MASTER'S THESIS | LUND UNIVERSITY 2018

Search-based Procedural Generation of Gameplay Content

Einar Nordengren

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-18

Search-based Procedural Generation of Gameplay Content

Einar Nordengren
`einar.nordengren@gmail.com`

June 12, 2018

Master's thesis work carried out at King Mobile AB.

Supervisors: Torbjörn Söderman, `torbjorn.soderman@king.se`
Michael Doggett, `michael.doggett@cs.lth.se`

Examiner: Jörn Janneck, `jwj@cs.lth.se`

Abstract

Over the years, game development has grown into a large business. Due to the ever-increasing demand for new content, there is much time, money, and effort that can be saved using automatic generation.

In this project, we have developed a procedural content generation tool to automatically define spawn points in a wave-based shooter game. Our intention was to create missions that follow the designers' desired difficulty and intensity levels, and that can mimic human behavior.

We evaluated the project through a survey, letting several persons play and rate the wave difficulty, intensity, and author. The results showed that it was possible to generate waves that somewhat followed the direction of the difficulty and intensity levels, but impossible to mimic human behavior.

We arrived at the conclusion that this was due to the nature of the problem: assuming what controls the difficulty and intensity measures is easier than predicting human design.

Keywords: genetic algorithm, evolutionary, direct evaluation, user survey, game design

Acknowledgments

Many thanks to Maria Karlsson for keeping the project alive during the first weeks, to Anna Huzelius for helping out with the survey, and to Jules Hanley for proofreading.

Contents

Acronyms **7**

1 Introduction **9**
1.1 Background . . . 9
1.2 Problem Description . . . 10
1.3 Related Work . . . 11
1.4 Scientific Contribution . . . 12

2 Theory **13**
2.1 Search-based Algorithms . . . 13
2.1.1 Content Representation . . . 14
2.1.2 Search Algorithm . . . 14
2.1.3 Evaluation Method . . . 16
2.1.4 Multi-objective Evaluation . . . 18
2.1.5 Online versus Offline . . . 18

3 Approach **21**
3.1 Method . . . 21
3.1.1 Program . . . 21
3.1.2 Project Evaluation . . . 21
3.2 Implementation . . . 22
3.2.1 Content Representation . . . 22
3.2.2 Search Algorithm . . . 24
3.2.3 Evaluation Method . . . 25
3.3 User Study Setup . . . 29

4 Evaluation **31**
4.1 Results . . . 31
4.1.1 Pareto Front . . . 31
4.1.2 Convergence Behavior . . . 31
4.1.3 User Study . . . 31

4.2 Discussion 37
4.2.1 Pareto Front 37
4.2.2 Convergence Behavior 37
4.2.3 User Study 37
4.2.4 General Discussion 39
4.3 Further Work 40

5 Conclusion 43

Bibliography 45

Appendix A NSGA-II 51
A.1 Background 51
A.2 Algorithm 51

Appendix B Questionnaire 55

Appendix C Test Results 57

Appendix D Enemy Statistics 61

Acronyms

DDA	dynamic difficulty adjustment
EA	evolutionary algorithm
ES	evolutionary strategy
FI-2Pop	feasible-infeasible two-population
GA	genetic algorithm
GP	genetic programming
MOEA	multi-objective evolutionary algorithm
MOP	multi-objective problem
NSGA	non-dominated sorting genetic algorithm
NSGA-II	non-dominated sorting genetic algorithm II
PCG	procedural content generation
SBPCG	search-based procedural content generation

Chapter 1
Introduction

This chapter includes an introduction to procedural generation and a motivation to why it is an important tool in today's game development. In addition, the chapter covers a description of the project and information about earlier work done in the area of search-based procedural generation.

1.1 Background

Developing games has become a complex process throughout the years, often demanding large teams of game designers, programmers, artists, sound engineers, and testers. At the same time, the players' demand for new content just keeps increasing, forcing the game companies to spend a lot of money hiring content creators. The most obvious way to treat these issues is to incorporate some automatic generation into the development process. In computer science, this is called procedural generation and deals with algorithmically creating data instead of doing it manually. In game development this is commonly known as *procedural content generation* (PCG), where content includes all different elements of a game such as rules, items, quests, spawn points, and structures.

One of the major issues with PCG is the difficulty of generating qualitative content in a wide context. Either the tools become very specialized at performing one task, such as in the commercial vegetation generator SpeedTree [1], or they try to encompass entire worlds as in No Man's Sky. In the latter case there have been problems with disappointed customers [2], possibly due to lack of diversity among the solutions. The ultimate PCG tool would be a *multi-level, multi-content* generator, as described by Shaker et al. [3, Chapter 1]: for a given game engine, generate all of the content while still guaranteeing high quality and full integration with the game. Although this is an unreasonable goal, every step toward it is valuable. In this thesis we will continue the development of content generators, with focus on the generation of qualitative and specialized content, similar to what SpeedTree did.

1.2 Problem Description

The application central to this project is a 3D mobile shooter game with a static player, whose objective is to shoot enemies spawning at different locations in a static environment. The gameplay is wave based, and the enemies are moving toward a fort that is to be defended by the player. If the player manages to defend the fort through a set number of waves, a mission is completed.

The task in this project is to partly define the spawn points for a certain mission, which here means that the enemy distribution of every spawn point should be determined for each wave in a mission. Every spawn point is responsible for spawning a group of enemies, assembled by different enemy archetypes. The parameter defining a spawn point is thus the *quantity of each enemy type* for a given wave. The actual content being generated is therefore just a set of numbers and falls into the category of non-visible gameplay, but how this content is generated is one of the most important factors concerning the game's dynamics and balance.

There are several different approaches to implementing procedural content generation, and it is not trivial to choose the best method. We will use a method known as *search-based procedural content generation* (SBPCG), a method that uses evolutionary, stochastic, or metaheuristic search techniques to find the content and an objective function to determine its quality. These different techniques will be explained later in the report.

In order to frame this project, we make a proposition that is going to be used as a reference point.

If we can create an algorithm that is able to define all the parameters concerning enemy spawning in a way that

- *follows the designer's desired mission difficulty and intensity and that*
- *is inseparable to a handmade level*,

that algorithm can be used to hasten the process of developing a game significantly. Regarding the second requirement, the goal is to construct an algorithm that strives toward eliminating the differences between the computer-generated waves trying to imitate a human and the waves designed by a human. The idea of imitating a human is really an alternate way of ensuring qualitative content. A human designer would create content with some kind of intention and believability, and those are the qualities we want to replicate.

The research and supplementary questions will then be:

RQ: *How can an algorithm that fulfills the above requirements be implemented using a search-based approach?*

SQ: *Is it possible to accommodate for both requirements? Why/why not?*

1.3 Related Work

As SBPCG is the main method used in this project, we present a few attempts that have already adopted this technique. The name *search-based procedural content generation* was first proposed by Togelius et al. [4] in 2010, but the core ideas had already been used to generate a wide range of content.

Also in 2010, Oranchak [5] used an *evolutionary algorithm* (EA) to generate Japanese Shinro puzzles based on the validity and entertainment value of each puzzle. The entertainment value was drawn from assumptions of what the player likes. They experienced challenges when trying to generate difficult puzzles since those puzzles required a certain type of symmetry.

Togelius et al. [6] generated racing tracks using a set target difficulty, evaluated by letting a neural network-agent drive through different types of tracks. They managed to produce drivable and seemingly well designed tracks, but as no humans took part in the evaluation, they had no idea whether the tracks were entertaining or not.

In the area of map generation, Togelius et al. [7], [8] generated StarCraft maps by optimizing the playability, fairness, and skill differentiation at the same time making sure the maps did not get bland and uninteresting. In order to achieve this, they had to make several assumptions about what a fair and interesting level was. They came to the conclusion that the use of a Pareto front (see section 2.1.4) is an excellent design support tool for human designers. Uriarte and Ontañón [9] also generated StarCraft maps, but with more focus on strategic balancing. By using a set of balance metrics, they showed the computer-generated maps to be comparable with those made by humans.

In the area of using questionnaires for evaluating these kinds of problems, Classon and Andersson [10] wrote a master thesis about how to procedurally generate levels with controllable difficulty based on player input. They selected a genetic algorithm and constructed the levels in an iterative procedure, letting the result from the user tests direct what parts had to be improved. They found that their method was a good way to ensure the objective functions' ability to evaluate the content correctly, but too slow for online generation.

Another thesis using input from users were written by Baldwin and Holmberg [11]. They conducted an in-depth user study, where a few professional game designers, animators, and developers were the test subjects. Their goal was to develop an AI-based, mixed-initiative tool to be used by the game designers, while still leaving them with a sufficient level of control. The main results from their study suggested that a mixed-initiative design tool is a good starting point when designing a game.

Similar to SBPCG, Jennings-Teats et al. [12] used an EA to generate levels for a platform game. Their goal was to dynamically adjust the difficulty of a level depending on the player performance, known as *dynamic difficulty adjustment* (DDA). After collecting data of the perceived level difficulty from over 200 players, they built a model using a neural network to correlate the obstacles with the difficulty. They developed a new strategy for DDA and highlighted the difficulty of making the player unaware of the algorithm.

The most famous game using DDA is probably Left4Dead [13]. They implemented an AI director responsible for controlling the game behavior. Their goals were to deliver robust behavior performances, provide competent human player proxies, promote replayability, and to generate a dramatic game pacing.

1.4 Scientific Contribution

Up to this point, search-based procedural generation has been used to produce puzzles, rules, terrains, maps, and stories. With this project we will add gameplay definition to that list and further discuss some of the questions stated by Togelius et al. [14].

- *Which types of content are suitable to generate?*
- *How is game content best represented?*
- *How can we best assess the quality and potential of content generators?*

as well as a question regarding the designer input:

- *How should the interaction between the designer and the algorithm work to accommodate for both controllability and ease-of-use?*

In addition, we will contribute with research to the rather unexplored area of using search-based content generators to mimic human designers.

Chapter 2
Theory

This chapter covers background theory about search-based algorithms and evolutionary search algorithms in general.

2.1 Search-based Algorithms

Search-based procedural content generation belongs to the family of generate-and-test algorithms. As opposed to constructive algorithms which generate the content just once, generate-and-test algorithms follow an iterative procedure and evaluate the generated content in every step.

The search-based approach is essentially an optimization technique that is able to solve problems with the following formulation:

$$\begin{aligned} &\textit{minimize}\ \ f(x) \\ &\textit{subject to}\ g_i(x) \leq 0,\ \ i = 1, \ldots, k \\ &\qquad\qquad h_j(x) = 0,\ j = 1, \ldots, p \end{aligned} \tag{2.1}$$

Here $g_i(x)$ are inequality constraints and $h_j(x)$ equality constraints. Constraints of all sorts are common in real-world problems, but they unfortunately make the optimization much more difficult to solve. Later on, we present different approaches on how to handle these constraints.

As described by Shaker et al. [3, Chapter 2], the search-based approach consists of three parts:

- a content representation,
- a search algorithm, and
- an evaluation method.

These parts are all described in more detail in the following sections.

2.1.1 Content Representation

In evolutionary algorithms, the content is usually expressed in two ways: as a genotype and a phenotype. The genotype can be seen as the genome or DNA from biology and is the representation the search algorithm uses. The phenotype is the representation used in the evaluation stage and could be compared to an organism's observable characteristics.

The mapping from genotype to phenotype could vary from direct to indirect. In a fully direct encoding, every part of the genotype maps linearly to the phenotype, whereas the genotype in a fully indirect encoding would just be a seed generating the phenotype. In general, the preferred mapping is somewhere between these extremes [3]. A direct representation works well for small problems, but for larger problems the size of the genotype grows too large and causes convergence problems due to the curse of dimensionality. The other end of the scale, where the genotype is just a seed number, would instead have a locality problem; a small change of the seed number would most presumably generate a completely different phenotype, making the likelihood of convergence infinitesimal.

If the task had been to generate a 10 by 10 grid of 90 white cells and 10 black cells, a possible representation of the genotype would be a 10 by 10 matrix of zeros and ones determining whether the color is black or white. This is an example of a fully direct representation. A semi-direct approach to the encoding problem, however, could be a vector of 10 coordinates specifying where the black cells are, which also would reveal where the white cells are. That would in most cases make the convergence process faster, and as long as the programmer knows how to map the genotype to the phenotype this representation works.

2.1.2 Search Algorithm

Although search-based methods often rely on evolutionary algorithms to do the actual search, other algorithms such as simulated annealing [15], particle swarm optimization [16] and stochastic local search are also included in this group, as stated in [14]. In this project we will use a *genetic algorithm* (GA), a subclass of EAs.

