FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Vegricht

Evolutionary Algorithm-Based Procedural Level Generator for a Rogue-like Game

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jakub Gemrot, Ph.D.
Study programme:	Computer Science
Specialization:	Programming and Software Systems

Prague 2018

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 in Prague 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 his encouragements, patient guidance, and advices throughout my time working on this thesis, as well as developing the game in form of individual software project.

Title: Evolutionary Algorithm-Based Procedural Level Generator for a Rogue-like Game

Author: Jan Vegricht

Department: Department of Software and Computer Science Education

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

Abstract:

Rogue-like games are genre with long tradition in game industry. One significant factor commonly associated with this genre is procedural level generation. The goal of this thesis is to design and implement a level generator for one concrete rogue-like game using evolutionary algorithms as main means of generation. Methods and results are then compared to non-evolutionary alternative algorithms, attempting to generate comparable solutions. The results seem to indicate that while evolutionary algorithms can be used to generate dungeons, practicality of this approach is for the most part limited.

Keywords: evolutionary algorithms, procedural generation, constrained optimization, rogue-like

Table of Content

1 Introduction 4
1.1 Rogue-like Games and Motivation 5
1.2 Procedural Map Generation 7
1.3 Thesis Structure and Goals 7

2 Problem Analysis 9
2.1 Game Design 9
2.1.1 Winning and Losing 9
2.1.2 Map and Visibility 9
2.1.3 Turns 9
2.1.4 Combat 10
2.1.5 Items 10
2.2 Formal Definition of the Game 10
2.2.1 Level Representation 13

3 Evolutionary Algorithms 16
3.1 Genetic Algorithms and Chromosomes 16
3.2 Initial Population 17
3.3 Selection 18
3.4 Fitness Function 20
3.5 Recombination 21
3.6 Mutation 22
3.7 Generation Replacement and Elitism 24
3.8 Terminal Condition 24
3.9 Multi-objective Optimization 25
3.9.1 Pareto-optimal Front 26
3.9.2 Locating Solution 27
3.9.3 Nondominated Sorting Genetic Algorithm (NSGA) 28
3.9.4 Nondominated Sorting Genetic Algorithm Revisited (NSGA-II) 29
3.9.5 Hypervolume 31

4 Approaches and Related Literature 35

4.1 Refining the Paradigm of Sketching in AI-Based Level Design 35
4.2 Automatic Generation of Fantasy Role-playing Modules 36
4.3 Game Level Layout from Design Specification 37

5 Proposing Evolutionary Solution 39
5.1 Phase 1 – Room Distribution Phase 40
5.1.1 Encoding of Individuals 40
5.1.2 Initial Population 41
5.1.3 Fitness Functions 41
5.1.4 Operators 42
5.2 Phase 2 – Structure Phase 43
5.2.1 Encoding of Individuals 43
5.2.2 Initial Population 45
5.2.3 Fitness Functions 46
5.2.4 Operators 47

6 Proposing Alternative Solution and Post-Processing 48
6.1 Phase 1 – Room Distribution Phase 48
6.2 Phase 2 – Structure Phase 49
6.3 Post-Processing and Populating the Dungeon 52

7 User Documentation 55
7.1 The Game 55
7.1.1 Loading a Map 55
7.1.2 Controls 55
7.2 Level Generator 56
7.2.1 Launching the Generator 56
7.2.2 Configuration File Formatting and Expected Variables 57

8 Code Decomposition 59
8.1 Technologies and External Dependencies Used 59
8.2 The Game 60
8.2.1 Component System 60
8.2.2 Scenes and Scene Manager 61

8.2.3 Arena Scene 62
8.2.4 Character Component, Pathfinding, and AI 63
8.2.5 Content Managers 64
8.2.6 Map Blueprint 65
8.3 Level Generator 66
8.3.1 Parameters Manager and Configuration Reader 66
8.3.2 Evolution Namespace 67
8.3.3 Individuals 67
8.3.4 Operators 69
8.3.5 Phase Bridge 70
8.3.6 Thread-Safe Static Random 71

9 Results and Discussion 72
9.1 Phase 1 – Room Distribution Phase 72
9.1.1 Evolutionary Approach 72
9.1.2 Alternative Approach 75
9.1.3 Practicality Comparison 77
9.2 Phase 2 – Structure Phase 78
9.2.1 Evolutionary Approach 79
9.2.2 Alternative Approach 81
9.2.3 Practicality Comparison 84
9.3 Final Populated Levels 85
9.4 Conclusions 89
9.5 Future Work 89

Bibliography 91

List of Tables 93

List of Figures 94

List of Abbreviations 96

Attachments 97

1 Introduction

Game content creation comes with no shortage of challenges to overcome, yet faster computers with more memory, as well as constant race among game developers to top each other, create pressure for production of larger, richer and more complex worlds. Procedural content generation (PCG) has always had its place in the game industry, for creating content algorithmically, as opposed to manually, comes with set of lucrative advantages.

In this thesis, understanding of *game content* as "*aspects of the game that affect gameplay other than non-player character (NPC) behavior and the game engine itself,*" (Togelius, et al., 2011) was adopted. Examples provided in the paper include terrain, maps, dialogue, quests, characters, etc. It is worth noting, that while most game content discussed in this thesis is video game content, aforementioned definition can easily extend to board games and table top role-playing games (RPG) (e.g., map generation for D&D in [Ashlock, et al., 2014]).

As discussed above, PCG benefits from number of advantages. Among the most obvious ones belong possibility of arbitrarily long stream of content and consequentially high degree of replayability. Skyrim (Behtesda Softworks, 2011) for example, an action RPG video game, is set in by hand designed world, but revels in usage of procedural quest generator, called Radiant Quest System, designed to create new quests on the fly, appropriate to player's current level. Generating the content while the game is running is referred to as *online generation* (Togelius, et al., 2011).

Another advantage comes in form of reduced time and money requirements, as manual content creation can be quite resource intensive task. Daggerfall (Bethesda Softworks, 1996) is notoriously known for featuring absurdly large world, which Bethesda claims is about the size of Great Britain (229,848 km^2), albeit most of it is actually barren wasteland.

Lastly, PCG can be less demanding on memory usage. In their survey, (Togelius, et al., 2011) describe possible dissections of PCG algorithms based (among other things) on their input (i.e. random seed or parameter vector) or determinism (i.e.

deterministic or stochastic). It would be easy to imagine deterministic (either mostly or completely) procedure to store large chunks of content compressed into comparatively small amount of data (the parameter vector) which would only get expanded when needed. Elite (Acornsoft, 1984) serves as a good example of this, storing entire complex star systems in just couple dozens of kilobytes (Togelius, et al., 2011).

1.1 Rogue-like Games and Motivation

One game genre in particular is often associated with procedurally generated content: rogue-likes. Rogue-like games have long tradition in the game industry and over the years had impact on no shortage of game developers. Games such as Rogue (A.I. Design, 1980) and NetHack (The NetHack DevTeam, 1987), old-school ASCII games, already featured online level generators in the times when games in general have just begun gaining mainstream popularity.

Defining genre of any kind is peculiar task, as rigorous definitions are of limited application when pigeon-holing art and sometimes can give impression of needlessly restricting the artists. That being said, it is undeniable that genres, rogue-likes included, exist in art precisely to loosely group together works that share similarities with one another. A definition of rogue-likes can be found in (İzgi, 2018), one which was originally proposed in 2008 at International Roguelike Development Conference in Berlin, thus commonly refer to as *Berlin Interpretation*.

Berlin Interpretation attempts to define the genre using high value and low value factors typically associated with rogue-likes. How many of these factors a game follows is then used as criteria to determine whether it is rogue-like or not, or how much of a rogue-like it is.

The game used in this thesis was made as Individual Software Project and is loosely based on Japanese strategy RPG God Catching Alchemy Meister[1] (Eushully,

[1] Kamidori Arukemi Maisutā (神採りアルケミーマイスター) in Japanese.

2011), or more precisely, on its combat system. Needless to say the original game is more complex and contain range of mechanics unrelated to the topic at hand, however its combat system exhibits number of factors from Berlin Interpretation, which have been even more exaggerated in the derived game. Some of the factors are listed below, with explanation of what they are.

Procedural world generation was already talked about earlier in this chapter. It is worth noting however, that online generation is typically used, rather than offline, which is used this thesis.

Permanent death of player controlled characters with no option to save the progress is another mechanics used in the game. Unlike some rogue-likes, the game used in this thesis have multiple such characters, but with only limited supply of them, with all of them being procedurally generated online.

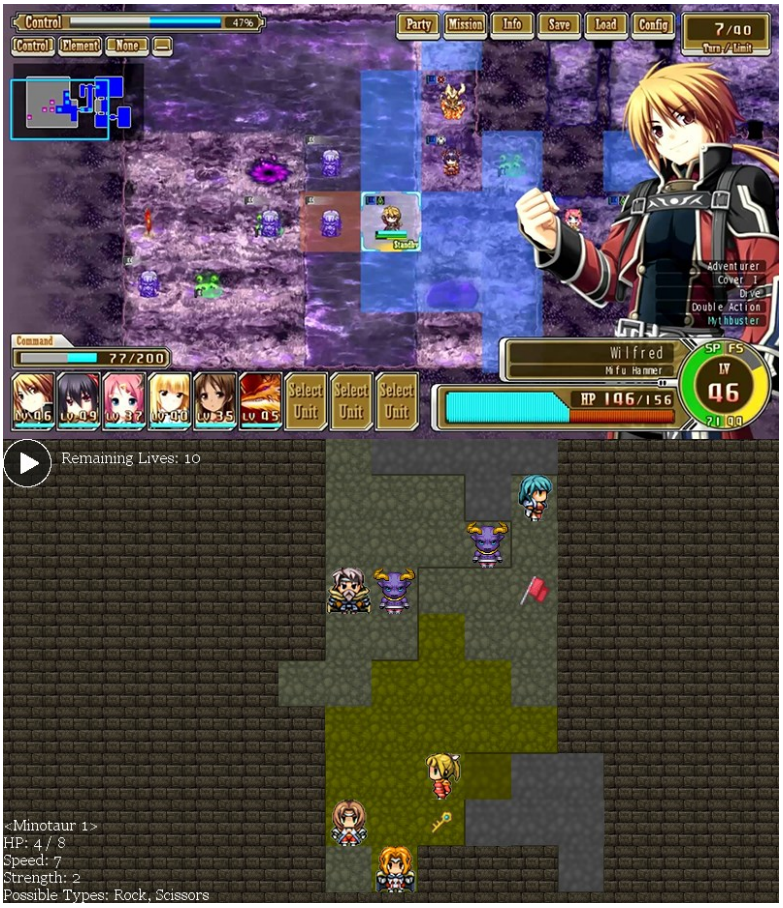

Figure 1: God Catching Alchemy Meister (top),
the game created for this thesis (bottom).

Combat is **turn-based** and takes place on **square grid-based world**. Time is not of much importance and every object belongs to a tile, with carrying capacity of one (two object cannot occupy same space at the same time). Player is **against the world**, fighting his way through hostile dungeon. Monsters do not communicate with each other.

1.2 Procedural Map Generation

The goal of this thesis is proposing an algorithm to generate levels for the game, including populating with loot and encounters. Subset of PCG approaches, called evolutionary algorithms (EA), was chosen as the main method of generation.

As briefly mentioned above, *online* generation refers to techniques where the algorithm runs while the game is running, creating content on the fly. On the contrary, *offline* generation refers to creating the content before the game is launched (Togelius, et al., 2011).

EA are a collection of methods inspired by principles of natural selection. The idea behind them is appealing and easy to understand, however they don't come without drawbacks. Generally speaking, to obtain decent solutions, one needs decently sized "population" and enough time for it to evolve, an aspect increasing required computation time. Consequentially, EA are notorious for converging rather slowly. For this reason, offline generation, rather than online, was chosen for this thesis.

1.3 Thesis Structure and Goals

The main goal of this thesis is to provide an evolutionary level generator for the game. The application will also offer non-evolutionary alternative algorithms attempting to generate similar looking results, so that the two different approaches can be compared to each other in terms of their practicality and capability to produce interesting or fun-to-play levels.

Both approaches will be capable of generating tile maps subdivided into tightly packed rooms. The arrangement of the rooms is otherwise of secondary importance, but the idea is to generate cave-like dungeons, so they should reflect that. As the

unvisited rooms in the game will not be visible for the player, the maps generated should encourage exploration, while challenging the player with monsters' lairs and boss encounters on the way.

Consequent chapters of this thesis are building of information presented in previous chapters. The second chapter sets out to provide in-depth analysis of the problem at hand, including overview of related literature, detailed description of the game and its mechanics, and formal definition of what *map* is. The third chapter provides more details on EA in general, with chapter four examining and discussing related work done on the topic. Chapter five then applies introduced concepts on the problem, with chapter six providing non-evolutionary solution for comparison.

Chapters seven and eight contain user documentation and code decomposition, respectively. Both of those chapters contain details on both the level generator and the game. Last chapter presents results of the experiment and provides discussion, as well as suggestions for future improvement.

2 Problem Analysis

The goal of this thesis, as stated in the introduction, is procedurally generating levels for a rogue-like game using EA. This chapter sets out to introduce the game in question, by first informally describing its mechanics and then describing them using formal mathematics. This description is then used to propose a matrix based mathematical object representing in a simple way all information relevant for the attempted PCG.

2.1 Game Design

2.1.1 Winning and Losing

In line with rogue-like tradition, the game punishes player with permanent death of their characters, with the main objective being getting through a level alive and retrieving special item at the end. Unlike most rogue-likes, in our game, player finds himself in control of multiple characters (heroes), rather than just one. New ones are then being generated in assigned room each turn in case some die, so that the number stays constant. Total number of hero deaths however, is limited by predetermined constant, whose reaching results in game over.

2.1.2 Map and Visibility

The map is placed on a square grid, each tile being assigned either floor or wall. Passable section of the map (i.e. floor tiles) is further subdivided into rooms. Heroes start in an allocated start room within the level. At least one hero needs to be present in a room in order for the player to see inside of it. Otherwise either only outline of the room is visible, in case the room has previously been visited by a hero, or the room is not visible at all, if it is yet to be visited.

2.1.3 Turns

During each turn, the player can move all their heroes, in any order, and can split their moves (e.g., move one hero by an amount, then move another, and proceed to move the first one again, if they did not run out of moves for the turn). Player's turn ends when *"next turn"* button is pressed. Skipping a turn when some (or even all of)

heroes can still move (albeit fully or partially) is permitted, but each hero can only attack once per turn.

After the player ends their turn, all non-player controlled characters (monsters) take their turns, one by one. Monsters' behavior is given by their artificial intelligence (AI) and will be discussed in *chapter 8*.

During movement, given hero or monster (unit) can skip over another unit, even if it seemingly blocks the path, but two units cannot occupy the same tile at the same time.

2.1.4 Combat

Combat occurs when a unit tries to enter a tile occupied by an enemy unit. The attacker performs his attack first, then the defender, if he is still alive. Each unit has a combat type, which determines their (dis)advantages against other units. This manifests as multiplier applied to the attacker's strength.

Combat type of an enemy unit is not known until enough fights reveal it based on the outcomes. If both units are still alive by the end of the fight, the attacker moves one tile back on his last path. As mentioned before, each unit can attack others only once per turn as an aggressor. Defeated units stay permanently dead.

2.1.5 Items

Items can be found lying in a level. There is no inventory system, any item collected is immediately consumed by the unit who has collected it. If a key is collected, all doors locked by it are automatically unlocked. As can be seen, the item system is rather simplified. This is due to its little role in the experiment.

2.2 Formal Definition of the Game

Given aforementioned rules, instance of the game can be formally described as a 14-tuple $(T, \tau, R, S, G, \lambda, \sim_d, P, E, I, \psi_t, \psi_p, \psi_i, L)$. It should be noted that while this definition suggests large search space, the actual dimension of generated game content will be reduced, as this thesis does not set out to generate everything. Combat types,

for instance, are given by used implementation and are not subject to change. *Table 1* and *table 2* provide overview of each element. More detailed descriptions are provided beneath. In the following section, $\mathcal{P}(X)$ is a power set of X.

Notation	Description
R	Finite set of rooms
$S \in R$	Start room
$G \in R$	Goal room
E	Finite set of monsters
I	Finite set of items
$\psi_i : E \cup I \to \mathbb{N}^2$	Initial positioning assignment function
$\lambda : \mathbb{N}^2 \to R$	Tile to room mapping function
$\sim_d (\mathbb{N}^2, \mathbb{N}^2)$	Door-tile pairing relation

Table 1: List of all elements in formal definition of the game defining map.

For better clarity, elements were dissected into two groups, those defining map (*table 1*) and the rest (*table 2*). Needless to say that the values from *table 1* are of higher importance for the purposes if this thesis.

Notation	Description
T	Finite set of combat types
$\tau : T \to \mathcal{P}(T)$	Combat type superiority function
P	Finite set of heroes
$\psi_t : P \cup E \to T$	Combat type assignment function
$\psi_p : P \cup E \to \mathbb{N}^3$	Properties assignment function (speed, strength, hit points)
$L \in \mathbb{N}$	Number of lives

Table 2: List of all remaining elements in formal definition of the game.

Some of these items are subject to further limitations, all of which are going to be discussed in this section.

Combat type superiority function τ maps all combat types to those that are to have disadvantage against it in case of a combat (see section about combat above). Complement of this set is implicitly to be assumed to contain combat types which are to have advantage in case of combat. Sole exception is the combat type mapped itself, which is to have neither advantage nor disadvantage in case of combat.

Finite set of rooms R seems straightforward, but it is important to note that the set must contain at least two different elements; one as a substitute for walls (i.e. all walls are considered a room with no door leading into them), and one for the starting room.

Tile to room mapping function λ assigns tiles to rooms. As such, it is expected to only generate valid rooms; that is such rooms where all pair of tiles have at least one path leading between them, which does not lead through a different room. Exception to this rule is the room which substitutes walls (i.e. walls *can* be discontinuous). Furthermore, all rooms need to be of finite size (except for walls) and contain at least one tile, and starting room needs to be at least the size of the heroes' set.

Door-tile pairing relation $\sim_d$ marks placement of doors by pairing two neighboring tiles that are in different rooms. Those tiles are then considered passable through their adjacent side and serve as a door between the two rooms. $\sim_d$ must be symmetric and it is required that each tile is in this relation with at most one other tile (i.e. $\forall(\alpha \in \mathbb{N}^2): |\{\beta \in \mathbb{N}^2 | \alpha \sim_d \beta\}| \leq 1$). We also explicitly forbid reflexive property, so $\sim_d$ is not equivalency[2].

Sets of heroes P, **monsters E**, and **items I** must not be empty. Heroes and monsters for obvious reasons, while items must contain special element that ends the game upon collecting. This item needs to be placed in the goal room.

[2] Forbidding reflexivity also destroys transitivity, because $\alpha \sim_d \beta \wedge \beta \sim_d \gamma \Rightarrow \alpha = \gamma \Rightarrow \alpha \nsim_d \gamma$

Combat type assignment function ψ_t must assign such combat types to the heroes, so that collectively they have advantage against all the combat types. This must also be respected when generating new heroes in case of death. This is in interest of fairness, so that for any monster, there is always a hero in the game who can defeat them with advantage.

Initial positioning assignment function ψ_i seems clear enough, but it is important to note that this function is required to be injective, i.e. for every tile there is at most one monster or item positioned there.

2.2.1 Level Representation

Generating a level could essentially be dissected to generating its *layout* and *population*, concepts tying closely to small subsets of items from previous section. This section attempts identifying those items and offers comprehensive way of their representation using rectangular matrices.

Layout in this context refers to the way rooms are distributed and connected on the grid. Without loss of generality, we can assume that every map is rectangular (if it is not, it can always be expanded by wall tiles to form a rectangle). This offers natural way of representing *tile to room mapping* λ in a form of a matrix $M_R \in \mathbb{Z}^{m \times n}$; assigning each element $r \in R$ a unique non-negative integer $id(r)$, its elements and dimensions are defined as follows:

$$M_{R_{i,j}} = id\big(\lambda(i,j)\big)$$

$$m = \max_{id(\lambda(x,y))>0} x$$

$$n = \max_{id(\lambda(x,y))>0} y$$

Note that these definitions assume few more things about λ, but all are without loss of generality. Namely it is assumed that for the room $w \in R$ containing wall tiles it holds that $id(w) = 0$ and that all tiles not reachable from the start room are contained within it, making maximal non-wall coordinates in each axis natural bounds for M_R (as anything beyond is irrelevant). In case this is not true, rooms can always be

reindexed and non-reachable tiles can be reassigned to a different room without affecting the game in a meaningful way[3]. All reachable tiles are also expected to be on positive coordinates, but this is guaranteed from initial requirements on λ.

Formally, we can also deal with *start room* and *goal room*, by defining them as minimal and maximal positive $id(r)$ over all $r \in R$ respectively. Again, if given start and goal rooms have $id(r)$ differing from this requirement, rooms can be reindexed without loss of generality.

In regards to connecting the rooms on the grid via doors, this can also be represented as a matrix of the same dimensions. As discussed in the previous section, *door-tile pairing* $\sim_d$ is not equivalency, but nonetheless it makes sense to dissect it into sets resembling equivalency classes.

