connect.
    for old_room in open_links:
        for old_link in open_links[old_room]:
            for new_link in new_room.links:
                if (old_link[3] % 2 == new_link[3] % 2) and (old_link[4] != new_link[4]):
                    # add this as a possible connection.
                    possible_connections.append([old_room, old_link, new_link])
    if not possible_connections:
        return 0
    else:
        chosen_connection = random.choice(possible_connections)
        return chosen_connection
```

Figure 7: Map Mutation: choosing connections

After the connection has been chosen, the new room may need to be rotated in order to fit with the rest of the dungeon. This is once again done using the directions of the links. In order for two links to connect, their directions need to oppose, for example, when one points to the left, the other should point to the right. In the program, this translates to a difference of 6 between the direction numbers, so the new room will be rotated until the direction of its chosen link has a difference of 6 when compared to the chosen link of the existing dungeon.

As a final part of mutating the map, the placement of components within the dungeon is also randomized. A list is made of all coordinates of the dungeon, and then each component gets one of these coordinates as its placement. For most of the components, this choice is made at random, but for the start locations, it is ensured that these are adjacent to a wall and grouped together.

Fix

The fix part of the program can be seen as an extension of mutation, except that it happens a step later. It is mainly there to ensure that the placement attribute of the new dungeon lines up with the actual components of the dungeon, which may happen since those are not mutated together. To fix this, the program goes through every component (start, monsters, obstacles, traps, hazardous terrain, difficult terrain, chests and coins) and compares the amount of that component with the number of coordinates in the placement attribute that is associated with that component. If the component has too many coordinates, it chooses a number of random coordinates from the associated list in the placement attribute equal to the component amount. If the component has too few coordinates, the program instead adds a number of new coordinates equal to the difference. These coordinates are chosen in the same way as the placement mutation described above.

Since the implemented version of the algorithm does not use goal or rules that reference specific components, the second part of the fixing process as described in Section 3.3 is not necessary, and thus has not been implemented.

Validity

No matter how good or bad the dungeon is, any dungeon in the population should be a valid dungeon. This means that someone should be able to physically make the dungeon using only what is in a single Gloomhaven box, and that the dungeon should theoretically be winnable, regardless of

the choice in characters. Using the check_validity function, the program ensures that this is true through three checks, namely overlap, reachability, and limits.

In the overlap check, the program searches the dungeon's coordinates list for any entries that are in there more than once. Since the coordinates are added to this list on a room by room basis, any duplicate values mean that the same coordinate is part of multiple rooms, and thus the dungeon cannot be physically created without overlapping rooms. This renders the dungeon invalid.

In the reachability check, the program checks whether the goal of the dungeon is reachable using only basic move and attack actions, since those are available to every character. While the exact goal depends on the dungeon, the current iteration of the program only creates dungeons with the "kill all monsters" goal. Since basic attack actions are usable when next to the target of the attack, it is important to make sure that every monster can be reached using basic movement.

In order to actually do this, the program takes the coordinates of the dungeon, and uses it as a grid for a search algorithm to find each starting location and each monster, beginning the search from one of the starting locations. Difficult terrain, hazardous terrain and traps, while preferably avoided, can be moved through, and thus do not matter for this search. However, characters are not guaranteed to have a way to move through obstacles, and thus obstacles are removed from the coordinates that are used for the grid before the search. This is done to prevent obstacles from, for example, blocking the path to a door.

For the search itself, a depth-first search is used, since it is only relevant if the monster can be reached, not how long it would take to reach the monster. If there is a monster that cannot be found in the search, not every monster in that dungeon can be killed using only move and attack actions, and thus there is no guarantee that the dungeon can be completed by any character. This renders the dungeon invalid.