Evolutionary Algorithms

Evolutionary algorithms are inspired by biology and are thus based on three operators: *recombination*, *mutation*, and *selection* [3]. Given a population, the process of finding the fittest individuals includes evaluating all the individuals in one or several attributes, select suitable parents, recombine them, mutate the offspring, and then select the next generation. This is done iteratively until a certain stopping criterion is met, usually when one individual within the current population has a satisfactory fitness value or when the maximum number of iterations is reached. The fitness value is in this context a number expressing how fit an individual is. This way of finding the optimal solution is better than a fully random search as long as there is a positive correlation between the difference in fitness and the distance to the optimal solution [17, p. 79]. This basically means that altering an individual toward the optimal solution should make the fitness value approach the optimal fitness value.

The most common type of EAs today are GAs, which use recombination as the primary search approach. Other EAs, such as the *evolutionary strategy* (ES), invented by Rechenberg and Schwefel [18, p. 101], instead utilize mutations to explore the search space. Which algorithm to choose depends mostly on the problem being solved, how the content is represented, and how large the search space is. The GA uses a representation of bit-strings, the ES works best with real-valued vectors, while a tree structure representation is preferred when using a method known as *genetic programming* (GP) [18, Chapter 6].

Genetic Algorithm

The genetic algorithm is a rather straight-forward algorithm, described in pseudo-code in algorithm 1.

Algorithm 1 Genetic algorithm

```
1:  Initialize a population of $\mu$ individuals
2:  Evaluate the population
3:  while maximum number iterations not reached do
4:      for $\mu$ times do
5:          Select two parents according to some criterion
6:          Recombine the two parents into a new individual
7:          Add the new individual to the offspring population
8:      end for
9:      Mutate the offspring
10:     Replace the population with the offspring
11:     Evaluate the population
12: end while
13: Choose the best individual from the population
```

The pseudo-code in algorithm 1 describes only an outline of the GA. By choosing different ways of implementing the inner steps, a large variety of GAs can be achieved. The most basic type of GAs is called Simple GA, and the sketch of its characteristics, as described by Eiben and Smith [18, Chapter 6], is shown in table 2.1.

Table 2.1: Simple GA

Representation	Bit-strings
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Roulette wheel
Survival selection	Generational

Representation: The representation referred to here is the genotype, the one that the search algorithm uses, and should not be confused with phenotype used in the evaluation stage. Figure 2.1 shows a possible bit-string representation of 8 bits.

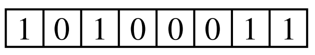


Figure 2.1: Example of a bit-string

Recombination: The 1-point crossover is a method of combining two parents into one child. A random number $r \in [1, l-1]$ (l is the length of the bit-string) marks where to split the parents' genotypes, giving the offspring a new, combined genotype. This is visualized in figure 2.2.

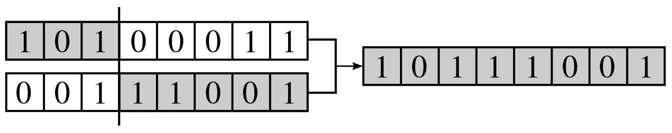


Figure 2.2: This figure shows the recombination of two bit-strings. Here $r = 3$, giving the child three bits from the upper parent and five bits from the lower parent.

Mutation: The mutation in a Simple GA is a bit flip of some bits controlled by a random process, often with a uniform distribution. Usually, there is a small probability of around 5 % that a bit is flipped. Even though mutation is not the primary search method in GA, it helps to avoid getting stuck in local minima.

Selection: The parent selection in the Simple GA is implemented using a roulette wheel technique. This method chooses a suitable parent by spinning a wheel where the sizes of the holes represent the individuals' fitness values.

The survival selection describes how the next generation is chosen. Letting the offspring replace the parents entirely is called *generational* survival selection.

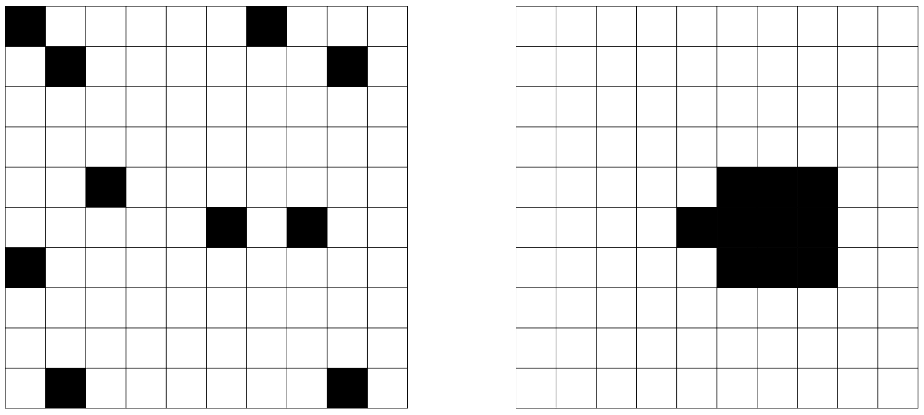
2.1.3 Evaluation Method

The choice of evaluation method is presumably the part with the greatest impact on how the final content will look, and again there are several options to choose from. Shaker et al. [3] divide the options into three categories: *direct*, *simulation-based*, and *interactive* evaluation. The evaluation method's task is to convert the phenotype into a single fitness value.

Direct Evaluation

This method evaluates the phenotype data of the individual. Either a theory-driven method is used where a designer determines what makes content good or bad (expressed in objective functions), or a data-driven approach is used where the decision is based on data gathered from questionnaires or some kind of user input.

Continuing on the example with the black and white squares, the phenotype would be a matrix where each cell indicates if it is black or white. Imagine the final goal is putting all the black squares close to each other. In that case, a theory-driven, direct evaluation function could be a measure of the largest distance between all of the black squares. The



(a) Unfit individual **(b)** Fit individual

Figure 2.3: If the algorithm works properly, the unfit individual in **(a)** is only apparent in the beginning of the search, while the population only contains individuals such as **(b)** toward the end.

search algorithm's goal would then be to minimize that number. A comparison between a fit and an unfit individual in this case is illustrated in figure 2.3.

Simulation-based Evaluation

This kind of evaluation method instead uses an AI agent to measure the quality of the generated content. In a platform game, where the goal might be to create a level with a certain difficulty rate, it can be complicated for a designer to determine what is difficult or not. In that case, measuring how well a bot performs would be a better way to evaluate the level.

Interactive Evaluation

In interactive evaluation methods, the reaction from one or several players is used to determine the quality. A simple way would be to ask the players directly about what they think, but another approach could be to measure the players' reactions during a game session.

Constraint Handling

In evolutionary computing, there exist several approaches of handling constraints. Kimbrough et al. [19] invented a method named *feasible-infeasible two-population* (FI-2Pop) that explored the search space by dividing the individuals into two populations: one feasible and one infeasible. The infeasible population does not fulfill the constraints, but is kept nonetheless to prevent discarding valuable information. While the evaluation method for the feasible population remains the same, the infeasible population is instead evaluated by measuring how much they have exceeded the constraints. Another method, invented by Deb [20], also keeps the infeasible solutions, but tries to discard them when there is a better choice. This is done during the selection phase in the GA by following two rules: always choose the feasible parent if possible, otherwise choose the least bad individual.

A third method [21] proposes a penalty function, effectively adding a large number to the individual's fitness value.

2.1.4 Multi-objective Evaluation

On top of all these choices there is also a question of how many objective (or fitness) functions to use. Up to this point we have only considered assigning one fitness measure to each individual, but an optimization problem in the real world generally consists of more than one function. In many cases it is impossible to satisfy all the objectives at once. When buying a car, for example, it would probably be impossible to optimize for both price and speed.

Problems with more than one objective are called *multi-objective problems* (MOPs), and there are many different ways to handle them. One way to deal with the conflicting objectives is to introduce weights that determine how important a certain measure is. This can be expressed by

$$\min_{x \in X} \sum_{i=1}^{k} w_i f_i(x) \tag{2.2}$$

where w_i are the weights controlling the fitness values $f_i(x)$, and the parameter x is the current phenotype from the set X, which in turn represents all the phenotypes in the population. This approach is called linear scalarization and effectively reduces several measures into one fitness value, which is handled by the same single-objective search algorithm described in section 2.1.2.

One issue with this method is that the designer must define the weights before the search algorithm starts. Another issue is that linear scalarization could miss an opportunity to minimize the fitness value even further if the underlying optimization problem is non-linear.

There are several methods to address these issues, where one alternative is to use a so-called *a posteriori* method. This method calculates several solutions between the objectives' different minima, and then a decision-maker can choose the desired solution. The solutions are commonly presented in a Pareto front that visualizes the trade-offs between the objectives. In a two-objective optimization problem where the ambition is to minimize the functions

$$\begin{cases} f_1(x) = x^2 \\ f_2(x) = (x-2)^2 \end{cases} \tag{2.3}$$

the Pareto-optimal front resembles a second-degree curve, see figure 2.4.

The domain of the solution space would here be between the two minima: $0 \le x \le 2$. In order to use this method, an algorithm that is able to handle MOPs must be used. One of the most used algorithms of the so-called *multi-objective evolutionary algorithms* (MOEAs) is *non-dominated sorting genetic algorithm II* (NSGA-II), invented by Deb et al. [22] (see appendix A for a detailed description of NSGA-II).

2.1.5 Online versus Offline

SBPCG works in both online and offline implementations. In the offline version, the content is generated before the game starts and relies on input coming from the designer. This

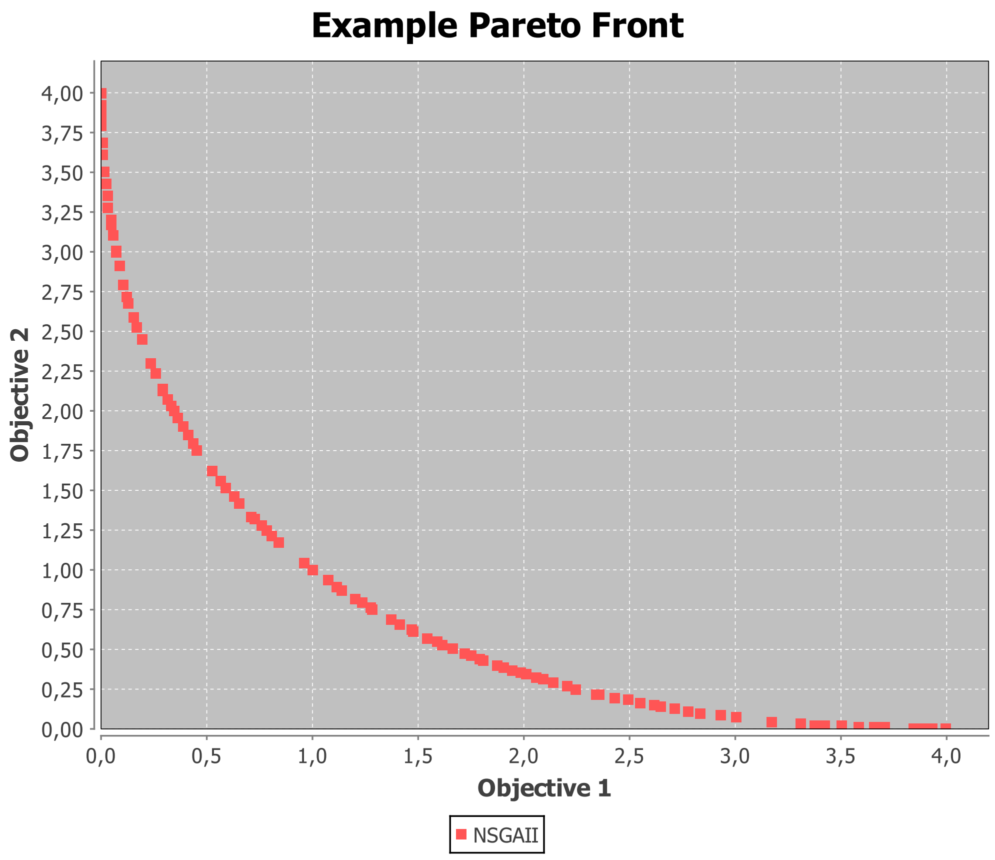


Figure 2.4: This graph shows an optimal Pareto front when minimizing equation 2.3. It is generated using the NSGA-II with the help of the library `MOEAFRAMEWORK` [23]. All of the dots are equal solutions with trade-offs between the two objectives.

is the method we are going to use in this project. Online generation, on the other hand, generates the content during gameplay and therefore puts a stricter speed requirement on the algorithm. The advantage of online generation is being able to adapt the game to the player, using some sort of DDA technique [24].