Definition: For the $\sim_d$ relation and an element $\alpha \in \mathbb{N}^2$, *quasi-equivalency class* of α, denoted as $[\alpha]_d$ is defined as the set:

$$[\alpha]_d = \begin{cases} \{\beta \in \mathbb{N}^2 | \alpha \sim_d \beta\} \cup \{\alpha\}, & \exists \beta \in \mathbb{N}^2 : \alpha \sim_d \beta \\ \{\beta \in \mathbb{N}^2 | \nexists \gamma \in \mathbb{N}^2 : \gamma \sim_d \beta\}, & otherwise \end{cases}$$

Note that from requirements on $\sim_d$, we can immediately conclude several properties of such an object. Firstly, every possible α belongs to exactly one quasi-equivalency class. Secondly, every quasi-equivalency class but one has cardinality of 2, with the remaining one being countably infinite (it groups all non-paired tiles). Lastly, number of quasi-equivalency classes is finite (because there is a finite number of finite rooms).

Using quasi-equivalency classes, we can easily represent $\sim_d$ relation with matrix $M_D \in \mathbb{Z}^{m \times n}$. Similarly to representing λ, we first assign each quasi-equivalency class q a unique non-negative integer $id(q)$, then define elements of M_D as follows:

$$M_{D\,i,j} = id([(i,j)]_d)$$

[3] It's impossible within the game ruleset for a player to tell the difference.

Matrices M_R and M_D fully describe layout of the map. What remains is a way to represent *initial positioning assignment* ψ_i to fully describe a level including its interactable content.

Luckily, representing ψ_i as a matrix is analogous to representing λ, with one key difference in form of working with inverse of ψ_i, due to its nature of assigning positions to other objects, rather than assigning objects to positions, as was the case with λ. As ψ_i is injective however, the inverse is guaranteed to exist, so this is not a problem.

After we assign each monster and item $x \in E \cup I$ a unique non-negative integer $id(x)$, elements of matrix $M_P \in \mathbb{Z}^{m \times n}$ are defined as follows:

$$M_{P_{i,j}} = id\left(\psi_i^{-1}(i,j)\right)$$

With M_P, population of the map is also described as matrix. Game content produced by matrices M_R, M_D, and M_P corresponds directly to the part of the game this thesis set out to generate. It is capable of describing any valid map for the game, as no restricting assumptions were made. Collectively, these three matrices are to be called *blueprint* in the rest of the text. For better illustration, *figure 2* contains an example of concrete blueprint instance.

$$\begin{pmatrix} 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 0 & 2 & 2 \\ 0 & 3 & 3 & 3 & 0 \\ 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 4 & 4 & 5 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 0 & 3 & 0 \\ 0 & 2 & 4 & 3 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 5 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 & 4 \end{pmatrix}$$

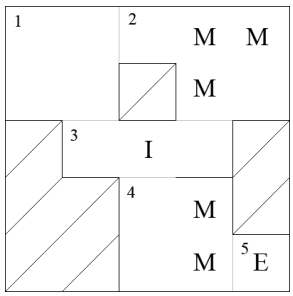

Figure 2: Concrete instance of a blueprint (top). From right to left: M_R, M_D, M_P. Corresponding map (bottom). M are monsters, E is ending, I is item, doors are shaded in grey.

3 Evolutionary Algorithms

Evolutionary algorithms provide us with flexible way to find heuristic solutions for optimization problems, using intuitive process inspired by natural selection. For millions of years, organisms on Earth have expressed remarkable ability to adapt to the environment they live in. This process relies on random mutation in their genes, as well as recombination of their parents' genes. In large enough population of individuals, and over long enough period of time, beneficial mutations will become dominant, creating a population of organisms better adapted to live in given environment. This is due to their carriers being more likely to survive long enough to reproduce, passing their genes onto the next generation.

In computer science, this process could be described as stochastic, population-based, heuristic optimization of an organism to survive in their surroundings. Its stochastic nature comes from mentioned reliance on mutation, as well as other mechanisms inspired by biological evolution, namely natural selection and recombination of parental organisms, all of which rely on element of randomness.

3.1 Genetic Algorithms and Chromosomes

Genetic algorithms (GA) are popular subset of EA, in which individuals are represented as vectors of symbols. Following biologically inspired terminology, this vector is commonly referred to as *chromosome*, while each of its components (symbols) is referred to as *gene*. Problem of finding optimal solution then corresponds to finding a chromosome with optimal genes.

```
1  Generate initial population
2  Evaluate fitness for all individuals
3  Repeat until terminal condition is met:
4          Select individuals to mating pool
5          Apply operators (crossover and mutation) with given probability
6          Evaluate fitness for all new individuals
7          Replace population
8  Return best solution in the population
```

Algorithm 1: Simple genetic algorithm skeleton.

As provided pseudocode suggests, GA (or any form of EA) allows for high degree of modulability and can easily be modified to best suit given problem. Individual principles that form GA are explored in depth in this chapter.

It is also worth mentioning that genes within a chromosome can, but do not necessarily have to be independent. For instance, in one classical binary encoding of partition problem (PP), presented for instance in (Junkermeier), each item corresponds to a gene, identifying the set it belongs to (zeroth or first). Observably, any item can belong to any of the two sets, regardless of where all the other items are, or even how many items there are in total.

On the other hand, the most natural encoding of travelling salesman problem (TSP) solution, where the genes form a permutation of N integers (there are other, mostly better possibilities how to encode it [Potvin, 1996]) does not work that way. If number x is on i-th position in a chromosome, we know that every other position is occupied by a different number, by nature of what permutation is.

3.2 Initial Population

As discussed above, process of natural selection affects entire population of organisms. In simulation of this process, EAs also work with pool of individuals, who are all being optimized (evolved) at the same time. Initial population of such process plays an important role, for it serves as starting point of the solution obtained by the end. Two classical strategies in generating such population exist, although former is typically preferred over latter (Reeves, 2010).

First strategy is generating all individuals randomly. This has obvious advantage of being simple to implement, but beside that it also guarantees variety among individuals, which is instrumental aspect we want our population to have. Argument for why randomized population should work is that even though chromosomes composed of completely random genes are probably not well suited to survive in any environment, some are still a little better than others. As the algorithm ensures that populations size stays constant among iterations, those individuals are

highly likely to survive and produce a little better offspring, starting the process of evolution toward well optimized solution.

Opposing strategy is generating individuals that are already somewhat suited for the environment. The idea behind this is that by generating at least a little competent chromosomes, we are helping the evolution, as it no longer needs to overcome the initial struggle from the firs approach, speeding up the process. In practice though, this approach has tendencies to push entire population into local optima (Reeves, 2010), which by their nature are difficult to navigate from.

Another important aspect of the population is its size. While large populations typically imply more variety, which we have already established as an important asset for EAs, they also mean more time spent on each generation. Considering that generally speaking, EAs tend to be slower when compared to problem specific optimization techniques, it is in our best interest to avoid prolonging the computation time even further. On the other hand, too small population can lead to ineffective exploration of the search space, which is also undesirable.

3.3 Selection

Organisms well fit for survival in their environment are more likely to reproduce and pass their genes onto the next generation. This process is arguably the defining part if natural selection (hence the name) and unsurprisingly plays an important role in evolutionary algorithms as well.

The idea behind selection methods is filling the need for authority (nature) selecting who gets to survive and who does not. Evidently, this process ties closely to the problem of quality measurement (fitness, discussed later in this chapter). In fact, selection methods use fitness as one factor in the decision process, but in addition account for small probability of well fit individuals dying prematurely, or poorly fit individuals getting lucky.

Among the most classical examples of selection method is *roulette wheel selector* (RWS) (Reeves, 2010). RWS assigns each individual part of an imaginary

roulette wheel, size of which is directly proportional to how well fit given individual is relative to all other individuals.

With the roulette wheel ready, it is spun N times, where N is size of desired mating pool, in which pairs of two (or alternatively different number of parents) are selected at random for recombination (see later in this chapter). Note that some individuals might mate with multiple individuals, or even with themselves. That does not matter, however, as former means the individual is probably producing high quality offspring, and while the latter might break analogy to real life biology, it does not negatively affect the algorithm in any way (the individual will simply produce copy of itself).

Clearly, the way fitness is measured is crucial in this case. With population consisting of two individuals, one of which has fitness value of 1 and the other fitness value of 2, latter is more fit and is twice as likely to be selected using RWS. On the other hand, a different fitness could be used in the same situation, assigning first individual a fitness value of 1 and the other fitness value of 100. The second individual is still better fit (both fitness functions agree), but this time, it is hundred times more likely to be selected using RWS.

Common alternative way of using roulette wheel is spinning the wheel just once, then turn it by $1/N$ of its circumference N times to select N individuals. This way of selecting potential solutions for recombination is commonly known as *stochastic universal sampling* (SUS).

Another common selector is *tournament selector* (TS). It comes in two different versions, the more basic one being referred to as *strict*. In strict TS, t candidates are selected at random and the fittest among them is picked. Repetition of the process N times will select mating pool of desired size. *Soft* version of the tournament selection works in the same way, but the best candidate will only win with certain probability.

Note that in case of TS, only comparison of two individuals' fitness values matters, not the values themselves, as opposed to RWS. This is potentially

advantageous, as we may cope with situations with no fitness available, so long we can at least compare two individuals against each other (Reeves, 2010).

3.4 Fitness Function

To determine the fittest individuals, a quality measurement technique, mapping individuals onto real values, is required. Using this function, the algorithm is able to generate populations of individuals that are better than those in previous generations (they yield higher fitness values). This part of the simulation is by its nature problem specific and no easy go to solutions exist.

The fitness function is directly derived from objective function. Objective function is higher level of abstraction, and expresses a property we care about, in units we understand. This is obviously useful for multitude of reasons, as easily understandable values are easier to work with, but might not always be best fit for efficient algorithm.

For instance, let's assume a set of intermediate candidate solutions for a hypothetical problem, all of which are equidistant from each other in terms of their objective values. Provided we are using selector such as SUS, probability of being selected grows linearly with each subsequent individual. However, it might be in our interest to transform the objective values before feeding them into SUS, so that mentioned probability grows faster, thus favor better fit individuals more.

Another situation when objective transformation comes to play is during solving minimization problems. GA always looks for maximum, so in cases when we are interested in minimum (such as partition problem, where we want to minimize the difference between weights of two bags), transformation is also needed.

This altered function is commonly referred to as fitness function, and it is the one the algorithm actually works with. Note that objective and fitness functions can be equal, i.e. transformation between them can be identity.

3.5 Recombination

Recombination (i.e. crossover) of parental organisms to form an offspring is another important mechanism of biological evolution. While it is not impossible (and EAs sometimes do function that way) for certain simple organisms to make offspring by simply copying themselves, sexual reproduction performed by two individuals of opposite gender is the way complex organisms function.

In similar manner, during each iteration (generation) of EA, all individuals are subject to be recombined with others with certain, typically low probability. When constructing EA, any number of parents is possible, yet again mimicking nature however, two are most common choice. As the idea of gender does not help in the optimization process (or rather would be straight counterproductive, as it would prune some potentially high quality recombination options), this concept is dropped from the simulation.

Note that as established previously in this chapter, well fit individuals are likely to be selected for recombination multiple times, which, as mentioned above, does not matter (and actually is beneficial for the simulation). Likewise, an individual can be chosen for recombination with itself, which also does not matter.

How recombination works depends on what kind of chromosome is being used, and sometimes even on objective. Crossovers that are aware of the objective, and are actively trying to improve the fitness, are commonly referred to as *informed* (problem dependent), and can highly increase efficiency of the algorithm. Research in area of solving TSP using GA for example, has produced numerous problem dependent crossovers on various different encodings of the individuals (Potvin, 1996).

It is also worth noting that recombination generally improves the simulation significantly, as it provides way of exploring new combinations of existing genes (Holland, 1992). That being said, crossover is not essential, and branch of EA, called evolutionary programming (EP), which deals with evolving programs, typically sees it as unneeded (Reeves, 2010).

Probably the most common example of *uninformed* (problem independent) recombination method is *one point crossover*. Idea behind one point crossover is taking two parents, splitting both at the same, randomly selected midpoint, and swap their second halves, producing two new individuals. Note that this method does not even rely on the encoding of the chromosomes, only on their length, which stays constant over the iterations.

Multipoint crossover is natural generalization of one point crossover. Instead of selecting single dividing point, multipoint crossover selects N points, then assembles children by swapping every other segment. Note that while dividing points are being selected at random, it should be ensured that no segment is empty.

Uniform crossover treats each gene in a chromosome individually, deciding in which of the children it will end up by flipping a coin. Consequentially, long chromosomes will converge to having 50:50 ratio of genes from both parents. Naturally, the crossover can be tweaked to prefer one of the parents, changing the ratio.

Lastly, common recombination method for real numbers is *arithmetic crossover*. This is prototype example from branch of EA known as evolutionary strategies, which deals with continuous optimization. Similarly to uniform crossover, arithmetic crossover treats each gene individually. The idea behind it is creating children by taking weighted average of genes at the same index, using following formula, where $\alpha \in (0,1)$:

$$Child_1 = \alpha Parent_1 + (1 - \alpha) Parent_2$$

$$Child_2 = \alpha Parent_2 + (1 - \alpha) Parent_1$$

3.6 Mutation

High variety in the gene pool is one of the most important aspects of biological evolution, as well as EAs. However, even with large gene pool, reliance on recombination only silently assumes that optimal or suboptimal genes are already present in the initial population, and the algorithm is just trying to figure out such sequence of crossover applications, that would assemble them into suitable solution

chromosome. Such assumption is not necessarily true, as presence of optimal genes cannot be guaranteed[4]. Crucial mechanism to avoid this problem (both in both biology and computer science) is random mutation.

Much like with crossover, mutation methods used depend on chosen chromosome encoding and potentially objective, in which case we classify them as informed, or problem dependent. As can be seen, mutations and crossover do not function in dissimilar ways and it makes sense to group them together in the algorithm. In such situation, they are sometimes referred to as *operators*.

Similarly to the case with generating initial population with the objective in mind, informed mutations showcase tendencies to get stack in local optima of the fitness function due to guiding the evolution process in certain direction, potentially demolishing much valued diversity in the population. When settling for usage of problem dependent mutations, it is a good idea to supplement them with an uninformed mutation, which can help preventing premature convergence, although this notion can be somewhat controversial (Reeves, 2010).

Classical example of problem independent mutation is so called *bit flip mutation*, which with certain probability flips value of a bit in binary encoded chromosome. Based on similar idea, another common example would be setting a gene to a new value uniformly selected from the alphabet of the chromosome. Notice that unlike bit flip mutation, this operation does not depend in the previous value of the gene. Mutations with this quality are known as *unbiased*.

A lot of commonly used encodings are numerical. Those might or might not be contained within a specific range. In case of uncontained values, natural mutation is adding a number from some distribution to the value of the gene. Those mutations are known as *biased* and are most common when working with real numbers in ES, in

[4] Although probability of optimal genes present in the initial population clearly increases with large population; which brings us full circle to the problem of population size, presented when discussing selection.

which case Gaussian distribution with mean zero is commonly used (known as *Gaussian mutation*).

ES are also noteworthy for their common practice of changing mutation parameters as the algorithm iterates. These are known as *adaptive operators*. Aligned with the philosophy of exploring the search space first and then exploiting promising candidates, prototypical example of adaptive operator is decreasing standard deviation of Gaussian distribution from which we generate differential values for the genes in previous example.

As mentioned previously in this chapter, genes in a chromosome can sometimes depend on one another (depending on chosen encoding). In such cases, it is essential that mutating a gene does not render the chromosome invalid, such as randomly changing an integer in permutation.

Lastly, it is important to note that determining probabilities for both recombination and mutation is problem specific (Reeves, 2010).

3.7 Generation Replacement and Elitism

By the end of each iteration of the algorithm, old generation is replaced by the new one. This poses an interesting problem in EA, one which does not have good parallel in Darwinian evolution. Simply replacing old generation with the new one potentially leads to losing unlucky high quality individuals. Simply ignoring this problem is the easiest way to go around solving it (and that is indeed how Holland's original GA worked [Reeves, 2010]), but efficiency of the algorithm can be improved greatly if some well fit individuals from the parental generation are allowed to cross over to the new one (even if it happens at the expense of some unsuccessful individuals in the offspring). This concept is known as elitism and was first introduced by De Jong (De Jong, 1975).

3.8 Terminal Condition

Some problems that can be effectively solved with GA have nicely defined objective functions with clear global optimum we ought to achieve. Example of such

problem can be previously encountered PP, where value difference of zero is desired solution, but letting the algorithm go until this result is achieved might not always be a good idea. If the only items in an instance of PP have values of two and three, the sole global optimum has difference of one, not zero, and the algorithm is now stack in an infinite loop, swapping the two objects indefinitely.

Instead of demanding global optima, it might be beneficial to settle for solutions that are "good enough". One strategy to achieve that would be to keep track of the best found fitness value and terminate the algorithm when no substantial change has been made to it for significant amount of time. Note that due to the stochastic nature of the algorithm, the fitness value will always be changing by at least some amount.

Other common strategies are waiting until certain number of generations have passed, or certain amount of time. These solutions are easy to implement and we are not running into risk of an infinite loop, but we might terminate the simulation with a solution that could have still been improved upon had we given it more time to do so.

3.9 Multi-objective Optimization

Above discussed approach allows for numerous operators to be independently applied. That raises a natural question of whether it is possible to apply multiple fitness functions in a similar manner. After all, many problems are not easily described with just one objective in mind, such as coachbuilder design, which needs not to be only aerodynamic, but also lightweight.

However, it is not difficult to imagine that two or more objectives might contradict each other. Knapsack problem for instance can be interpreted as bi-objective optimization problem, in which we try to maximize value of the objects taken, but at the same time not to take more objects than we can carry, preventing us from taking everything. In aforementioned coachbuilder design example, we want the solutions to be lightweight, but the most lightweight solution is the one with no material, which is obviously undesirable.

This section attempts to clear such questions by explaining common principles in approaching multi-objective optimization using EA and presenting examples of some well-known algorithms. Those algorithms are collectively known as multi-objective evolutionary algorithms (MOEA).

3.9.1 Pareto-optimal Front

To tackle raised concern, let's refocus on slightly different problem: what happens, if two solutions cannot be compared to each other? So far, our only way of comparing individuals was to compare their fitness values, but with multiple objectives, this method is no longer valid, as comparing non-trivial vectors is not well defined operation.

One way of solving this issue is called domination. We say that one vector dominates other, if and only if all of its components are equal to or higher[5] than those of the other vector on the same indexes. In addition, strict dominance further requires at least one value to not be equal. Being able to sort candidate solutions based on dominance is a key gimmick in solving multi-objective optimization. For better clarity, formal definition follows:

Definition: Let u and v be real vectors of the same dimension $N \in \mathbb{N}$. We say that u *weakly dominates* $v \Leftrightarrow \forall i \in \{1 \dots N\}: u_i \geq v_i$. We say that that u *dominates* $v \Leftrightarrow$ u *weakly dominates* $v \wedge \exists i \in \{1 \dots N\}: u_i > v_i$, denoted as $u \succcurlyeq v$ and $u \succ v$ respectively. Otherwise, we say that u *does not (weakly) dominate* v.

Set of solutions not dominated by any other solution is called Pareto-optimal front. Rigorously mathematically correct at this point would be terminating the simulation upon reaching one such solution (or sufficient approximation), but not all that glitters is gold and not all the Pareto-optimal solutions are universally best for the

[5] In this thesis, domination is defined for maximization problems, but it's important to note that for minimization problems, it makes sense to define it with the other inequality (i.e. to call the lower solution dominating).

problem at hand. Some might be more suited than others depending on the context, and selecting just the right one is an interesting problem on its own.

3.9.2 Locating Solution

One classical approach of finding solution to the multi-objective problem is objective weighting. This and other classical methods rely on scalarizing the vector and reducing the problem to single-objective optimization (Srinivas, et al., 1994). In case of objective weighting, the reduction is done with the following formula:

$$f(x) = \sum_{i=1}^{N} w_i f_i(x),$$

where N is number of objectives, f_i is i-th fitness function, and w_i is weight of i-th objective. Weights should always sum up to 1. Obviously, designer's choice of distributing weight across the objectives highly impacts the results. An illustrative example of this can be found in (Srinivas, et al., 1994), the gist of which lies in bi-objective problem assigned as minimization of

$$f_1(x) = x^2$$
$$f_2(x) = (x-2)^2.$$

Analyzing the problem, we can conclude that solutions satisfying $0 \leq x \leq 2$ form Pareto-optimal front, but depending on the weight distribution, actually found solution may steer toward each end.

This may or may not be what the designer intended, but the point stands that some deeper understanding of the problem, in regards to how to assign the weight vector, is essential to perform the reduction. Unfortunately, other classical methods, such as method of distance functions, share similar drawbacks (Srinivas, et al., 1994).

Lastly, mentioned methods will only find one solution, for post-reduction they become single-objective. Trying to find multiple Pareto-optimal solutions then requires running the simulation multiple times with different weight distributions (or other parameters in case of different methods).

3.9.3 Nondominated Sorting Genetic Algorithm (NSGA)

The problem concluding previous section might raise a question: if GAs work with population of individuals, is it necessary that we steer its run to favor one Pareto-optimal solution over the others? Maybe there is a way to breed entire population so that by the end of the simulation, it approximates entire Pareto-optimal front.