```python
def reachability_check(dungeon):
    reachability_valid = True
    start_locations = dungeon.placements["start"]
    monster_locations = dungeon.placements["monsters"]
    main_start = start_locations[0]
    dungeon.get_coordinates()
    coordinates = dungeon.coordinates.copy()
    coordinates.extend(dungeon.connection_coordinates)
    if len(dungeon.placements["obstacles"]) != 0:
        for obstacle_hex in dungeon.placements["obstacles"]:
            coordinates.remove(obstacle_hex)
    # do the actual search
    for start in start_locations:
        path = search(coordinates, main_start, start)
        if not path:
            reachability_valid = False
    for monster in monster_locations:
        path = search(coordinates, main_start, monster)
        if not path:
            reachability_valid = False
    return reachability_valid
```

Figure 8: Reachability Check

In the limits check, the program checks whether the dungeon can be created using the limited amount of components that come in the Gloomhaven box. This is necessary since the Gloomhaven box only comes with limited components, and it is not possible to quickly create more of those components when physically building the dungeon in the way that would be possible when creating the dungeon digitally. While it could be relevant for the theme of the dungeon, the limits check does not check whether there are also enough components that fit with the theme of the room they are placed in. In some cases, this check is relatively simple. In the cases of coins, chests, doors and traps, there is simply a number of these components in the box, and should the created dungeon exceed this number, it is not a valid dungeon.

The limits check for obstacles, hazardous terrain and difficult terrain is slightly more involved, since these are often part of double-sided tiles, which may have a different type on either side, such as those in Figure 9b. In addition to that, some of these tiles are also part of a two-hex or three-hex component, which cannot be split up into single-hex tiles. In order to resolve the first problem, new inequalities are added to the check, to make sure that there are always enough tiles of

each type. While first the limit of every type is tested, the program then tests the limits of all combinations of types. In these combinations, each tile that has one type on one side but another type on the other side is only counted once. For example, there are 24 tiles that are obstacles on one side and difficult terrain on the other. So even though the obstacle limit is 95 and the difficult terrain limit is 40, the limit of obstacles + difficult terrain is only 111, since these 24 tiles are only counted once.

To resolve the problem of two-hex and three-hex components, the decision was made to only include the two-hex tiles in the limit. This decision was made since it is likely that there are adjacent hexes of the same type if the limit of that type becomes relevant. This is less likely to be the case with three-hex components, especially since these are all shaped in a triangle, and not including the two-hex components would make the values for the limits very low. For example, hazardous terrain has only six one-hex components, and only one of those does not have difficult terrain or an obstacle on the other side.

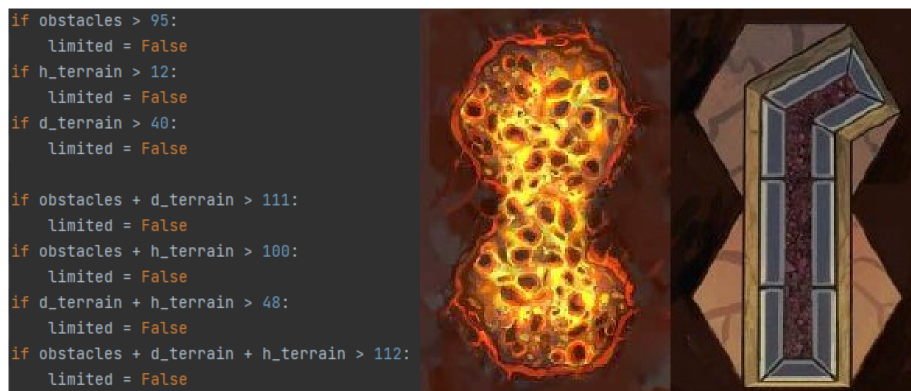

Figure 9: (a) combined limits, (b) double-sided tiles

The final component limit that is included in the limits check is the limit of monsters. There is a limited number of standees for each monster type in the box. This limit is even more important than the other limits, since the standees also come with a number that is used to track which exact monster it is, so quickly adding more monsters is a more complex task than quickly adding more terrain or obstacles. While the original Gloomhaven scenarios only practice this limit per room and not per dungeon, the program limits the amount of monsters of a single type over the entire dungeon, since the program is made with the assumption that the entire dungeon is set up before playing.

