Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eirik H. Skjærseth and Harald Vinje

Evolutionary algorithms for generating interesting fighting game character mechanics

Master's thesis in Computer Science (MTDT)
Supervisor: Ole Jakob Mengshoel
July 2020

NTNU
Norwegian University of Science and Technology

Eirik H. Skjærseth and Harald Vinje

Evolutionary algorithms for generating interesting fighting game character mechanics

Master's thesis in Computer Science (MTDT)
Supervisor: Ole Jakob Mengshoel
July 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Procedural content generation (PCG) is the process of generating video game content through algorithms and has been used in the game industry for a long time. *Content* refers to elements in a game such as levels, terrain, or game rules. PCG has several benefits for the industry since computer-generated content can inspire human designers, and be used to create games with endless content. Research interest in the PCG field has grown over the last couple of decades, where PCG problems are often formulated as search problems. Evolutionary algorithms have frequently been applied as the search mechanism, with an objective defined by a desired content quality. The optimal quality in entertainment games can be formulated as the question *will a player enjoy this content?* This quality has been quantified by heuristics within different game genres, with success. A promising approach is based on evaluating content while it is played in a game, referred to as simulation-based evaluation.

Constraint novelty search is an evolutionary algorithm that has emerged and shown promise in the field of PCG. The commonly used objective given by a fitness function is replaced by the objective of novelty alone.

This thesis is a first attempt at using constraint novelty search, with constraints based on simulation-based evaluation, to generate character mechanics in a fighting game. Character mechanics refer to the technical design of characters, i.e., how they may behave in a game. Aesthetics like graphics and sound are not considered. A simple two-player fighting game will be used for this research, developed by the thesis authors. The game is based on design patterns seen in commercial fighting games.

User studies are conducted to evaluate the generated character mechanics, based on the enjoyment of test subjects. The results show that the applied generation method can produce multiple characters that are perceived as more interesting than human-designed characters. However, characters of high quality are not generated consistently. The amount of test subjects is limited, such that further research is needed to verify our results.

The proposed generation method can be applied to other fighting games on the conceptual level. However, the implemented system is dependent on technical aspects and heuristics of the game used for this thesis.

Sammendrag

Prosesuell generering (Procedural Content Generation, PCG på engelsk) er en metode brukt i spillutvikling for å generere spillinnhold ved bruk av algoritmer fremfor menneskelig arbeid. Spillinnhold er elementer som for eksempel baner eller terreng. PCG er nyttig siden det kan inspirere til nytt design, spare arbeidstimer, og produsere spill med ubegrenset innhold. Som forskningsfelt har interessen for PCG økt de siste tiårene, hvor PCG problemer ofte blir formulert som søkeproblemer. Evolusjonære algoritmer er en av de mest brukte algoritmene i PCG, hvor algoritmens mål (objective) er spillinnhold av høy kvalitet fra et menneskeperspektiv. Tidligere forskning på PCG har lyktes med å kvantifisere kvaliteten på spillinnhold ved å bruke heuristikker basert på kunsitg intelligente algoritmer som spiller gjennom innholdet. Denne kvantifiseringsteknikken kalles simuleringsbasert evaluering.

Denne oppgaven er et forsøk på å benytte evolusjonære algoritmer i PCG-feltet for å generere fyiskken til spillkarakterer i “fighting game”-sjangeren. Vi bruker en variant av evolusjonære algoritmer kalt ”Constrained novelty search”, som fokuserer på å genere unike karakterer som oppfyller visse kvalitetskriterier under en simuleringsbasert evaluering. Med fysikken til en spillkarakter, refererer vi til karakteresn fysiske egenskaper, som f. eks farten, styrken og størrelsen til karakteren. Spillkarakterenes grafiske elementer og lydelementer er ikke vurdert i denne oppgaven.

Vi konstruerte et system som genererer spillkarakterer for et enkelt 2D-spill i fighting game-sjangeren. Vi utførte brukertester for å vurdere systemet ved å trekke ut et sett med karakterer for testing. Resultatene viser at karakterene generert av systemet vårt i flertallet av tilfellene var mer interessante enn de andre, menneskekonstruerte karakterene, sett fra brukernes perspektiv. Enkelte av de genererte karakterene var dog lite interessante, så systemet vårt var ikke helt konsistent. I tillegg var antallet brukertestere lavt, så mer forskning behøves for å trekke klarere konklusjoner. Systemet for karaktergenerering kan brukes til spill i fighting game-sjangeren generelt, men mye av systemet er implementert med hensyn på spillet som er brukt i denne oppgaven.

Preface

This thesis was conducted for the course TDT4900 - Master thesis at the Norwegian University of Science and Technology (NTNU). This thesis was was written and carried out by Eirik H. Skjærseth and Harald Vinje, both master level students at the Department of Computer and Information Science, Faculty of Information Technology, Mathematics and Electrical Engineering. The project was supervised by professor Ole Jakob Mengshoel.

Many thanks are given to our supervisor, Ole Jakob Mengshoel, who supported our idea for this thesis. He has given us a lot of help for this thesis. He has also been a supporting factor throughout this process, and we have really appreciated his good mood at all times.

This thesis was inspired by our interest in game mechanics and artificial intelligence. The game Sol was initially prototyped in the summer of 2016. In 2017, we got a summer scholarship to continue the project, with focus on creating a fighting game from the bottom, including physics calculations, graphics and game mechanics. Our interest in game mechanics grew, and many days were spent investigating low-level mechanics of games like Mario. Our interest in artificial intelligence grew as well, so we wanted to write a thesis on the intersection of these two fields.

User studies have been conducted for this thesis, where eight user contributed. We would like to thank the users for participating. We also appreciate that Jens-Andreas Rensaa lent us computer power for our experiments.

We would like to thank Blank as well, who gave us the a summer scholarship for initially creating the Sol game.

The last six years in Trondheim have probably been the best years of my life (even though childhood was easy and fun). I would like to thank everyone that has contributed to this amazing time. Thanks to Abakus, Bedkom, Abakusrevyen, Scenegruppa, Shape of You, Revystyret, Nattforestillingen, UKErevyen, the 4^{th} basketball team of NTNUI and flat mates. I would also like to thank my great family, *Famtastic*, and grandparents for all their support. They gladly listen to me rambling on about projects I'm engaged in and are an incredible motivational factor. Also a special thanks to my grandpa who has boosted curiosity and practical skills, that are essential for who I am today. With all the knowledge I have obtained through these six years, I'm really looking forward to the future.
- Eirik H. Skjærseth

Many good years at NTNU in Trondheim have come to an end, and I look back on my time here with some nostalgia and many precious memories. I have gained

warm and meaningful relationships through my time in the Computer Science classes, Abakus, and the NTNU rowing crew. I am grateful for the opportunity to spend the last five years in such friendly inspiring environments.

- Harald Vinje

Eirik H. Skjærseth and Harald Vinje

Trondheim, July 1, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces our thesis. We present the context and motivation for our research in section 1.1. The goal of the research and research questions are stated in section 1.2. We provide a definitions of key terms in section 1.3. A disclaimer of the research is given in section 1.4.

1.1 Context

Artificial intelligence (AI) has seen a lot of progress lately through improving methods such as reinforcement learning, artificial neural networks, tree search, and evolutionary algorithms. Games have been used to utilize and experiment with AI, both in academia and the industry. The most common application of AI in the realm of games is agent playing behavior. AI playing classic games such as Chess and Go have had breakthroughs with the use of reinforcement learning and Monte Carlo tree search, and video games such as Star Craft[1] and DOTA[2] have also seen great advances when it comes to optimizing player behavior.

Besides playing games, however, there are many other applications within games and other simulated environments where AI can be applied. Computational generation of game content is one such field, a field known as *Procedural Content Generation* (PCG). PCG has been around since the 1980s, with early applications seen in the game *Elite*[3] (by Acornsoft) in the year 1984, among others. Recent

[1] `https://starcraft.com/en-us/`
[2] `https://www.dota2.com/play/`
[3] Elite: `https://en.wikipedia.org/wiki/Elite_(video_game)`

titles such as *No Man's Sky*[4] (by Hello Games) have made PCG a selling point. PCG attracts players because of the excitement of experiencing novel maps and creatures, giving games plenty of replay value.

Generating content is a central part of video games. It is time-consuming, labor heavy, and requires creativity. Humans are only able to create a fixed set of content to ship with a game, while computers can in principle create infinite content, even while the game is being played. This opens the possibility for infinite replay values of games.

PCG can be used to inspire and extend the creativity of a human. Tools have been made, such as Sentient Sketchbook [26] and Restricted Play [16]. These tools complement a human designer by proposing alternatives and possibly improved content as the designer progresses with manual design.

Research shows that generating complex content[5] in the field of PCG shows most promise by search-based methods (search-based PCG). Content generation is formulated as a search problem, and an objective is defined based on the desired content *quality*. The literature shows that evolutionary algorithms are the most widely applied search algorithms for this purpose. Novelty search is a variant of evolutionary algorithms that shows promise in search-based PCG, as shown by Liapis et al. [25].

Search-based PCG can be used to evaluate content by letting computer players, based on game-playing AI techniques, take the role of a human player. This technique is called simulation-based evaluation and is necessary if the generated content can not be evaluated outside the context of a game. The optimal content quality of entertaining games can be seen as the enjoyment of a player that plays with the given content. Evaluating player enjoyment computationally is a hard task. Heuristics on specific game genres have priorly been used to estimate player experience, through simulation-based evaluation.

The research within the field of PCG has seen a noticeable growth over the last decade. The General Video Game AI (GVGAI) competition has promoted research within both game-playing AI and AI methods for PCG, with individual competitions for generating game levels and game rules. The games examined in the GVGAI competition are limited to simple action games.

Research on AI in the context of fighting games has also seen growth over the last decade. The Fighting game AI (FTGAI) competition has been held yearly since 2013. This competition promotes research on game-playing AI in fighting games. Fighting games are complex real-time games, compared to the games used

[4]No Man's Sky: `https://en.wikipedia.org/wiki/No_Man%27s_Sky`

[5]Complex content is game content that is integral to a game and hard to evaluate.

for the GVGAI competition. However, there has not been any research towards generating content in the FTGAI competition.