The first genetic algorithm to do that was Vector Evaluated Genetic Algorithm (VEGA) (Schaffer, 1985), based on simple idea of dividing the mating pool into equisized subsets, each of them filled by independent selection based on different fitness. The population was then shuffled, crossed over, and mutated. This approach however, carries numerous disadvantages. Mainly, it is criticized for creating specialized individuals, i.e. individuals that perform well with one fitness, but poorly with others (Srinivas, et al., 1994).

An algorithm that tackled many drawbacks of VEGA was Nondominated Sorting Genetic Algorithm (NSGA). The idea behind it was first proposed by Goldberg in 1989 and later elaborated upon further in (Srinivas, et al., 1994)[6]. Its two main steps consist of finding good point in the search space and using niche methods to maintain population diversity.

The first part is achieved by ranking individuals based on dominance. Nondominated individuals in the population are ranked as 1, then each subsequent rank is assigned by removing individuals that have already been ranked and finding new nondominated individuals in the remainder. NSGA assigns dummy fitness to each candidate solution based on its rank, so that individuals within the same front have the same fitness.

Maintaining diversity is more peculiar task. In nature, niches have carrying capacity to support certain number of organisms and in turn, organisms have ability to

[6] Other than (Srinivas, et al., 1994), at least two other studies have been published based on Goldberg's idea, proposing similar algorithms.

exploit that carrying capacity. Consequentially, overpopulated niche will cause lack of food and extinction of the weakest, and fertile niche with only few organisms will soon be populated to meet its carrying capacity, by said organisms reproducing.

In *sharing*, used by NSGA, every individual is essentially considered center of a niche, and reduces fitness of all nearby (similar) individuals. The idea is that punishing similar individuals and in turn implicitly rewarding unique ones encourages diversity (Miller, et al., 1996).

Sharing fitness is then derived for each individual by dividing their dummy fitness with their *niche count*; summarization of *sharing function* over the entirety of the population. Formulas on how to calculate both are as follow:

$$m_i = \sum_{j=1}^{N} sh(d_{i,j})$$

$$sh(d_{i,j}) = \begin{cases} 1 - \left(\frac{d_{i,j}}{\sigma}\right)^2, & d_{i,j} < \sigma \\ 0, & otherwise \end{cases},$$

where m_i is niche count of i-th individual, sh is sharing function, N is population size, $d_{i,j}$ is distance between i-th and j-th individual (how similar they are), and σ is maximum distance allowed between two individuals of the same niche.

Nondominated sorting procedure only alters selection process, other steps remain unchanged when compared to single-objective GA. Once the simulation is terminated, entire first nondominated front in the final population is considered solution.

3.9.4 Nondominated Sorting Genetic Algorithm Revisited (NSGA-II)

Presented algorithm has been target of criticism, mainly for three reasons: its high computational complexity, lack of elitism, and need of specifying sharing parameter σ. An improved version of NSGA was proposed (Deb, et al., 2002), addressing all of these issues.

The study suggests faster way of performing nondominated sorting with improved computational complexity of $O(MN^2)$, where M is number of objectives and N is size of the population. The procedure, as well as in depth analysis and comparison to its predecessor can be found in the paper.

Need for elitism has been briefly explored earlier in this chapter. It comes as no surprise that elitist algorithms have tendencies to outperform their non-elitist counterparts (Zitzler, et al., 2000). The way NSGA-II implements elitism is based on simple idea: merging the old generation with the new one, then sorting it and only taking the better half (which will be of expected size, since population length is immutable).

This forces us to revisit the problem of comparing two individuals. In case of NSGA, a method was proposed how to assign fitness based on nondominated sorting rank and niching (sharing). NSGA-II bypasses assigning fitness altogether by defining partial order on individuals, which it uses for tournament selection. Before delving into definition of said order, understanding of one new concept is required.

Crowding distance essentially replaces sharing, which also solves last of aforementioned problems; a need for determining optimal sharing parameter σ. Besides problem of choosing σ, sharing also requires every individuals to be compared to every other individual, approaching complexity of $O(N^2)$ (Deb, et al., 2002), which is also undesirable.

To calculate crowding distance of an individual, candidate solutions must be sorted in ascending order in respect to one of the objectives. Distance to boundary individuals is set to infinity, while distance to others is derived by taking absolute normalized difference of their neighbors' function values. This procedure is then repeated for each objective and summed up. *Algorithm 2* is as proposed in (Deb, et al., 2002):

1 $l \leftarrow |P|$
2 For each $p \in P$:
3 $\quad \text{dist}(p) \leftarrow 0$
4 For each objective m:
5 $\quad P \leftarrow \text{sort}(P, m)$
6 $\quad \text{dist}\big(\text{first}(P)\big) \leftarrow \text{dist}\big(\text{last}(P)\big) \leftarrow +\infty$
7 $\quad$ For $i \leftarrow 1$ to $l - 2$: *(we are excluding the first and the last element)*
8 $\quad\quad \text{dist}(P[i]) \leftarrow \text{dist}(P[i]) +$
9 $\quad\quad + \big(m(P[i+1]) - m(P[i-1])\big)/\big(f_m^{\max} - f_m^{\min}\big)$

Algorithm 2: Calculating crowding distance for set of candidate solutions

With understanding of crowding distance, we can now compare two individuals based primarily on their rank (lower is better) and secondarily on their crowding distance (higher is better). As individuals with low crowding distance correspond to crowded regions of the search space, this order encourages diversity among them. Formally put, given two different candidate solutions i and j, their order is defined thus:

$$i \prec_n j \;\Leftrightarrow rank(i) < rank(j)$$
$$\vee \big(rank(i) = rank(j) \;\wedge\; dist(i) > dist(j)\big)$$

3.9.5 Hypervolume

As was established, EAs are designed to find heuristic solutions. When comparing different results acquired by multiple runs of single-objective evolutionary algorithm, fitness function used provides obvious means of solution quality measure. After all, that is what it was designed for.

However, in case of multi-objective optimization, a question of quality evaluation in analogous situation does not have such an obvious answer. Results generated by MOEAs are sets of mutually nondominated individuals, which are supposed to approximate Pareto-optimal front. Thus a reliable quality comparator quantifying how good of an optimal set approximation given solution is, is needed.

When comparing two optimal sets, two main criteria are believed to be of importance (Nicola, et al., 2009):

1) Points of the approximated solution are close to the actual Pareto-optimal front.

2) Points of the approximated solution are well distributed along the Pareto-optimal front.

One such quality indicator respecting those requirements, used frequently in modern MOEAs is *hypervolume indicator* (Nicola, et al., 2009). Hypervolume uses dominance as means to quantify quality of a solution. The idea is that every point in the solution set, in respect to the same shared reference point, defines a hyper-cuboid of points it dominates. All points in the solution set then define multidimensional orthogonal polytope, whose volume can be calculated, yielding a scalar number easily comparable to volumes of other solution sets. Formal definition presented by (Nicola, et al., 2009) is as follows:

Definition: Given a finite set P of points in the positive orthant $\mathbb{R}_0^d$, the *hypervolume indicator* is defined as the d-dimensional volume of the hole-free orthogonal polytope

$$\Pi^d = \{x \in \mathbb{R}_0^d | x \preccurlyeq p \in P\}.$$

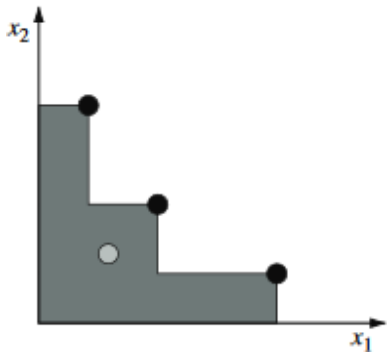

Figure 3: Two dimensional hypervolume with the origin as the reference point. The dominated part is shaded in grey. Pareto-optimal front consists of three points, depicted as black dots. *(Source: Nicola, et al., 2009)*

This polytope corresponds to the space which is dominated by at least one point in the set P.

It should be noted that the reference point mentioned earlier coincides with origin in the definition above. The definition also assumes positive objective values, but that is not at expense of generality, as a transformation can be applied to an objective function to guarantee it to be the case (Nicola, et al., 2009).

In two dimensions, hypervolume indicator corresponds to the area of orthogonal polygon (see *figure 3*). Consequentially, an algorithm calculating said area can be constructed easily (see *algorithm 3*).

1 $\text{sort}(P)$ *(ascending order with respect to X-coordinate)*
2 $V \longleftarrow \text{first}(P).Y * \text{first}(P).X$
3 For each $p \in P \setminus \{\text{first}(P)\}$:
4 $\quad V \longleftarrow V + p.Y * (p.X - \text{predecessor}(p).X)$
5 V is hypervolume indicator

Algorithm 3: Hypervolume indicator in two dimensions

In three dimensions, constructing such algorithm is more difficult task. An approach proposed in (Nicola, et al., 2009) suggests reducing 3-dimensional problem into series of much easier to solve 2-dimensional problems. By sweeping the third dimension in descending order and remembering the (x, y) projection of the boundary of the dominated volume above the sweeping plane, hypervolume can be calculated (see *figure 4, left*). Each new projection, combined with knowledge of its distance from the last in the third dimension, defines an extruded rectilinear polygon. Hypervolume can then be obtained by summing up volumes of all those objects.

Note that only the boundary of the dominated volume above the sweeping plane needs to be remembered in suitable sweeping structure. Structure proposed in (Nicola, et al., 2009) is balanced binary tree, storing the boundary line as series of points in increasing order with respect to first dimension. This can be done, because the boundary line is monotone rectilinear polyline (see *figure 4, right*). Pseudocode and in-depth analysis of the algorithm can be found in the original paper.

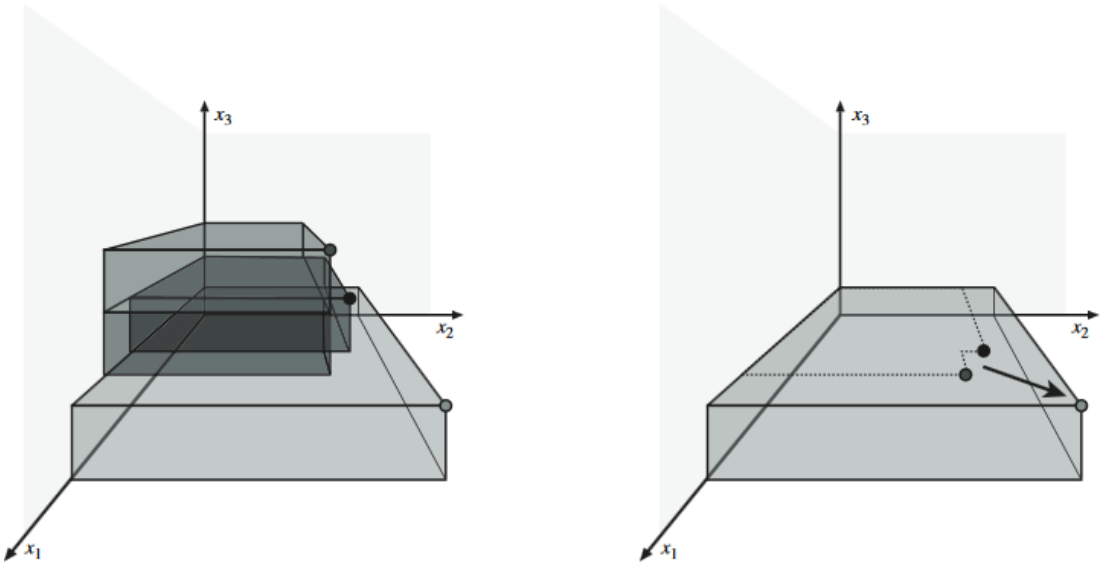

Figure 4: Calculating the hypervolume by sweeping the third dimension in decreasing order (left). Boundary line of the dominated volume in a given instance of the two dimensional sub-problem (right). *(Source: Nicola, et al., 2009)*

4 Approaches and Related Literature

Introduction discussed prominence of PCG in game industry. As such, it comes as no surprise that various content generation methods have been suggested and thoroughly tested for variety of games, including rogue-likes and similar genres. This chapter will examine several of those and compare them to methods used in this thesis.

4.1 Refining the Paradigm of Sketching in AI-Based Level Design

When human an artist works on a level design, their typical workflow follows rough sketching, with the sketch then being refined into the final product. In fact, most artistic workflows follow this pattern. Base on the same idea is a PCG approach proposed by (Liapis, et al., 2015), which attempts to simulate this workflow to generate levels for various games, independent of the genre. In fact, the paper presents three case studies, each done on a game of a different genre and using different of three presented sketch refining methods.

While Liapis et al. use GA as a means to generate the levels, just like this thesis, it is noted in the paper that proposed principles can be carried out with different generation method.

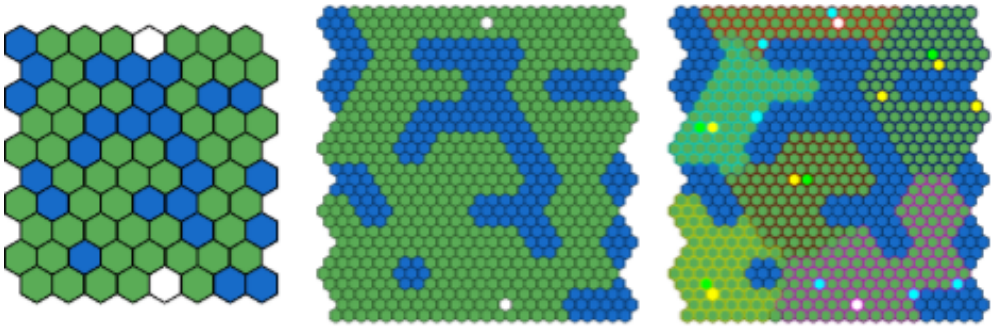

Figure 5: Evolved rough sketch (left), sketch scaled up to higher resolution (middle), evolved final refined level (right) *(Source: Liapis, et al., 2015)*

First presented refinement method is based on increasing resolution of the map and adding more tile types. According to the paper, this is supposed to approximate the iterative refinement of artwork, as artists typically begin with rough pencil strokes and proceed with colors, shading, as well as more delicate details.

This lays ground for dissecting the algorithm into two stages. Case study present in the paper, using a strategy game Endless Legend (Amplitude, 2014) as a guinea pig, uses the first stage to generate traversable and impassable chunks of land, packed with players' starting positions. The second stage increases the resolution, proceeding to distribute resources. The idea of dissecting the algorithm into two stages responsible for generating different essentials served as an inspiration for this thesis, although methods presented here do not necessarily follow the sketch-refinement dissection presented in the paper.

4.2 Automatic Generation of Fantasy Role-playing Modules

Besides video game content, PCG can be utilized to generate content for table top RPGs. This topic was touched upon in the introduction, with mention of (Ashlock, et al., 2014), a study proposing procedural generator for modules into games such as Dungeons & Dragons, with focus on automatic map design. The study also utilizes GA to achieve its goals, with subdividing map into rooms and even representing maps in form of integer matrix, just like is the case with this thesis.

Although the idea of using subdivision into rooms is similar, what each of our approaches understand under the term "room" differs greatly. (Ashlock, et al., 2014) defines room as "*a part of the dungeon containing a 3×3 area of open space that forms a contiguous open space.*" This obviously differs from the idea of specifying what room each tile belongs to by assigning each tile some sort of room identification. Ashlock's definition also allows for the existence of "corridors," a parts of the map not belonging into a room. It also disallows for two rooms to be directly adjacent on the grid, as there must explicitly be a wall to separate them.

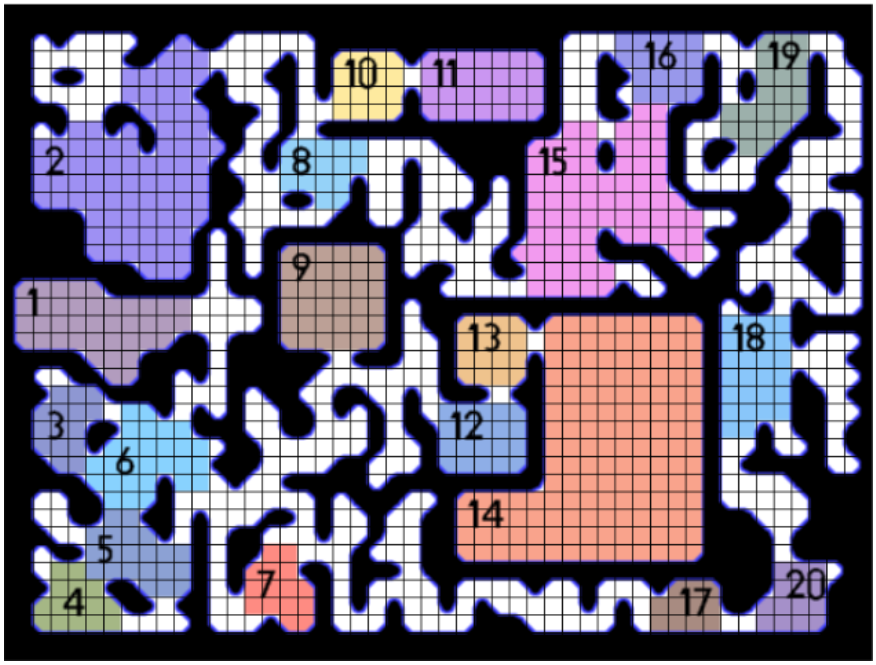

Figure 6: Example of an evolved map. Rooms are highlighted and labeled with numbers. *(Source: Ashlock, et al., 2014)*

The GA proposed in (Ashlock, et al., 2014) is single-objective, with fitness function based on lengths of shortest paths between user specified checkpoints. For example, one version of the fitness requires the goblin lair to be far from the entrance, the armory to be near the goblin lair, and the magical being's lair to be far from the entrance. On the contrary, for this thesis, it was decided that MOEA would be used and the algorithm dissected into multiple stages.

4.3 Game Level Layout from Design Specification

In traditional level design process, designers can control gameplay flow by assembling (typically reusable) chunks of dungeon, building blocks, in desired fashion. Procedural generation normally takes that ability away, as the algorithm is now in charge of designing the dungeon. The approach to solve this problem, presented in (Ma, et al., 2014), attempts to keep the gameplay flow control in hands of the designers.

The algorithm takes planar graph and set of building blocks as input. The graph represents desired connectivity among the building blocks and is subject to by-hand design, allowing the designers to control gameplay flow as promised. Each node of the

graph is then assigned a building block from the library, so that neighboring building blocks do not intersect and connect at valid places only.

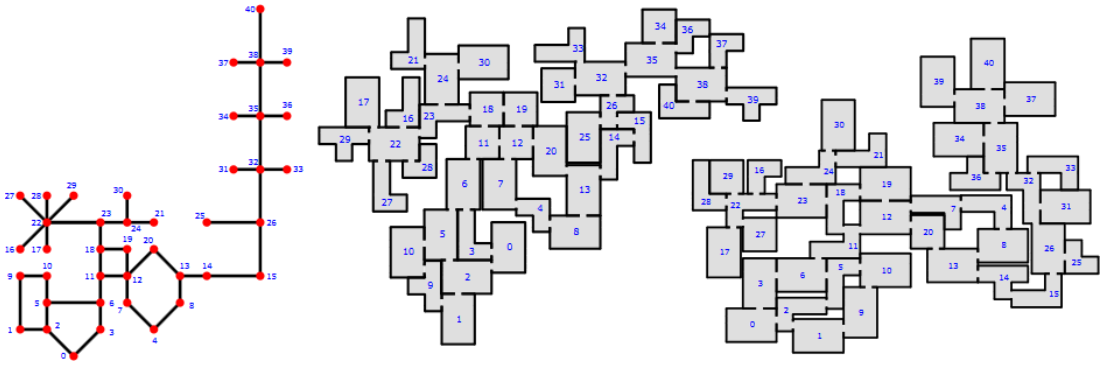

Figure 7: Two levels generated using the same building blocks and input connectivity graph. *(Source: Ma, et al., 2014)*

Proposed approach is noteworthy in context of this thesis, particularly due to different philosophies of how the final result should look like. The building blocks are in some sense similar to rooms in the approach proposed in this thesis, however the point of having predefined building blocks lays mainly in the ability to have the dungeon made up from by-hand designed chunks of content.

On the contrary, the algorithm proposed in this thesis makes active effort to *shape* all the rooms so that they fit tightly together on a rectangular grid. The results of both algorithms lead to dungeons with very different feel to them.

5 Proposing Evolutionary Solution

With thorough theoretical understanding of the objects to be generated, this section proposes EA with numerous operators to do the job. The idea is to generate each blueprint matrix separately in a way that does not violate requirements set out in *chapter 2*. Generated blueprint is then used to represent the map.

Good map in any game represents collection of complex ideas that collectively make it interesting to play in some sense. These can be challenging to accurately represent by small number of objective functions. Inspired by (Liapis, et al., 2015) it was thus decided to dissect the generation process into two successive stages.

First phase of the algorithm should generate tightly packed rooms, as proposed in previous chapter, which is followed by the second phase, generating connections between them. The second phase should also lay out easy ground for distributing loot and encounters throughout the level.

Both phases will be described in further detail in this chapter. Collectively, they both use NSGA-II algorithm for selection, but differ greatly in encoding of individuals, fitness functions, used operators, and even philosophy of evolution guidance. Both phases furthermore deviate somewhat from the idea of simple GA to lesser or greater extend, a topic also discussed in this chapter. Note that in-depth exploration of implementation of operators is subject to *chapter 8*, this chapter merely showcases basic ideas behind their concepts.