To determine whether the monsters in the dungeon conform to this limit, the program goes through each used monster type, and then adds the normal and the elite monsters of that type together to get the number of standees of that type that are necessary to create the dungeon. It then compares this number to the amount of standees of that type in the box, which is an attribute that is saved in the Monster class of each specific type. If any of the monster types in the dungeon exceed this number, there are not enough standees of that monster, and thus the dungeon is invalid.

```
for monster_type in dungeon.monsters:
    monster_count = 0
    monster_data = dungeon.monsters[monster_type]
    monster_class = monster_data[0]
    monster_count += monster_data[1]
    monster_count += monster_data[2]
    if monster_count > monster_class.max_amount:
        limited = False
return limited
```

Figure 10: Monster Limits

After the validity check, three different things can happen. If the dungeon passes all three checks, it is considered a valid dungeon, and it continues to the fitness step where it will get a score and be put into the population. If the dungeon is invalid because of reachability or limits, it is discarded as it is an invalid dungeon. If the dungeon is invalid because of overlap however, the dungeon map, which is the only thing that can cause the overlap, is mutated until there is no longer any overlap. This is done because there is a relatively high chance that a randomly generated map has overlap, and if such dungeons would simply be discarded, there would be a strong and unintended bias to dungeons that have no mutated map, since the maps of the initial population never overlap.

Fitness

The final part of the evolutionary cycle is the application of the fitness function. Every dungeon that makes it through the validation stage, as well as the initial population dungeons, will get a fitness score that is a rough indication of how enjoyable the dungeon would be to play. The dungeon is judged on five categories, with each category returning a score between 0 and 1, based on how much the dungeon deviates from an ideal value. The five categories are difficulty, size, complexity, theme, and clutter.

The difficulty score tells how hard it is to win the dungeon. This difficulty is separate from the game inherent difficulty-level of 0 to 7 that is chosen by the players. The difficulty is calculated by adding up all the difficulty values of the monsters in the dungeon, which is a number given for each monster type, with elite monsters having their difficulty doubled. The difficulty numbers for each monster type come from the Gloomhaven website [13]. The ideal difficulty is based on a statement by Gloomhaven designer Isaac Childres [16], which should be between 15 and 18 for a two-player dungeon. However, since all monsters have a difficulty that is either a whole number or a whole number and a half, the choice has been made to double all difficulty values. This places the ideal difficulty between 30 and 36. Based on this, an ideal difficulty of 34 was chosen for the fitness function.

The size score reflects the number of hexes that are part of the dungeon. An ideal size of 90 was chosen for the fitness function. This number is the mean size of all dungeons that were used as the initial population.

The complexity score tells how much complexity the additional rules add to the dungeon. Since most dungeons have at least one special rule, the ideal complexity is set to 1, which means one extra rule is preferred to make the dungeon slightly different, but not too complex. In later iterations that also have varied goals, more complex goals will also be judged by this score.

The theme score reflects the thematic consistency of the dungeon. Theme is separated into two categories; monster theme, and map theme. The ideal theme is 0, which means that there are no different themes than the main themes. Any additional theme used reduces the theme score. The choice was made to use the factorial of the number of themes for rooms and dungeons separately. This was done to ensure that every additional theme added has more impact than the previous one. The theme of a room tile is one of four themes based on its art style. The theme of a monster is based on their name and on how often they appear with other monsters, as well as their concept. These themes were decided by the writer of the program, so opinions on the choices might differ.

The clutter score reflects how full the dungeon is. It is calculated as a ratio between the number of occupied hexes as compared to the total amount of hexes in the dungeon. Hexes can be occupied by starts, obstacles, hazardous or difficult terrain, traps, coins, chests or monsters. This fitness score makes sure that the output dungeon is neither too cluttered, nor too empty. The ideal clutter score is 0.344. This is the mean value of all clutter scores of the dungeons in the initial population.

For every score with the exception of theme, the score is calculated from the dungeon's value using the following formula.