In the field of AI and games set out in this section, this research will first contribute to the use of PCG in the context of fighting games, to generate interesting character mechanics. Secondly, to evaluate player experience through simulation-based evaluation in a fighting game. To our knowledge, these two areas have not been covered by previous research.

1.2 Research goal

This thesis presents a system to procedurally generate content for a fighting game, created by the authors. It is an attempt to combine methods from the PCG field, with player experience estimated through simulation-based evaluation in the fighting game domain. Fighting games propose several different characters that can be chosen by a player. Thus, the technical design of the characters is a large part of the mechanics in a fighting game. The technical design refers to how a character may behave in a given game, and is referred to as the *mechanics* of a character. This thesis will focus on generating *interesting character mechanics*. The goal of this research is summarized as follows:

Research goal: Explore generation of character mechanics in fighting games through the use of evolutionary algorithms.

The concept of *interestingness* in the context of fighting game characters are based on theories of what makes games fun and creative. Interestingness in the context of this work is defined as: *novel*, *fun* and *balanced*. Novelty is the property of a character being dissimilar to other characters. Fun is determined by how a player experiences a character. Balance relates to characters where no character is superior to another.

Character mechanics are generated for a two-player fighting game developed by the authors. It is a simplified version of a typical fighting game and shares many of the design principles commonly found in fighting games. The rules of characters are given by the game itself, and parameterized representations of the mechanics of characters are evolved.

Our system is based on a search method known as *constrained novelty search*, which is a type of evolutionary algorithm that uses novelty as the main objective. Constraints are defined based on the concept of fun and balance through simulation-based evaluation. A simulation is performed with a computer-controlled player that plays a generated character, while the character is evaluated. The character is evaluated according to a set of criteria that evaluates the player

experience with regards to *fun* and *balance*.

Three research questions were defined for meeting the research goal:

RQ1 How can constrained novelty search be utilized to evolve interesting character mechanics?

RQ2 Can we measure fun and balance of character mechanics in a simulated game?

RQ3 Can constrained novelty search yield character mechanics that humans find interesting?

As mentioned in section 1.1, the most common application of AI in games is game playing behavior and decision making. In this thesis, the research is focused on *optimizing the game rather than the player*. To illustrate what this means we present a research metaphor.

Imagine the act of driving your car to work. In this case, you are the agent performing actions. There are several actions you, as the agent driving the car, can perform to optimize your commute to work. External entities, such as your car, the road, traffic lights, etc. are part of your environment. They will affect your experience of driving to work. Instead of searching for the optimal strategy for the agent, we aim to optimize parts of the environment that the agent operates in. In this thesis, an analogy to character mechanics is the car. We seek interesting character mechanics in a video game. Similarly, a car manufacturer seeks a diverse set and interesting cars for any driver to enjoy. The metaphor is visualized in figure 1.1.

Game playing AI refers to the behavior of an agent operating in an environment. Wide known examples of game-playing AI are the Deep blue chess computer, and the AlphaGo computer playing Go.

AI for content generation refers to using AI methods for generating content artifacts. The word "Content" has a broad meaning, and generally refers to any non-player elements within a game, such as game levels, maps, obstacles or game rules.

The main focus of this research is AI for content generation.

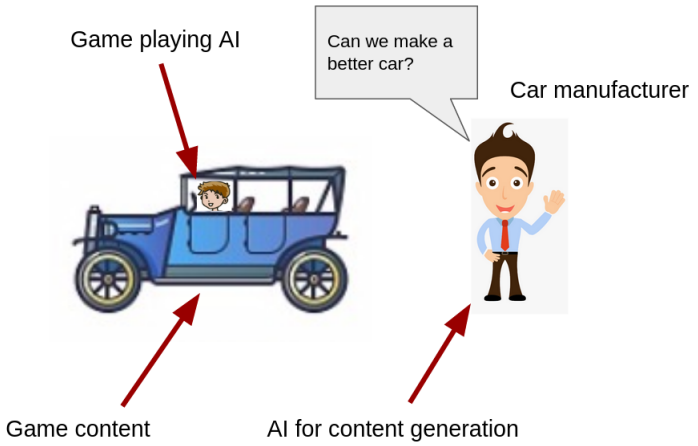


Figure 1.1: A metaphor for our research. An analogy to the game is the activity of driving a car on the road. The car can be viewed as content within the game. We explore the use of AI for generating content, as opposed to game playing AI.

1.3 Definitions of terms

There are some terms used frequently in this thesis that can be confusing. As such, this section provides definitions for these terms and their meaning in this thesis.

Character mechanics

Character mechanics can be seen as the "technical" design of a character. It refers to how a character may behave in a game. Aesthetics like graphics and audio are not considered mechanics. Our representation of character mechanics is given by properties that affect the mechanics. Typical examples of character mechanics are movement speed, jump height, and attack strength. A "character configuration" refers to a given character in the described representation. A "character" refers to character mechanics in the context of generating characters. In the context of a game, a "character" refers to an actual character that can be played and experienced by a player.

Computer player

Computer-controlled player, or just computer player, refers to a computer algorithm controlling a character in a game. The term computer player could be seen as a substitute name of "AI player". We prefer the term "Computer player" since "AI player" indicates methods considered as artificial intelligence methods, whereas Computer player allows for both simple and complex methods.

Interestingness

Although not a commonly used word, interestingness is the word we use to describe the quality of a character with regards to how it is experienced by a player. Interesting characters are characters that are novel, fun, and balanced.

1.4 Disclaimer

This thesis is a continuation of the specialization project conducted in the fall of 2019 by the thesis authors (Skjærseth and Vinje). The resulting report based on the project is titled "Automatic generation of character mechanics in fighting games", and contains a literature review, as well as a proposed framework for generating character mechanics in fighting games.

Large parts of chapter 2 and chapter 3 are based on the specialization project report. Some sections in these chapters contain similar paragraphs that are modified to a varying degree. We will mark these paragraphs with footnotes throughout this thesis, to clearly state that they are not original work.

Chapter 2

Background

In this chapter, we present an overview of the concepts and research areas related to our goal of procedurally generating fighting game character mechanics.

We introduce the research area of procedural content generation (PCG) in section 2.1. Evolutionary algorithms is a search algorithm often applied in PCG problems, and will be presented in section 2.2. This thesis focus on PCG in the fighting game domain that is presented in section 2.3. To evaluate fighting game characters according to *interestingness*, we need to know how this term can be interpreted in the context of video games and for characters in a fighting game. This is discussed in section 2.4. PCG problems need a method to evaluate the generated content. In this work, characters will be evaluated according to interestingness, i.e., how a human player would experience a character. This can be done through simulation-based evaluation that is covered in section 2.5.

2.1 Procedural content generation

Procedural content generation (PCG) is the process of generating game content through the use of algorithms and computational methods, rather than manually through human labor. In PCG, game content typically refers to artifacts such as game levels, maps, terrains, items, enemies, rules, or game parameters. Non-player character (NPC) AI is typically not viewed as content in PCG [48].

The appropriate methods for generating content will vary greatly based on the nature and complexity of the content. We will therefore make some distinctions on the methods used in PCG.

Constructive methods vs generate-and-test

The first way we will distinguish PCG methods is between *constructive* methods, and methods described as *generate-and-test*. Constructive methods are methods in which content is generated once, and then either used or discarded. There may be constraints on the generated content during the generation process to steer the content toward a desirable solution, but once a solution is produced, the constructive method is finished. Typical examples of content created using constructive methods are backgrounds or landscapes in a video game. Constructive methods are more suitable for less complex content with fewer constraints.

On the other end are the methods we describe as generate-and-test. These methods are divided into two parts, a generator and an evaluator (or tester). The generator produces solutions for the evaluator to test given some criteria. If the test fails, the solution, or a part of the solution, is discarded and the generation process restarts. This continues until a desirable solution is produced. The generate-and-test approach is suitable for more essential and complex content of a video game, such as game levels in a platform game, or game enemies. In these situations, the search space may be large, but the desirable solution space is typically a very small subset of the search space. The quality of the content will also greatly impact the gaming experience, so we need strict criteria for the solutions.

Autonomous vs mixed-initiative

Another way to distinguish PCG methods is by the degree of human involvement in the generation process. We separate here between *autonomous* and *mixed-initaitve* methods. Content generation can be viewed on a continuum between fully autonomous methods on one end, and entirely human-designed on the other end. In a fully autonomous system, the computer system drives all the creative processes of the generation of content. Mixed-initiative methods lie somewhere in the middle of the continuum of fully autonomous methods and entirely human-based content creation. Yannakis et. al defined mixed-initiative PCG as a "process that considers both the human and the computer *proactively* making the content contributions to the game design task although the two initiatives do not need to contribute to the same degree" [47].

2.1.1 Search-based procedural content generation

The search-based approach is the approach often investigated in academic research of PCG [44]. It is a variant of the *generate-and-test* procedure discussed previously. The idea behind search based PCG is that content design in video games can be viewed as a search problem. The search is performed in the domain

of artifacts (a piece of generated content) with an objective, given by a desired content *quality*. The objective is defined by a *fitness function* that assesses an artifact and outputs a fitness score. Content quality can be given by a set of constraints defined by a designer. An example is a search for game levels that can be completed by a human within a reasonable time frame.

Common for all search-based PCG methods are three major components:

- A **search algorithm**
- A way to **represent the content**
- A **fitness function** to evaluate any given solution (or partial solution). A solution in this context refers to an artifact.

We cover these three components briefly here.

The search algorithm is what drives the exploration of new and better content. The most common search algorithms seen in complex PCG problems are evolutionary algorithms. Evolutionary algorithms are further presented in section 2.2.

The content can be represented in various ways, but a useful distinction between representations is how closely the content is tied to the representation of the content. In the case of evolutionary algorithms, an artifact in its representational form is the *genotype*, while the artifact as it is presented in the game is the *phenotype*. An example of a direct representation of a game level is one where the genotype represents a grid of the game world, with each grid cell either being empty or containing an object in the game world. An example of a more indirect representation of the game level is one in which the genotype only specifies some desired properties of the level, and it is up to the *genotype to phenotype mapping* function to define a useful transformation from the genotype to the phenotype. Indirect representations tend to be less fine-grained and have smaller search spaces, which can be beneficial in certain cases. However, they also tend to have less correlation between adjacent points in the phenotype and genotype spaces. This is undesired as small changes to the genotype can yield a large difference in the phenotype. The genotype to phenotype mapping is usually more complex as well.