Chapter 3
Approach

In this chapter we present and motivate the choices made when using and applying SBPCG on this problem.

3.1 Method

To be able to answer the questions presented in section 1.2, we will construct a computer program and then evaluate it using bottom-up methods. This means we will evaluate the program using incoming data (in this case data from a user study) instead of using top-down methods which evaluate the results by decomposing and interpreting the output from the algorithm.

3.1.1 Program

The program should consist of an implementation of the search-based algorithm and must include the three parts mentioned in section 2.1. The initial structure of the program is prototyped in Java and later ported to C# when finalizing and merging it with the game.

3.1.2 Project Evaluation

To be able to evaluate the research questions given in section 1.2, there must be a way to measure the quality of the two claims made.

... fulfills the designer's desired mission difficulty and intensity: To succeed at measuring these features, we want to find correlation between the input parameters – provided by the designer – and the output parameters. The data will be collected from a set of test subjects that will play through a version of the game and then rate the waves' difficulties and intensities. This data will later be compared to the designer input data.

... is inseparable to a handmade level: This quality is, in similarity with the difficulty and intensity, hard to measure using a top-down approach, and we will therefore use the same bottom-up evaluation method once again. The comparison in this case is not input-output oriented; instead, the test subjects will play through one handmade and several computer-generated waves and then try to classify them. The computer-generated waves will be divided into two different types: one that tries to mimic a human-made wave and one that does not. The goal is to find significant evidence that the players cannot determine who or what created the content, and further to find data that suggest mimicking the human designer works.

Bottom-up

The test group must play through ten different waves: one designer-made and nine computer-generated, where all the waves should have a set difficulty and a set intensity. Four of the nine computer-generated waves will try to mimic a human designer. After each wave, the test group is required to answer three questions:

- *On a scale from 1 to 10, how difficult was this wave?*
- *On a scale from 1 to 10, how intense was this wave?*
- *Do you think this wave was made by a human or constructed by an algorithm?*

3.2 Implementation

In this section we present and motivate our different choices of the different parts in the SBPCG implementation. First off, we chose the method SBPCG due to its versatile nature and because of its relevance in the academic PCG research the recent years [3, Chapter 2].

3.2.1 Content Representation

Before we describe the content in numbers, we present how the game level looks, shown in figure 3.1. Every spawn point is responsible for generating a number of enemies of different enemy types, which only differ in speed, health, and attack power. To make the player aware of the presence of the different types, they are rendered in separate colors.

Although a semi-direct representation of the data is preferred, as described in section 2.1.1, we decided to use a fully direct representation of the data. The main reason is that the data being generated is already in a very compressed state; it is not possible to reduce the amount of data it takes to describe the spawn point characteristics without losing locality (earlier described in section 2.1.1). The only difference between the genotype and the phenotype is therefore how the data is stored. The genotype is a long vector of integers:

$$\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}, x_n], x_i \in [0, K] \tag{3.1}$$

where K is the maximum number of enemies of one type for every spawn point and n = `numberSpawnPoints` · `numberEnemyTypes`. The phenotype, on the other

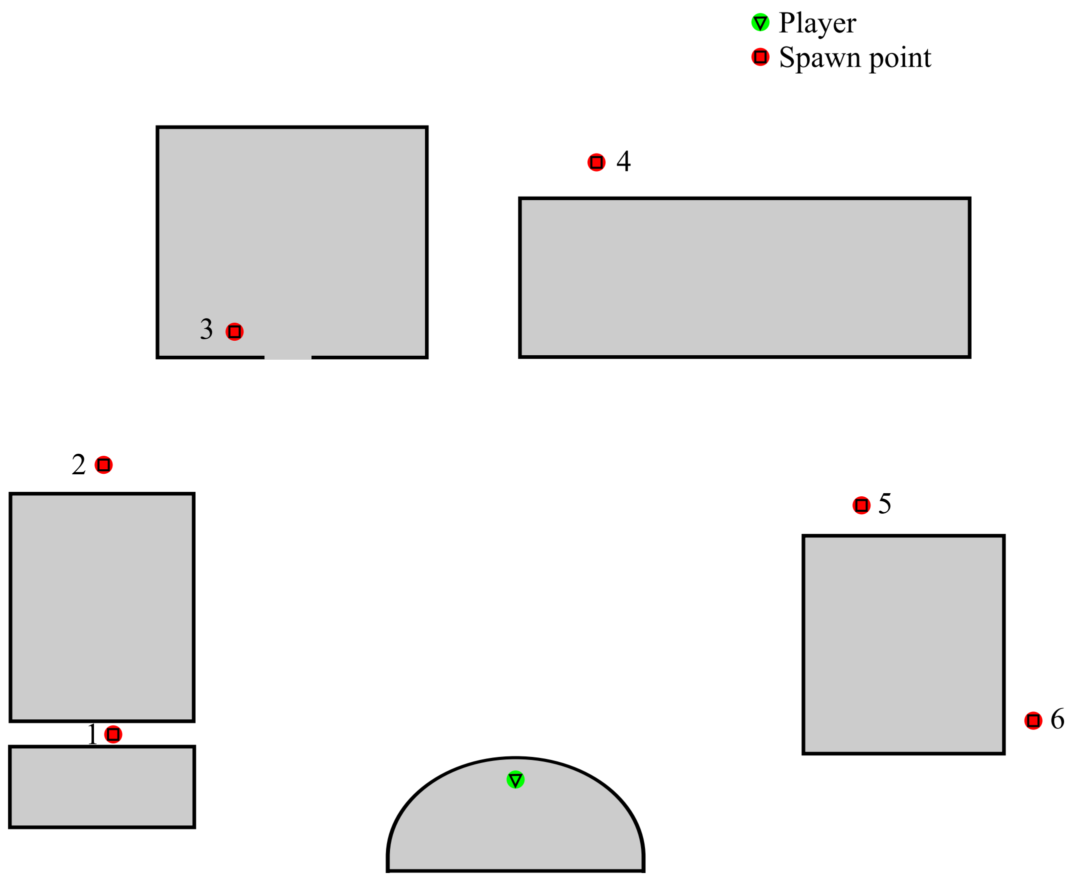


Figure 3.1: This is a sketch of the game level used in the project. The gray boxes represent buildings and the half-moon shaped area represents an elevated platform which acts as the fortification.

hand, better visualizes the distribution of the enemies across the spawn points, and contains the same data as the genotype, put in a matrix. This matrix consists of M rows representing the enemy archetypes and N columns representing the spawn points. Since all the parts of the spawn point definition involve discrete numbers, we chose to use an integer representation of the data. A phenotype with two types and two spawn points is visualized in table 3.1. In this small example, we see that the first spawn point will release

Table 3.1: Example phenotype

	SP 1	SP 2
Type 1	2	3
Type 2	0	1

two enemies of type 1 and zero enemies of type 2 while the second spawn point will release three enemies of type 1 and one enemy of type 2. The corresponding genotype in this example would be the vector

$$\mathbf{x} = \begin{bmatrix} 2 & 0 & 3 & 1 \end{bmatrix} \tag{3.2}$$

Most of the parameters controlling the algorithm setup are adjustable. In the user test survey, however, a setup of six spawn points and six enemy archetypes is used, and the parameter controlling the maximum amount of each type for every spawn point is set to $K = 10$. The first genotype is assembled by random numbers in the range $[0, K]$ and could result in the following phenotype:

$$\begin{bmatrix} 2 & 9 & 6 & 4 & 9 & 9 \\ 8 & 7 & 4 & 9 & 7 & 6 \\ 7 & 5 & 1 & 8 & 2 & 3 \\ 1 & 4 & 8 & 4 & 0 & 7 \\ 7 & 1 & 7 & 5 & 8 & 8 \\ 0 & 4 & 8 & 7 & 9 & 1 \end{bmatrix} \tag{3.3}$$

3.2.2 Search Algorithm

We decided to use a genetic algorithm as search engine, mostly because of its compatibility with integer representations (as long as the recombination uses the same set of operators as for binary representations [18, Chapter 4]). We set the population size to $\mu = 40$ after some experimentation. This value does not seem to alter the convergence behavior of the GA significantly.

Genetic Algorithm

As the content is represented using integers, we had to change the Simple GA on some points.

Recombination: As described earlier, the recombination still works using a 1-Point crossover with the same operators as in the binary case. The only difference is that the resulting genotype will inherit the integers from its parents.

Mutation: We had to change the mutation operator a bit more drastically than in the recombination case in order for it to work with integers. Two alternatives, presented in [18, Chapter 4.3.1]: *random resetting* and *creep mutation*, handle the integer representation in different ways. The former method is most suitable when there is no clear relation between the numbers, but as the integers in our case represent the actual number of enemies, we do have a clear relation between the numbers, encouraging us to choose the latter method. The idea behind creep mutation is to add a small number to each gene with a probability p, drawn from a distribution symmetric about zero. We chose to use a uniform distribution generating one number among $\{-1, 0, 1\}$, and later adding that number to a gene with the probability $p = 0.15$.

Selection: Not because of the integer representation, but due to the roulette wheel selection not giving good sample of the population [18, p. 84], we instead chose to use another method called *tournament selection*, which selects k individuals randomly and then picks the fittest one. In our implementation we use $k = 2$.

Regarding the survival selection, we kept the generational method, completely replacing the parent generation with its offspring.

Due to all these changes, a summary of the Altered GA version used in this project is described in table 3.2.

Table 3.2: Altered GA

Representation	Integer vector
Recombination	1-Point crossover
Mutation	Creep mutation
Parent selection	Tournament
Survival selection	Generational

3.2.3 Evaluation Method

Presented with the phenotype matrix, we want to evaluate the two measures proposed in section 1.2: how well the solution follows the designer's wanted difficulty and intensity, and the ability to mimic human behavior.

Given the problem is formulated using several objectives, we are committed to use some sort of multi-objective optimization. If the measures had been independent of each other, simply adding all the fitness values together would suffice. As the final application is supposed to function as a non-iterative black box, using an MOEA is not preferred in our case. Theoretically, we could let some algorithm be the decision-maker after the Pareto front has been created, but that would only be advantageous if the front is discontinuous, visualized in figure 3.2. If the weights had been ordered in a way that would put the final solution close to the discontinuity in figure 3.2, it would be hard for a designer to choose the right solution without looking at the Pareto front. A solution on the right side of the discontinuity would have lower and therefore better values in both objectives, but this would be impossible to know if only working with a priori weights. The final optimization problem in this project, however, turned out to be continuous, shown in figure 4.1 in section 4.1, and an MOEA was not needed. Even so, as these measures still are in conflict with each other to some extent, we utilize an a priori linear scalarization, which becomes an additional input for the designer.

Considering the impossibility in creating an unplayable wave just by generating the amount of enemies of different types, there should not be any need for constraints. However, as some characteristics of a wave are more important than others, we chose to pack all the measurements addressing the human behavior into one constraint, while letting the intensity and difficulty be normal fitness values. This choice also suits the purpose of the algorithm well; a designer would most probably prefer losing some control over the difficulty and intensity in favor of generating a well designed solution.

After experimenting with different ways to implement this constraint, we found that the best solution was to enforce the constraint using a penalty function, as described in the fourth subsection of 2.1.3. This means we treated the human factor as a fitness measure with higher, non-normalized numbers. The high correlation between the human factor versus the difficulty and intensity measures is probably the reason this method performed better than the other methods. By not separating the constraint and the fitness functions, the search algorithm receives more information about good versus bad individuals, making the convergence process quicker. In an end version of the application the goal might however be to generate human-like individuals, with less emphasis on strictly following the designer input. In those cases, the second option incorporating constraint handling in the tournament selection would be preferred.