Both phases also have several free parameters which can be altered to achieve different results. List of those parameters is present in *table 3*. *Chapter 7* provides comprehensive guide on how to change all of these parameters in the application that implements proposed PCG methods. Note that most of these parameters are also free parameters for alternative solutions presented in *chapter 6*.

Parameter	Notation	Phase
Probabilities of each operator occurring	*none*	Both
Width and height of the map	*none*	First
Probability of a wall during initial population generation	p	First
Desired number of rooms	C	First
Branch rooms to total number of rooms ratio	ζ	Second
Desired heroes' party size	D	Second
Desired number of branches	B	Second

Table 3: List of free parameters of the evolutionary solution.

5.1 Phase 1 – Room Distribution Phase

5.1.1 Encoding of Individuals

As explained above, first stage of the process sets out to generate rooms on the map. Chromosomes of this stage directly correlate to the first matrix of the blueprint. Where the idea differs from that of simple GA is that individuals provide additional data structures to help operators when manipulating the rooms. Their nature will be discussed in greater depth in *chapter 8*, but the idea is to keep list of all the rooms, while all the rooms will keep fast and easy access to their tiles (coordinates in the matrix), as well as fast way of comparison between their sizes (number of tiles contained within).

Important constrain to point out is the necessity of only generating valid rooms, i.e. not disjointed ones. The second important constrain is keeping the rooms tightly packed, as discussed above. Additionally, it also helps rooms from being completely disjointed from the rest and the second phase to have more potential in exploiting interesting possibilities to connect them.

The first constrain can easily be taken care of by careful design of the operators used, but the second one is more challenging to tackle. It was decided that the algorithm should encourage tightly packed rooms, but not guarantee them. This leaves

the door open for invalid maps being generated, however such cases can be prevented by making them so unlikely (by used operators and fitness), they will for all practical purposes be as good as impossible.

5.1.2 Initial Population

In line with closure remark of the last section, initial population generation ignores the latter of the concerns. The algorithm simply goes through each locus in the matrix and leaves a wall tile (represented as 0) with certain probability p, or changes it to a room tile as complementary event. To satisfy the first constrain, room identification is copied from one of the four neighbors, or generated anew if the tile is only surrounded by walls.

5.1.3 Fitness Functions

The first phase uses three fitness functions, whose main objectives are encouraging tightly packed rooms (f_1), punishing deviations from desired number of rooms (f_2), and encouraging room spreading (f_3). In the following formulas, R is the set of all rooms as defined in *section 2.2*, except without substitute for walls. It is available as extra data structure in every individual. Description of each formula is provided below. It is worth reminding at this point, that fitness functions as described in *chapter 3* are always subject to maximization (see the difference between objective and fitness in *section 3.4*).

$$f_1(R) = \frac{1}{|R|}\sum_{r \in R} |neighbors(r)|$$

Why tightly packed rooms are important was already discussed. The way this is actually achieved is by taking average number of neighbors as one of the fitness functions. In this context, neighbor is considered any such room containing such a tile having (Manhattonian) distance of 1 from some of our tiles. First fitness obviously has global minimum at 0, if no rooms are connected. Global maximum is tricky to determine exactly, but since R is finite by definition, upper bound of some sort is guaranteed.

$$f_2(R) = exp\left(-\left|C - |R|\right|\right)$$

Second fitness punishes deviations from the desired number of rooms, denoted C in the formula. This is important, because maps with too few or too many rooms would not be fun to play; while former contains little challenge, the latter would be too tiring. Global maximum of the second fitness is clearly 1, if $C = |R|$, while for growing deviation it quickly shrinks to 0.

$$f_3(R) = \sum_{r \in R} |r|$$

Propose of the last fitness is to ensure that the rooms are spacious. Without it, the algorithm might steer toward solutions with miniature rooms tightly packed in one corner of the map, leaving vast majority filled with walls. Such levels are obviously undesired and countered by summing up all non-wall tiles and using it as the final fitness. Global maximum of this fitness equals to the total number of tiles, while global minimum is 1, disregarding case with only walls on the map.

5.1.4 Operators

Floor type mutation is operator with least knowledge of the problem and was introduced mainly to prevent premature convergence. It simply selects one tile at random, and swaps it from wall to floor or vice versa.

In case the tile was originally floor, this seems straightforward, but it is essential to keep an eye out for two special cases: splitting a room into two, in which case new room is created and tiles must be properly reassigned, and erasing last tile in a room, in which case the room is removed from any additional structure in the individual. In case the tile was originally wall, a room is assigned to it in the same fashion as during the initial population generation (see above).

Join rooms mutation is more sophisticated operator. It utilizes helping structures to pick a room at random, with bias toward small rooms. Simple heuristic is used to attempt locating an attached room, which upon success is merged with the original room.

Split room mutation provides counterbalance for joining rooms. It picks a room at random, biased toward large rooms, whereupon the room is split into two. An attempt is made to generate each of them of the same size (i.e. split the room precisely in half tile-wise).

Remove room mutation is simpler compared to last two operators. As implied by the name, it removes a room completely from the individual. This operator is strongly biased toward small rooms, which are deemed generally uninteresting, and was introduced primarily to eradicate large number of rooms during early generations.

Fill hole mutation scans the map from random locus in reading order and attempts to locate a tile surrounded from all sides by tiles in different rooms (in this context, wall is also considered a room). These can be either wall tiles surrounded by rooms, or single tiled rooms. Upon finding first such "hole," it fills it by reassigning it to a room from one of the neighboring tiles.

Extend edge mutation is the most sophisticated operator used in this phase. It picks a room at random and proceeds to pick a random *edge*. By edge, we understand a continuous straight line of tiles in the same room, which all neighbor a different-room tile on the same side, and cannot be lengthened on either side to form a longer object with the same properties.

Given the side where the edge neighbors different-room tiles, chances are all of them are wall tiles. In an attempt to generate tightly packed rooms, this empty space can be filled by tiles extending from the original edge.

5.2 Phase 2 – Structure Phase

5.2.1 Encoding of Individuals

Second phase takes slightly different approach in producing its results. It takes one incomplete map from the room distribution phase and extracts *room adjacency graph* out of it. Room adjacency graph is simple structure describing in straightforward manner whether two rooms are connected by drawing an edge between their respective nodes.

Connection in context of the extraction means that the rooms are neighboring, as described in the room distribution phase, however edges on this graph are subject to mutation (indirectly, see “Branches” below), with the solution graphs describing where to put doors and where not to by the end.

The graph was decided to be represented by list of nodes, each remembering list of references to their neighbors. This decision clashes with basic idea behind GA, as the chromosomes are not simply strings, but it is important to note that they are still *not* of variable length, as neither nodes, nor connections are actually subject to change directly. Instead, concept of branches, described below, is used. This decision was made due to number of operators being judged to be easier to implement with proposed encoding.

Second purpose of the structure phase is to define starting room, which extends the length of the chromosome by few extra bytes. Similarly to the first phase, individuals contain additional data structures in order to increase overall efficiency.

Lastly, it is worth pointing out that unlike the room distribution phase, structure phase will always guarantee valid individuals, due to its operators.

5.2.1.1 Branches

The structure phase attempts to generate connections by utilizing idea of *branches*. Branch can be thought of as a mask, covering some of the existing connections. Each node of the graph remembers what branch it belongs into (a piece of data which is actually going to be evolved), with rules governing them not being dissimilar to those governing rooms and tiles. Namely, it is essential that branches are not disjoined.

Figure 8: Room adjacency graph with three branches, highlighted in red, green and blue. Their starting rooms are shaped as octagons. Dashed edges showcase connections masked out by the branches.

Each branch defines its starting room, which is the only point connecting the branch with the outside world, as it does not mask any connections out. All the other nodes within the branch cover the connections leading into rooms outside, creating a branched area of rooms with possible content (or negative possibility space) at the end.

For the purposes of this thesis, all branches are locked by a key (an item within the game) which first needs to be retrieved, but this is not subject to evolution, see post-processing in the next chapter.

5.2.2 Initial Population

As mentioned above, structure phase guarantees valid individuals at any point throughout the algorithm. This also includes all individuals in the initial population and the generation algorithm must reflect that. Note that this assumes valid output from the first phase, but as mentioned above, with reasonable parameters, that will for all practical purposes always be the case (not a single test run produced invalid outputs, in fact, all invalid outputs were selected out relatively early on, see *chapter 9* for more details).

Fortunately, this is easy to do, once we realize that branches of length 1 do not mask out any door connections whatsoever, since the only room contained within is branch starting room, therefore we can just spawn those at random B times without

any worries. Starting room for the heroes can picked at random as well, as long as it is large enough to hold the entire party. If there is no such room, the algorithm will fail, but such situation suggests poor (and actually borderline impossible, see *chapter 9*) results from the room distribution phase.

The only remaining problem is to be careful with placement of the keys, as some areas might be inaccessible from the starting room (due to locked doors), but that is not concern of the structure phase. For further details, see post-processing in next chapter.

5.2.3 Fitness Functions

Second phase is bi-objective, with its two fitness functions keeping distance between heroes' starting room and each branch starting room as low as possible (f_4), while encouraging desired proportion of rooms contained in a branch compared to the total number of rooms (f_5). In the following formulas, $G = (V, E)$ is room adjacency graph, $d(v, u)$ is number of edges between v and u, $s \in V$ is the starting room, and $V_{BS} \subseteq V$ is set of branch starting rooms. Note that all the information beyond the pair (V, E) are easily available either through help structures or other means.

$$f_4(G, s, V_{BS}) = \frac{|V_{BS}|}{\sum_{v \in V_{BS}} d(v, s)}$$

As can be seen, the first fitness simply takes average distance between starting room and each branch starting room, then inverts the expression. Its meaning is straightforward, however it is worth pointing out that from definition of a branch, the sum cannot be 0 if there is more than one. Furthermore, the case of single branch is also covered, due to the algorithm actually disallowing starting room to be part of a branch. Global maximum of this fitness is equal to 1, if they are all directly adjacent to the starting room, while global minimum is at $\frac{|V_{BS}|}{|E|}$.

$$f_5(G, s, V_{BS}) = exp\left(-\left|\frac{\sum_{v \in V} in(v)}{|V|} - \zeta\right|\right)$$

Second function minimizes deviation between desired proportion $0 \leq \zeta \leq 1$ of rooms contained in a branch compared to the total number of rooms. Function $in(v)$ used in the formula simple equals to 1 if and only if v is part of a branch, or 0 otherwise. Similarly to f_2, the global maximum lays at 1, if the ratio is perfect match, and continuously shrinks to 0 for greater deviations.

5.2.4 Operators

All operators in the structure phase function a little differently than standard EA operators; the algorithm provides a quick way to check whether validity of a solution has not been breached, in which case the change is reverted. With such procedure, validity is always guaranteed by induction.

Extend branch mutation simply picks a branch at random, locates the end and extends it in random available direction. Counterforce for this is *compress branch mutation*, which does the exact opposite, with further restriction in form of not being able to reduce a branch to zero (empty list).

Replace branch mutation picks a branch at random and removes it from the graph, then proceeds to place a new branch at random. The new branch is of length 1, just like during initialization.

Reverse branch mutation is simple operator, which picks a branch at random and swaps its starting and ending nodes. While presented definition of a branch does not guarantee uniquely identifiable ending nodes, it does not matter. When traversing the branch from the starting node, there will always be some sort of "last" node.

Lastly, **shift starting room mutation** is the only operator not altering branches at all. Instead, it provides the algorithm with a way to wiggle placement of the heroes' starting room by shifting it by one edge on the graph. If the new room isn't big enough, the operator is applied again starting from it, until it reaches a suitable sized room. The size required is equal number of heroes D.

6 Proposing Alternative Solution and Post-Processing

For the purposes of comparison, this chapter sets out to propose two simple algorithms achieving by non-evolutionary means results similar to those of the algorithms proposed in previous chapter. Furthermore, a post-processing algorithm converting room adjacency graph from the structure phase (both evolutionary and alternative) into valid game map is proposed, as it is also non-evolutionary.

6.1 Phase 1 – Room Distribution Phase

Room distribution utilizes one simplification when compared to its evolutionary counterpart: all rooms are assumed to be rectangular. This allows for easy generation, where rooms are represented as set of non-intersecting rectangles. With those in place, all that remains is making sure they are as tightly packed as possible, which can be easily achieved by extruding them, in manner not dissimilar to what extend edge mutation does.

One case to look out for in this process is situation, where rooms were generated in such unfortunate manner that required amount cannot be fit onto the map. However, given reasonable restrictions on rooms' initial size, making such situation unlikely, it can be avoided by letting the algorithm to try again couple of times. As a safety net, the algorithm will simply give up trying to distribute the remaining rooms after too many attempts failing, yielding a map with fewer rooms than required. Note that evolutionary version of the room distribution phase also run into similar situation.

Second case to look out for is making sure there is at least one room large enough to hold heroes' party. With the extrusion part of the proposed algorithm, generating such room is highly likely, but to ensure its existence, it is a good idea to further restrict first rectangle generated to be at least required size. Note that this implies that unlike its evolutionary counterpart, this algorithm needs to know heroes' party size.

```
rooms ⟵ [] (the size of C)
last(rooms) ⟵ generate new rectangle with area at least the size of D
Repeat until rooms is filled, or enough attempts have failed:
    Until last(rooms) does not overlap with any other rectangle:
        last(rooms) ⟵ generate new rectangle
For each r ∈ rooms:
    Extrude right edge as far as possible without overlapping
    If the resulting edge would touches map boundaries:
        Cancel the extrusion
    Extrude bottom edge as far as possible without overlapping
    If the resulting edge would touches map boundaries:
        Cancel the extrusion
```

Algorithm 4: Alternative room distribution phase.

6.2 Phase 2 – Structure Phase

In evolutionary version of the structure phase, numerous connections were pruned by masking them out using the concept of branches. Non-evolutionary version starts from the same point, but instead of masking connections out, it prunes them by finding a spanning tree of the room adjacency graph and then assigning branches to the nodes so that the definition of a branch stays intact.

```
s ⟵ highest degree node corresponding to large enough room
Reduce G to spanning tree using BFS and calculate distances
n ⟵ round(ζ ∗ |V|)
nodes ⟵ sort(V) (descending order with respect to distance from s,
                 leaves have priority over inner nodes in case of match)
remains ⟵ n
i ⟵ 0
l ⟵ dist(first(nodes))
While l > 0 and remains > 0:
    While dist(nodes[i]) = l:
        If nodes[i] is leaf:
```

```
12                          Put nodes[i] in a new branch
13                  Else:
14                          children ⟵ children of nodes[i]
15                          If |banches| − |children| + 1 < B:
16                                  Abandon outer loop
17                          b ⟵ branch(first(children))
18                          branch(nodes[i]) ⟵ b
19                          For each child ∈ children:
20                                  branch(subtree(child)) ⟵ b
21                  remains ⟵ remains − 1
22                  i ⟵ i + 1
23          l ⟵ l − 1
24
25  sort(branches) (ascending order with respect to size)
26  i ⟵ 0
27  While i < |branches| and |branches| ≥ B:
28          remains ⟵ |branches[i]|
29          Remove branch branches[i]
30          For each b still existing branch:
31                  If remains = 0:
32                          Break inner loop
33                  While not fail:
34                          If branches[i] = b:
35                                  Fail
36                          v ⟵ startingRoom(b)
37                          If parent(v) = s:
38                                  Fail
39                          siblings ⟵ children of parent(v) w/o v
40                          If v has siblings who are not in any branch:
41                                  Fail
42                          If |banches| − |siblings| < B:
43                                  Fail
44                          branch(parent(v)) ⟵ b
```

```
45                          startingRoom(b) ⟵ parent(v)
46                          For each child ∈ siblings:
47                                  branch(subtree(child)) ⟵ b
48                          remains ⟵ remains – 1
49          i ⟵ i + 1
50  Only take B largest branches
```

Algorithm 5: Alternative structure phase.

The algorithm first assigns suitable starting room. Unlike in the evolutionary version, the starting room is selected once and then stays unchanged. Selecting it at random could still work well enough, but in interest of achieving more interesting results, more sophisticated method was chosen.

To allow the player to choose from multiple paths, the starting room is selected as a room with highest degree node in the room adjacency graph, large enough to hold the heroes' party. This room is then used as root for the spanning tree, so that the starting room is in certain sense located in the middle of the map.

The process of assigning branches on the tree can be split into two parts. The first part traverses the tree node by node in depth order, with deeper nodes going first (variable l in *algorithm 5* stands for *level*). Those are all assigned a new branch if they are leaves (line 12), or their first son's branch, whereupon branches of their other sons are merged with it as well (lines 14 – 20). If the number of branches was to drop under required number of branches (line 15), the loop is abandoned early.

Drawback of leaving the algorithm after part one is that it has tendencies of creating larger number of extremely short branches. To counter that, the second part of the algorithm loops through the branches from shortest to longest and redistributes nodes from the short branches into all the others, until desired number of branches is acquired. In case we are not able to redistribute nodes in that way (for instance because all branches have reached the starting room), we only consider desired number of branches that are largest. It should also be noted that this algorithm is deterministic.

6.3 Post-Processing and Populating the Dungeon

Either version of the structure phase generates graph that governs connections between rooms, but this structure needs to be translated to actual map, and the map then needs to be populated with loot and encounters. Besides the graph, post-processing has also map, as outputted by the room distribution phase, available. Note that this map is assumed to have connection between all neighboring rooms and our goal is to mask some of those out.

```
1   branchToKey ⟵ []
2   For each door X ~_d Y on the map:
3       If branch(λ(X)) ≠ branch(λ(Y)):
4           If one λ(·) is branch starting room and the other is not
5           in any branch:
6               b ⟵ branch(X) if X is in a branch
7               b ⟵ branch(Y) if Y is in a branch
8               If branchToKey does not contain (b,·):
9                   k ⟵ newKey(b)
10                  Add (b, k) into branchToKey
11          Else:
12              Disregard X ~_d Y relation
```

Algorithm 6: Applying modifications from the room adjacency graph on the map.

Algorithm 6 loops through all the connections from the room distribution phase and judges whether they should stay. There are three possibilities to consider:

First one is case when one side of the door is a branch starting room, while the other one is not in any branch at all. In this case, we want to keep the connection, but lock the door. This is good opportunity to map out which doors are locked and by which key. Method $newKey$ in the algorithm generates new key based on the branch it unlocks.

Second possibility is when branch of one side of the door matches the branch of the other side. In this case, no action is required, the connection is kept as is.

Last case is when branches on each side are different, but the condition from the first case is not met. In this case, we want to delete the connection. After the map connections have been pruned, we can proceed to populate the dungeon.

```
1   contentType ⟵ [] (mapping from rooms to type of content)
2   endings ⟵ []
3   keys ⟵ [] (mapping rooms that should contain key on the key)
4   For each r ∈ R:
5           Add (r, vacant) to contentType
6   For each v ∈ V_BS:
7           Add ending(v) to endings
8           Add (ending(v), key & boss) to contentType
9   sort(endings) (ascending order with respect to distance from
10                  starting room)
11  Add (last(endings), ending & boss) to contentType
12  Add (startingRoom, branchToKey(branch(first(endings)))))
13          into keys
14  For each i ⟵ 0 to |endings| − 2 (we are excluding the last element)
15          Add (endings[i], branchToKey(branch(endings[i + 1]))) into keys
16  For each r ∈ R:
17          If contentType(r) contains vacant:
18                  With small probability:
19                          contentType(r) ⟵ encounter
20          Place desired content on random tile within r.
21          In case of non-boss encounter, populate multiple random
22          tiles, difficulty and numbers grow with growing distance
23          from the starting room. Place a loot with small probability.
```

Algorithm 7: Populating the dungeon.

First and possibly most important thing to talk about are content types. Each room is first assigned a content type or multiple content types, which are then used to actually populate the dungeon.

Vacant content type is default value, indicating an empty room. *Key* indicates that there is a key in the room, with $keys$ containing additional information on what key exactly. Note that lines 12 – 15 populate this structure in following manner: key to the first branch is in the starting room, and every subsequent key is inductively located in previous branch. The ending of the last branch doesn't have a key, but is marked with *ending* and *boss* content types. *Ending* marks the goal room and *boss* marks a room with unique strong monster.

As discussed in *chapter 2*, content is described as matrix, with number at each element determining what kind of content should be present. While items have uniquely determined number tied to them, monsters do not. Encounters are only identified by range of numbers indicating their difficulty. The difficulty is simply distance from the starting room, offset to put the number into the range the game will interpret as monsters.

With this setup, populating the dungeon is simple matter. Starting at line 16, the rooms are looped and the content is placed at random. In case of non-boss encounters, number of monsters is random between 1 and 10, with bias towards higher end based on encounter difficulty (see uneven random number generation at the end of *chapter 8*). Items can randomly appear along encounters with certain probability. Note that in case all the content can't fit into the room, simple priorities come into play: *ending* and *key* are more important than *boss*, is more important than *encounter*.

7 User Documentation

Algorithms described in previous chapters are contained in project separate from the main game. This is to not create dependency on the level generator; in principle, with the game existing independently, it is possible to supply maps created by different means, such as user created, or procedurally generated using techniques dissimilar from those used in this thesis.