The final major component of search-based methods is the fitness function. We distinguish here between *direct* fitness evaluations and *simulation based evaluation*. Direct evaluations use explicit functions or formulas to evaluate the fitness of the phenotype. Direct evaluations may use statistics or data of what constitutes desirable solutions to define the evaluation function. In other situations, there might be no reliable way to evaluate content without putting it in a game.

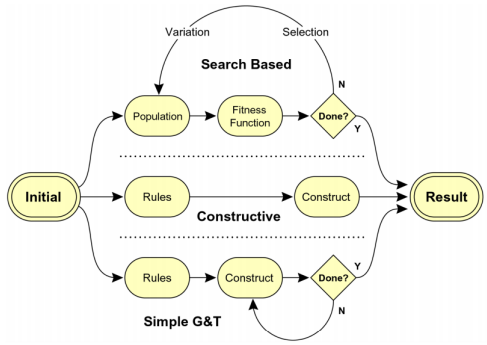


Figure 2.1: The constructive, generate-and-test and search-based approach to PCG. Figure by Togelius et al. (2011) [45]

This might be due to the fitness being time-varying or involve randomness and requires a simulation-based evaluation. In simulation-based evaluation, a computer player plays through the game with the artifact to be evaluated. The played game is referred to as a *simulation*. To evaluate the artifact, data is extracted from the simulation and used to determine the fitness. This approach introduces several new challenges. We need an environment that can spawn game simulation instances with the artifact we wish to evaluate. We also need to define the content quality we wish to evaluate such that it can be quantitatively measured through the simulation. Simulation-based testing is further covered in section 2.5.

The reader is encouraged to read chapter 4 of the 2018 academic book "Artificial Intelligence and Games" by Georgios N. Yannakis and Julian Togelius [48] if they wish to get a more detailed and thorough picture of PCG than we provide in this chapter.

2.2 Evolutionary algorithms

Evolutionary algorithms are a class of search algorithms that are widely used in optimization problems and is one of the most commonly used algorithms in search-based PCG. As such, this section provides an introduction to evolutionary algorithms. The reader is recommended to read Introduction to evolutionary computing book by Eiben and Smith for a more thorough introduction [11].

Evolutionary algorithms are stochastic global optimization algorithms that draw inspiration from the principles of biological evolution through natural selection,

reproduction, and mutation. Evolutionary algorithms are classified as stochastic since they use randomness during the search, and global since they are intended to determine optimal solutions within the entire search space. Evolutionary algorithms can be used for a variety of problems, but are especially useful for problems where a large portion of the search space needs to be explored for solutions, and where there is little prior knowledge to where in the search space desirable solutions may be.

In an evolutionary algorithm, we refer to a proposed solution as an *individual* and a set of individuals as a *population*. An individual has two different manifestations; a genotype and a phenotype. The genotype is the representation used in the search algorithm, where evolutionary operators (reproduction and mutation) can be applied. The individual components of a genotype are referred to as *genes*. The phenotype refers to all the observable characteristics of the individual in a practical setting. In the case of generating video game characters, the phenotype is the character as it appears when playing the game. Evolutionary algorithms give no constraints on the genotype of an individual, so it is up to the programmer to decide a useful representation. A trivial genotype representation often used is simply an array of numbers, where each number corresponds to an attribute of the individual.

The algorithm itself starts by generating an initial population of individuals either randomly or based on some heuristic. The initial population should contain individuals that are widely distributed in the search space. Otherwise, the algorithm could easily conform to local search within a small subset of the search space.

Each individual in the population is evaluated using a *fitness function* which assigns individuals a fitness value. Individuals are selected as parents for reproduction of new individuals based on their fitness. This process is referred to as *parent selection* (or just *selection*) and is an integral part of the performance of an evolutionary algorithm. The selection scheme is independent of the search domain, thus general purpose schemes have been investigated and presented, by for example Blickle and Thiele [3]. Selection schemes have varying dependence on the fitness of a population. This is referred to as the *selection pressure* and is described in detail by researchers like Back [2]. A selection that highly prioritizes individuals with high fitness applies high selection pressure.

The reproduction process is referred to as a *crossover*. A crossover combines two parents to produce one or more (typically two) offspring individuals. The crossover operator is highly dependent on the domain and the genotype representation of a solution. The crossover operation promotes exploration of the search space, along with a *mutation* operator. A mutation is a process that alters some

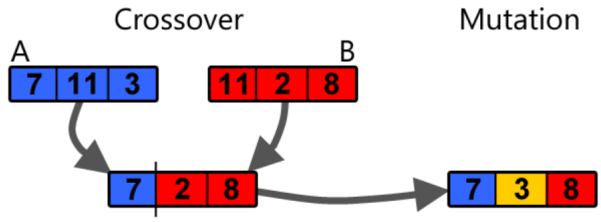


Figure 2.2: Crossover and mutation of an evolutionary algorithm. Individuals are represented as an array of three numbers in this case.

of the genes of an individual's genotype. Genes can be replaced or swapped internally. The mutation operator promotes local search, as mutations on the genotype should yield new individuals where the phenotype is adjacent, or close in the search space. A visualization of crossover and mutation can be seen in figure 2.2[1]. There is often randomness involved when selecting, crossing, or mutating individuals. For example, a mutation policy could be that attribute a of individual i has a probability p of being changed during mutation.

The last step of one iteration of an evolutionary algorithm is to decide which individuals to keep among the current population and its offspring, and which to discard. The individuals that are kept constitute a new population of a new *generation*. The algorithm proceeds with evaluation of the new generation. These steps are repeated to produce multiple new generations until a termination condition is met. A termination condition could be based on whether a solution discovered is good enough. In many cases of evolutionary algorithms, the goal is to find a solution as optimal as possible. It is therefore more common to define a termination condition based on a set amount of time, computational resources, number of generations, or other resource constraints. Pseudocode of a general

[1]This fugre was taken from "Automatic generation of character mechanics in fighting games" project report by Skjærseth and Vinje (2019)

evolutionary algorithm is shown followingly.

Algorithm 1: Pseudocode of a general evolutionary algorithm

```
Result: Population
Population=generateInitPopulation;
EvaluatedPopulation=evaluatePopulation(Population);
while not terminationCondition do
    Selection=selectIndividuals(EvaluatedPopulation);
    Offspring=crossover(Selection);
    mutate(Offspring);
    Popualation=newGeneration(Selection, Offspring);
    EvaluatedPopulation=evaluatePopulation(Population);
end
```

2.2.1 Exploration vs exploitation

The concept of *exploration* and *exploitation* are important within evolutionary algorithms. Exploration refers to exploring a large part of a search space, while exploitation refers to exploiting good solutions, and searching for slightly better ones nearby in the search space.

Local search methods, such as hill climbing, have no mechanism to ensure a wide exploration of the search space. They may arrive at a local maxima even though it might not be a global maxima. Several other points in the search can represent better solutions, far away from the discovered local maxima. This problem can also occur in global search methods, such as evolutionary algorithms. An early convergence toward a local maxima may discourage the search to explore other areas of the global search space. Important factors of an evolutionary algorithm that support exploration are how the initial population is created, how parents are selected, and the crossover scheme that is used. The initial population should be distributed across the search space, and parent selection should not strictly be based on the individuals with the highest fitness.

The opposite aspect of exploration in the context of evolutionary algorithms is the exploitation aspect. When selecting parents based on a population's fitness, we are exploiting the best solutions found to form even better solutions. This is important as the search space might be large, so exploiting seemingly fit landscapes of the search space for further evolution will lead to faster convergence.

Evolutionary algorithms need an appropriate balance between exploration and exploitation. Too much exploitation will promote fast convergence at the cost of missing globally better solutions. On the other hand, too much exploration can

lead to no or very little convergence. The right balance between exploration and exploitation is highly dependent on the search domain, the evolutionary operators used, parent selection scheme, and computational budget.

In many domains where evolutionary algorithms are used, a single optimal solution is desired. However, in other problems, a set of good solutions might be the goal of the search process. Other goals may be to discover a diverse set of desirable solutions. A set of solutions where each solution is similar to one another might not yield much value. In this case, exploration becomes even more important.

Because the search space might be sparse (meaning that local and global maximas are far apart) and the need for multiple dissimilar solutions, boosting exploration might be desired. Several techniques for encouraging population diversity have emerged and been widely studied in academic literature.

De Jong introduced the concept of niching in 1975 [9]. Common for niching techniques is maintainance of several diverse subpopulations to be explored in parallel and can be used to converge to multiple fit and divergent solutions. Different niching techniques discussed and tested in academic research include deterministic crowding [31], fitness sharing [13], and clustering [49], among others. The crowding approach encourages exploration by having a bias toward divergent individuals during the selection phase of an evaluated population. Fitness sharing cuts off the fitness of individuals in densely populated regions and share the value with less populated regions. In the clustering approach, the k fittest individuals are split into disjoint species, and the remaining individuals are clustered into one of the species. Individuals can only mate with individuals within the same species.

Another approach, called *Novelty search* was introduced by Lehman and Stanley [22]. It takes diversity to the extreme by replacing the objective of maximizing fitness by the objective of novelty. Novelty search is further presented in section 3.2.

2.3 Fighting games

In this section, we present the fighting game genre. We present what a fighting game is, the common design patterns of games from the genre, and artificial intelligence research within the domain.

The fighting game genre has been around for decades and is a popular genre to this date. The two fighting games *Super Smash Bros: Ultimate* and *Super Smash Bros: For Wii U* are seen on the top 20 list of most popular fighting

games in 2020, published by Gaming Gorilla[2]. Both these games are from the *Super Smash Bros* franchise by Nintendo. Other popular fighting game franchises include *Street Fighter* by Capcom, *Tekken* by Bandai Namco Entertainment, and *Mortal Kombat* by Midway Games, among others.

Fighting games are multiplayer games where two or more players control one character each. Fighting games usually take place in a two-dimensional stage seen from the side. A snapshot of a game of Super Smash Bros: Ultimate can be seen in figure 2.3 and Street Fighter 2 in figure 2.4. The goal is to eliminate the opponent's character by moving and using a set of attacks while defending yourself with defensive abilities. Each character has a fixed number of so-called *hit points*, or a *health bar*, that depletes when hit by an enemy attack. Characters are eliminated when their hit points reach zero. The Super Smash Bros. franchise differs from this traditional system. Instead of having a number of hit points, each character has a damage value that goes up every time the character gets hit. When getting hit, the character gets knocked backward with a force relative to the accumulated damage taken. If a character gets knocked off the stage, it is eliminated.