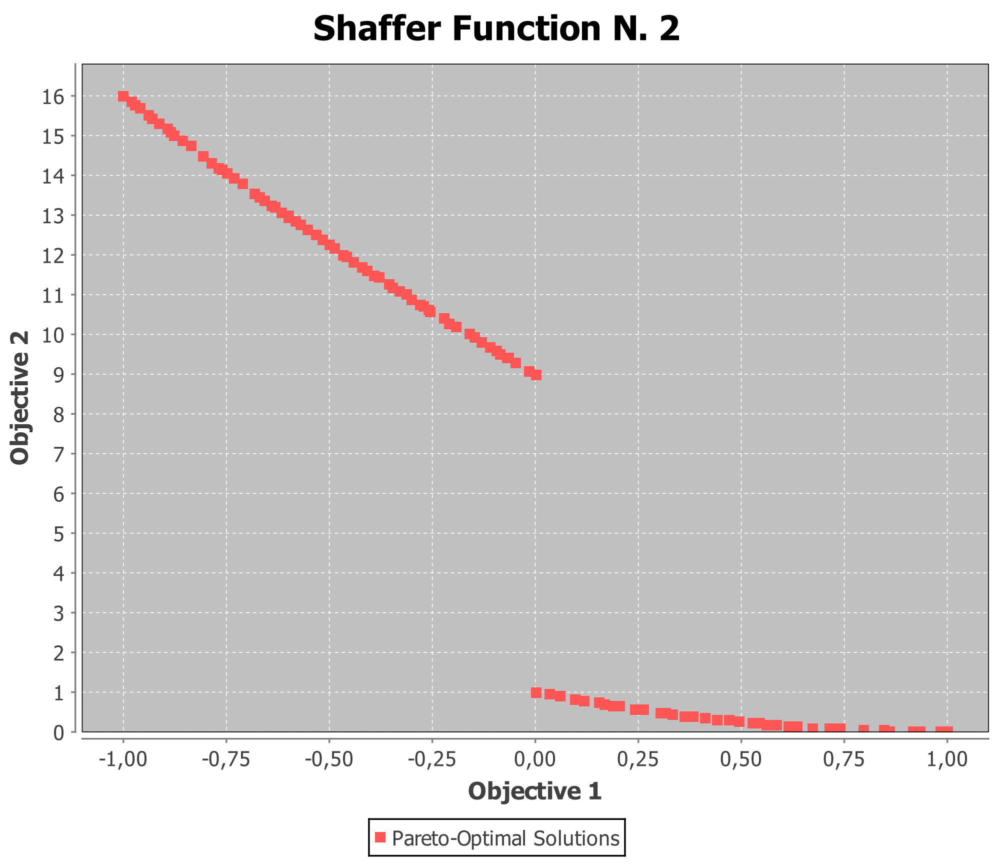


Figure 3.2: This is an example of a discontinuous Pareto front. The test function is called Shaffer N. 2 and was developed by Schaffer [25].

Using the problem formulation in section 2.1, the final fitness measure to minimize is:

$$f = w_d f_d + w_i f_i \tag{3.4}$$

subject to the equality constraint:

$$h = c_h \tag{3.5}$$

The indices here are d: difficulty, i: intensity, and h: human factor. The weights w_d and w_i control how important the difficulty and intensity are, and are kept normalized: $w_d + w_i = 1$.

The following part is a breakdown of the three factors f_d, f_i, and c_h, which together form all the rules of what makes a good or bad individual. Let capital `D`, `I` and `H` denote difficulty, intensity, and human factor accordingly.

Difficulty: This fitness measure is a number $f_d \in [0, 1]$ and determines how well the difficulty of an individual's phenotype matches the designer's desired difficulty level. This is described by equation 3.6:

$$f_d = |\texttt{preferredD} - \texttt{actualD}| \tag{3.6}$$

Both `preferredD` and `actualD` range from very easy (0) to very difficult (1). While the number `preferredD` is just a number chosen by the designer, `actualD` is a number that must describe how difficult a wave is. This is done using a theory-driven direct evaluation method, simply assuming what makes a wave difficult or easy. The relation controlling this is given in equation 3.7.

$$\texttt{actualD} = \sum \sum P_{i,j} d_j \tag{3.7}$$

where **P** denotes the phenotype matrix and **d** is a vector that contains a difficulty value for each enemy type. The total difficulty is thus a sum of all the enemies multiplied by their difficulty value. This method works well if the difficulty vector is correctly assessed. We calculate the difficulty of the enemy types by multiplying their health, damage, and speed together. These values are found in appendix D. We use the normalized values from D.2 to assert `actualD` is also normalized.

Intensity: The corresponding equation describing how well the intensity matches the preferred intensity is:

$$f_i = |\texttt{preferredI} - \texttt{actualI}| \tag{3.8}$$

The quantities in equation 3.8 have the same intervals as in the difficulty equation (3.6), and the function describing the estimated intensity is given in equation 3.9.

$$\texttt{actualI} = \sum \sum P_{i,j} s_j \tag{3.9}$$

Here, **s** is instead a vector of the speeds of the enemy types. An intense individual would thus contain many high-speed enemies, while a low-intense individual would only consist of a few slow enemies. The intensity measure is similar to the difficulty measure – both of the measures increase with the number of enemies – which is why we anticipate some correlation between them.

Human factor: The number that controls how well a wave imitates a human designer does not require any input, and always strives to be zero. If the algorithm generates an individual that trespasses any of the rules considered to describe human behavior, a number is added to this measure depending on how much the rule has been violated. The four following rules are based on a few assumptions of what characterizes a human-made wave:

- One spawn point must not contain more than two different types.
- The total number of enemies per wave must be in the interval [5, 50].
- Slow enemies should not spawn too far away from the player.
- The wave should not contain more than four different types.

The implementation of the first rule adds a number, c_1, equal to the number of types for every spawn point that contains more than two different types. This is formally described in equation 3.10, where i denotes the index of the current spawn point.

$$c_1 = \sum_i max(0, \texttt{nbrTypes[i]} - 2) \tag{3.10}$$

The second rule adds a number equal to the amount of enemies above or below the limit. An individual with 2 enemies would, for example, get a penalty value equal to $c_2 = 3$. This is formulated in equation 3.11 when there are too few enemies and 3.12 when there are too many enemies.

$$c_2 = max(0, 5 - \texttt{totNbrEnemies}) \tag{3.11}$$

$$c_2 = max(0, \texttt{totNbrEnemies} - 50) \tag{3.12}$$

In the third constraint, only the two slowest types are being kept off the two most remote spawn points. If this rule is violated, the number of incorrectly generated enemies is added to c_3. Equation 3.13 is only active if enemies from type 1 or type 5 are present at spawn point 3 or 4.

$$c_3 = \texttt{nbrOfEnemies} \tag{3.13}$$

Similarly to the two first constraints, this constraint adds a number if there are more than four different types.

$$c_4 = max(0, \texttt{totNbrTypes} - 4) \tag{3.14}$$

The final constraint value is then the addition of these four values:

$$c_h = c_1 + c_2 + c_3 + c_4 \tag{3.15}$$

which together with equation 3.5 finalizes the constraint expression.

At this step in our implementation, we do a division between the user test version of the generator and the final product. In the final product the designer's task is to construct a mission, and this is achieved by modifying two graphs controlling the difficulty and intensity levels. An example of one of these graphs is found in figure 3.3. In addition, the designer must define the weights using two interdependent sliders, shown in figure 3.4.

In the user test, on the contrary, we do not allow any modification of the weights. In other words the difficulty and intensity levels are kept equally important in the testing phase. The reason for this is that we want be able to measure how well the algorithm performs when it is trying to accommodate for both measures at the same time.

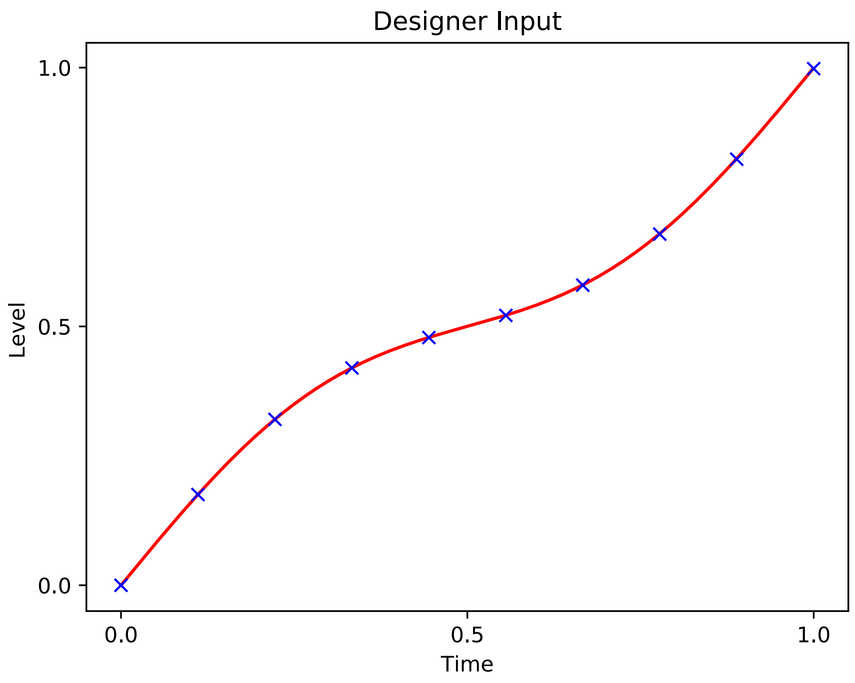


Figure 3.3: This figure shows the graphical layout of how the designer controls the algorithm's behavior. The red curve is interactive and sampled at the blue crosses. The number of samples is determined by the amount of waves for the current mission and are distributed equidistantly.

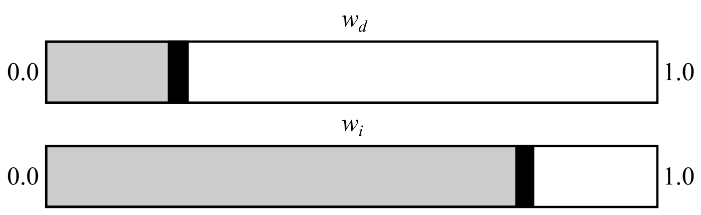


Figure 3.4: These two sliders control the weights w_d and w_i. Moving one slider will automatically move the other one, making sure the weights are kept normalized.

3.3 User Study Setup

The user test is designed to last for approximately fifteen minutes. First, the test group is introduced to two waves with known intensity and difficulty. The purpose of these waves is to familiarize the player with the game and to set reference levels regarding the difficulty and intensity. After the two introductory waves, the test group must play through another eight waves with different levels of difficulty and intensity. Some of the waves are generated using the constraint functions and thus try to simulate human design, while the other waves are only using the fitness measures. In addition, there is one real human-made wave that works as a reference point. The test group must not be aware of anything but the intensity and difficulty of the two first levels. The entire testing phase is shown in table 3.3.

Table 3.3: Input values to the 10-wave mission used in the survey

Wave Index	Difficulty	Intensity	Human-made	Constraint Functions
1	1.0	1.0	Yes	-
2	10.0	10.0	No	Yes
3	5.4	3.7	No	No
4	2.3	3.3	Yes	-
5	2.9	1.8	No	No
6	4.0	3.7	No	Yes
7	3.2	5.4	No	No
8	3.3	6.8	No	Yes
9	7.5	7.8	No	Yes
10	4.5	2.8	No	No

Along with a copy of the game, the test group is provided with a questionnaire concerning the eight test waves. After every wave, they have to answer the three questions presented in section 3.1.2. The test group is restricted to answering the questions using integers between 1 and 10 (and one yes/no question), which might seem rather peculiar considering the true solutions use decimal values. There are two reasons for this: we want to give the designer full control but still keep the common style of survey response scales. In addition, the main goal is to be able to control the difference in difficulty and intensity, not acquiring the exact values. The full questionnaire can be found in appendix B.

As described in section 2.1.3, we chose the human factor to be implemented together with the fitness measure. This gives us the two following evaluation expressions:

$$f_1 = f_d + f_i + c_h \tag{3.16}$$

$$f_2 = f_d + f_i \tag{3.17}$$

Here, f_1 is used on waves 2, 6, 8, and 9 while f_2 is used on waves 3, 5, 7, and 10.

Chapter 4
Evaluation

This chapter contains the outcome of the project, starting with a presentation of the results, followed by a discussion, and finally a section about further work in the area of SBPCG.

4.1 Results

The results are divided into three parts: first plots concerning the use of an MOEA, then the results from the algorithm's convergence behavior, and at last the outcome of the user study.

4.1.1 Pareto Front

The arguments presented in section 3.2.3 about not using an MOEA for this project are only valid if the optimal Pareto front is continuous. In figure 4.1, the optimal Pareto front between the two objectives f_d and f_i is shown.

4.1.2 Convergence Behavior

The convergence behavior was investigated mainly in order to obtain a stopping criterion for the search algorithm. In figures 4.2a, 4.2b and 4.2c the convergence of the total fitness ($f = f_d + f_i$) for the entire population is shown. The parameters changed are the desired difficulty and intensity values.