7.1 The Game

7.1.1 Loading a Map

Upon launching, main game automatically looks for a file $map.bin$ located in the same folder as the executable. This file is expected to contain a valid map for the game. If such file is not present or does not contain valid map, player will be notified in form of a message and the game will proceed to shut itself down.

7.1.2 Controls

When in game, the player is welcomed by simplistic interface. Its main two features are located in top left and bottom left corners of the screen. Top left corner contains button for ending a turn, as well as information on player's remaining lives. Bottom left corner meanwhile pops up information on various in-game objects when hovered over with mouse.

These objects include heroes, monsters, and items. Note that in line with rules as presented in *chapter 2*, information on monsters' combat types remain ambiguous until enough fights have revealed the truth. For this reason, corresponding caption states "possible types."

Additional information provided to the player are reachable tiles for selected hero (if any is selected), which are highlighted in yellow. This information accounts for steps the hero has already taken within the round, but does not let the highlights spill into the rooms without vision (except through doors).

Besides above described features, the game can be interacted with using mouse clicks. Usages for left and right mouse buttons are as follows:

Mouse button	Effect
Right (hold)	Pan the map
Left	Select, Deselect, Move, Use, Attack

Table 4: Mouse buttons functionality descriptions.

Lastly, it should be noted that locked doors are represented with pair of small red tiles on the floor, while unlocked doors are shown in green. Upon either collecting the *Ending orb* or losing all lives, the game will show a message informing the player about the outcome.

7.2 Level Generator

7.2.1 Launching the Generator

Unlike the main game, the level generator is not a graphical application and requires command line parameters to be launched. One of these parameters is compulsory and informs the application of which phase and version are to be executed. Four of the possibilities are listed in *table 5*. In case this parameter is not specified, an error message will be shown to the user and the application will proceed to terminate itself.

Parameter	Description
-le	Evolutionary room distribution phase
-la	Alternative room distribution phase
-se	Evolutionary structure phase
-sa	Alternative structure phase

Table 5: All possibilities of which version and phase should be executed.

Second parameter specifies where to look for configuration file and is formatted thus `-config [path to the file]`. This parameter, however, is not

compulsory, as every value expected in the configuration file has clearly defined default value, thus not including any such file simply results in taking all the default values for given phase.

It should be noted that any unexpected variables in the file, as well as variables with invalid or unexpected types, will prevent the application from running to avoid typos, and any unneeded values will simply be ignored.

7.2.2 Configuration File Formatting and Expected Variables

Formatting of the configuration file is fairly straightforward: any defined variable is expected to be on a separate line, with the name (case insensitive) separated by equal sign ("=") from the value. Any leading or tailing white characters, as well as empty lines, are ignored. List of all possible variables is given in *table 6*.

Variable	Deflt.	Description	Phase
Runs	10	Number of independent runs	All
Gencount	1500	Number of generations to run	le
Gencount	20	Number of generations to run	se
Popsize[7]	100	Population size	le, la
Popsize	50	Population size	se
Runpath	(8)	Folder to store results in	All
FloorTypeMutProb	0.5	Operator application probability	le
JoinRoomsMutProb	0.8	Operator application probability	le
SplitRoomMutProb	0.5	Operator application probability	le
RemoveRoomMutProb	0.8	Operator application probability	le
FillHoleMutProb	0.4	Operator application probability	le

[7] In case of non-evolutionary room distribution method, population refers to number of solutions to be generated.

[8] Creates new folder "results" in current folder.

ExtendEdgeMutProb	1.0	Operator application probability	le
RoomCount	20	Number of rooms	le, la
Width	30	Width of the map (in tiles)	le, la
Height	20	Height of the map (in tiles)	le, la
WallProb	0.5	Wall probability in initial population	le
InputPath	(9)	Path to the input object[10]	se, sa
ExtendBranchMutProb	0.4	Operator application probability	se
CompressBranchMutProb	0.3	Operator application probability	se
ReplaceBranchMutProb	0.2	Operator application probability	se
ReverseBranchMutProb	0.3	Operator application probability	se
ShiftStartingRoomMutProb	0.3	Operator application probability	se
BranchRatio	0.333	Proportional number of rooms in a branch	se, sa
BranchCount	3	Number of branches	se, sa
PartySize	5	Number of heroes	Not le

Table 6: List of all possible variables in the configuration file.

[9] InputPath by default looks for a file "input.bin" in current folder.

[10] Input object is a selected solution from the room distribution phase.

8 Code Decomposition

This chapter sets out to provide comprehensive overview of the concrete implementation techniques used in the projects. As mentioned in previous chapter, the generator is separated from the main game. It is important to note that while the game project does not depend on the generator, the generator requires access to some of the structures present in the game, such as class representing the map.

8.1 Technologies and External Dependencies Used

Both projects also rely on several external dependencies, list of which can be found in *table 7*. Both projects were created using Visual Studio 2017, version 15.5.7, with .NET Framework version 4.7.03056 and MonoGame Framework version 3.6. Visual Studio solution containing both projects can be found in attachment [1].

Name	**Link to GitHub page**
net-object-deep-copy	https://github.com/Burtsev-Alexey/net-object-deep-copy
RockCollections	https://github.com/OndrejPetrzilka/Rock.Collections
AVL Tree	https://gist.github.com/yutopio/5643839
FibonacciHeap	https://github.com/sqeezy/FibonacciHeap

Table 7: List of external dependencies.

All external dependencies are to be copied into "Externals" folder in root of each project. *FibonacciHeap* belongs to *RoguelikeEva* project, while all the other dependencies belong to *LevelGenerator* project.

Furthermore, from *FibonacciHeap* repository, only files *FibonacciHeap.cs* and *FibonacciHeapNode.cs*, both located in "src/FibonacciHeap" folder, are to be copied. From *RockCollections* repository, only files *OrderedDictionary.cs* and *OrderedHashSet.cs*, as well as folder "Internals," all located in folder "Rock.Collections" are to be copied.

8.2 The Game

8.2.1 Component System

The game project runs using MonoGame Framework 3.6. The philosophy of the development process is to imitate component structure found in most modern high-end game engines, such as Unity or CryEngine, and develop the game on top of that.

The application can be understood as a set of *scenes*, each of which defines its own virtual world by set of related *GameObject* instances. Every object in the virtual world is an instance of this class. Inheritance from it is not allowed, instead, a system is set in place where *components* can be attached to a game object and define all of its gameplay properties and behavior. Each game object can have multiple components attached, creating more complex behaviors.

Each component follows single responsibility principle, e.g. *Transform* is only responsible for defining position and scale, while *SpriteRenderer* is only responsible for remembering and drawing visual representation associated with the object. It should be noted that each component has ability to perform update every frame, as well as render on the screen, in case it is inherited from special *RenderableComponent* class. Information about updates and renders are delivered to the MonoGame Framework by propagating it up the class hierarchy to the instance of *SceneManager* class.

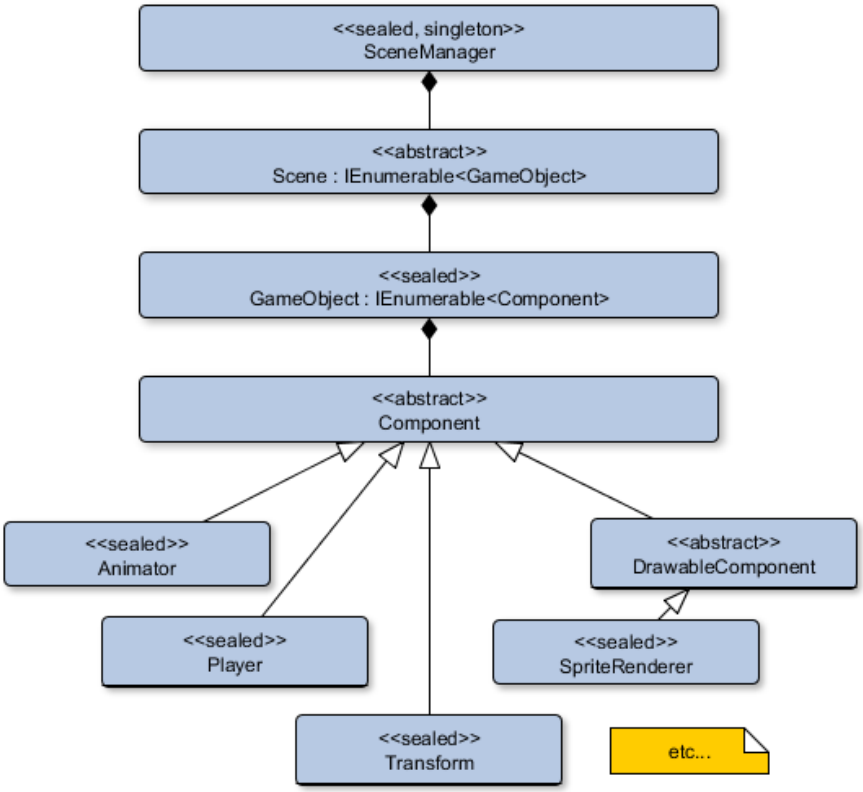

Figure 9: Component system hierarchy with examples of concrete components. In this figure, black arrows show relation "is in collection," and white arrows relation "is inherited from."

8.2.2 Scenes and Scene Manager

In previous section, scenes were briefly introduced as containers for game objects. They are represented as an instance of class inherited from *Scene* class and are responsible for loading related resources, instantiating their game objects and providing them for enumeration in *SceneManager* class.

SceneManager and *RoguelikeEva* are classes serving as a bridge between MonoGame Framework and the rest of the project. *RoguelikeEva* is inherited from MonoGame's *Game* class and is instantiated upon launching the game. It forwards update and render requests from *SceneManager* to the framework and handles game pre-processing, such as defining initial scene.

SceneManager meanwhile is a singleton class, available to any object in the game. It is responsible for identifying currently active scene, game objects and

components, and making sure only requests from those are being forwarded to the *RoguelikeEva* class. It should be noted that in case of rendering, *SceneManager* remembers each instance of a class inherited from *RenderableComponent* present in a current scene in dynamic array. This is done assuming that vast minority of components present will require to be drawn, thus remembering them at the side improves efficiency when rendering. Renderable components are automatically identified upon initialization.

One last thing *SceneManager* is capable of, is defining transformation matrix used when rendering. This is actually C# *Func<Matrix> event*, with *Matrix* being representation of right-handed 4x4 floating point matrix provided by MonoGame Framework. This is useful for components such as *Camera*, which alters what is being rendered based on its position.

8.2.3 Arena Scene

ArenaScene is the most crucial piece of the game. As discussed above, scenes group together related game objects, handle their initialization, and thus define a virtual world. *ArenaScene* in particular handles loading of the map and all gameplay elements once the map is loaded. In this section, some of more interesting approaches used to achieve this are going to be discussed.

Notable game object in this scene is globally available object containing components defining game mechanics. These mainly include *Player*, *TurnManager*, and *CombatManager* components.

Player component defines most of the player's interactions with the game and stores certain information, such as remaining number of lives, or hash set containing all retrieved keys. The component basically handles all player related aspects of the game.

TurnManager is component that governs flow of the game. Its main parts revolve around ending the player's turn (callback on clicking the "next turn" button), making each monster to take their turn, and observing when they are done doing so, whereupon the player is allowed to take their new turn.

CombatManager defines procedures that handle combat, based on combat types, combat initiator, as well as other factors. Since combat is the only way a hero can die, combat manager is also convenient place to check for losing condition.

It also handles awareness of combat types based on the combat outcome. Simple procedure is defined returning multiplier of the attacker's strength, given combat type superiority function τ. Since combat type awareness is implemented as *enum* with *Flags* attribute, new information learned from the combat can be obtained by anding old awareness with all combat types that would have resulted in the same multiplier. By nature of bitwise "and" operation, this will further restrict the possibilities, based on information learned.

It should also be noted here that combat types and their superiorities over each other are implemented simply as *rock, paper, scissors, lizard, Spock*, more complex variant of *rock, paper, scissors*, popularized by TV sitcom The Big Bang Theory (Warner Bros. Television, 2007 pilot).

8.2.4 Character Component, Pathfinding, and AI

Character component is abstract class, a predecessor to all heroes and monsters. It defines all characteristics of a unit, including hit points and strength, and handles one of the most important aspects of a unit: the ability to move between tiles.

Pathfinding is actually what majority of the *Character* component's code deals with. It move the unit along given path (array of tiles), or does nothing, if there is not any path available. This way, a unit can be moved by simply assigning a path to it. *Player* and *TurnManager* components are responsible for making sure at most one unit is moving at any given time.

Character also handles path finalization, i.e. initiation of an action upon reaching the destination, such as snapping to grid, consuming an item, or engaging in a combat. Lastly, it is capable of finding reachable tiles, based on the environment information and remaining speed.

Path initiation happens via vastly different means for heroes and monsters. While heroes' path are determined by the player, monsters are driven by artificial intelligence. As AI is not by any stretch focus of this thesis, the rules governing monsters' behavior are rather simplistic in nature. Given a state machine, monsters can be attacking, chasing, patrolling, or retreating, based on their awareness about nearby heroes' presence and combat types.

Pathfinding is implemented in *AStarPathFinder* class, whose instance is created in *Character* component. *AStarPathFinder* itself is standard C# class (not a component). As the name suggests, it contains implementation of A* algorithm.

Data structure used to store fringe is implemented in *HashedFibonacciHeap* class, which in itself is a wrapper around *FibonacciHeap* external dependency, enriched by ability to map objects stored in the heap to their position in the heap. This allows for quick checking whether an element is present in the heap or not. This is useful in pathfinding, as the ability to know whether an element is present can be used to determine whether it should be added, or its priority in the heap altered (both operations are $O(logN)$).

8.2.5 Content Managers

Besides above described components and classes, *Content* namespace contains few other important classes. Most of those follow naming convention *[Thing]Manager* and are responsible for spawning of actual game objects into the virtual world.

This for instance includes *HeroManager*, a class responsible of generating new heroes at the beginning of the game, as well as in case of hero's death. This is a good place to remind that monsters are only defined on the map as numbers expressing their difficulty. For this reason, *MonsterManager* is responsible for creating actual enemies based on that number, mostly through random means. Simple class *MonsterPrototype* is used to encapsulate all data shared among a monster species, such as their name, their sprite location, combat type, or combat types the player should suspect they might have (*enum* with *Flags* attribute, see above).

Lastly, *ItemManager* contains information on all items. Unlike the other managers, items are provided as fixed list. Note that some of the information contained within those classes, such as item list, are exposed for the generator.

8.2.6 Map Blueprint

The level itself is defined by three different classes. *Room* class is needed mainly to provide a way to update visibility in each room based on the ruleset. If each tile remembers which room it belongs to, it is easy to keep track of required visibility status.

Perhaps more interesting are classes *Map* and *MapBlueprint*. *Map* is used in the game itself and contains all kinds of game related content. This includes all the graphics used, tile size, list of rooms, and most importantly, all the tiles in form of two dimensional array of *Tile* class instances. Those recursively link to all the objects in the game, which creates complex, but convenient and easy to navigate structure, helpful for running the game.

MapBlueprint on the other hand, is more lightweight. The intent should be clear by now; unlike *Map*, *MapBlueprint* is meant to be serialized. Instance of this class is also output of the structure phase of the level generator. Its only content boils down to one byte defining the starting room and two dimensional array of one dimensional byte arrays, with the inner array representing three values of the three blueprint matrices on the position defined by the outer array (`byte[,][]`). For better clarity, this relation is described in *table 8*.

MapBlueprint class array positions	Blueprint matrices
`[i,j][0]`	$(M_R)_{i,j}$
`[i,j][1]`	$(M_D)_{i,j}$
`[i,j][2]`	$(M_P)_{i,j}$

Table 8: Mapping between *MapBlueprint* C# class arrays positions and blueprint matrices.

8.2.6.1 Serialization

Serialization and deserialization process utilizes *BinaryFormatter* class from namespace *System.Runtime.Serialization.Formatters.Binary*. A PNG image of the map can also be generated. This process utilizes standard *System.Drawing* tools, coming together to create simplistic looking version of the map.

8.3 Level Generator

Upon launching the level generator, the algorithm run is determined based on user's input. This offers an easy way of representing algorithms in form of an interface, called *IPhase*. Before phase is decided however, input parameters must be analyzed and configuration file parsed, if any is specified.

8.3.1 Parameters Manager and Configuration Reader

These are handled by classes *ParameterManager* and *ConfigurationReader*, respectively. *ParametersManager* determines which phase is supposed to run and where the configuration file is located. If any parameters are invalid, exception is thrown. *ParametersManager* is also used to store all parameters from the configuration file. This is handled by hash tables, mapping a string key onto supported types (integer, string, and double) and hash set of strings, making sure no key is used more than once, even across types.

ConfigurationReader uses *ParametersManager* to locate configuration file and store all the variables. Unlike *ParametersManager*, *ConfigurationReader* is aware of what to expect, when it comes to configuration file; it contains information about all expected variables as well as their default values. Default values are all set before the file is even opened, with the file then being able to rewrite them if required. *ConfigurationReader* is also responsible for making sure that values provided are valid, e.g. probabilities are within 0 and 1, etc. After *ConfigurationReader* and *ParametersManager* are done pre-processing, phase is properly defined and can be run as many times as required.

8.3.2 Evolution Namespace

This namespace provides a framework for EA. It further contains nested namespaces for fitness functions, hypervolume indicators, individuals, and operators. Two core classes in this namespace are *EvolutionaryAlgorithm* and *Generation*.

EvolutionaryAlgorithm, as the name suggests, is a wrapper for all components necessary for the generation process, and implements the actual loop of the algorithm. Terminal condition is handled by *Predicate<EvolutionaryAlgorithm>* delegate, which can contain arbitrary code yielding a boolean value every iteration. However, each single iteration is handled by *Generation* class, which defines immediate population.

Generation can only be instantiated from within (private constructor), or using factory method. The reasoning is, that the private constructor can be given any population and be defined as labeled with any ordinal number. When creating *Generation* instance outside however, this kind of freedom is not desired. Initial generation will always be the first and every individual in the population is randomly generated. For this purpose, the factory method accepts one individual as a sample, which is then deep copied using *net-object-deep-copy* external dependency, and initialized using abstract method from *Individual* class. Desired size of the population is also decided at this stage.

It should also be noted that since all EAs in this project utilize NSGA-II, instance of the class representing this algorithm is present in the *Generation* class and the code performing the iteration accounts for that. Furthermore, since initialization of individuals is independent, multithreading is utilized with .NET *Tasks*.

8.3.3 Individuals

As mentioned above, individuals define method for their random initialization. Beside that however, they also store information about all their fitness values (stored in array), as well as which front they belong to and what is their crowding distance. As we can see, knowledge of utilizing NSGA-II is exploited here as well. It should also be noted that individuals are serializable. This is important, because bridging first and second phase of the generation process requires storing individuals into files.

8.3.3.1 Layout Individual

It was mentioned in earlier chapters, that individuals in room distribution phase contain some additional data structures. There are two noteworthy, both dealing with simplifying the process of mutating rooms.

The firth is simply a hash table mapping bytes (used as identifications for rooms) onto ordered hash sets of coordinates. This hash set is actually *OrderedHashSet<T>* from *RockCollections* external dependency, which contains the same operations with the same complexities as its .NET counterpart, bat additionally guarantees order on the set. This is important for certain heuristic used in one of the operators. Usefulness of this structure should be obvious: operators now can grab a room quickly and process its tiles without first having to locate the room in the matrix.

The second is standard .NET *SortedSet<T>* (not to be confused with *RockCollections*' *SortedSet<T>*), containing room identifications (bytes). This data structure is special in a sense that it does not actually compare the bytes themselves, but rather sizes of their corresponding rooms, using custom comparer. Usefulness of this structure might not be as obvious right away, but it provides quick way to grab a room based on its size.

Note that the second of the structures presented does not update automatically. In case size of a room is modified, corresponding identification should first be removed and then re-added into the structure.

8.3.3.2 Room Adjacency Graph Individual

Similarly, individuals in structure phase also have some additional data structures. Namely, they contain hash table mapping branch identifications (bytes) onto graph nodes, labeling branch starting rooms. This structure is used numerous times in related operators.

The second structure is another hash table, this time mapping door identifications onto coordinates. Note that doors are defined by two tiles, not one, but for each door, given location of one tile, the other can be located almost immediately

in $O(1)$, so it does not matter. This structure is actually not used in any of the operators, but plays an important role in post-processing.

8.3.4 Operators

Operators are implemented as boxes eating an array of individuals, and spitting modified array as an output. This way, all individuals can be processed in parallel as per operator. Offspring individuals start as deep copies of their parents, using *net-object-deep-copy* external dependency. A lot of operators have rather straightforward implementation, however few of the more interesting ones are presented in this section.

Split room mutation has probably the most noteworthy implementation. Once a room is selected, an attempt is made to split it as evenly as possible. This is done using two-way flood fill algorithm. A pair of tiles, as far apart from each other as possible is taken, with both of its participants used as starting points for a flood fill. The algorithm stops when one of the flood fills runs out of open tiles, at which point closed tiles of the corresponding flood fill are assigned new room identification.

Finding the pair of tiles with required condition is potentially expensive operation, demanding $O(N^2)$ comparisons, but as hinted in previous chapters, a heuristic is used instead, making the operation much faster. The gist of this heuristic lays in simply taking first and last tile as present in ordered hash set in the corresponding room. Since tiles were being added into the structure in reading order, first and last tiles will in most cases be sufficiently far apart. Obviously, cases exist where this is not true, however potential benefits weighted against the odds were deemed more favorable.