$$1 - \left| \frac{ideal - value}{ideal} \right|$$

The overall fitness score of a dungeon is determined by these scores, which each specific score multiplied by a weight for that category. Using the results of the research in Section 2, the weights have been chosen to be 4 for difficulty, 2 for theme and clutter, and 1 for size and complexity. Any references to weighted dungeons in this report refer to dungeons generated using these weights. During the experimentation in the next section, there is also a need for a reference to test these weights. This reference has all weights set to 1, which means that the fitness function gives the same results as when no weights would be used in the first place. Therefore, dungeons generated with this fitness function will also be referred to as unweighted dungeons.

5. Experimentation

While all of the research questions have been answered in theory, a user test is needed to see if the answers are actually correct. The main research question, "*How can evolutionary game design be used to generate random dungeons of a given game system?*", is answered in the implementation section. After all, any dungeon that passes the validity check should be a viable dungeon, and thus the algorithm generates dungeons. However, there is also the implication that these generated dungeons should be "good". While it is debatable what would actually make a dungeon good or bad, it seems reasonable to say that enjoyment is at the very least a significant factor. Because of this, the questions of the user test will be about the participant's enjoyment. Since the enjoyment of something is difficult to grade without context, all questions that ask for a degree of enjoyment do so relative to a chosen reference dungeon, which is the first dungeon in the Gloomhaven scenario book.

The main aim of the user test is to check the answers two of the research questions. For the main research question, this was done indirectly. Should the participants experience the evolutionary generated dungeons to be at least as enjoyable as the original dungeons, it would strongly imply that the used algorithm does indeed generate proper dungeons, and that it is indeed a proper answer to the main research question. The second research question that the user test would answer is the sub-question "*What are the optimal weights in the fitness function in order to generate the best Gloomhaven dungeons?*". This was done by presenting the participants with two generated dungeons, one dungeon with weights that were optimized using theory, and the other with equal weights for all factors. If the theoretically determined weights are indeed the optimal weights, the weighted dungeons should have a higher enjoyment score than the unweighted dungeons.

In addition, the user test will provide additional insight on the factors that are used by the fitness function. This includes new factors which might contribute to the enjoyment of the dungeon but were not included, as well as whether the chosen factors have been implemented in a way that matches with user experience.

5.1 Experiment Setup

For the user test, the participant was asked to play through three different Gloomhaven dungeons. The first of these dungeons is also the first dungeon in the Gloomhaven scenario book. This first dungeon was used to teach the participant the game, and to provide context and a reference for the other dungeons. The other two dungeons were ones generated by the evolutionary algorithm, with one dungeon using the predicted optimal weights, and the other dungeon using uniform weights. After playing one of these dungeons, the participant was asked to fill in a questionnaire about the enjoyment of the dungeon, as compared to the reference dungeon that was played first. For the user test, some additional choices were made that warrant additional explanation:

- While the algorithm usually has a default ideal difficulty of 32, the reference dungeon has a difficulty of 24, which seemed like too big of a difference. Therefore, the choice was made to put the ideal difficulty at 28 for the dungeons that would be used in the user test. This number is closer to the reference difficulty, but still slightly more challenging. This slightly

increased difficulty is supposed to compensate that for many participants, the first dungeon was the very first time they have played Gloomhaven.

- The choice was made to generate two new dungeons for each participant, instead of each participant testing the same two dungeons. This is because the true goal of the experiment is to test the evolutionary algorithm that created these dungeons. If the same two dungeons were used for the entire user test, the test results would give information on those specific dungeons instead of on the evolutionary algorithm and the chosen weights.
- Since Gloomhaven is a game that generally requires at least two characters to play through a dungeon, the dungeons were tested with both the participant and the researcher as players. While this does add more risk that the researcher influenced the research outcome, it was considered better than the two alternatives. The first alternative was to have two participants play the game at the same time, but this would require twice the amount of testers, since it is important that different dungeons are tested each time, as stated above. The second alternative was to have the participant play two characters at once. While Gloomhaven does support this playstyle, it would make the game significantly more complicated, especially for new players, to the point that it did not seem reasonable to ask this from participants.

5.2 Results

The relevant information for testing the answer to the first research question is the enjoyability of the dungeons. That can be summarized as in table 1. The number in each cell represents the amount of times that answer was given. An empty cell means that the answer was not given.