A fighting game contains a set of characters that the player can choose from, where each character is equipped with a unique set of offensive and defensive abilities. The attacks are mainly divided into two groups; melee and projectile attacks. Melee attacks are short-ranged and persistent with the reach of the character, like punches, and kicks. Projectile attacks are attacks emitted from a character and may travel for a longer distance, for example shooting a fireball. The differences between characters yield different playing strategies and styles.

2.3.1 Technical characteristics of fighting games

In this section[3], we look deeper into the design patterns and mechanics that fighting games are built around. These aspects will be central to how we represent a character, later seen in section 4.2.

When an attack is performed, it hits the opponent if the *hitbox* that corresponds to the attack intersects with the opponent's corresponding *hurtbox*. Hitboxes and hurtboxes are geometrical areas which correspond to attacks and characters, respectively, and are used to calculate intersection. The hitbox of an attack by

[2]Gaming Gorilla has made a ranking of the most popular video games in 2020: `https://gaminggorilla.com/most-popular-video-games-now`. The list is based on human voting from other internet sources.

[3]This subsection contains figures and paragraphs that are slightly modified versions of paragraphs taken from "Automatic generation of fighting game characters" by Skjærseth and Vinje (fall 2019), section 2.1.1.

Figure 2.3: Two characters can be seen on a stage in the game Super Smash Bros: Ultimate. The accumulated damage taken is presented at the bottom, next to the characters icons.

Figure 2.4: Two characters in combat in Street fighter 2. The character on the right performs an attack that hits with the character on the left. The hit points of each character can be seen at the top as a yellow bar with a red background

the character Ryu in Street Fighter 2[4] is displayed in figure 2.5. Although their names imply that they are boxes, hitboxes and hurtboxes can in principle have any geometrical shape.

A common attack pattern in fighting games is the concept of combinations or *combos*. A combo is a string of attacks performed sequentially. Hitting the opponent often opens up an opportunity for more hits, but typically requires skills and precision to effectively master.

Figure 2.5: The hitbox of a kick attack performed by the character Ryu in Street Fighter 2. The character's hurtbox is also displayed.

As we are concerned with fighting game character mechanics in this thesis, we will emphasize how we define a *character* in the context of fighting games for the purposes of this thesis. A character is an entity that is controlled by a player and interacts with the game environment. Aesthetic factors like visuals and audio are left out, such that there will be a focus on the game rules and mechanics.

A character and its interactions in an environment can be modeled as a finite state machine[5] (FSM). Common states include standing still (idle), moving, jumping, and attacking, as seen in figure 2.6. The transitions between these states may be caused by player input, timed events, or external environmental factors like getting hit by an opponent's attack. Given this perspective, the *character mechanics* are defined by the states and transitions of a character along with its hurtbox. Their mechanics also encompass the mechanics of their combat actions, which will now be presented.

[4] `https://en.wikipedia.org/wiki/Street_Fighter_II:_The_World_Warrior`

[5] For an overview of finite state machines, visit `https://en.wikipedia.org/wiki/Finite-state_machine`

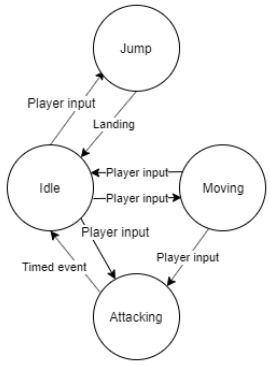


Figure 2.6: Fighting game character modeled as a FSM. This is an example of a subset of states and transitions commonly present in a character. The circles are the states, and the arrows represent transitions.

According to Yu and Sturtevant, there are three main design elements in fighting games [52], namely *timing*, *spacing* and *effectiveness*. *Timing* refers to the timing of combat actions, hereby referred to as just *actions* in this section. An action is split into four states: Input time, lead time, execution time, and lag time. We can expand the attack state of a character, seen in figure 2.6, to visualize the flow of an attack, see figure 2.7. In some fighting games, some actions require a sequence of button inputs, while others may only require a single button input. In the first case, the input time will be longer, whereas in the latter case, the input can be turned into an action immediately. Lead time is the time from when an input has been registered until the execution of the action. In this state, the character is uncontrollable and is usually accompanied by an animation where the character, for example, starts to lift his foot to perform a kick. During the execution time, the effect of an action is active. That is, a melee attack may hit an opponent, a projectile attack is emitted or a defensive action may block an attack. The lag time is similar to the lead time, where the characters settle back to an idle state. The timings are usually short, where the whole duration of one action may generally last for 0.2 seconds to 1 second, but can last longer. The timings differ between actions and are often aligned with the strength or quality of an action.

Spacing refers to the relative distance between the two characters given the range of their attacks. If the hitbox of an attack overlaps with the hurtbox of an opponent character, the opponent will be hit. Thus, spacing is the relative distance between a character's hurtbox and the hitboxes of an opponent's attacks.

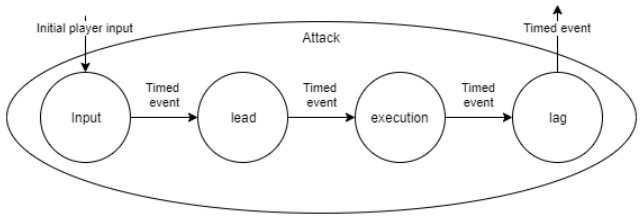


Figure 2.7: The common flow of a fighting game attack represented as a FSM, based on Yu and Sturtevant [52]. The circles represent states while arrows represent transitions. This may be seen as an expansion of the attack state seen in figure 2.6

Effectiveness of an action is the gain of performing the action when it achieves the desired effect. For an attack, it is usually the amount of damage applied to the enemy, while for a defensive action, it could be the time gained where a retaliation attack may be performed. The effectiveness of an action is limited by the counteractions the enemy may perform. An attack might not be effective against a defensive action, while there may exist specific attacks that counter a defensive action.

Yu and Sturtevant also describe a *rock-paper-scissors* pattern often present in fighting games. Combat actions are constructed in a way where they are divided into three types. Each action type is effective against one other type and weak against another type, forming mechanics comparable to the game of rock-paper-scissors.

2.3.2 Fighting games and AI

Fighting games are real-time[6] continuous[7] multiplayer games. The number of actions that can be performed in a given game state, called the action space, is typically large and includes movement and combat actions. The total amount of actions available are in the range of 30 and above, where certain actions can only be performed in certain states. Fighting games usually run at 60 frames per second (FPS), which means that inputs are registered and the game state is updated every $\frac{1}{60}^{\text{th}}$ second. A single battle in fighting games typically lasts

[6]Real-time is the property of a game that seemingly progresses in time like the real world, as opposed to turn-based games. At a computational level, real-time is commonly discretized to game-state updates every $\frac{1}{60}^{\text{th}}$ of a second

[7]A continuous game is not bounded by positions inside a predefined grid as opposed to typical board games like chess

between 5 and 10 minutes. This leaves a lot of possible states and state-action pairs to be explored. These characteristics arise challenges in the context of AI, especially for AI agents adapted to play games, known as *game-playing AI*. Fighting games and other real-time video games have seen less research focus than classical board games like chess and Go. These technical challenges might be a part of the reason.

Fighting games have received a considerable amount of interest in the AI research community. The game industry has created AI-based computer players for their fighting games and academia has seen a growth of interest over the last several years. AI used for playing fighting games in the human player role has clearly been the most contested topic. Dynamic difficulty adjustment has also seen research, mainly focusing on adjusting computer-controlled opponents to the skill level of a player as in the work of Danzi et al. [8] and Ortiz et al. [32]. Player modeling with regards to player type and player skill has been also been seen, such as in the work of Konecný [21]. A player model is to computationally capture characteristics of a player and can be for example be used in the aforementioned task of dynamic difficulty adjustment. Mimicking human players is another research area applied to fighting games, for example by Saini [37].

In the game industry, it is common to ship fighting games with the ability to play against a computer-controlled player. This lets a player play by themselves and practice. Computer agents produced by the game industry are historically based on human-designed rules, denoted rule-based agents. These rules are based on a handcrafted function of the game state to produce an action. The tendencies of a rule-based agent are that it is predictable. This might be a benefit for a developer as it gives full control of how a game is played, but the predictability might be boring for a human player. [8]

The fighting game AI (FTGAI) competition [9] was introduced in 2013 as a platform promote research on game-playing AI in fighting games [28]. Agents are compared on performance, such that the goal is to design an agent to beat other designed agents. The competition has been held yearly since 2013 and has lead to a great advance in fighting game-playing AI.

Early entries to the FTGAI competition were rule-based, similar to the common approach in the industry. This is a type of a reflex agent, with rules that act as state-action functions. Opponent prediction was introduced to counter the rule-based agents [46]. This technique is used to predict the opponent's attack actions

[8] This paragraph and the following are slightly modified versions of paragraphs taken from "Automatic generation of fighting game characters" by Skjærseth and Vinje (fall 2019), section 2.6.1.

[9] The fighting game AI competition's official website: `http://www.ice.ci.ritsumei.ac.jp/~ftgaic/`

and/or movement and taking countermeasures based on the prediction. Other attempts include dynamic scripting, where a set of predefined rules are adopted dynamically by reinforcement learning, as seen in the work of Sato et al. [38] and Majchrzak et al. [29]. The latter method won the 2015 FTGAI competition.

Monte Carlo Tree Search (MCTS) has been a dominant method among the best entries in the FTGAI competition since 2016 when it was introduced by Yoshida et al. [51]. MCTS has been used in many variations, including combinations with predefined rules [15], reinforcement learning [35] and evolutionary algorithms [19]. Some attempts at using Q-learning have been seen [50], but has not performed well.

Reinforcement learning (RL) has seen use outside the context of scripts and MCTS as well. Deep Q-learning (DQL) was explored by Takano et al. [41], and Yoon and Kim explored the possibility of DQN directly on the visual pixel values of a game state [50]. These approaches have not yet proven competitive among the other methods used in the FTGAI competition. A hybrid reward function with RL has also been explored, where different value functions are learned separately, by Takano et al. [42].