4.1.3 User Study

In this section we present plots summarizing the answers collected from the questionnaire. All the answers can be found in appendix C. In figures 4.3 and 4.4, the perceived and true

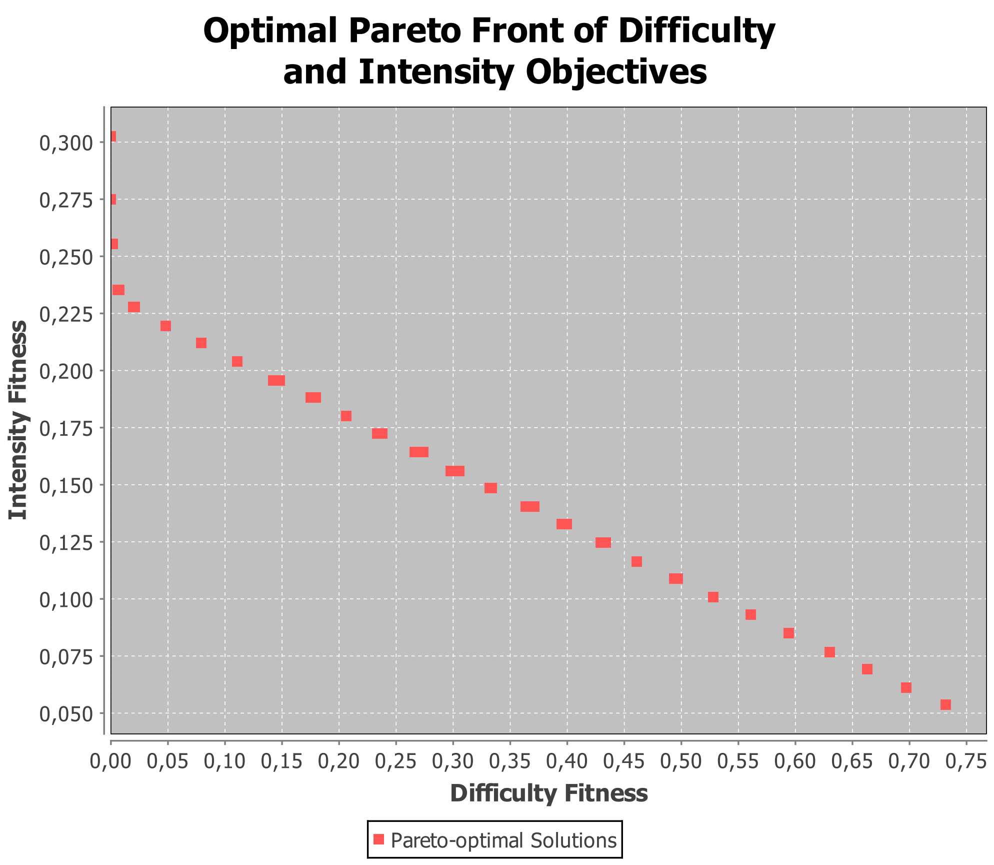
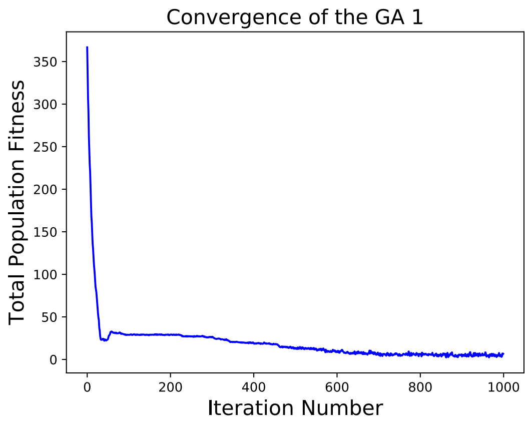


Figure 4.1: This graph shows the optimal Pareto front of the final version of the generator between the conflicting objectives f_d and f_i.

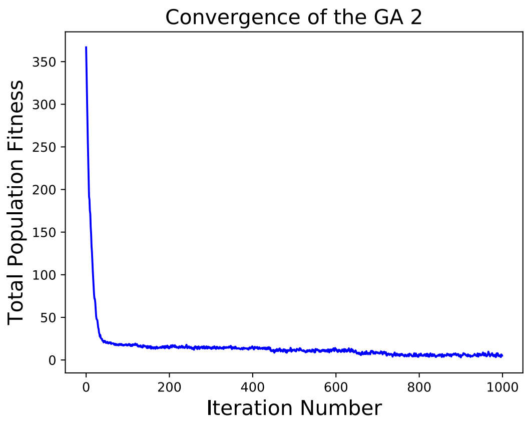
difficulty and intensity values are shown.

In an attempt to accommodate for the offset between the perceived and true values, the plots in figures 4.5 and 4.6 show a re-scaling of all the values into the range [1, 10], done separately for the true and perceived values. Further, these plots highlight the change of the values by drawing lines between the points. This might seem a bit misleading since the values are discrete, but they are there to easier see if the perceived values follow the true values to some extent.

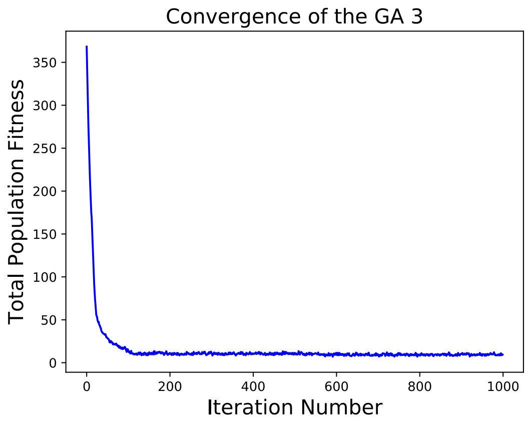
In figure 4.7, data from of the test group's ability to determine who created each wave is presented. The true values are found in table 3.3 in section 3.3.



(a) `preferredD` = 0.2
`preferredI` = 0.8



(b) `preferredD` = 0.5
`preferredI` = 0.5



(c) `preferredD` = 0.8
`preferredI` = 0.2

Figure 4.2: Convergence of the total fitness values

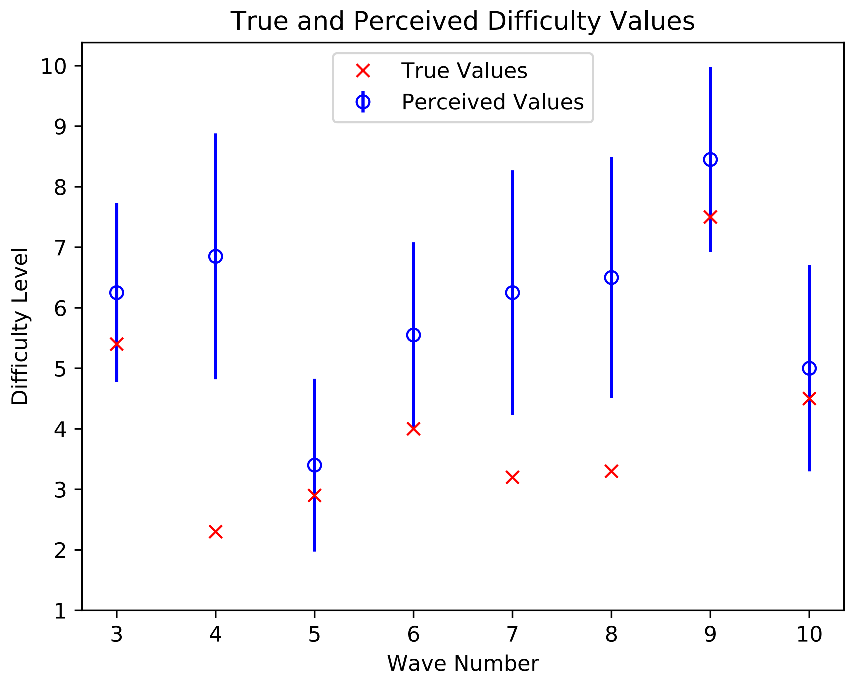


Figure 4.3: This plot visualizes the true and perceived difficulty values for all the test waves. The perceived difficulty values are average values visualized with one standard deviation.

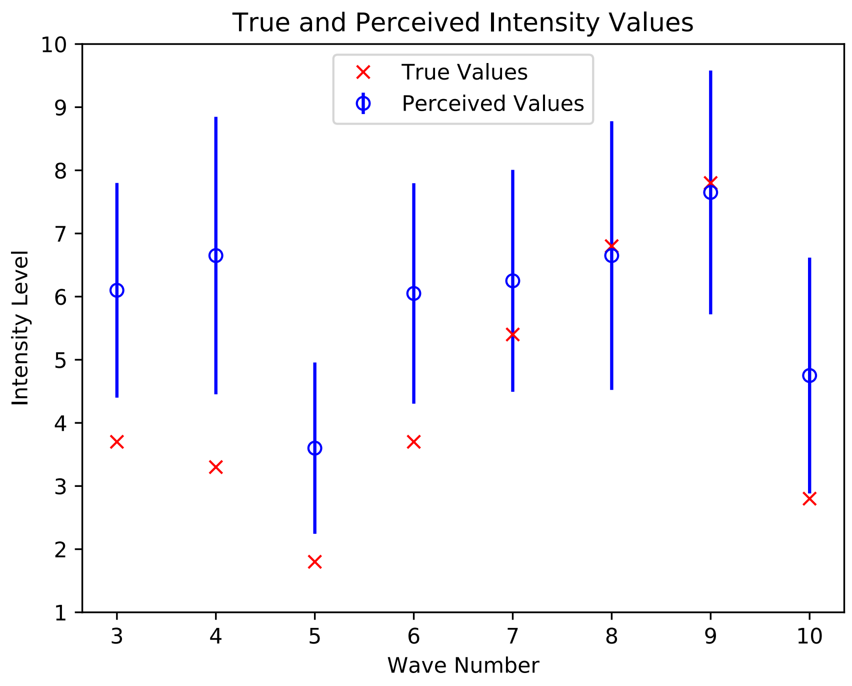


Figure 4.4: This plot visualizes the true and perceived intensity values for all the test waves. The perceived intensity values are average values visualized with one standard deviation.

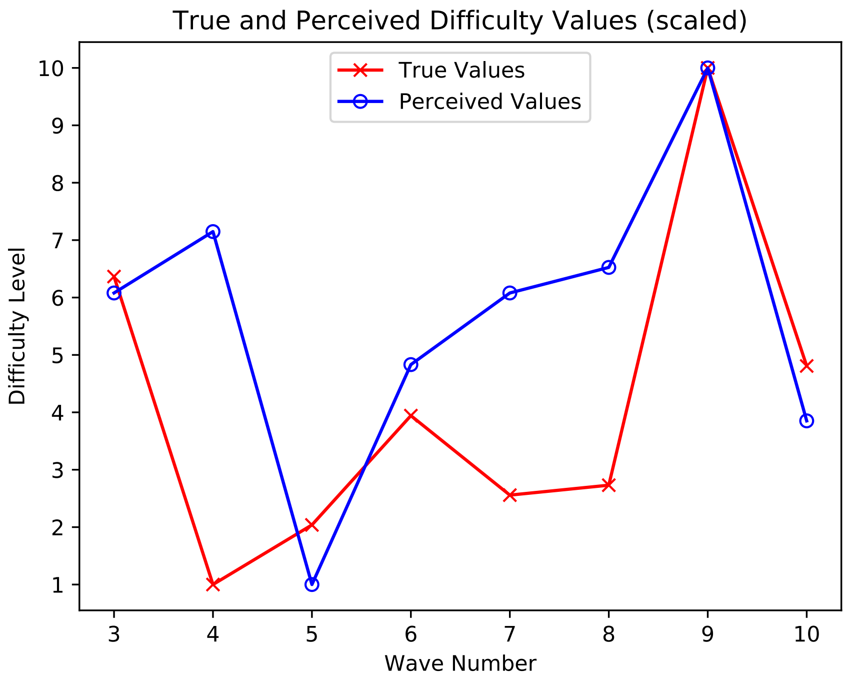


Figure 4.5: A normalized version of the true and perceived difficulty values for all the test waves