	Significantly less enjoyable	Less enjoyable	Similar	More enjoyable	Significantly more enjoyable
Overall experience		1	1	5	1
Difficulty	1	1			6
Dungeon size			4	4	
Complexity			1	3	4
Theme			1	4	3
Layout			1	3	4

Table 1: User test result of 4 participants and 8 dungeons.

The relevant information for testing the answer to the second research question is the enjoyability of the weighted and unweighted dungeons. This is summarized in table 2.

	Significantly less enjoyable	Less enjoyable	Similar	More enjoyable	Significantly more enjoyable
Overall experience (weighted)		1	1	1	1
Overall experience (unweighted)				4	

Table 2: User test result comparing weighted and unweighted dungeons. Result of 4 participants, each playing 1 dungeon of each category.

5.3 Conclusion

The user test was done to help answer two research questions, namely "*How can evolutionary game design be used to generate random dungeons of a given game system?*" and "*What are the optimal weights in the fitness function in order to generate the best Gloomhaven dungeons?*".

While these questions have already been answered in theory, the user test is meant to test whether these answers also hold up in practice. Therefore, the questions that the user test will try to answer are "*Are the dungeons generated with the implemented evolutionary algorithm at least as*

enjoyable as the original dungeons?" and "*Are the dungeons generated with theoretically optimized weights more enjoyable than the dungeons generated with uniform weights?*"

In order to do the statistical analysis on these results, the ordinal results have been translated into numerical ones. Significantly less enjoyable will be treated as a value of -2, less enjoyable as -1, similar as 0, more enjoyable as 1 and significantly more enjoyable as 2.

Dungeon Enjoyability

To answer the question "*Are the dungeons generated with the implemented evolutionary algorithm at least as enjoyable as the original dungeons?*", the overall experience of the generated dungeons as compared to the reference dungeon will be tested. The following data was used for this test.

	Significantly less enjoyable	Less enjoyable	Similar	More enjoyable	Significantly more enjoyable
Overall experience		1	1	5	1

Table 3: Excerpt from table 1

Because of the small sample size, the statistical analysis of the overall enjoyability will be done through a T-test. The aim is to show that the generated dungeons are more enjoyable than the reference with 95% confidence. While the research question asks whether the dungeons are at least as enjoyable, changing it to the dungeons being more enjoyable should not change the outcome.
Form the table above, the following values are collected:
Sample mean: $\overline{X} = 0.75$
Sample Variance: S = 0.89
H_0: $\mu = 0$
H_1: $\mu > 0$

The formula for test statistic T is as follows:

$$T = \frac{\overline{X} - \mu_0}{S / \sqrt{n}}$$

Using the values above, the test statistic is T = 2.384. For 95% certainty, the rejection region with 7 degrees of freedom is $T \geq 1.895$. Therefore, H_0 can be rejected. This means that the dungeons generated with the evolutionary algorithm are more enjoyable than the reference dungeon with a 95% certainty.

Dungeon Weights

Unfortunately, due to the small sample size and the fact that the weighted experience has a significantly larger variance than the unweighted experience, it is very difficult to do a statistical analysis on the difference in experience between weighted and unweighted dungeons. In fact, the sample variance of the unweighted dungeons being 0 means that several relevant statistical techniques become impossible. However, since the mean of the weighted dungeons is significantly lower than the mean of the unweighted dungeons, namely 0.5 as compared to 1, it seems unlikely that the weighted dungeons provide a better experience than the unweighted dungeons. This means that the proposed optimal weights are unlikely to be the actual optimal weights, and that the real optimal weights are probably closer to the uniform weights.

6. Discussion

First of all, it is worth noting that the algorithm works. It does indeed generate entirely new Gloomhaven dungeons using evolutionary game design, which seem to have a high chance of being at least as enjoyable as the official dungeons. However, there are still many improvements that can be made in future iterations.