Apart from the FTGAI competition, research on game-playing AI has been conducted on other fighting games, including Super Smash Bros Melee (SSBM). where DQL has been applied by Firoiu et al. [12]. The results of Firoiu et al. were promising as their method was able to beat some of the best professional SSBM players in the world.

We have not seen attempts at procedural content generation (PCG) in fighting games. PCG may draw parallels with dynamic difficulty adjustment, as computer-controlled players are adjusted, though game-playing AI is typically not thought of as content within PCG. Player modeling can be used as a basis for tuning game parameters, as pointed out by Konecný [21]. His work focus on modeling player types in a multiplayer game, a game could be tuned to allow an equal challenge to all players based on their characterized player type.

2.4 Interestingness in video games

Our method of generating characters should promote *interesting* characters. But what makes games interesting? Theories on what makes games fun and computational creativity will be briefly presented in this section. This gives the reader a theoretical background on the concept of interestingness and how it can relate to fighting game characters.

Fun in video games

Malone published a paper on what makes video games fun back in 1980 [30]. He presented three characterizing aspects of enjoyable video games: *Challenge*, *fantasy* and *curiosity*. *Challenge* is described as having clear goals, and uncertainty of the outcome of a game. There should be an appropriate difficulty level, where success is possible, but by no means guaranteed. *Fantasy* refers to the level of fantasy in the game environment. *Curiosity* means having a certain amount of information hidden so the player stays curious and wants to keep exploring the game. A more recent example of research on what constitutes fun in video games is the paper on GameFlow, by Sweetser and Wyeth [40]. GameFlow describes eight elements that characterize a *flow* state when playing a game. The concept of flow was earlier introduced by Csikszentmihalyi [7] and adopted in the domain of games by Sweetser and Wyeth. The flow state is identified by being intensely engaged in a task, feeling a sense of control, and losing the sense of time and the real world. There are four of the eight elements presented by Sweetser and Wyeth we find especially relevant since they are tightly connected to the mechanics of a game, namely *Challenge*, *player skills*, *control* and *clear goals*. Challenge and player skills are closely related, where the challenge must adapt to the skill of a player. Control indicates that a player should feel a sense of control over his or her actions in a game. Clear goals consider the presence of goals, and that they are easily understood by a player.[10]

Computational Game Creativity

Computational creativity is a computational field concerned with creativity in a computational context and mainly relates to visual art, narrative, and audio. Liapis et al. [24] presents how computational creativity can be seen in the context of generating content in a game[11]. Liapis et al. base their work on three essential properties of creativity: *Novelty*, *quality* and *typicality*, presented by Ritchie [36]. The three properties are stated as follows:

- Novelty: "To what extent is the produced item dissimilar to existing examples of its genre?"
- Quality: "To what extent is the produced item a high-quality example of its genre?"

[10] This subsection is a modified version of the subsection "What is fun?" found in section 2.2, page 12 of "Automatic generation of character mechanics in fighting games".

[11] This subsection is a modified version of the subsection "Computational Game Creativity" found in section 2.2, page 13 of "Automatic generation of character mechanics in fighting games".

- Typicality: "To what extent is the produced item an example of the artifact class in question?"

We will further elaborate on these three aspects in the domain of fighting game characters. In this domain, the *item* would be a character and the *genre* would correspond to other fighting game characters.

2.4.1 Interestingness of fighting game characters

Novelty is the property of a character that corresponds to being different from other characters. The other characters are characters already defined in a given game or basic character design seen in the fighting game genre as a whole. Novelty also corresponds with the curiosity aspect presented by Malone [30].

Quality in the domain of games is tightly related to what makes games fun, as presented previously. For a generated fighting game character, it should promote the right amount of challenge. This relates to how hard a character is to play. It also relates to the concept of balance in multiplayer games. Characters should have a similar level of strength, such that players have a fairly even chance of winning. A character should also promote a flow state of the player. The theories of fun presented previously will work as a basis for what we consider quality.

Typicality refers to characters being familiar to each other and the genre, i.e, the fighting game genre. It is partially achieved through the game environment specified for a given game and should contain rules and mechanics that promote typicality toward fighting games. Typicality between different characters is also important. If a player has learned how to play a game with a single character, the player should be able to adapt to another character with parts of the obtained knowledge. Typicality will promote the flow state of a player for new characters as well.

The concepts presented in this section form our definition of interestingness in the perspective of fighting game characters: Novel, balanced, and fun.

2.5 Simulation based evaluation

Simulation-based evaluation is the process of evaluating a generated artifact by loading it into a game and let computer players play the game with the given artifact. Measurements of the played game are calculated and used for the evaluation of the artifact. The process of playing a game with the generated artifact will be referred to as a *simulation*. This approach might be used for evaluating the *quality* of content in search-based PCG (see section 2.1). Simulation-based

evaluation is used instead of *direct* or *interactive* evaluations, as they might not be suitable, as discussed in 2.1.

Simulation-based evaluation consists of three components:

- A **game** that can load a generated artifact.
- **Computer players** that give inputs to the game. The computer players take the role that human players would normally have in a game.
- **Criteria** that evaluate some aspect of a game through measurements with respect to the desired quality of a generated artifact.

It should be noted that these principles also apply to other simulation domains, apart from games.

The desired *quality* of the content from a designer's perspective will decide suitable criteria for the evaluation. The desired quality is dependent on the game domain and the purpose of the content that is generated. An example is to evaluate if a player can reach a goal in a game with a generated level. A computer player may be designed to move along the shortest path to the end of the level, and a single criterion may be used that is based on whether or not the end was reached.

Human-perceived enjoyment of an artifact is a more complex quality criterion. This can be seen as the "ultimate" criteria for entertainment games, as the overall goal is to enjoy the game. Defining criteria to capture a human's experience of a game is, however, not a straight forward task. Relevant theory for understanding a player's experience related to enjoyment through fun and balance was presented in section 2.4. The concepts presented are not quantifiable. A computational model to capture the player experience is therefore needed. We call this *player experience modeling*, which refers to a model based on a set of criteria. This will be further discussed in 2.5.1.

How to design the computer player is also dependent on the desired quality of generated content. In the case where the desired quality is the player experience, the controller should play similarly to a human player. We will further look at how such a computer player can be designed in section 2.5.2.

2.5.1 Player experience modeling

When evaluating game artifacts by simulation, we need a way to assess the quality of the artifact. The quality, as discussed earlier, depends on the goal of the generation process and the type of content. We will here focus on the quality given by the *player experience*. That is the human experience of the content.

Note that the quality is based on simulation-based evaluation, where a computer player is applied instead of a human player.

When generating content with the goal of satisfying human players, it would be optimal to have a computational model of human player satisfaction based on the outcome of a computer player. This is a rather hard model to construct, as it ideally should model the brain of a human and track its emotions. Since this is not practical, we use simpler heuristics of player experience for this task.

Previous work on quantifying player experience has been based on theories regarding what makes games fun and enjoyable, as presented in section 2.4. These theories are hard to quantify, and there is no general computational model to capture player experience. Previous work has defined criteria as heuristics for player experience, but are heavily dependant on certain game genres, where domain knowledge is applied. Applications of simulation-based evaluation will be further presented in section 3.3.

The type of content evaluated will impact how the player experience can be modeled. Generated content may be essential or optional as seen in section 2.1. The generated content can greatly impact how a game is played. It can even be a complete set of game rules, as seen in the work of Brown and Maire [4], where puzzle game rules were generated in a domain specific syntax. The more impact the generated content has on a game, the harder it is to constructing player experience heuristics.

Criteria for measuring player experience were grouped into three categories by Brown and Maire, where two categories were related to simulation-based evaluation and proved the most significant [4]. The two categories are *quality* criteria and *viability* criteria. Quality criteria aim at evaluating player experience based on how a computer player *interacts* in a game. The optimal quality criteria would be a human player that reason about a game and the human could evaluate their experience. That is however not possible with a computer player. An approximation can be made by assessing the actions of the player. Viability criteria are based on the *outcome* of several played games. Viability criteria are usually easy to compute, as the outcome of a game is trivial to record. It may correspond to which player won and the length of the game, as seen in the work of Brown and Maire. Quality criteria, on the other hand, are harder to construct as they estimate the player experience during a game. These categories can be seen in figure 2.8.

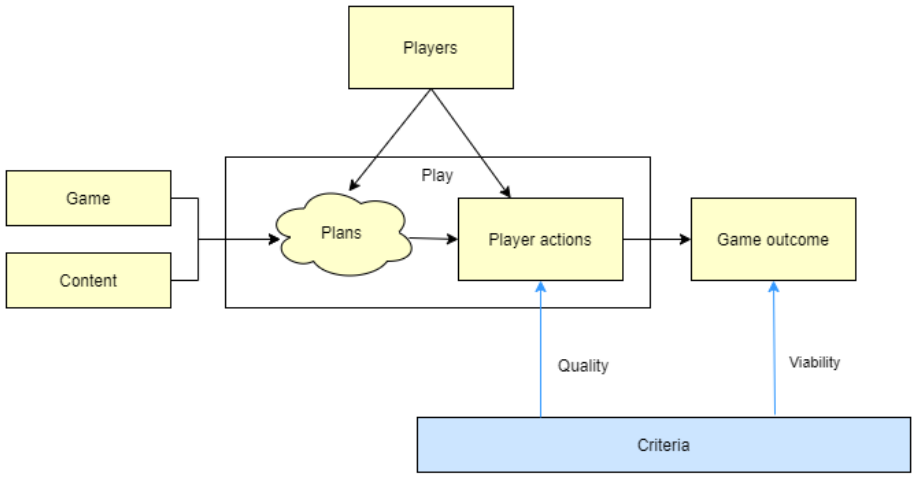


Figure 2.8: Categories of criteria in simulation based evaluation. The criteria are divided into two categories that are based on measurements through simulation based evaluation. Quality criteria considers the behaviour of a player and viability criteria considers the outcome of simulated games. Based on the work of Brown and Maire [4]

2.5.2 Computer controlled players

Simulation-based evaluation uses a computer-controlled player, or *computer player*, that replaces the role of a human player. The computer player allows content to be evaluated through play, without the need for human player interaction. This is desired in the phase of game development before there exist players to play a given game.