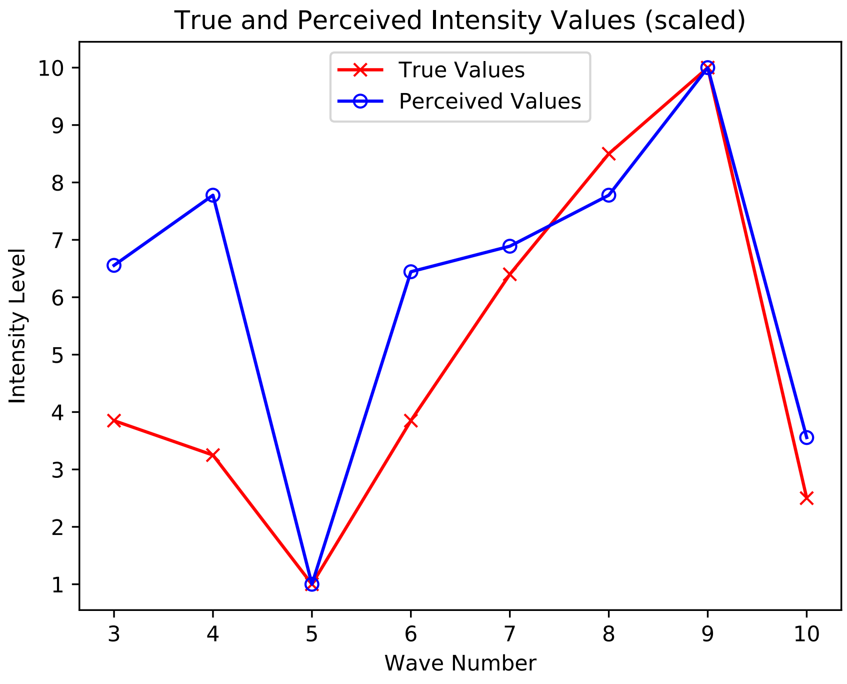


Figure 4.6: A normalized version of the true and perceived intensity values for all the test waves

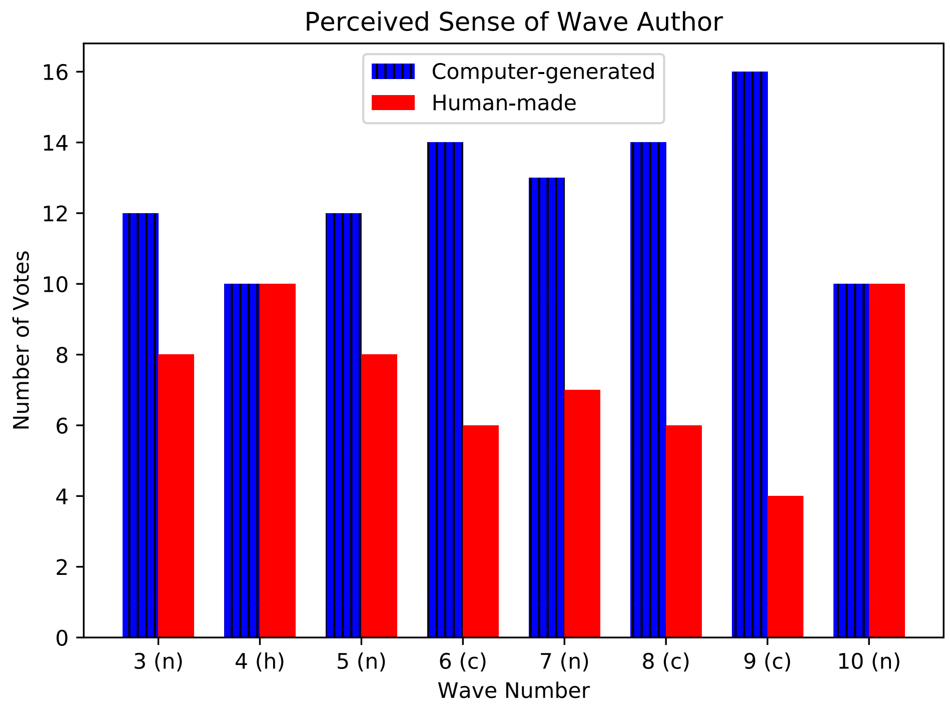


Figure 4.7: This figure shows the score of the test group's perceived sense of whether the waves were constructed by a human or by a computer algorithm. The letters in the parentheses represent the actual wave author: (h) if made by a human, (c) if trying to imitate a human using constraint functions, and (n) if generated without any constraint functions.

4.2 Discussion

In this section we discuss the results in the same order as they were presented.

4.2.1 Pareto Front

Even if there seems to be a discontinuity when the difficulty fitness approaches zero in figure 4.1, this is not considered a problem. The reason for this is that the discontinuity would only appear when the intensity weight is zero, and in that case the designer would not care. If the discontinuity would be in the middle, however, it would probably be a good idea to involve an MOEA in the search process.

4.2.2 Convergence Behavior

In a one-objective GA, the most straight-forward approach to stop the search would probably be to use a threshold value, representing the boundary between a satisfactory and a non-satisfactory individual or population. If the search algorithm cannot find a solution that fulfills the requirements, the search should then terminate when a certain number of iterations is reached.

In our case, however, we have an MOP, and setting a stopping criterion for only one objective would not be ideal. Generally we do not want to stop searching for the solution if only one fitness value is satisfied; it might be possible to improve the other objective while still keeping the already satisfied value. Another approach would be to add the two fitness values together and then use a relevant, possibly larger threshold value. This method also has a drawback because it would be difficult to control which objective is performing better.

Both of the above approaches have the same problem of risking a too early stop as the search algorithm might find an even better solution. To address this issue, we tried to implement a way of determining when the search is no longer improving the fitness values, and then stop the search at that point. This was however rather challenging due to noise from the mutations.

We came to the conclusion, after several tests, that using a set number of iterations was, after all, the best method for this project. In order to not waste a lot of computing time, we graphically studied the point where the algorithm stops improving. This is shown in figures 4.2a, 4.2b and 4.2c. This point is very quickly reached in figure 4.2c, where the preferred difficulty level is significantly larger than the intensity level. The slowest convergence to this point is seen in 4.2a, and in this case the algorithm reaches the point after around 800 iterations. Conclusively, it should be safe to terminate the search after 800 iterations in any case. This is important when using the SBPCG in online fashion, during gameplay.

4.2.3 User Study

As is evident from figures 4.3 and 4.4, the perceived values and the true values are considerably far apart. Ideally, the perceived values should be closer to the true values and have shorter corresponding vertical lines to indicate a lower spread. There are many potential reasons for these results.

Firstly, one problem seems to be a very large offset between the perceived and the true values. As long as this offset is consistent, this is not a problem. The final goal is to be able to control the difficulty and intensity values to match the user experience of the same measurements, and a constant offset could easily be corrected by redefining the minimum and maximum difficulty and intensity parameters.

Another issue is that the difficulty and intensity values resemble each other to a great extent, even when they should not. Although the users had to play through two reference waves, marking the extremes of the algorithms, they only covered the two parameter settings: $\{d, i\} = \{1, 1\}$ and $\{d, i\} = \{10, 10\}$. A better introduction could probably be achieved by adding two additional waves showing different combinations of the difficulty and intensity values. In that case, the perceived difficulty and intensity values would hopefully not coincide as much.

The third factor we must consider is that the players improve their own skills while taking the test, possibly making the perceived difficulty values in relation to the actual values higher at the beginning of the test. Yet, there is no indication that the distance between the perceived and actual values decreases throughout the test. In any case it would have been better to let the test subjects play the game for a longer while before starting the test. In addition, the level order could have been randomized to assure that everyone would have followed a different learning path.

One thing that also has a large impact on the perceived difficulty and intensity levels is how accurate our assumptions about these values were. We assumed the difficulty could be expressed as a product of the enemy health, speed, and attack power, but we did not consider weighing these factors any differently. Perhaps attack power is twice as relevant compared to the other factors. If we experiment with these factors we might achieve more accurate results, but it is probably better to not use assumptions at all. We will discuss other methods to evaluate the content in section 4.3.

In conclusion, the most important result is that the perceived values at least change in the same direction when altering the input values. By compensating for the offset difference, the results shown in figures 4.5 and 4.6 look promising. The intensity values seem to follow more accurately than the difficulty values, which is probably due to the intensity being constant disregarding the skill of the players. The accuracy of the difficulty values will suffer more the larger the skill variation among the test subjects is.

The results concerning whether the test group could distinguish between human-made and computer-generated waves are shown in figure 4.7. The first noticeable tendency among the test subjects is to vote for the computer as the wave author. As the majority of the waves were in fact generated using the computer algorithm, this might be considered a good thing. Following the goal stated in section 1.2, this is however not true; what we would like to see is instead similar results between the waves that imitates human behavior and the human-designed wave (wave 4). There is no correlation here whatsoever.

If the waves that were constructed using the constraint functions had received more votes for human-made than the other waves, there would at least have been some indication that the assumptions about what makes a wave look human-made were correct. There is, however, no indication of this either. On the contrary, these waves (6, 8, 9) are the waves receiving the largest amount of votes for computer-generated, which is the opposite of the presumed results. More extensive tests would have to be done to assert if this is a coincidence or not.

If there is no correlation between the usage of constraint functions and the wave author, why is there a difference at all between the different waves? Comparing the perceived intensity values in figure 4.4 to the bars representing the computer-generated votes in figure 4.7, there seems to be a slight correlation between increasing intensity and a tendency to vote for computer-generated. It is possible that an increasing amount of enemies makes the wave look disordered, no matter how well designed it is.

In summary, it is difficult to make correct assumptions about what makes a computer game look human designed. Human intuition about design uses a large amount of parameters, and many are more or less impossible to conceptualize. Our conclusion is that a better approach to make an SBPCG implementation to mimic human behavior is to incorporate humans in the process. We will discuss this more in section 4.3 about further work.

The difficulty and intensity measures are in theory much easier to determine using a direct approach, and from the results we see that it works in practice to some extent. Even if the variance among the players is large, which generally would indicate more testing of this direct approach is needed, the results of our survey revealed that the fitness functions were not ideal. Thoughts about how to improve the fitness functions are also found in section 4.3.

4.2.4 General Discussion

Although the outline for this thesis is similar to the work of Classon and Andersson [10], there are three major differences: they had a single objective problem, they used an interactive evaluation method, and they generated visible gameplay content. In this section we will further discuss different implementations by answering the questions formulated in section 1.4. The first question concerns which types of content are suitable to generate. In this project we have shown one example of non-visible content, and even if the most documented variant of content is visible, this works just as well. In this regard, SBPCG is very versatile. Problems did however arise when we tried to add more fitness functions to the algorithm. If, for example, the task would be to both define the content and the position of the spawn points, our approach would not be as effective. As described in section 1.1, the ultimate PCG tool would be able to do this and much more. When developing small scale PCG, SBPCG works more than adequately. If the generator must be able to handle more complex and comprehensive content, the best way to improve it is to represent the content more intelligently.

This leads us to the second question: how is game content best represented? We used a direct mapping between the genotype and the phenotype, and this worked very well for our problem. Adding the position of the spawn points to the requirements, as discussed earlier, would force our algorithm to use a semi-direct mapping instead. We believe the easiest way to discover the best representation is to experiment with different alternatives. Also, plotting the convergence behavior is a good way to determine how well the algorithm performs. If a less direct representation is not needed for the algorithm to converge, as in our case, the only reason to still implement it is to make the search process quicker.

The third question is about how to assess the quality and potential of content generators. We used a bottom-up approach in terms of a user study. This method is possibly the best way to evaluate the capability and accuracy of the generator, but needs a lot of resources. We recommend using a bottom-up approach whenever possible, mainly because all the content generated is going to be used and judged by the players. As discussed in [3, Chapter 12], the preferred way is to either use few dedicated players that play through the game and answer a set of in-depth questions, or to collect data from many players to be used for machine-learning the evaluation functions.

The last question is about how the interaction between the game designers and the algorithm should be arranged in a useful way. The answer to this question is specific to each implementation, but the main goal is to make the process of designing a game faster. The vegetation generator SpeedTree, mentioned in section 1.1, generates all kinds of vegetation with just a few parameter settings. We aimed for a similar setup, but as the original workflow of creating a mission mainly dealt with experimenting with numbers, we wanted to make the control of the generator even simpler. The interactive graph, shown in figure 3.3, was the result of this aim. With the help of this graph, the designer is able to sketch the difficulty and intensity throughout the entire mission. This is advantageous if they are striving for a flow-like mission design [26]. The idea of keeping the player in a so-called flow-channel is to alternate the mission difficulty up and down while still increasing it continuously. This keeps the players in a state between boredom and anxiety, making them play for longer sessions.

4.3 Further Work

In this section we describe potential ways of improving the current implementation. This is followed by a general outline of what can be developed further in the area of SBPCG.