For this discussion, two types of improvements will be mentioned. Necessary improvements are ones that resolve a problem with the current iteration that prevent it from working perfectly. Additional improvements, on the other hand, are not truly necessary, but do improve the quality of the resulting dungeons.

6.1 Necessary Improvements

There are four necessary improvements that will have to be made before this algorithm would be available for public use. These improvements are visual output, adapting for player count, preventing flawed maps, and preventing incorrect limitations. Visual output is necessary for users to be able to physically create the generated dungeons, without having to decipher the output. The current output uses rooms, connections and component coordinates to describe dungeons, and while that does describe the entire dungeon, creating the dungeon from this output takes a lot of time.

Adapting for player count is necessary to allow people to play the dungeon with three or four people, instead of only with two people as the current algorithm allows. In the original game, the only part of a dungeon which changes with the amount of players is the monsters. More specifically, the total difficulty of the monsters is based on the total amount of players. So in the algorithm, the only part that really needs to change is that the ideal difficulty will be dependent on the amount of players that the dungeon is made for. However, since the monster difficulty after mutation is based on the monster difficulty before mutation, the algorithm will need many cycles to create dungeons that are close to the new ideal difficulty. This can be resolved by using the original dungeons in the initial population for the specific player count, instead of defaulting to two players.

The other two necessary improvements should remove two remaining flaws in the algorithm. While both of these are rare, they make that the algorithm may still make invalid dungeons. The first of these improvements is to fix the flawed map generation. This flaw has to do with the overlap of edges of one room tile with hexes of another room tile. The other improvement has to do with the limitations, which currently does not take into account that two-hex and three-hex components exits. In order to resolve this, additional steps should be added to determine where those components can be placed, in order to get full certainty that any dungeon within the limitations can be created.

6.2 Additional Improvements

There are also several improvements possible that are not necessary, but do significantly improve the algorithm and its outputs. The main additional improvement is the revision of the fitness scores. While the fitness function itself seems to work well, how the scores are obtained is rather simplistic and misses out on a lot of detail. Instead of each score only depending on one attribute, and each attribute only contributing to one score, a better scoring system would be more complex. While further research is required to find out what exactly contributes to each factor of the fitness function, there are some examples. Difficulty, for example, currently only depends on the difficulty of the monsters within the dungeon. While this is the main factor of a dungeon’s difficulty, more difficult terrain, hazardous terrain and traps, as well as many special rules, also increase the dungeon’s difficulty and should ideally weigh in on this factor. As another example, complexity currently only depends on the special rules that are added to the scenario. However, many monsters have high shield values, such as flame demons, inflict a condition when hitting a character, such as black imps, retaliate, such as harrower infesters, or even have unique abilities or ability cards. Including such monsters in a dungeon forces the players to use different tactics, which increases a dungeon’s complexity.

Continuing with the fitness function, an improvement can be made by making the scoring system less punishing for values close to the ideal. For example, the vast majority of dungeons created for the user test had a difficulty score of exactly 1, which means that the difficulty of those dungeons was exactly at its ideal value of 28. However, difficulties of 27 or 29 would probably not have a massive impact on enjoyability, and only taking dungeons with a difficulty of exactly 28 means that a lot of other, still enjoyable dungeons were not considered. This problem can be

resolved in two ways. Either the ideal values can be replaced with ideal ranges instead, or the formula that calculates the score from the value can be adapted to be less punishing for values near the ideal.

Another additional improvement rarely shows up, but solving it could remove some of the disappointing output dungeons. This improvement has to do with the fact that placement only mutates when the map does. This is not a problem if the output dungeon has a mutated map. However, if the output dungeon does not have a mutated map, it has one of the maps of the original Gloomhaven dungeons that were used as the initial population. Because placement only mutates with the map, this means that this map will look very similar to a dungeon in the book, except for probably different monsters and maybe a few components that are missing or added. Since the algorithm is supposed to generate unique dungeons, this is not desired. This problem could be solved by always mutating the placement if a new dungeon has inherited the map attributes directly from one of the initial dungeons.