The computer player can essentially be based on any game-playing AI technique. We have seen examples of game-playing AI in the context of fighting games in section 2.3.2. The algorithm used depends on the content that is evaluated and the defined content quality that constitutes the objective for the evaluation.

We focus on simulation-based evaluation to evaluate the quality of *player experience*. In this case, considerations should be given based human qualities of the computer player adopted. The first aspect to consider is whether the computer player can play at the same level as humans. Real-time video games feature reaction time and input accuracy as mechanical aspects of the game. This is especially true for fighting games, which often are fast-paced. Computers are typically good at fast and accurate computation, compared to the human brain. Thus a computer player could abuse its computational speed to act much faster than a human. That would not seem human-like. This has been handled within research on game-playing AI for fighting game by introducing state perception delay. That is, delaying the perception that the computer player has on the game by a fixed amount, d. Even though computer players may react immediately, with this delay, the player would react immediately to what happened in the past. The delay, d, is usually defined based on an estimation of human delay time. The game FightingICE, used in the fighting game AI competition, issues a static perception delay with d =0.25s [28]. With a delay like this, a computer player is noticeably weaker, as presented by Asayama et al. [1] for the FightingICE game. Asayama et al. also presented a method for predicting the opponent to partially cope with this delay. Experiments have also been conducted to model noise in human input [43].

Creating human-like computer players have also been conducted in the context of other video games. For example the work of Schrum et al. [39], Bryant [6] and Phuc et al. [34].

Chapter 3

Related work

This chapter contains summaries of research closely related to our research goal and research questions. Relevant research within search-based procedural content generation (search-based PCG) is first presented in section 3.1. The class of evolutionary algorithms has seen a lot of use in Search-based PCG, and one such algorithm, novelty search, is further presented in section 3.2. In section 3.3, applications of simulation-based evaluation in games are presented. There will be a focus on the evaluation of player experience.

3.1 Research on search-based PCG

This section evaluates research using similar methods to ours, such as search-based PCG, and how it can be utilized in different game domains.

Research on PCG, and search-based PCG (presented in section 2.1) in particular, has seen a sharp increase in academic interest over the last two decades [45]. By reviewing the literature, we found several examples from recent years where evolutionary search was implemented for generating content. As recently as in January 2020, Zafar et al. published a study on generating general game levels using search-based PCG [53]. The aim of the study was to generate challenging and aesthetically appealing game levels in a more general way than in previous studies, where there often has been only one game used for experimentation. Their conclusion was that "The results indicated that the levels are aesthetically more appealing and challenging". To evaluate levels, Zafar et al. used a direct fitness evaluation of the levels instead of using simulation-based testing. This approach is suitable when the content is appropriate to evaluate purely based on

its properties, as discussed in section 2.1. An example of research on search-based PCG using simulation-based evaluation is the work of Liu et al. [27]. Their study evaluated evolutionary search to generate levels of the Tower Defense genre[1]. To evaluate the levels, Liu et al. implemented an AI agent based on reinforcement learning to play through the generated levels for evaluation. Their experimentation yielded three generated game levels, later tested by humans and considered to be enjoyable.

An example of research more closely related to generating character mechanics is the research of Hastings et al. [14]. Neuroevolution was applied to generate weapons for the game *Galactic arms race*. In Galactic arms race, the player controls a spaceship and engages in firefights with other players online. The generated weapons were put online for players to use, and the fitness of the weapons was calculated based on how frequently they were used by the players.

A similar approach was taken by Pantaleev [33]. Pantaleev searched for new abilities for characters in a small text-based role-playing game[2] he had developed. Each player could choose a number of abilities from a given set of abilities for their character. The evolved content was put online for players to test, and the fitness of an ability was directly evaluated based on the number of times it was chosen by players. The content Hastings et al. and Pantaleev generated is more similar to ours, since the weapons and abilities available for the player could be viewed as a subset of the character mechanics. The most significant difference between these last two examples and the method proposed in this work is in the evaluation of the generated content. The applied fitness functions in the work of Hastings et al. and Pantaleev can be described as direct interactive fitness functions. They required human interaction and can thus be classified as a form of mixed-initiative PCG. In contrast, this work aims to utilize simulation-based evaluation, and thus remove all need for human involvement during the generation process.

Novelty search is a variant of evolutionary algorithms that have also been used in the context of search-based PCG. The following section will present the algorithm.

3.2 Novelty Search

Novelty search is an evolutionary algorithm that heavily focuses on exploration, and presented by Lehman and Stanley [22]. Novelty search promotes diversity of individuals through generations, such that solutions represented by the individu-

[1] `https://en.wikipedia.org/wiki/Tower_defense`

[2] For info about the role-playing game genre, visit: `https://en.wikipedia.org/wiki/Role-playing_game`

als are novel, relative to each other. Novelty search has been successfully applied as a method of PCG, as seen in the work of Liapis et al. [25]. As such, we will cover the Novelty search algorithm, and how it was applied by Liapis et al.

There is a trade-off to be made between exploitation and exploration for evolutionary algorithm, as discussed in section 2.2. Finding the right balance between exploitation and exploration depends on the problem at hand.

The techniques briefly discussed in section 2.2 include deterministic crowding, sharing, and crowding. These techniques have proven useful to avoid the problem of premature convergence in many applications of evolutionary search by boosting exploration.

When great diversity of solutions is desired, exploration can be taken to the extreme. This is the case for Novelty search. Novelty search abandons an objective entirely and focuses merely on exploring the search space by encouraging *novel* individuals to be generated. The higher distance between an individual an the other explored individuals, the more likely this individual is to be explored further. Abandoning the objective and the fitness function may seem very counter-intuitive, but Lehman and Stanley argued that objective functions themselves can actively direct searches to dead ends. To evaluate the hypothesis of novelty being more suitable for certain domains, a comparison between NeuroEvolution of Augmenting Topologies (NEAT)[3] with fitness-based search and NEAT with novelty search was performed on a two-dimensional maze navigation task. The two different maps are depicted in figure 3.1. The fitness function was defined as $f = b_f - d_g$, where b_f is a bias constant and d_g is the distance to the goal from the maze-navigating robot. The results of the experimentation showed that *using novelty was three times faster*, as well as a much more reliable way to reach the goal than using fitness.

Lehman and Stanley also experimented on an unenclosed maze, as seen in figure 3.2. In this case, fitness-based NEAT and NEAT with novelty were both inefficient in solving the maze, and one method did not significantly outperform the other. NEAT with novelty search solved the maze five out of 100 runs, whereas fitness-based NEAT solved the maze in two out of 100 runs.

The research of Lehman and Stanley has taught us that novelty can be a more suitable mechanism for driving the search than fitness in certain domains. The extent to which we value exploration is highly dependent on the domain of the problem. The question we need to answer is whether our domain of searching for character mechanics is more similar to the domain of the mazes in figure 3.1, the

[3]NEAT is beyond the scope of this thesis, but a detailed explanation can be found here: `http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`

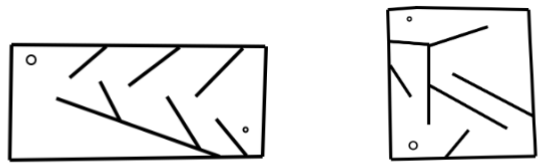


Figure 3.1: The two mazes used by Lehman and Stanley for comparison between objective based fitness and novelty search. The small circles represent the goal, and the large circles represent the starting point.

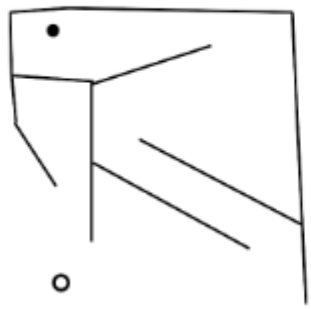


Figure 3.2: An unenclosed maze used as a search domain

unenclosed maze in figure 3.2, or a different domain, where greedily approaching the objective is the best path.

The paper "Constrained Novelty Search: A Study on Game Content Generation" published by Liapis et al. [25] explored the use of Novelty search for generating game content. The aim of their study was to generate a diverse set of 2D video game levels. In their study, Liapis et al. tested two different implementations of novelty search for game levels that included the constraint of being a solvable level. The two algorithms implemented by Liapis et al. were called *Feasible-infeasible novelty search* (FINS) and *feasible-infeasible dual novelty search* (FI2NS). In both algorithms, they divided the search space between *feasible* and *infeasible* individuals. A feasible individual is an individual that satisfies the given constraints, whereas infeasible individuals do not satisfy the given constraints. Feasible parents may generate new infeasible offspring and vice versa. The goal of novelty search is to produce a set of unique and diverse solutions within a given search space. To avoid generating similar solutions across different generations, the novelty search algorithm keeps track of previous novel solutions in the *novel archive*. The experimentation compared FINS and FI2NS, and was performed on a set of small, medium, and large game levels. In conclusion, Liapis et al. stated that "both FINS and FI2NS seem particularly useful for the procedural generation of game content, which requires its generated artifacts to be diverse yet playable".[4]

The difference between FINS and FIN2S is depicted in figure 3.3. This research is relevant to generating interesting characters because it focuses on generating a diverse set of content solutions. As seen in section 2.4, novelty is an important factor. In our context, *interesting* could be seen as a constraint, whereas *diverse* could be seen as the novelty of generated characters. A significant difference between this research and our own is that interestingness is a much more complex constraint than the constraint put on each game level in the game used by Liapis et al. Whereas we need to decide on an appropriate balance between fitness and novelty of our solutions, novelty was the main focus of the FINS and FIN2S algorithms, as long as the levels were solvable.

[4]This paragraph is a slightly modified version of a paragraph taken from "Automatic generation of fighting game characters" by Skjærseth and Vinje (fall 2019), page 25, section 2.4.

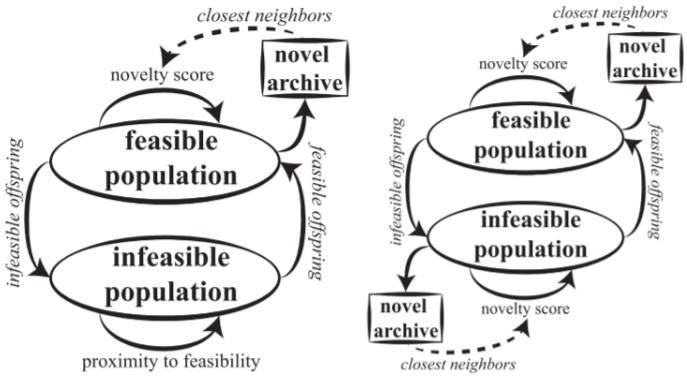


Figure 3.3: The two constrained novelty search algorithms, FINS and FIN2S, implemented by Liapis et al. [23]. In FINS, the infeasible population evolves to minimize the distance to feasibility. In FIN2S, both the feasible and infeasible population evolve toward novelty using a novel archive.