We would like to theorize how to improve two aspects about the algorithm: the fitness and constraint functions. One way to increase the accuracy of the fitness functions is to continue the work of Classon and Andersson [10], running several iterations of the user test. This would work to some extent, given they are functioning reasonably well from the beginning. Another approach would be to collect data from the players, possibly during a beta version of the game. The players would either have to answer questions during gameplay, or an algorithm could collect data about how they perform. This data would then be used to reverse engineer fitness functions that more accurately coincide with the true difficulty and intensity levels. Both of these methods, however, rely heavily on information gathered from the players, which is often a time-consuming and tedious task. Without using any player input, the best approach would presumably be to use a simulation-based evaluation. By letting an agent analyze the levels, the entire process would become automated. The issue with this method is that the agent might behave differently compared to actual human beings. Nevertheless, a framework combining SBPCG with the latest advances in reinforcement learning would be extremely powerful.

When it comes to the constraint functions controlling the ability of the algorithm to mimic human designers, an agent would not be very helpful. Real human beings would have to be used in the process, either by continuously collecting data from them or by letting designers construct a large amount of human-made waves. In the latter method, we would have to construct constraint functions that give these handmade waves a high rating, though all this work would somewhat contradict the purpose of having an automatic algorithm from the beginning.

Conclusively, our recommendation for problems similar to this is to prefer using some kind of simulation-based evaluation when possible, and otherwise somehow collect data from players. We believe that SBPCG displays more advantages the closer the evaluation method is to the end product's purpose.

The last possible continuation of this project would be to implement an online version of it. In an online version, as mentioned in 2.1.5, using DDA to adapt the difficulty and intensity would be useful way to tailor the experience of the game to each individual player. One thing to be aware of, however, is to not make the algorithm's behavior visible to the players, as noted in [12]. If they notice the game changes depending on their playing style, they might try to exploit it.

Chapter 5
Conclusion

In this project we have applied SBPCG exclusively to define spawn points. The content was represented in two ways: one integer array being the genotype and one integer matrix being the phenotype. The search for a good solution was done by a modified version of a GA, and the fitness functions determining what makes a good solution were derived from three assumptions: what makes an enemy wave difficult, intense, and look human-made.

The results from the test were collected using a bottom-up approach, asking a test group of 20 persons questions about the waves' difficulty and intensity levels and one question about whether the waves were computer-generated or not. The study showed that the algorithm performed better at following the designer's desired difficulty and intensity levels than trying to mimic a human designer. Our conclusion is that this mainly depends on the nature of the different problems; it is much easier to estimate a difficulty value than to predict how a human being thinks and acts. In other words: humans are unpredictable, numbers are not.

Returning to the research question, we have shown one way to implement SBPCG. The program was not able to fulfill both requirements, but we did get much insight into how to further develop PCG and SBPCG algorithms. Our main recommendation is to steer away from direct evaluation methods toward either simulation-based evaluation using reinforcement learning or some kind of interactive process, closer to the players. After all, we are trying to develop games for the players.

Bibliography

[1] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2013.

[2] Chandranil Chakraborttii, Lucas N. Ferreira, and Jim Whitehead. Towards generative emotions in games based on cognitive modeling. In *Proceedings of the International Conference on the Foundations of Digital Games (FDG)*, FDG'17, 2017.

[3] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[4] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In *Applications of Evolutionary Computation*, pages 367–377, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] David Oranchak. Evolutionary algorithm for generation of entertaining shinro logic puzzles. In *Applications of Evolutionary Computation*, pages 181–190, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making racing fun through player modeling and track evolution, 2006.

[7] Julian Togelius, Mike Preuss, and Georgios N. Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM.

[8] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010*, pages 265–272, 2010.

[9] Alberto Uriarte and Santiago Ontañón. Psmage: Balanced map generation for starcraft. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013.

[10] Johan Classon and Viktor Andersson. Procedural generation of levels with controllable difficulty for a platform game using a genetic algorithm. Master's thesis, Linköping University, Human-Centered systems, 2016.

[11] Alexander Baldwin and Johan Holmberg. Mixed-initiative procedural generation of dungeons using game design patterns. Master's thesis, Malmö University, Department of Computer Science and Media Technology, 2017.

[12] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: A model for dynamic level generation. In *AIIDE*, 2010.

[13] Michael Booth. The AI Systems of Left 4 Dead. Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference, 2009.

[14] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey, 2011.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[16] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[17] Franz Rothlauf and David E. Goldberg. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, 2002.

[18] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[19] Steven Orla Kimbrough, Gary J. Koehler, Ming Lu, and David Harlan Wood. Introducing a Feasible-Infeasible Two- Population (FI-2Pop) Genetic Algorithm for Constrained Optimization: Distance Tracing and No Free Lunch. *European Journal of Operational Research*, 2005.

[20] Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2):311 – 338, 2000.

[21] Biruk G. Tessema and Gary G. Yen. A self adaptive penalty function based algorithm for constrained optimization. *2006 IEEE International Conference on Evolutionary Computation*, pages 246–253, 2006.

[22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[23] MOEA Framework, 2017. URL `http://moeaframework.org`.

[24] Bruno Baère Pederassi Lomba de Araujo and Bruno Feijó. Evaluating dynamic difficulty adaptivity in shoot'em up games. In *Proceedings of the XII Brazilian Symposium on Games and Digital Entertainment - SBGames 2013*, pages 229 – 238, São Paulo, Brazil, 2013.

[25] J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[26] Jesse Schell. *The Art of Game Design: A Book of Lenses*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[27] Wei Peng, Qingfu Zhang, and Hui Li. *Comparison between MOEA/D and NSGA-II on the Multi-Objective Travelling Salesman Problem*, pages 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[28] Ming-Der Yang, Yi-Ping Chen, Yu-Hao Lin, Yu-Feng Ho, and Ji-Yuan Lin. Multiobjective optimization using nondominated sorting genetic algorithm-ii for allocation of energy conservation and renewable energy facilities in a campus. *Energy and Buildings*, 122:120 – 130, 2016.

Appendices

Appendix A
NSGA-II

In this appendix, we describe the NSGA-II that handles MOPs in more detail.

A.1 Background

The NSGA-II was first mentioned by Deb et al. [22], and is a successor of the original *non-dominated sorting genetic algorithm* (NSGA). Both of these algorithms have the characteristic of generating several Pareto-optimal solutions in one single simulation. The Pareto-optimal solution is further described in section A.2. The problems concerning high computational complexity, lack of elitism, and the need for specifying a sharing parameter apparent in the NSGA is addressed in the NSGA-II version, which today is generally considered to be the most popular Pareto dominance MOEA [27].

A.2 Algorithm

A Pareto-optimal or a non-dominated solution is a solution that is better than all other solutions in at least one objective [28]. The NSGA-II tries to find all the non-dominated solutions, while avoiding the more dominated solutions. The result is a set of equal solutions with respect to all of the objectives. In figure A.1, a method of how to graphically distinguish between dominating and dominated solutions is shown. Note that this only applies to minimization problems using two objectives. The method when using three or more objectives is similar, however much harder to visualize.

The Pareto-optimal solutions can be found analytically by using algorithm 2 several times. The NSGA-II approach of finding the solutions is described in the following four steps:

```
1: Given a population, form a new population twice the size of the original population.
```

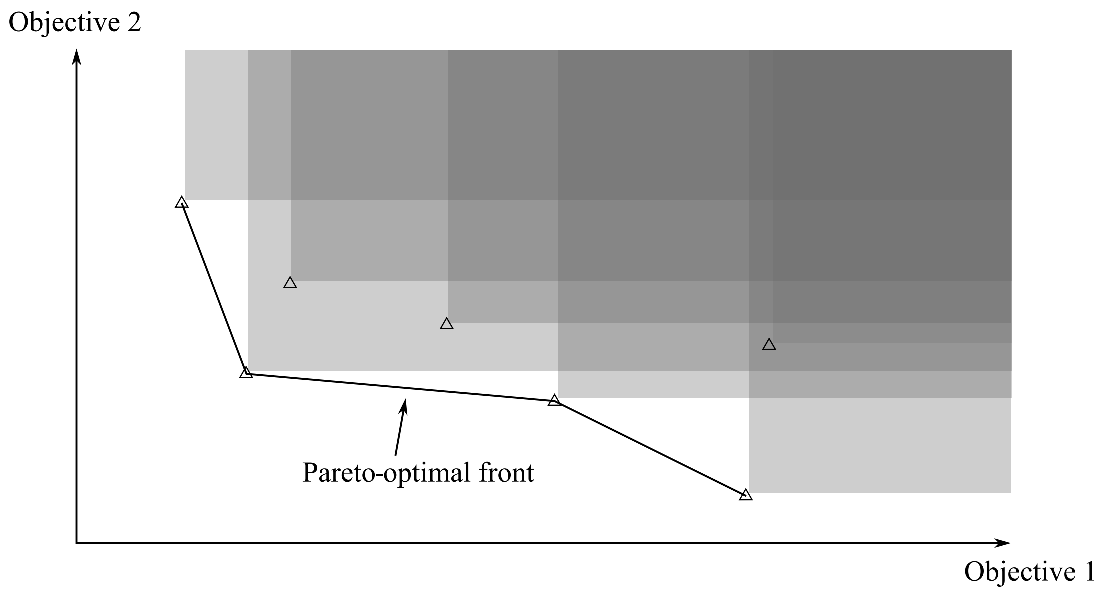


Figure A.1: This plot shows a set of dominated and non-dominated solutions. The non-dominated solutions must not be shadowed by any quadrant emerging from the other solutions.

Algorithm 2 Find best front

```
 1: Sort OrigPop on Objective1
 2: Front ← <empty list>
 3: MinIndividual ← SortedPop[0]
 4: for all individuals in OrigPop do
 5:     if CurrentIndividual.Objective2 < MinIndividual.Objective2 then
 6:         MinIndividual ← CurrentIndividual
 7:         Front.add(CurrentIndividual)
 8:         OrigPop.remove(CurrentIndividual)
 9:     end if
10: end for
```

If using a GA as search method, the original population could be extended by its offspring, effectively creating a population double the size.

2: Use algorithm 2 until the original population OrigPop is empty. Add the layers in the order they were produced to a new population. Go to step 3 if the current front does not fit the new population. This population will be sorted according to non-domination, with the first front being the only front with truly non-dominating solutions.

3: If one front does not fit in its entirety, pick the correct amount of individuals from this front by ensuring a uniformly spread-out Pareto-optimal front. This is achieved by selecting the individuals that give rise to the largest crowding distance if they were to be inserted in the new population. The crowding distance is calculated as the distance between the nearest neighbors of the new individual.

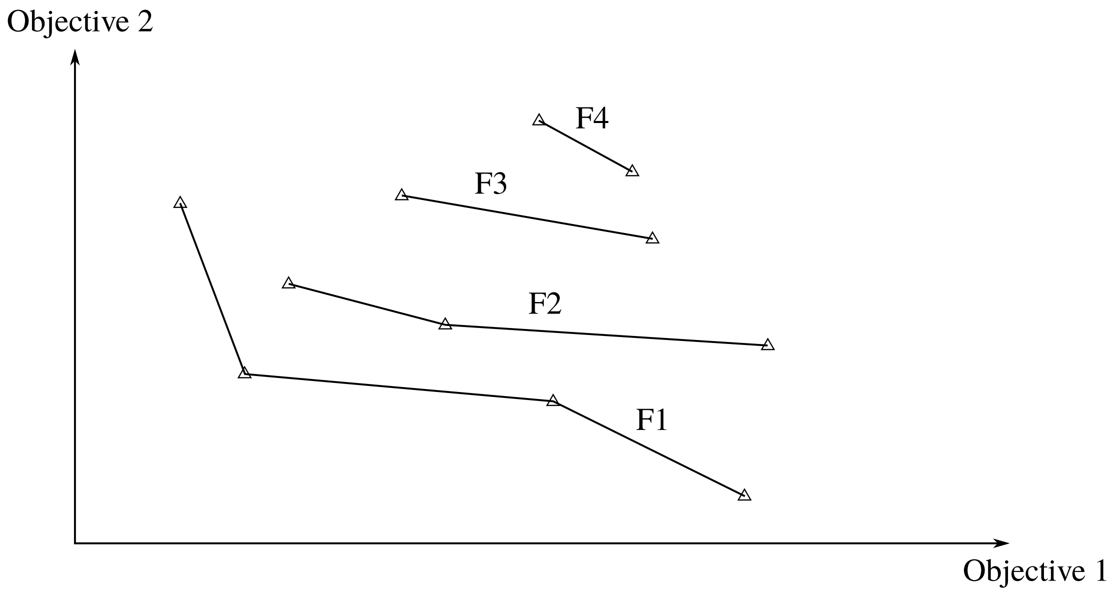


Figure A.2: This figure shows the sorted fronts after step 2. In step 3, the first front chosen is going to be F1.