Finally, a significant improvement can be made by including more parts of the original game, such as goals that are different from “kill all monsters” and rooms that are combined into a single larger room instead of being connected with doors. While this is not required for the algorithm to work, it does create more possibilities for unique dungeons. This is, however, also the most extensive improvement, since it requires changing a lot of steps of the algorithm, including the mutation process and the fitness function.

6.3 User Test Validity

While the results of the user test were positive, there are several factors that may have created a bias to more positive results.

- Since the games were played together with a researcher, as explained in Section 5.1, there might have been unintended social pressure towards the participants to judge the scenarios more favorably.
- All results of the user test were based on the experience of the dungeon as compared to a reference dungeon. Since the reference dungeon was the same for all participants, there is the assumption that it represents the official Gloomhaven dungeons well, which is not guaranteed to be the case. Especially since it is the first dungeon of the game, it is less complicated and less difficult than most Gloomhaven dungeons, which may have created a bias for people who enjoy more difficult and more complicated games.
- Since the reference dungeon was always played first and also used to teach players the rules of Gloomhaven, the players had more experience with the game and its rules when playing through the generated dungeons.
- In an attempt to balance the learning issue above, the generated dungeons were made slightly more challenging. However, this may have created a bias for players who enjoy more challenge or who found the reference dungeon to be too easy.

6.4 Future Research

There are several parts of this project that still require further research. One of those, which attributes exactly contribute to which part of the fitness score, has been mentioned before. In addition to that, the user test has shown that the dungeons created with uniform weights of the fitness function outperformed those created with the proposed ideal weights, so the actual ideal weights seem to be closer to the uniform weights than the proposed ideal values. More research is required to find out what the actual ideal values for the fitness function’s weights are.

Finally, while the current algorithm works for the generation of Gloomhaven dungeons, the original research question was about dungeon generation for a given game system. For this project, Gloomhaven was chosen as the given game system, but further research is required to see if a variant of this algorithm can also be used to generate dungeons of other systems, or if those systems will require an entirely new algorithm.

In theory, the basis of the used algorithm can be used for a variety of other systems. While the exact code will differ, the crossover and mutation categories of rules, map, monsters, environment and treasure, as well as the fitness categories of difficulty, size, complexity, theme and clutter, can be used as a basis of an evolutionary algorithm generating dungeons for another game or system. The other system can be one without limited components, such as dungeons for a tabletop roleplaying game like Dungeons & Dragons, in which case only the limits of the validity check need to be removed. A very similar algorithm could be used to create maps for other games that also use modular maps, but not necessarily the same dungeon structure. This could be used to replace random map generation, for example to create a more thematically consistent layout for Betrayal at the House on the Hill. It could also be used as a tool to help in the design of new scenarios, such as to create a framework for a scenario for Mansions of Madness. In these situations, the crossover and mutation categories need to be adapted to suit the system, but the fitness categories are general enough that they may work for such systems as well.

References

[1] I. Childers, “Gloomhaven” (2017)
[2] Boardgamegeek ranking, https://boardgamegeek.com/browse/boardgame (accessed 17/02/2023)
[3] J. Togelius, E. Kastbjerg, D. Schedl, G.N. Yannakakis, “What is procedural content generation? Mario on the borderline.” In: *Proceedings of the 2nd Workshop on Procedural Content Generation in Games* (2011)
[4] J. Brown, M Scirea, “Procedural Generation for Tabletop Games: User Driven Approaches with Restrictions on Computational Resources.”, doi: 10.1007/978-3-030-14687-0_5.
[5] N. Shaker, J. Togelius, and M.J. Nelson, “Procedural content generation in games.” Springer, 2018
[6] V. Valtchanov, & J.A. Brown, “Evolving Dungeon Crawler Levels with Relative Placement”, doi: 10.1145/2347583.2347587
[7] C. Browne, & F. Maire, “Evolutionary game design,” in *IEEE Transactions on Computational Intelligence and AI in Games, 2(1)*, 2010, pp. 1-16. doi: 10.1109/TCIAIG.2010.2041928
[8] R. Hunicke, M. Leblanc, & R. Zubek, “MDA: A formal approach to game design and game research,” Paper presented at the *AAAI Workshop – Technical Report, WS-04-04*, (2004), pp. 1-5.