3.3 Applications of simulation based evaluation

In this section[5], we will present related work on simulation-based evaluation with the goal of evaluating player experience. We have not found related work on simulation-based evaluation in fighting games, so other game domains will be explored. Based on the background of simulation based evaluation presented in section 2.5, we will focus on criteria on simulated games that are heuristics on player experience. The computer player applied for simulation will also be mentioned.

A paper on generating board game rules was presented by Browne and Maire [4]. They present a framework, called Ludi, for generating two-player puzzle games rules, presented on a fixed grid board. The games produced by Ludi were similar to board games like Tic-tac-toe, Checkers, and Go. The goal of the Ludi system was to generate games that were novel, interesting, and publishable, meaning that they can be directly published without human interaction. The Ludi system is divided into modules. The module named the *criticism module* is responsible for evaluating the quality of a given game. This module tries to evaluate how interesting a game is to a human player, similar to our notion of player experience, as seen in section 2.5.

[5]This section contains paragraphs that are slightly modified versions of paragraphs taken from "Automatic generation of fighting game characters" by Skjærseth and Vinje (fall 2019), section 2.5.1.

Brown and Maire use simulation-based evaluation with human-designed criteria that capture different aspects of a simulated game with regards to the player experience. 57 metrics that are presented by Browne [5] were initially considered. Experiments were conducted with human playtesters to evaluate the correlation between the 57 metrics and the perceived human playing experience of 79 already existing games. The 16 most important metrics where then chosen to be used for fitness evaluation in an evolutionary search for better games. Browne and Maire presented three categories of criteria, as discussed in 2.5.1. *Quality* and *viability* criteria were presented, while *intrinsic* criteria is the third category. Intrinsic criteria are based directly on the representation of the generated puzzle rules in the work of Brown and Marie. Intrinsic criteria are not used for the method presented in our work but are included here to give a broader picture. Of the 16 criteria that were extracted by Browne and Marie, 4 were in the intrinsic, 10 in quality, and 2 in viability category. The significance of the criteria was examined by the error in correlation with human preferences when each criterion was removed. The intrinsic criteria were the least significant of the 16 presented, while the two most significant were the quality criteria *uncertainty* and *lead change*.

A state evaluation function is central in multiple of the criteria and is a function that yields a score to each player based on their current state in the game. Following, we present 6 quality metrics and 3 viability criteria from the work of Browne and Maire:

Quality criteria:

- Uncertainty: Evaluates the uncertainty of a simulated game by looking at the state evaluations of the winning player, and its advantage during a game. The advantage should not be strictly growing over the whole game. If it does, the game could easily be perceived as boring, with a player falling behind early is ensured to lose.
- Lead change: The tendency for the lead to change between the two players, given by the state evaluation function.
- Killer moves: The tendency of killer moves, that is, moves that drastically change the state evaluation for a player.
- Drama: Evaluates the drama given by the amount of time the winning player is in an unfavorable state, given the state evaluation function.
- Momentum: The steady increase in state evaluation over n consecutive moves. Browne and Maire considered $n = 1, 2, 3$ as separate criteria. The version with $n = 1$ was the most significant.

- Board coverage: The number of board positions that have been visited by any game piece over a game.

Viability criteria:

- Duration: Duration is based on the length of simulated games. It is given by the deviation of the game length deviation from a preferred game length set by the designer.
- Balance: This criterion considers the outcome of a set of games with respect to the players winning equally often.
- Depth: *Dumb* computer players (that represent human beginners) playing against each other and *smart* computer players (representing experienced human players) playing against each other should both be balanced, beginners playing against experienced players should not be balanced. This proves that players have the possibility to play better as they learn the game.

The quality criteria board coverage and the viability criteria *balance* and *depth* are not present in Brown and Marie's 16 most significant criteria. Their criteria were evaluated in the domain of board games but might find a correlation with player experience in other domains, e.g., fighting games.

The simulation-based evaluation uses a computer player and is based on self-play. That is, the same AI algorithm is used for both players. The player uses minimax tree search with alpha-beta pruning. The state evaluation function used for the criteria is also used for the player for non-terminal states. It consists of a set of 20 hand-made evaluations of a game state that Browne and Marie call *advisors*. The advisors are weighted, and the weights are learned over multiple play troughs of a game that is evaluated. This is similar to dynamic scripting, presented in the context of fighting games in section 2.3.2.

The General Video Game AI (GVGAI) competition[6] lately opened two new tracks in their competition introduced by Khalifa et al. The level generation track was introduced in 2016 [18], and the rule generation track in 2017 [17]. In both papers, they introduce three example generation techniques, where one from each paper featured a search-based approach with simulation-based evaluation. It should be mentioned that the games presented in the GVGAI competition's level and rule generation tracks feature single-player games. In both simulation-based approaches, the simulation-based evaluation used a *learnability* criterion. They look at the difference in score outcome of one *dumb* and one *smart* computer player. Both generation techniques also based their fitness on the number of

[6]More on the GVGAI competition can be found at `http://www.gvgai.net/`

unique interaction events that happened during a simulation. The technique for level generation further defined a set of feasibility criteria, with *solution length* being one of them. This criterion is based on a minimal length of playtime by every computer player. Solution length corresponds to a *viability* criterion.

Drageset et al. further present a set of criteria, calling them *factors*, that are proposed for evaluation in the GVGAI level generation track [10]. The hypothesis was that the presented criteria should provide a stronger evaluation compared to the criteria for the search based generator given by Khalifa et al. [18]. Khalifa et al. also used three computer players of different level in their evaluation, where one of the players were designed to do nothing. The relevant criteria will be presented, and we divide them into *quality criteria* and *viability criteria* as follows.

Quality criteria:

- Danger factor: Considers how close the character is to death averaged over a game. This is measured by random rollouts of length n, that is, simulating the game n frames forward and seeing how many rollouts lead to death. n is a hyperparameter.
- Danger rate factor: Similar to the *danger factor*, but measures the amount of time the character is close to death. That is, at least one rollout leads to death.
- Interaction factor: How many interactions take place between the player-character and its attacks, and the other characters and environmental objects.
- Unique interaction factor: Similar to *interaction factor*, but only counts unique interactions.

Viability criteria:

- Win factor: Considers how much better the best computer player performs than the two other players with regard to win percentage.
- Score factor: Considers how much better the best computer player performs compared to the other two players with regard to the game score.

Chapter 4

Methodology

Our goal is to generate interesting fighting game character mechanics. Interesting is defined to mean novel, fun and balanced, as discussed in section 2.4. This chapter presents a system for generating such characters in the fighting game Sol, a fighting game developed by the authors. The system will be used to conduct experiments to examine research question RQ1, RQ2, and RQ3 from section 1.2. There are several parameters and variants of the suggested system, and these will be presented in this chapter. Chapter 5 will cover experiments to determine the value of those parameters and to compare the different variants.

The system we developed is based on the related work presented in chapter 3. The approach is within the category of search-based procedural content generation (see section 2.1). A feasible-infeasible 2 population constrained novelty Search (F-I 2pop novelty search) algorithm, as seen in section 3.2, will be used as the search method. Novelty search is a form of evolutionary algorithm that highly prioritizes exploration, where the objective is the novelty of the evolved characters. Constraints are given on characters by criteria, evaluated through simulation-based evaluation, inspired by methods presented in section 3.3. The constraints aim to capture fun and balance that a player would experience. Characters that fulfill the constraints are referred to as *feasible* and those that violate the constraints are *infeasible*. Two variants of F-I 2pop novelty search are implemented in the system and will be discussed later.

A desired *quality* of generated content must be defined, as seen in section 2.1, and an evolutionary algorithm must be designed with respect to that quality. We define the quality of generated characters as novel, balance, and fun. Through F-I 2pop novelty search, this quality can be focused. Novelty is the objective of

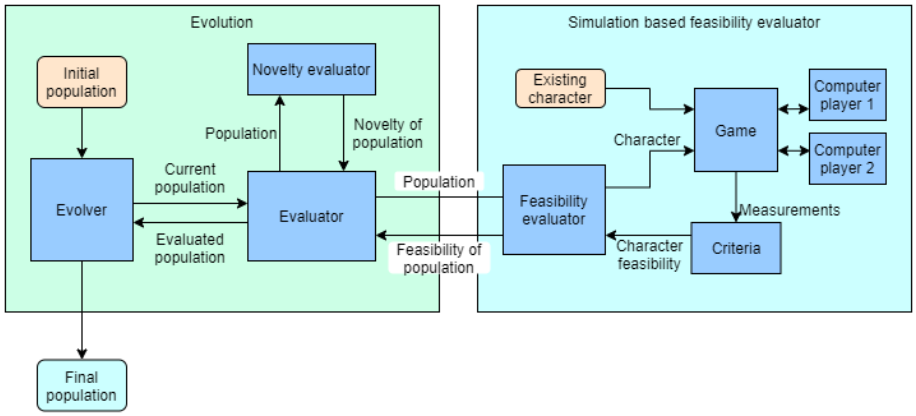


Figure 4.1: The generation system architecture: The green box on the left represents the evolutionary algorithm performing the search for character mechanics. The evolution is split into several components with different responsibilities. The blue box on right represents the simulation-based feasibility evaluation module. This service is responsible for spawning game instances, controlling the characters' behavior during games, and measure the outcomes of games.

novelty search, such that characters are optimized for this aspect. Constraints on balance and fun ensure that a minimum level of these aspects is present for all generated characters. Our approach to computationally quantify these three aspects will be presented later.