Floor type mutation hides similar problem. As discussed in previous chapters, few special cases need to be looked out for. One of those was accidently splitting a room into two or more. In such case, some tiles need to be reassigned into new room identifications, as not to break room continuity. This is achieved again by flood fill algorithm, this time in its standard form.

The first tile in ordered hash set of the corresponding room is taken as a start and flood fill then locates all reachable tiles without spilling into neighboring rooms.

As long as content of this set is different from the content of the ordered hash set (i.e. expected number of tiles), all reachable tiles are reassigned and the flood fill is repeated. Note that during the reassignment, the tiles are also removed from the ordered hash set, so the first tile in the ordered hash set changes. Also note that this procedure will repeat upmost three times, in an unfortunate case when the tile removed in the operator was the only bridge among four disjoined areas, one from each side.

8.3.5 Phase Bridge

PhaseBridge class represents the bridge between door distribution and structure phase. Output file of first phase is actually serialization of this object. It encapsulates *LayoutIndividual*, *RoomAdjacencyGraphIndividual*, and *MapBlueprint*. Two ways of obtaining instance are creating it with constructor, which is how room distribution phase does it, or reading it from file using factory method, which is how structure phase does it.

A question might emerge how does *PhaseBridge* come to obtain *RoomAdjacencyGraphIndividual* and *MapBlueprint* objects. If *PhaseBridge* is instantiated manually, *Process* method needs to be called before its stored into a file. *MapBlueprint* is instantiated internally, which is necessity coming from the fact that *LayoutIndividual* only contains information about walls and floor (i.e. only first matrix from blueprint, no information on doors). For this reason, conversion takes place and doors are added between all neighboring rooms. *LayoutIndividual* instance is kept around for its helpful additional data structures, which are utilized in post-processing.

Room adjacency graph is then extracted from the blueprint and used to instantiate *RoomAdjacencyGraphIndividual*, which is in turn used as sample individual for the second phase. In case of non-evolutionary structure phase, the individual is still utilized, as the graph (chromosome) still serves as core for the algorithm, and additional data structures are used in post-processing. Note that in case of non-evolutionary room distribution phase, *LayoutIndividual* needs to be created before serialization.

Serialization and deserialization process itself uses *BinaryFormatter* class, just like was the case with *MapBlueprint*. Voluntarily, a PNG image of the map can be

generated, a process handled directly by *MapBlueprint* class (see earlier in this chapter).

8.3.5.1 Room Adjacency Graph Extraction

Perhaps the most interesting snippet of this class is algorithm extracting room adjacency graph out of the provided *LayoutIndividual*. The graph itself is represented as hash table, mapping room identifications onto instances of *GraphNode*, which is simple class remembering bunch of things about a graph node, such as its neighbors, its room identification, etc. *Neighbor* class is essentially representing one side of an edge in the graph, remembering the node we came from and door identification used.

The algorithm is principally breadth first search, with a twist. *LayoutIndividual* data structures are utilized to grab a room (first one without a loss of generality), which is then used as a starting point of the search. Neighbors are found by standard flood fill algorithm on the tiles, started from a random location (first tile in the room hash set, without a loss of generality). Wherever the flood fill spills into another room, new *Neighbor* instance is created.

8.3.6 Thread-Safe Static Random

The generator utilizes on multiple occasions multithreading using .NET *Tasks*. Unfortunately, standard *Random* class provided by .NET is not meant to be used for multithreaded applications, requiring an alternative to be used. Solution used utilizes multiple instances of *Random* class, one per thread, and one global to generate seeds for the thread instances. This approach requires usage of expensive locks when generating seeds, but since those are only utilized once per thread, overall efficiency is higher than naïve implementation with one global *Random* instance, locking single every request for a random number.

This random number generator also includes method for generating integer biased toward lower of bigger numbers. These are generated using following formula, where $\lfloor x \rfloor$ is x rounded down, max and min are boundaries, d is uniformly distributed random number between 0 and 1, and p is method parameter:

$$r = \lfloor \min + (\max - min) * d^p \rfloor$$

9 Results and Discussion

All approaches presented in the body of this thesis were run in all six possible arrangements: evolutionary and non-evolutionary for the first phase and all combinations of versions and outputs from the first phase in the second phase, which there is four of. All runs were realized using default parameters as presented in *chapter 7*, if not stated otherwise. It should be reminded here, that it implies ten independent runs for each combination.

9.1 Phase 1 – Room Distribution Phase

9.1.1 Evolutionary Approach

Evolutionary version of the room distribution phase showcased tendencies to heavily oscillate. As can be seen in *figure 10*, the difference between low ranking and high ranking solutions can be quite significant. Particularly interesting are spikes in hypervolume that happened in generations 941 and 1154.

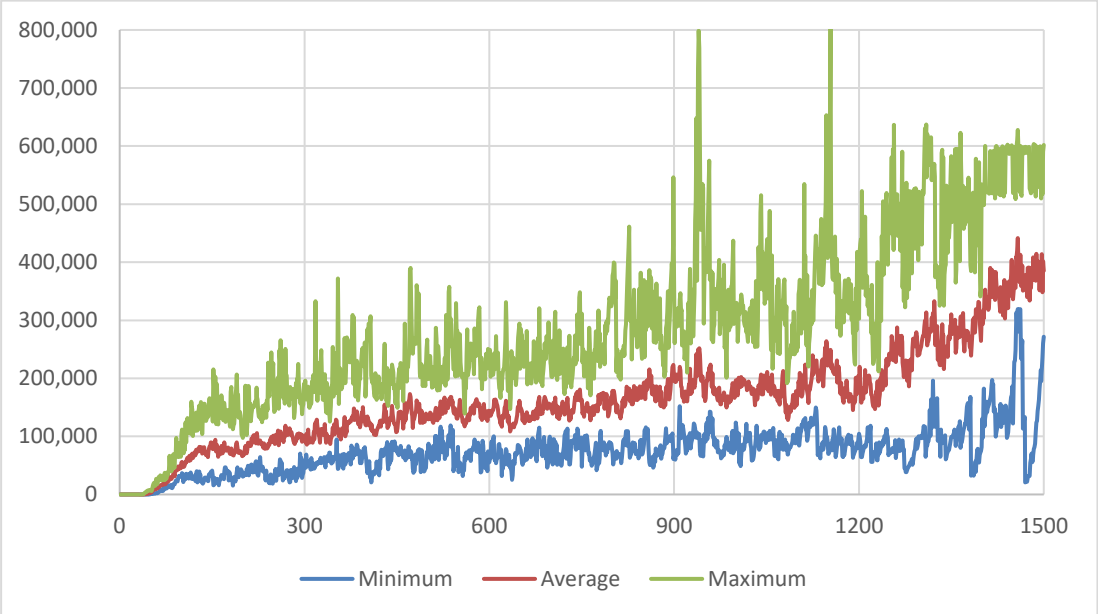

Figure 10: Evolutionary room distribution phase results.
Y-axis is hypervolume, X-axis is number of generations.

At larger number of generation, the algorithm showcased strong tendencies to stop improving and the maxima began to oscillate between 500,000 and 600,000. One

run was realized over the course of 5000 generations, which did not show any improvement of this trend. Detail on last 300 generations from data from *figure 10* can be seen in *figure 11*. The oscillation mentioned above can be seen to begin at around generation 1400, although average value still seems to be growing slightly.

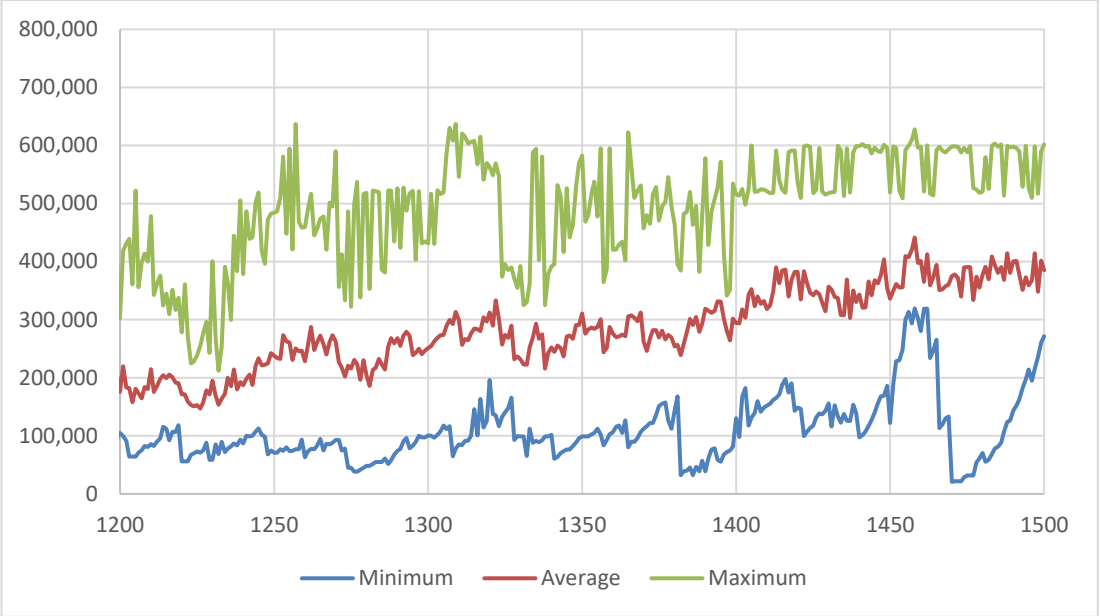

Figure 11: Evolutionary room distribution phase results, last 300 generations. Y-axis is hypervolume, X-axis is number of generations.

As expected, resulted rooms tend to be tightly packed together and collectively cover large area (leaving only little room to walls). In fact, maps with little no walls or even no walls at all seem to be relatively common. When it comes to sizes of the rooms, large rooms have tendencies to form on the outskirts of the map, while small ones form in the middle. This can probably be attributed to the extend edge mutation, since it extrudes rooms in random direction as far as it can.

Total number of rooms does not deviate too much from the desired value. Average value of f_2 in the first run, over all solutions in Pareto-optimal front approximation is 0.294847, therefore average deviation from the desired number of rooms is just 1.2213 rooms. Average number of neighbors and average number of floor tiles are 5.478561 and 592.7 respectively. These number are representative. All data can be found in attachments. Note that all runs have managed to eradicate invalid solutions within first 400 generations at worst (mostly within first 300 generations).

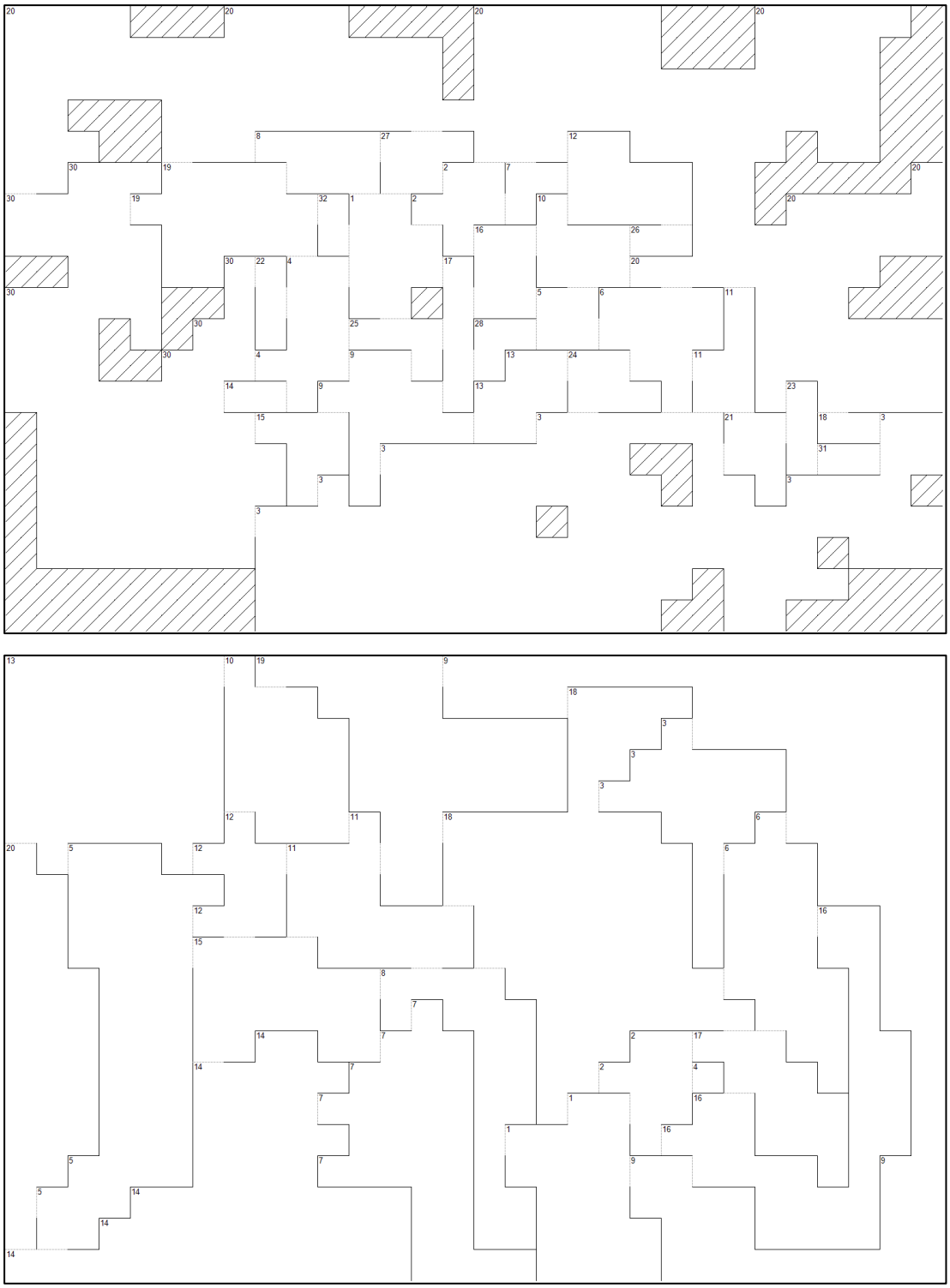

Figure 12: Two selected maps outputted by evolutionary room distribution phase. The bottom one was deliberately picked as an example of map with no wall tiles. Both can be found in attachment [2] (from top to bottom) under `\selected examples\phase 1 evolution (1).bin` and `\selected examples\phase 1 evolution (2).bin`.

9.1.2 Alternative Approach

Solutions from non-evolutionary version of the algorithm can be easily identified due to the restriction on shape of the rooms. They have noticeable tendencies to not occupy as much space as is the case with the evolutionary method. The rooms also tend to group in the middle of the map, leaving the edges empty (filled with wall tiles).

The extrusions of rooms were disallowed to take place if the rooms were to touch edges of the map. This decision was made due to the concern of making absurdly large rooms on the outskirts of levels, however that did turn out to be the case with the evolutionary method. Allowing extrusions all the way to the edges then might have been the right choice in retrospect, if the results are supposed to mimic the evolutionary version. That being said, large blocky rooms might also be less interesting when compared to cave-like structures outputted by the evolution.

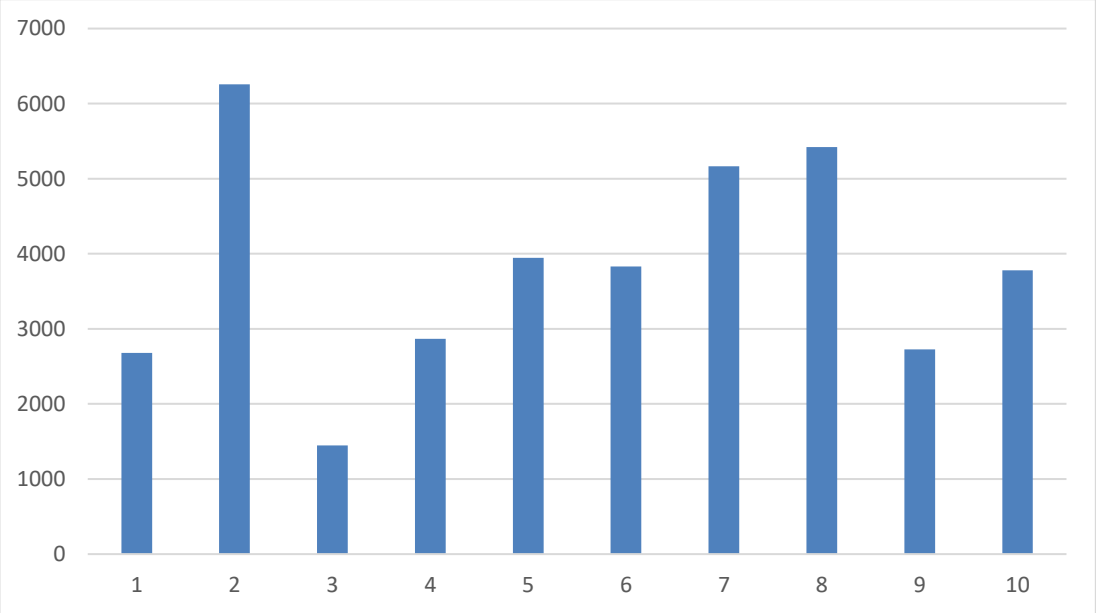

Figure 13: Hypervolumes of populations of 10 independent runs of the non-evolutionary room distribution phase.
Y-axis is hypervolume, X-axis is ordinal number of the run.

As can be seen in *figure 13*, in terms of hypervolume, non-evolutionary method performs 2 orders of magnitude worse when compared to its evolutionary counterpart. The most obvious culprit is the fact that the evolution was designed to optimize the fitness functions, while the alternative method was oblivious of them. Indeed, when evaluating fitness functions on the results, the first and the last of them suggest just that: the evolution did its job and optimized the solutions. In particular, majority of the hypervolume difference can be attributed to the large amount of wall tiles in the non-evolutionary results. Average number of neighbors does not seem to be as much lower.

However, non-evolutionary version has strong tendencies not to deviate from required number of rooms. In fact, no solution generated throughout the 10 runs deviated at all. Average fitness values across all runs (i.e. 1000 solutions) are listed in *table 9*.

Fitness	**Value**
Average number of neighbors (f_1)	3.4433
Deviation from required number of rooms (f_2)[11]	1
Number of floor tiles (f_3)	279.895

Table 9: Average fitness values of non-evolutionary solutions from the room distribution phase.

This seems as a good place to point out that results from non-evolutionary method are not required to dominate in order to be stored. When dominance is enforced, only about 3 to 5 solutions survive, but above noted observations still hold, i.e. the numbers do not improve significantly.

[11] It should be reminded that f_2 is actually exponential of negative deviation.

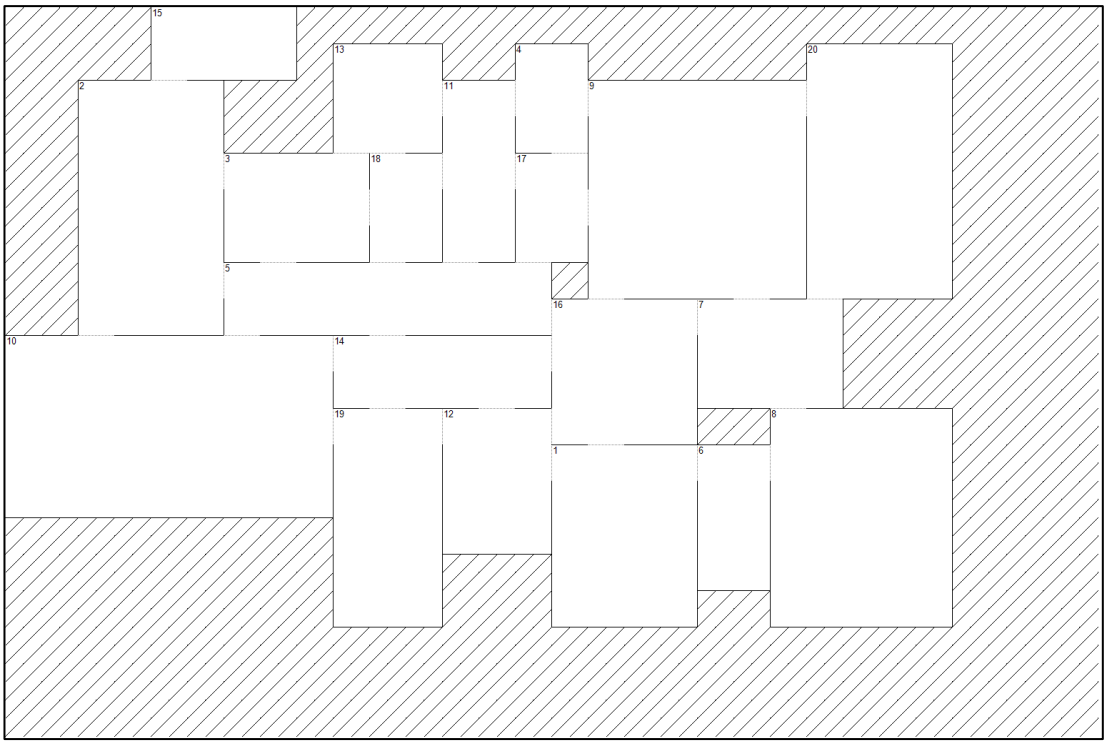

Figure 14: Selected map outputted by non-evolutionary room distribution phase. It can be found in attachment [2] under `\selected examples\phase 1 alternative.bin`.