[9] G. Costikyan, “I have no words & i must design: Toward a critical vocabulary for games,” (F. Mäyrä, Ed.), in *Proceedings of Computer Games and Digital Cultures Conference*, (Tampere, Finland: Tampere University Press), 2002, pp. 9-33
[10] J.M. Thompson, “Defining the Abstract.” The Games Journal, July 2000.
[11] W. Kramer, “What Makes a Game Good?” in *Game & Puzzle Design*, vol. 1, no. 2, 2015, pp. 84-86
[12] C. Browne, “Automatic generation and evaluation of recombination games,” Ph.D. dissertation, FIT, Queensland Univ. Technol., Brisbane, Australia, 2008.
[13] Gloomhaven Monster Spoilers, by Cephalofair Games, https://cephalofair.com/pages/gloomhaven-monster-spoilers (accessed 16/02/2023)
[14] Red Blob Games, “Hexagonal Grids.” redblobgames.com, https://www.redblobgames.com/grids/hexagons/ (accessed 16/02/2023)
[15] Virtual Gloomhaven Board, by Purple Kindom Games, https://vgb.purplekingdomgames.com/ (accessed 17/02/2023)
[16] Submissions for fan-made scenarios are open! https://boardgamegeek.com/thread/1430857/submissions-fan-made-scenarios-are-open (accessed 17/02/2023)

Appendix A: User Test Questionnaire

This questionnaire is specifically about your personal experience of the dungeon you just played through in comparison to the reference dungeon you played as introduction.

Below is a table listing several categories that may have impact on the enjoyability of a dungeon. How did you, for each category, experience the enjoyability of the dungeon you just played through in comparison to the reference dungeon?

	Significantly less enjoyable	Less enjoyable	Similar	More enjoyable	Significantly more enjoyable
Overall experience					
Difficulty					
Dungeon size					
Complexity					
Theme					
Layout					

For each of the categories that were significantly less or significantly more enjoyable, what was the cause of this?

Where there any other things that were significantly **more** enjoyable in this dungeon than in the reference dungeon? If so, please mention what.

Where there any other things that were significantly **less** enjoyable in this dungeon than in the reference dungeon? If so, please mention what.

Appendix B: User Test Dungeons

Images were made using the Virtual Gloomhaven Board [15]

Weighted Dungeons:

Aurora Ruins:
Additional Rules: Add three -1 cards to each character’s attack modifier deck as a scenario effect.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
9.68	1.0	0.92	1.0	0.9	0.98

User Test: More enjoyable than the reference.

Raided Barrow:
Additional Rules: All characters start with DISARM as a scenario effect.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
9.64	0.93	0.93	1.0	1.0	0.99

User Test: Significantly more enjoyable than the reference.

Mountaintop Temple:
Additional Rules: None.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
9.71	1.0	0.89	1.0	0.95	0.96

User Test: About as enjoyable as the reference.

Slumbering Volcano:
Additional Rules: None

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
9.72	1.0	0.94	1.0	0.95	0.94

User Test: Less enjoyable than the reference.

Unweighted Dungeons

Bandit Hideout:
Additional Rules: All characters start with IMMOBILIZED as a scenario effect.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
4.82	1.0	0.94	1.0	0.9	0.97

User Test: More enjoyable than the reference.

Temple of Flame:
Additional Rules: None

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
4.81	0.96	0.92	1.0	0.95	0.98

User Test: More enjoyable than the reference.

Drake Kennels:
Additional Rules: Add three CURSE cards to each character’s attack modifier deck as a scenario effect.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
4.85	1.0	0.97	1.0	0.95	0.93

User Test: More enjoyable than the reference.

Overgrown Ruin:
Additional Rules: All characters start with DISARM as a scenario effect.

Dungeon Results:

Total Score	Difficulty	Size	Complexity	Theme	Clutter
4.84	0.93	0.96	1.0	1.0	0.95

User Test: More enjoyable than the reference.