4: When the new population has the same size as the original population, one iteration of NSGA-II is completed, and any GA can be used to apply the other evolutionary operators.

This entire workflow of the algorithm is summarized in figure A.3, inspired by [22, Figure 2].

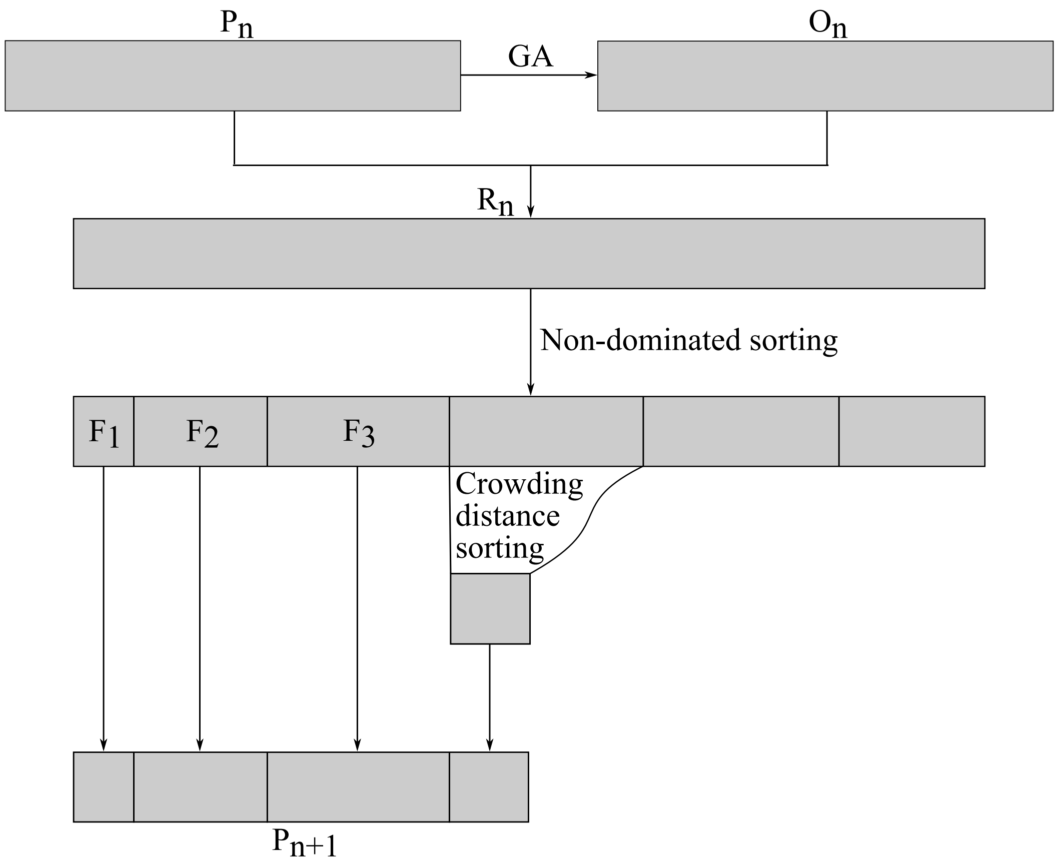


Figure A.3: This figure visualizes the entire NSGA-II workflow with a GA. The initial population P_n and its offspring O_n creates a new population R_n, which is sorted producing several Pareto fronts. The three first fronts are transferred directly to the next generation P_{n+1}, while the crowding distance sorting is used to select survivors among the fourth front.

Appendix B

Questionnaire

You have been chosen to participate in a user test about a mobile shooter game. Your task is to play through ten waves and try to grade them in a few ways. The two first waves are not included in the test, but they will introduce you to the game and set a few reference levels.

Each wave has a difficulty (1-10) and an intensity (1-10) rating. The waves are either human-made or computer-generated. Your task is to try to assess each wave's difficulty and intensity values and try to determine if it is made by an algorithm or a human.

Table B.2: Have you played mobile shooters before?

Yes	No
☐	☐

Table B.2: Difficulty ratings

	1	2	3	4	5	6	7	8	9	10
Wave 1	■	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 2	☐	☐	☐	☐	☐	☐	☐	☐	☐	■
Wave 3	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 4	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 5	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 6	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 7	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 8	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 9	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 10	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Table B.3: Intensity ratings

	1	2	3	4	5	6	7	8	9	10
Wave 1	■	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 2	☐	☐	☐	☐	☐	☐	☐	☐	☐	■
Wave 3	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 4	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 5	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 6	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 7	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 8	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 9	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Wave 10	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Table B.4: Algorithm ratings

	Human-made	Computer-generated
Wave 3	☐	☐
Wave 4	☐	☐
Wave 5	☐	☐
Wave 6	☐	☐
Wave 7	☐	☐
Wave 8	☐	☐
Wave 9	☐	☐
Wave 10	☐	☐

Appendix C
Test Results

A number of 20 persons participated in the survey, 19 of which had earlier experience with mobile game shooters.

Table C.1: Perceived difficulty levels

	W3	W4	W5	W6	W7	W8	W9	W10
Test Person 1	8	5	3	6	10	10	10	3
Test Person 2	7	6	3	6	8	5	6	6
Test Person 3	6	7	5	6	6	5	9	6
Test Person 4	8	10	5	3	9	6	10	7
Test Person 5	6	6	6	6	7	9	9	5
Test Person 6	8	3	2	4	3	1	10	3
Test Person 7	7	5	1	3	3	6	9	5
Test Person 8	4	9	2	7	5	9	8	5
Test Person 9	6	7	4	7	8	6	10	2
Test Person 10	7	5	4	6	5	6	9	5
Test Person 11	7	6	5	4	2	4	5	3
Test Person 12	2	9	3	7	5	5	7	6
Test Person 13	6	9	4	5	7	7	7	6
Test Person 14	6	9	2	5	6	6	6	6
Test Person 15	8	4	3	6	8	8	10	6
Test Person 16	5	5	6	7	7	7	8	9
Test Person 17	7	6	3	5	6	7	8	2
Test Person 18	5	9	3	9	7	7	10	5
Test Person 19	5	10	1	6	8	9	10	4
Test Person 20	7	7	3	3	5	5	8	6

Table C.2: Perceived intensity levels

	W3	W4	W5	W6	W7	W8	W9	W10
Test Person 1	8	10	4	7	8	7	10	5
Test Person 2	7	8	3	6	5	4	5	4
Test Person 3	6	7	5	7	7	7	10	6
Test Person 4	9	10	5	5	10	10	8	7
Test Person 5	7	6	6	6	6	9	10	7
Test Person 6	6	5	2	3	2	1	5	1
Test Person 7	6	2	1	5	6	7	4	3
Test Person 8	6	5	3	8	7	9	8	5
Test Person 9	8	8	5	7	8	7	7	3
Test Person 10	7	10	2	6	5	8	10	4
Test Person 11	5	8	3	7	6	6	8	2
Test Person 12	2	8	4	9	7	5	6	6
Test Person 13	4	3	2	4	5	4	7	4
Test Person 14	3	5	3	3	4	7	4	4
Test Person 15	6	7	3	7	6	7	8	5
Test Person 16	5	5	6	7	8	8	8	9
Test Person 17	8	7	4	5	6	8	9	3
Test Person 18	5	4	3	8	7	6	8	5
Test Person 19	7	7	5	8	8	9	10	7
Test Person 20	7	8	3	3	4	4	8	5

Table C.3: Perceived sense of whether the waves were made by a human (H) or a computer (C)

	W3	W4	W5	W6	W7	W8	W9	W10
Test Person 1	C	C	C	C	C	C	C	C
Test Person 2	H	H	C	C	C	C	C	C
Test Person 3	C	C	C	C	C	C	C	C
Test Person 4	H	C	H	H	C	C	C	H
Test Person 5	C	H	H	C	H	C	C	C
Test Person 6	C	H	C	H	H	C	C	C
Test Person 7	H	H	C	C	C	H	C	H
Test Person 8	C	H	H	C	C	C	H	H
Test Person 9	H	C	C	H	H	H	H	H
Test Person 10	C	C	C	H	C	C	C	C
Test Person 11	C	H	C	C	H	C	C	C
Test Person 12	C	C	H	C	H	C	C	H
Test Person 13	C	H	H	H	C	H	H	C
Test Person 14	C	H	C	C	H	H	H	H
Test Person 15	C	C	C	H	C	C	C	C
Test Person 16	C	C	C	C	H	C	C	H
Test Person 17	H	H	H	C	C	C	C	H
Test Person 18	H	H	C	C	C	H	C	C
Test Person 19	H	C	H	C	C	H	C	H
Test Person 20	H	C	H	C	C	C	C	H

Appendix D
Enemy Statistics

Table D.1: Enemy statistics

	Health	Speed	Attack power
Type 1	300	0.8	320
Type 2	50	1.5	75
Type 3	150	2.8	150
Type 4	50	2.8	34
Type 5	275	0.8	325
Type 6	75	1.4	50

Table D.2: Normalized enemy statistics

	Health	Speed	Attack power
Type 1	1.000	0.286	0.985
Type 2	0.167	0.536	0.231
Type 3	0.500	1.000	0.462
Type 4	0.167	1.000	0.105
Type 5	0.917	0.286	1.000
Type 6	0.250	0.500	0.154

EXAMENSARBETE Search-based Procedural Generation of Gameplay Content
STUDENT Einar Nordengren
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Jörn Janneck (LTH)

Procedurell generering av spelinnehåll

POPULÄRVETENSKAPLIG SAMMANFATTNING **Einar Nordengren**

På grund av datorspelsindustrins stora framväxt de senaste åren är behovet av nytt och varierat spelinnehåll större än någonsin. I detta arbete har vi tagit fram och testat en algoritm som kan underlätta skapandet av spelinnehåll.

Tack vare den snabba utvecklingen av datorer de senaste decennierna har möjligheterna inom datorspelsbranschen exploderat. Begränsningar om hur mycket innehåll man kan inkludera i ett datorspel finns knappt i dag, vilket både har fört med sig nya problem och ny potential. I takt med att hårdvaran utvecklas ökar samtidigt spelarnas behov av nytt material.

En stor utmaning spelbranschen har ställts inför är att kunna skapa allt detta spelinnehåll på ett effektivt sätt. Att lösa problemet genom att bara anställa fler grafiker och designer är inte bara problematiskt ur ett ekonomiskt perspektiv; att skapa allt spelinnehåll för hand tar även ofta lång tid.

I detta examensarbete har vi utvecklat ett sätt att procedurellt (eller automatiskt) skapa det önskade innehållet i stället för att behöva göra det för hand. Spelet i fokus är ett skjutspel för mobil som går ut på att, som statisk spelare, eliminera fiender som anfaller i vågform. Eftersom spelinnehåll kan betyda så många olika saker, har vi avgränsat oss till att endast definiera och generera antalet fiender av olika typer i varje våg.

Det finns många utmaningar med att låta automatisk generering av spelinnehåll ta över: dels måste det genererade innehållet gå att kontrollera, och dels måste det se ordnat ut. Målet med arbetet var därför att skriva en algoritm som automatiskt definierar fiendevågor så att en viss svårighetsgrad och intensitet följer speldesignerns önskade värden. Samtidigt var avsikten att spelaren inte skulle ha någon aning om att det faktiskt var en algoritm som hade skapat innehållet.

Arbetet utvärderades i form av en studie där ett antal testpersoner fick spela igenom spelet och svara på frågor om hur svåra och intensiva fiendevågorna var, samt en fråga om de kunde avgöra om det var en algoritm eller en människa som skapat innehållet.

Resultatet av studien visade att det var svårt att få algoritmen att skapa innehåll så naturligt som en designer kan, men att det var möjligt att skapa ett verktyg som till viss grad följde svårighetsgraden och intensiteten.

Den metod vi använde för att generera innehållet är baserad på en genetisk algoritm. Precis som man kan vänta sig bygger genetiska algoritmer på liknande principer som evolutionen bygger på. I början genereras helt slumpmässigt innehåll, vilket i vårt fall endast var en massa tal som representerade antalet fiender på olika ställen i spelet. Efter det utvärderas dessa tal utifrån vissa kriterier. En våg med högt betyg var här en våg som uppskattades följa designerns önskade värden och som bedömdes likna en människogjord våg. De individer som får bra betyg överlever och används senare som mall för att generera nya individer. Denna process upprepas sedan flera gånger ända tills en tillräckligt bra våg har skapats.