9.1.3 Practicality Comparison

It comes as little surprise that by far the biggest drawback of the evolutionary approach is the time required to generate the maps. Another aspect to point out is that the alternative approach generates completely independent maps, therefore each is different from the others. From 100 generated maps using this method, all could potentially be used as subsequent levels.

On the contrary, the evolutionary approach tends to create subpopulations on the Pareto-optimal front approximation, which tend to be very similar. This inevitable leads to throwing a lot of levels, that we have invested so much valuable time in, out of the window. Furthermore, we have little control over the number of those subpopulations. While the first independent run seems to contain couple of them, the second only contains one, which translates to only one usable level from the second run.

Time required to generate Pareto-optimal front approximation using the evolution with used parameters on Intel i7 2.20 GHz CPU was on average 3,195,767.1 milliseconds, which translates to roughly 52 minutes per run.

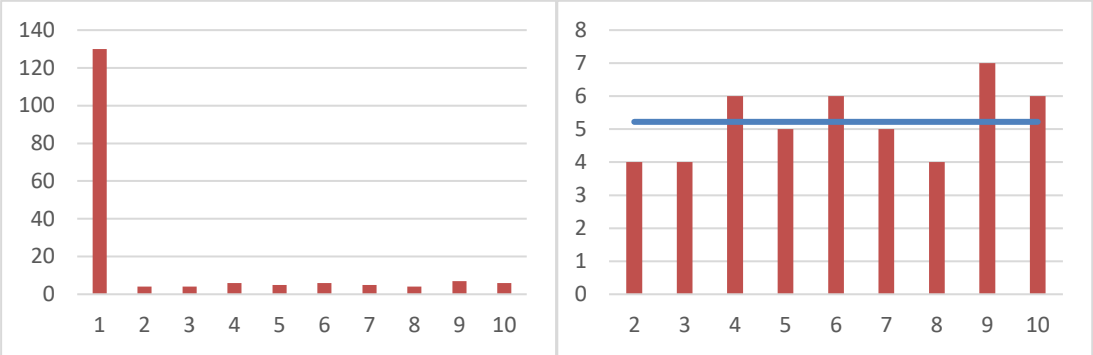

Figure 15: Elapsed time for each non-evolutionary run of room distribution phase. The chart on the right excludes first run and marks the average value (blue line). Y-axis is time in milliseconds, X-axis is ordinal number of the run.

As can be seen on *figure 15*, average time to generate a population with the alternative approach using the same hardware was just 5.222 milliseconds. This is denoted as the blue line in *figure 15*. Note that the execution times exclude all IO operations.

The very first run took about 130 milliseconds to generate however, a value odd enough that it should be addressed. Considering it only affected the first of independent runs, it is highly likely that this oddity can be attributed to .NET JIT compiler. For this reason, the first value was disregarded when calculating the average.

9.2 Phase 2 – Structure Phase

The second phase of the process requires an input file in form of the output of the first phase. This applies regardless of the approach used. As stated at the beginning of this chapter, all combinations of evolutionary and non-evolutionary methods were run during the experiment. Maps from *figures 12* and *14* were used as representatives from evolutionary and alternative methods of room distribution respectively. Besides them, different input files were also used in case of the alternative approach, due to its deterministic nature.

9.2.1 Evolutionary Approach

Evolution showcased quick convergence regardless of the input. Furthermore, evolved solutions are remarkably close to theoretical global optimum of hypervolume of 1. Actually, in case of non-evolved input, given that found solution has exactly 20 rooms and that we have required portion of 0.333 of them to be in branches, the best we can do in terms of the fitness measuring branch ratio is 0.983144 with 7 rooms in branches. Which is exactly what was found during the evolution.

Coupled with the fitness measuring distances from the starting room being equal to 1 (global optimum), the solution found is the best achievable solution in terms of fitness and hypervolume, given the input layout into account. Consequentially, the solution set only contains one individual, as it clearly dominates any other solution.

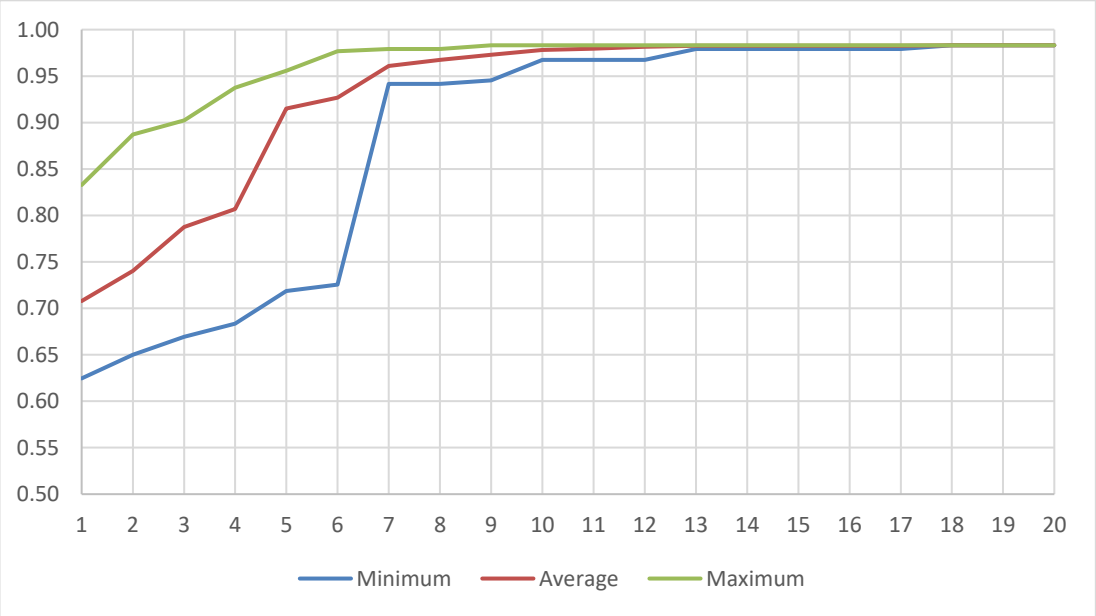

Figure 16: Evolutionary structure phase results evolved from non-evolutionary input. Y-axis is hypervolume, X-axis is number of generations.

Similarly, in case of evolved input, the solution found in several runs has evolved to have the first fitness value of 0.989635, which is the best possible value given the input containing 31 rooms, with 10 of them ending up in branches. The other fitness value ended up as 1, which is global optimum. Again, the best possible solution was found in terms of fitness and hypervolume, given limitations of the input.

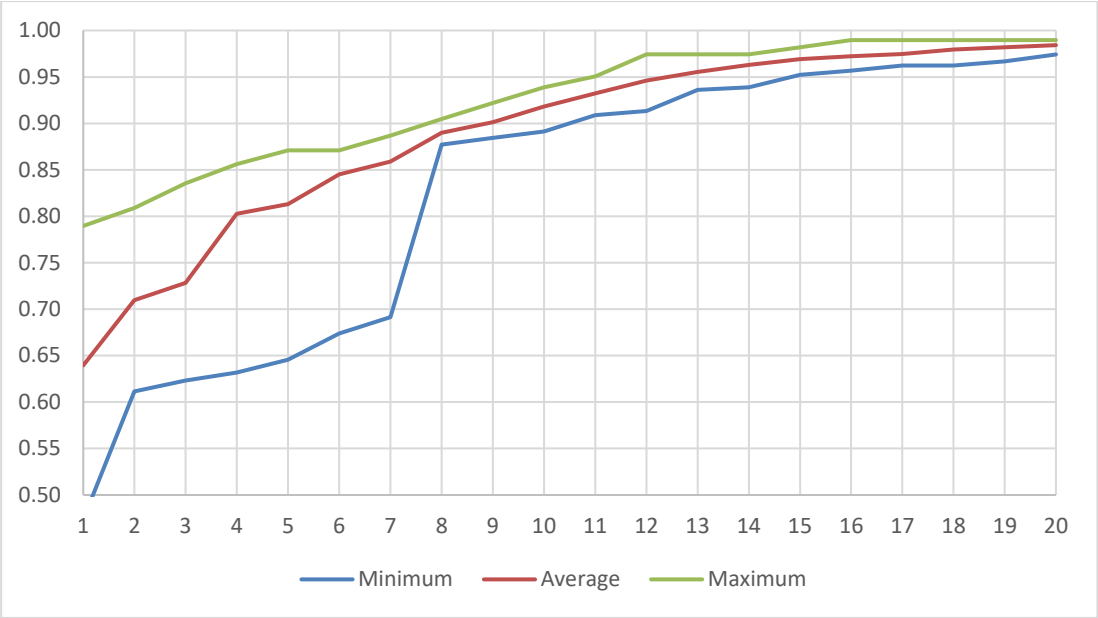

Figure 17: Evolutionary structure phase results evolved from evolutionary input. Y-axis is hypervolume, X-axis is number of generations

Convergence seems to be slower when initiating from evolved input, as can be seen in *figure 17*, but this phenomenon can be easily attributed to the fact that the input map in this case was considerably larger (31 rooms, i.e. 31 nodes for the graph, as opposed to 20 rooms in the other case).

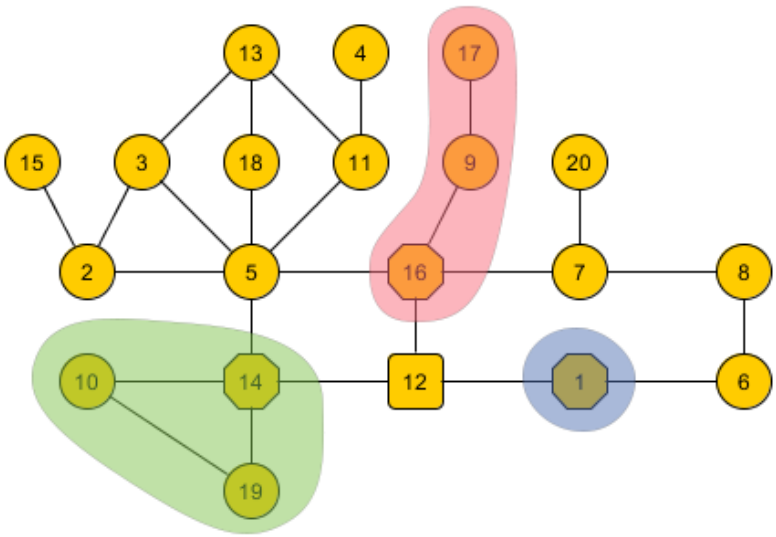

Figure 18: Selected room adjacency graph outputted by evolutionary structure phase evolved from non-evolutionary input. Branches are highlighted in red, green and blue, branch starting rooms are shaped as octagons, and the starting room is shaped as square. The map file can be found in attachment [2] under `\selected examples\phase 2 evolution (input alternative).bin`.

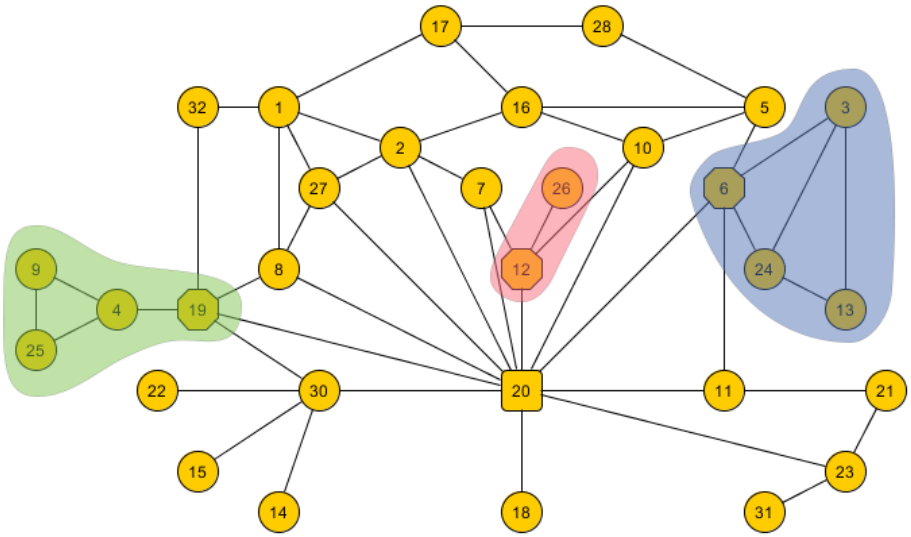

Figure 19: Selected room adjacency graph outputted by evolutionary structure phase evolved from evolutionary input. Branches are highlighted in red, green and blue, branch starting rooms are shaped as octagons, and the starting room is shaped as square. The map file can be found in attachment [2] under `\selected examples\phase 2 evolution (input evolution).bin`.

9.2.2 Alternative Approach

Non-evolutionary method of performing the structure phase is deterministic. This implies that the solution found will always be the same for the same input. Due to this reason, there is no point in running the algorithm multiple times on the same input, therefore multiple other maps were used to complete the statistics. List of all maps used is provided bellow. All of the files can be found in attachments [2] and [3].

- `\selected examples\phase 1 alternative.bin` *(attachment [2])*
- `\Results\phase 1 alternative\results, run 1\RESULT 45 suitable.bin` *(attachment [3])*
- `\Results\phase 1 alternative\results, run 3\RESULT 24 suitable.bin` *(attachment [3])*
- `\Results\phase 1 alternative\results, run 6\RESULT 42 suitable.bin` *(attachment [3])*
- `\Results\phase 1 alternative\results, run 10\RESULT 84 suitable.bin` *(attachment [3])*
- `\selected examples\phase 1 evolution (1).bin` *(attachment [2])*

- `\Results\phase 1 evolution\results, run 3\pareto\RESULT 46 suitable.bin` *(attachment [3])*
- `\Results\phase 1 evolution\results, run 2\pareto\RESULT 40 suitable.bin` *(attachment [3])*
- `\Results\phase 1 evolution\results, run 5\pareto\RESULT 31 suitable.bin` *(attachment [3])*
- `\Results\phase 1 evolution\results, run 8\pareto\RESULT 82 suitable.bin` *(attachment [3])*

All the results showcase problems with the fitness measuring distances between the starting room and branch starting rooms. This can be attributed to the fact that the algorithm constructs branches from leaves of a spanning tree whose root is the starting room. This can (and does) lead to the fitness being not as good, if required number of branch nodes is not large enough to reach the starting room.

Values of this fitness for evolutionary and non-evolutionary inputs are similar, averaging at 0.683214, although evolution-based inputs seem to perform slightly better with average of 0.735714. The difference is small enough that this could be attributed to random chance.

The algorithm performed in terms of the other fitness significantly better, scoring average of 0.945789. The difference between average values based on the method used to generate the input is negligible in this case.

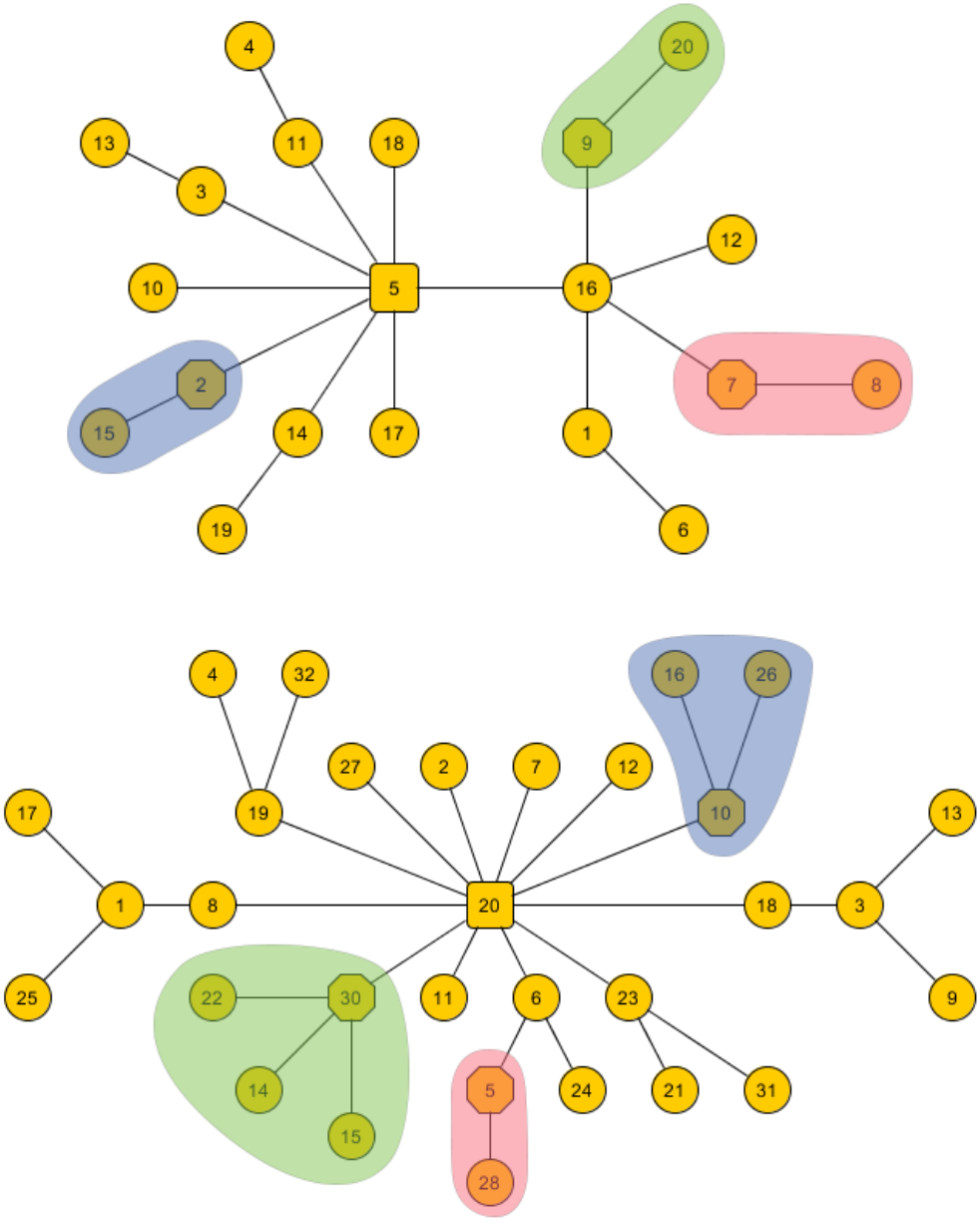

Figure 20: Selected room adjacency graphs outputted by non-evolutionary structure phase evolved from non-evolutionary input (top) and from evolutionary input (bottom). Branches are highlighted in red, green and blue, branch starting rooms are shaped as octagons, and the starting room is shaped as square. Both map files can be found in attachment [2] (from top to bottom) under `\selected examples\phase 2 alternative (input alternative).bin` and `\selected examples\phase 2 alternative (input evolution).bin`.

9.2.3 Practicality Comparison

Unsurprisingly, alternative method was yet again much faster, taking only 9.9 milliseconds to compute. Note an additional "dummy run" was performed this time, in order not to attain results muddied by .NET JIT compilation times, as was the case with the first phase.

Interestingly, evolution-based inputs consistently showcased prolonged computation times, averaging at 13.8 milliseconds, while most of the other solutions only took half, or in one case even third of that time to compute. Given that the evolutionary first phase tends to overshoot number of required rooms, this phenomenon could still easily be attributed simply to larger input graphs. After investigating fitness values of the individual solutions one by one, the one which was processed the fastest scored the highest in terms of the room fitness, while those who took the slowest scored the lowest, an observation which seems to strongly support the suspicion.

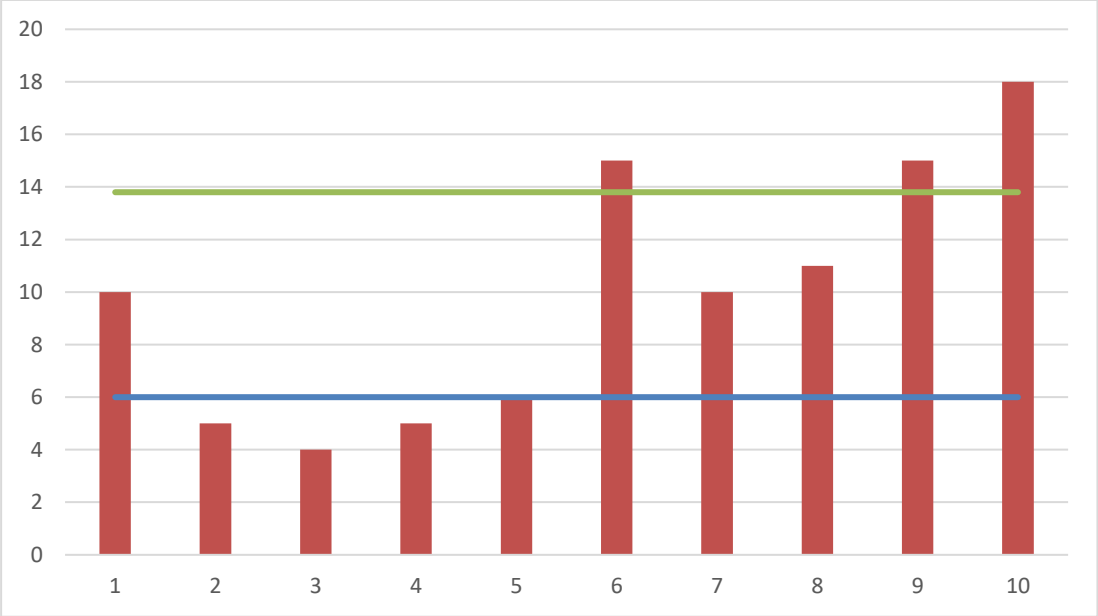

Figure 21: Elapsed time for each non-evolutionary run of the structure phase. Runs 1 – 5 are based on non-evolutinary inputs and runs 6 – 10 are based on evolutionary inputs. The chart marks the average values based of the method that generated the input, with evolutin-based inputs marked with the yellow line and the alternative with the blue line. Y-axis is time in milliseconds, X-axis is ordinal number of the run.

Average times required to evolve the results starting from evolutionary and non-evolutionary inputs were 2884 and 2127.2 milliseconds respectively, so only few seconds. These times, while still much larger when compared to their non-evolutionary counterparts, are short enough to even be used in online generation.

The evolutionary algorithm is easy to understand with operators and fitness functions that can be quickly adjusted to achieve different results. On the contrary, the alternative approach in this case was difficult to tackle via non-evolutionary means and the resulting algorithm is complex, comparatively more difficult to navigate, and was prone to bugs.

When choosing an algorithm to perform the structure phase, the elegant evolutionary solution should definitely be considered for those reasons, despite performing worse in terms of computation time.

9.3 Final Populated Levels

The four combinations produced several populated dungeons, ready to be played (see post-processing in *chapter 6*). Of those, four, one for each combination, were selected for discussion. All can be found in *figure 22*. Meanings of the letters in *figure 22* are explained in *table 10*.

Notation	Meaning
M	Monster
K	Key
B	Boss
I	Item
E	Ending

Table 10: List of letters from *figure 22* with descriptions.

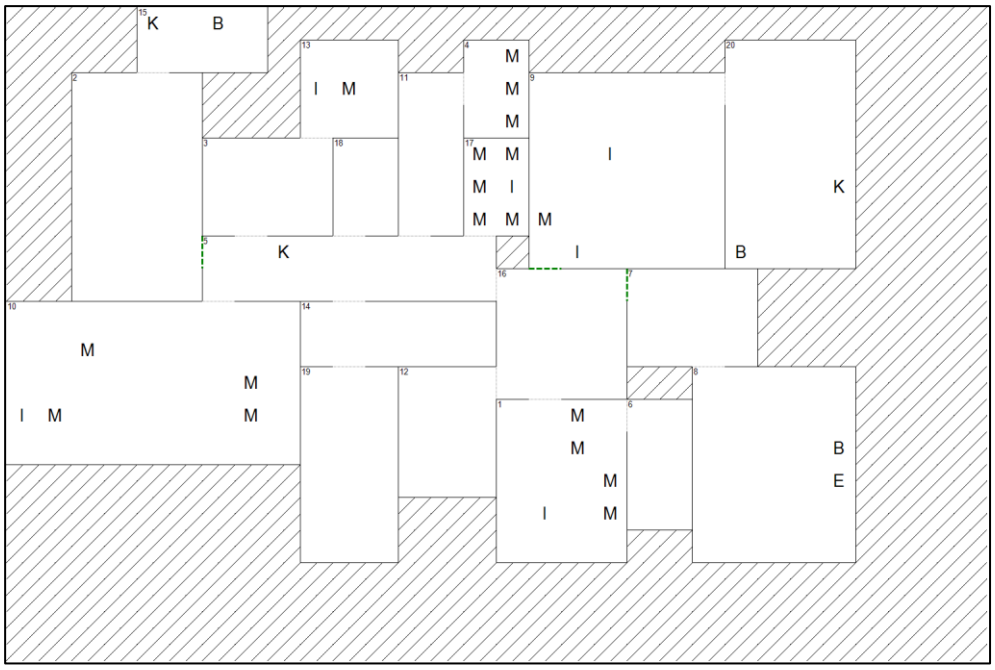

Figure 22a: Final map, constructed by completely non-evolutionary means. The file can be found in attachment [2] under `\selected examples\phase 2 alternative (input alternative).bin`.

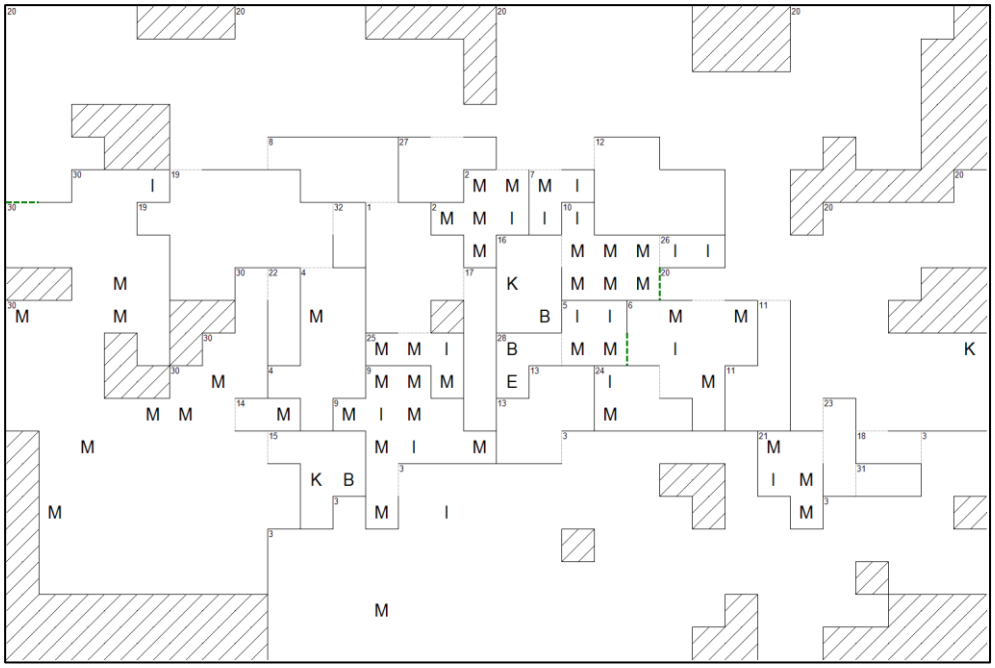

Figure 23b: Final map, constructed by evolution-based room distribution phase and alternative structure phase. The file can be found in attachment [2] under `\selected examples\phase 2 alternative (input evolution).bin`.

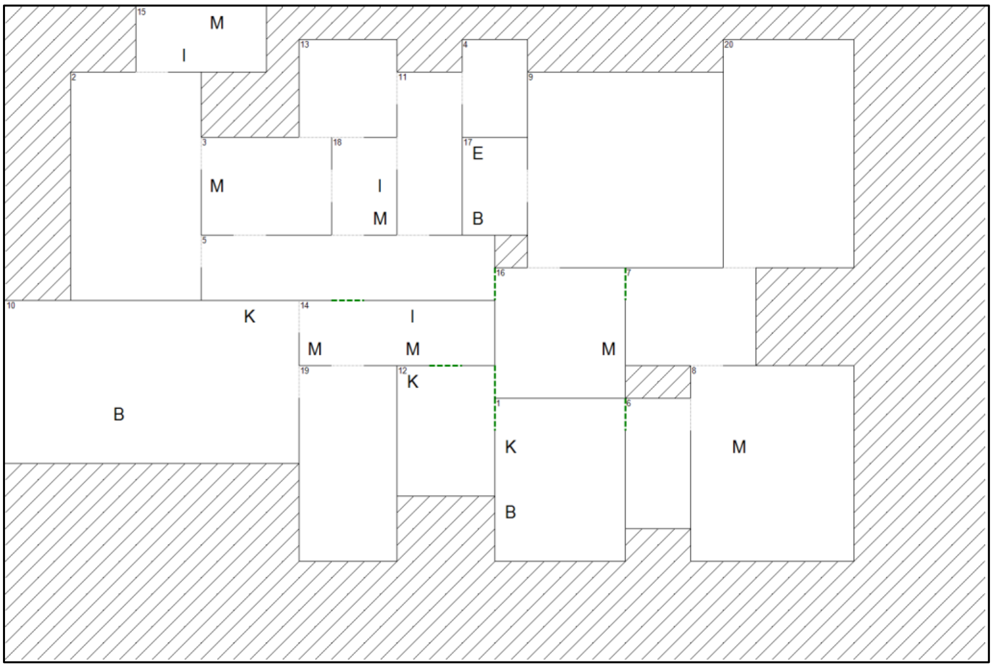

Figure 24c: Final map, constructed by non-evolutionary room distribution phase and evolution-based structure phase. The file can be found in attachment [2] under `\selected examples\phase 2 evolution (input alternative).bin`.

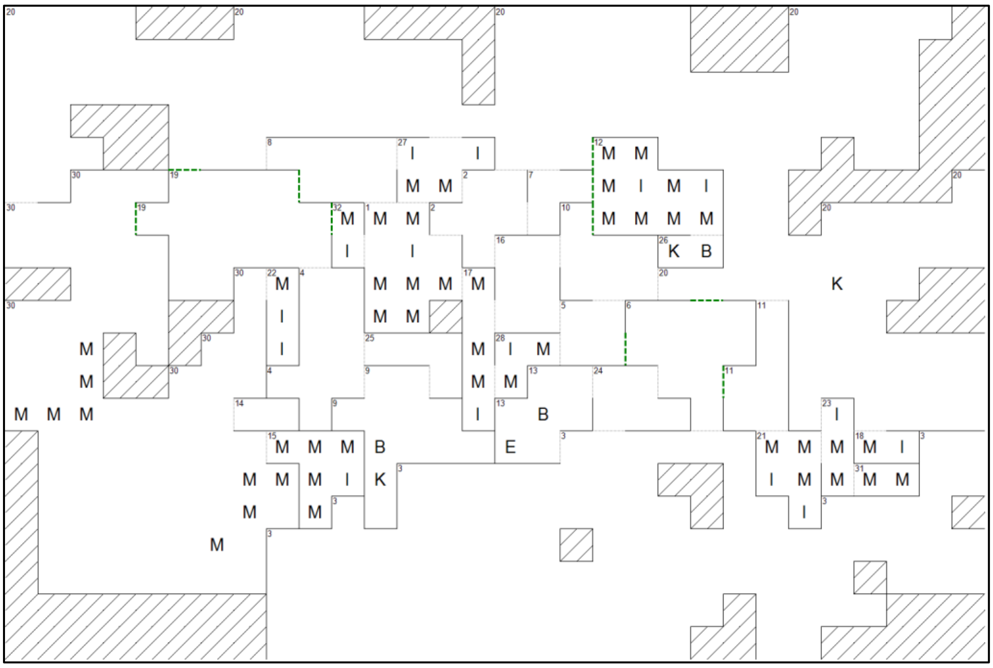

Figure 25d: Final map, constructed completely by evolutionary means. The file can be found in attachment [2] under `\selected examples\phase 2 evolution (input evolution).bin`.

Green dashed lines in *a* – *d* signify locked doors.

The maps tend to develop areas with high density of encounters. From the perspective of the gameplay, those could be viewed as “monster lairs.” Notably, map from *figure 22c* is lacking any such area, although this is likely result of random chance, given that map from *figure 22a*, with the same room distribution, does contain one “monster lair.” The map from *figure 22b* has evolved particularly interesting lair setup near the middle, where a room filled with monsters is the only point through which a small room with only loot can be accessed. This can be seen as reward in the game.

The maps with evolution-based room distribution seem to develop more lairs, but that can easily be attributed to the selected example having considerably more rooms. That being said, as discussed above, evolution-based room distribution phase showcases tendencies to overshoot number of required rooms on average, so these results might be relatively common.

Evolutionary room distribution phase generates very cave-like structures, as opposed to rectangular rooms of its non-evolutionary counterpart. This is in line with the initial intention, although large rooms it tends to generate along the edges might not always be desirable. That being said, maps from *figure 22* with evolution-based room distribution in particular resemble to some degree areas with some sort of structure in the center, surrounded by gardens on the outside, which is an interesting result that might benefit from corresponding graphical makeup.

Currently, the algorithm handling populating the dungeon bumps up stats of monsters with growing distance from the start. Chances of generating more enemies also grow with growing distance. This was supposed to create growing tension with peaks when facing a boss encounter. Maps from *figures 22c* and *22d* seem to handicap from this, as one lacks challenging encounters, while the other contains a room filled with monsters adjacent right to the starting room. In fact, the room in question has more monsters in it, than how many monster there are in entire map from *figure 22c*. While the idea of increasing difficulty seems to hold in general, mentioned cases are clearly undesirable.

9.4 Conclusions

Evolutionary room distribution phase generates dungeons for the most part in line with the initial intention; the rooms are tightly packed and cave-like, albeit some of them are fairly large (for better or worse). This effect is interesting and something its non-evolutionary counterpart doesn't mimic very well. However evolutionary solutions don't seem to offer that many advantages in exchange for their long computation times. As those computation times prevent the algorithm from being used in online environment, they make usefulness of EA in this area rather limited. For this reason, perhaps investing some extra time into more convoluted non-evolutionary generator is the better option when it comes to room distribution (or distribution of building blocks in more general sense).

Evolutionary structure phase performed much better in comparison. It was fast enough to be potentially even used in online generation and outputted good solutions, which were difficult to mimic with non-evolutionary algorithm. It suffers from inhered need to express desired structure using mathematical functions, but there seem to be a potential in employing similar methods in practice. That being said, the concept of branches and their distribution across graph nodes is idea unique to the game at hand and thus of limited application.

9.5 Future Work

This chapter has provided detailed discussion of the results of the algorithms, mostly in terms of fitness values and computation times. In terms of actual "fun" gameplay, some points were made at the end, but most of those were either basic observations, or guesses based on assumptions of what an average gamer would consider intriguing. Perhaps more interesting results might be obtained by performing survey with number of people put into blind test. Blind in this context means that they would be unaware of which of the four generation combinations they are playing. With data from such survey, more solid conclusions could be made about the capabilities and merits of the generation methods presented.

Were the results from such survey to support conclusions made in previous section, the non-evolutionary room distribution coupled with evolutionary structure

combo could be altered to be used in online generation. This kind of level generator would better suit the roguelike aspect of the game and as it would effectively eliminate the long generation times (that were mainly present due to evolutionary version of the first phase), the idea is feasible.

The non-evolutionary room distribution phase can currently only generate rectangular rooms. In interest of more intriguing and hopefully more fun to play results, modifying the algorithm to work with more complex shapes might be desirable. Perhaps adding random noise after all rooms have been distributed would be enough to solve – or at least improve upon – the issue.

The structure phase could use some minor adjustments in its operators and fitness functions. As can be seen in *figure 18*, it was evolved properly with the rules given, but nodes 13, 4, 15, 3, 18, 11, 2, 5, 20, 7, 8 and 6, despite not being part of any branch, are still effectively locked away from the player, as passing through a branch is required to get to them. Adding some sort of insurance that nodes not in any branch are all directly accessible from the starting room might be desirable.

Lastly, the generator itself might benefit from graphical user interface. Such modification would make it easier to use, although were the generator be integrated into the game as an online generation module as suggested above, this would not be needed. Future work related to the game and not related to the generator is purposely left out from this section, as it is not of interest in the context.

Bibliography

Ashlock, Daniel and McGuinness, Cameron. 2014. Automatic generation of fantasy role-playing modules. *Computational Intelligence and Games (CIG), 2014 IEEE Conference on.* 2014.

De Jong, Kenneth. 1975. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Michigan : Thesis (Ph.D.) - University of Michigan, 1975.

Deb, K, et al. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation.* 2002.

Holland, John H. 1992. Genetic algorithms. *Scientific american.* 1992, pp. 66 - 73.

İzgi, Erdi. 2018. Framework for Roguelike Video Games Development. Prague : Thesis (Mgr.) - Charles University in Prague, 2018.

Junkermeier, Jordan. A Genetic Algorithm for the Number Partitioning Problem.

Liapis, Antonios and Yannakakis, Georgios N. 2015. Refining the Paradigm of Sketching in AI-Based Level Design. *AAAI Publications, Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.* 2015.

Ma, Chongyang, et al. 2014. Game level layout from design specification. *Computer Graphics.* May 2014, Vol. 33, Issue 2, pp. 95 - 104.

Miller, B. L. and Shaw, M. J. 1996. Genetic algorithms with dynamic niche sharing for multimodal function optimization. *Evolutionary Computation.* 1996.

Nicola, Beume, et al. 2009. On the Complexity of Computing the Hypervolume Indicator. *IEEE Transactions on Evolutionary Computation.* August 18, 2009, pp. 1075 - 1082.

Potvin, Jean Yves. 1996. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research.* June 1996, Vol. 63, pp. 337 - 370.

Reeves, Colin. 2010. Genetic Algorithms. [book auth.] Fred Glover and Gary A. Kochenberger. *Handbook of Metaheuristics.* 2nd Edition. s.l. : Springer, 2010, Chapter 3, pp. 55 - 82.

Schaffer, James David. 1985. Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. *Proceedings of the 1st International Conference on Genetic Algorithms.* 1985, pp. 93 - 100.

Srinivas, N and Deb, Kalyanmoy. 1994. Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation.* September 1994, pp. 221 - 248.

Togelius, Julian, et al. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games.* 2011.

Zitzler, Eckart, Deb, Kalyanmoy and Thiele, Lothar. 2000. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation.* 2000.

List of Tables

Table 1 (p. 11) List of all elements in formal definition of the game defining map.

Table 2 (p. 11) List of all remaining elements in formal definition of the game.

Table 3 (p. 40) List of free parameters of the evolutionary solution.

Table 4 (p. 56) Mouse buttons functionality descriptions.

Table 5 (p. 56) All possibilities of which version and phase should be executed.

Table 6 (p. 57) List of all possible variables in the configuration file.

Table 7 (p. 59) List of external dependencies.

Table 8 (p. 65) Mapping between *MapBlueprint* C# class array positions and blueprint matrices.

Table 9 (p. 76) Average fitness values of non-evolutionary solutions from the room distribution phase.

Table 10 (p. 85) List of letters from *figure 22* (p. 86) with descriptions.

List of Figures

Figure 1 (p. 6) Screenshots of God Catching Alchemy Meister and derived game.

Figure 2 (p. 15) Concrete instance of a blueprint.

Figure 3 (p. 32) Two dimensional hypervolume with the origin as the reference point.

Figure 4 (p. 34) Calculating the hypervolume by sweeping the third dimension in decreasing order and boundary line of the dominated volume in a given instance of the two dimensional sub-problem.

Figure 5 (p. 35) Evolved rough sketch, sketch scaled up to higher resolution, and evolved final refined level (related literature).

Figure 6 (p. 37) Example of an evolved map (related literature).

Figure 7 (p. 38) Two levels generated using the same building blocks input and connectivity graph (related literature).

Figure 8 (p. 45) Room adjacency graph with three branches.

Figure 9 (p. 61) Component system hierarchy with examples of concrete components.

Figure 10 (p. 72) Evolutionary room distribution phase results.

Figure 11 (p. 73) Evolutionary room distribution phase results, last 300 generations.

Figure 12 (p. 74) Two selected maps outputted by evolutionary room distribution phase.

Figure 13 (p. 75) Hypervolumes of populations of 10 independent runs of the non-evolutionary room distribution phase.

Figure 14 (p. 77) Selected map outputted by non-evolutionary room distribution phase.

Figure 15 (p. 78) Elapsed time for each non-evolutionary run of room distribution phase.

Figure 16 (p. 79) Evolutionary structure phase results evolved from non-evolutionary input.

Figure 17 (p. 80) Evolutionary structure phase results evolved from evolutionary input.

Figure 18 (p. 80) Selected room adjacency graph outputted by evolutionary structure phase evolved from non-evolutionary input.

Figure 19 (p. 81) Selected room adjacency graph outputted by evolutionary structure phase evolved from evolutionary input.

Figure 20 (p. 83) Selected room adjacency graphs outputted by non-evolutionary structure phase evolved from both non-evolutionary and from evolutionary inputs.

Figure 21 (p. 84) Elapsed time for each non-evolutionary run of structure phase.

Figure 22 (p. 86) Selected final maps from all four phase combinations.

List of Abbreviations

PCG (p. 4)	Procedural content generation
RPG (p. 4)	Role-playing game
EA (p. 7)	Evolutionary algorithm
AI (p. 10)	Artificial intelligence
GA (p. 16)	Genetic algorithm
PP (p. 16)	Partition problem
TSP (p. 17)	Travelling salesman problem
RWS (p. 18)	Roulette wheel selector
SUS (p. 19)	Stochastic universal sampling
TS (p. 19)	Tournament selector
ES (p. 22)	Evolutionary strategies
MOEA (p. 26)	Multi-objective evolutionary algorithm
NSGA (p. 28)	Nondominated sorting genetic algorithm
VEGA (p. 28)	Vector evaluated genetic algorithm

Attachments

[1] Visual Studio 2017 solution containing both projects

- “Level Generator” only contains the level generator
- “RoguelikeEva” only contains the game project

[2] Example results presented in this thesis

[3] All generated results