A modular overview of our system is shown in figure 4.1. It is divided into two main modules, *Evolution* and *Simulation based feasibility evaluation*. Evolution is based on constrained novelty search. An *initial population* of characters is first generated. Each character is then evaluated according to constraints and the novelty objective in the *evaluator*. First, constraints are handled through the simulation-based feasibility evaluator. Each character is assessed individually and determined either feasible or infeasible. This is done by starting a game, where the character to be evaluated is loaded with one of four existing characters, priorly designed for the game. Both characters are played by a *computer player*, and measurements are taken of the game, with regards to the character that is evaluated. These measurements are used by a set of *criteria* that are either fulfilled or violated. The character is classified as feasible if all the criteria are fulfilled in the *feasibility evaluator*. The characters are then inserted into the feasible or infeasible population. A novelty score is given to each individual of the feasible and infeasible population in the *novelty evaluator*. Parent selection is performed, and

crossover is applied to produce offspring that are mutated. The new population with the offspring are then carried over to the new generation. The output of the system is the *final population* of characters. The source code for the system is available at `https://github.com/sol-ai-master/solai_project`.

The Sol game is presented in section 4.1. We use a parameterized representation for the characters, discussed in section 4.2. The two variants of the F-I 2pop novelty search that are used in our system are presented along with the initial population, parent selection method, and evolutionary operators in section 4.3. The simulation-based feasibility evaluation is discussed in section 4.4.

4.1 The Sol fighting game

The Sol game was originally developed in 2017 by the thesis authors, Eirik Skjærseth and Harald Vinje. Sol is a simple, multiplayer fighting game that can be played with two players in the 1 vs 1 mode and four players in the 2 vs 2 mode. We will only consider the 1 vs 1 mode for this work. A single game lasts for about 1 to 2 minutes. There are four main elements in the game; player-controlled characters, hitboxes, holes, and walls, as seen in figure 4.2. The figure presents two characters that have both used a projectile attack, such that a projectile hitbox is emitted. The characters have a corresponding *hurtbox* and offensive abilities have a corresponding *hitbox*, as described in section 2.3.1. The stage area is bounded by holes and walls. The walls are impassable objects for the characters, and holes are objects where a character is eliminated upon colliding. A player interacts with a character by four movement keys (up, down, left, right), that gives a total of 8 possible movement directions. Three ability inputs are available, used to perform attacks, and each character is aimed toward the mouse cursor of the player.

The goal of the game is similar to that of the *Super Smash Bros* fighting game franchise seen in section 2.3. When an attack hits an enemy character, the enemy takes damage that is accumulated for every hit. An attack also applies a knockback force to the enemy given by a function based on the enemy's accumulated damage. Each character has three stocks (lives). When colliding with one of the holes in the map, the character loses one of its stocks. The first to eliminate all of the opponent's stocks is the winner of the game.

The game was created with a roster of four available characters: *Frank*, *Schmathias*, *Brail* and *Magnet*. A character has a unique set of three combat actions, that we refer to as *abilities*, that are either of the type *melee* or *projectile*. These attack types are common for fighting games and were presented in section 2.3. Character specifics will be further discussed in section 4.2. The four existing characters

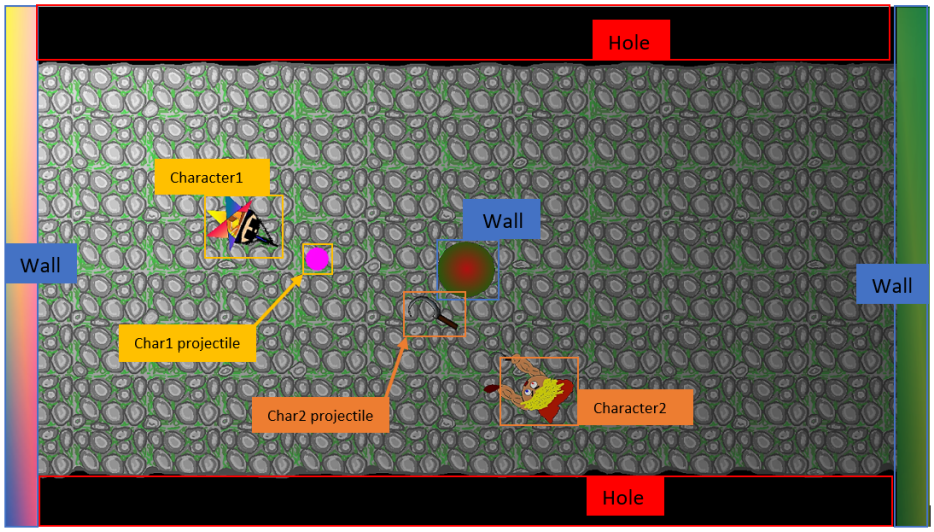


Figure 4.2: Elements in the Sol fighting game. Impassable walls are outlined in blue, and holes are outlined in red. The holes are areas where a character dies (loses a stock) upon collision. Two characters are displayed, outlined in different shades of orange, both performing a projectile ability.

were manually designed and tuned to be diverse, fun, and balanced. They were created by the thesis authors.

The game was altered for the purpose of this work. The game was made able to load characters in a given representation, further discussed in section 4.2. The original graphics seen in figure 4.2 were simplified, where objects are represented by a single color. As this work doesn't focus on the visual aspect of generated characters, a simple visual representation was implemented. The simple interface is later shown in the context of user studies in chapter 5. The game was also made able to run without graphics for the purpose of simulation-based evaluations.

The game has many similarities to existing fighting games seen in section 2.3, but with simplified mechanics. The game is in a top-down perspective. The *height* dimension usually seen is removed, such that jumping is not possible. This removes some complexity compared to other fighting games. Defensive abilities are omitted, such that offensive attacks are the only possible combat actions. Combat is thus simplified, but the *rock-paper-scissor* effect presented in section 2.3.1 is still present as the offensive attacks have different timing, spacing and effectiveness.

4.1.1 Technical aspects of Sol

When Sol is played by humans, they perform actions with the keyboard and computer mouse and get feedback on the game state through a graphical computer display. This is impractical for computer players, and when measuring the game. Thus the game implements a programmatic interface for both perceptions of the game state and game inputs. The game state is given by objects in the game, with positions, size, and other properties related to the characters and their abilities. The inputs are given by nine values. Four boolean movement inputs: mv_l (move left), mv_r (move right), mv_u (move up), mv_d (move down), three boolean ability inputs: ab_1, ab_2, ab_3, and two aim inputs that are floats corresponding to the aim position coordinates: aim_x and aim_y. The inputs correspond to inputs available to a human player through the keyboard and the mouse.

The game normally runs in 60 frames per second (FPS), i.e., state updates happen every $\frac{1}{60}^{\text{th}}$ second. That is common for video games and is enough for players to believe the game is continuous in time. The computational time used for one such state update is much lower than $\frac{1}{60}^{\text{th}}$ second, such that the game can run faster if human perception and reaction time is not a concern. That is the case when using computer players to play the game.

Speed tests of Sol were conducted to assess the speed of which the game could be run. The test were performed on an Intel(R) Core(TM) i5-8300H cpu, a relatively new laptop processor. 1000 games were played by two computer players. The results are as follows:

Timing	Average (ms)	Min (ms)	Max (ms)
Game update	0.0161	0.0078	33.0235
Players update	0.0038	0.0020	15.4066
Total update	0.0252	0.0130	33.0371

Table 4.1: Speed test of Sol with two computer players on an Intel(R) Core(TM) i5-8300H cpu. Computational time of a single game update, computer players update and the total update time are given. The average time over each state update from 1000 games are given in milliseconds, as well as the minimum and maximum update time.

The average game length was 7649 updates, which corresponds to 128 seconds of human playtime, with a maximum of 15825 updates and a minimum of 2381. This means that a game that takes 2 minutes for a human, would take about 116 ms on average when running as fast as possible. With the computer players, it would be an average of 143 ms.

4.2 Character representation

In order to search and generate new characters, we need a way to represent a character, as discussed in section 2.1. The representation will be the genotype in the context of an evolutionary algorithm, where considerations must be given. The search space is also defined based on the representation.

Fighting game character mechanics are usually defined by similar rules, where the character distinction lies in numerical properties. These properties will be used for our representation of a character, and are given as key-value pairs. Character mechanics can be seen as the rules that determine how a character can behave alongside the properties of those rules. The character rules are defined in the game, but changing their properties greatly alters how they behave, thus, their mechanics.

The complete set of properties of one of the existing characters in Sol, namely *Frank*, can be seen in listing 1. The representation of all four existing characters (seen in section 4.1) are given in Appendix A. The "name" properties of the character and the abilities are meta-attributes and not significant to the representation. The remaining properties solely correspond to the mechanics of the character, where aesthetics such as graphical and audio are left out.

All characters in the Sol game are represented as circles, which fairly well captures the physical shape of a character seen from the top. The "radius" and "moveVelocity" properties describe the size and moving speed of a character, respectively, and are inherent to all characters. They are referred to as the *character body properties*, and can be seen in figure 4.3. In the figure, these properties belong to the red character body object.

```
{
  "name": "Frank",
  "radius": 32,
  "moveVelocity": 500,
  "abilities": [
  {
      "name": "rapid shot",
      "type": "PROJECTILE",
      "radius": 8,
      "distanceFromChar": 32,
      "speed": 1200,
      "activeTime": 30,
      "startupTime": 2,
      "executionTime": 0,
      "endlagTime": 2,
      "rechargeTime": 30,
      "damage": 100,
      "baseKnockback": 200,
      "knockbackRatio": 0.5,
      "knockbackPoint": -128,
      "knockbackTowardPoint": false
    },
    {
      "name": "hyper beam",
      "type": "PROJECTILE",
      "radius": 20,
      "distanceFromChar": 32,
      "speed": 1500,
      "startupTime": 15,
      "activeTime": 120,
      "executionTime": 1,
      "endlagTime": 10,
      "rechargeTime": 120,
      "damage": 300,
      "baseKnockback": 400,
      "knockbackRatio": 0.8,
      "knockbackPoint": -256,
      "knockbackTowardPoint": false
    },
    {
      "name": "puffer",
      "type": "MELEE",
      "radius": 98,
      "distanceFromChar": 0,
      "speed": 0,
      "activeTime": 2,
      "startupTime": 8,
      "executionTime": 2,
      "endlagTime": 8,
      "rechargeTime": 180,
      "damage": 20,
      "baseKnockback": 1300,
      "knockbackRatio": 0.1,
      "knockbackPoint": 0,
      "knockbackTowardPoint": false
    }
  ]
}
```

Listing 1: An example character configuration from the Sol game for the existing player “Frank”. These properties constitute the search domain.

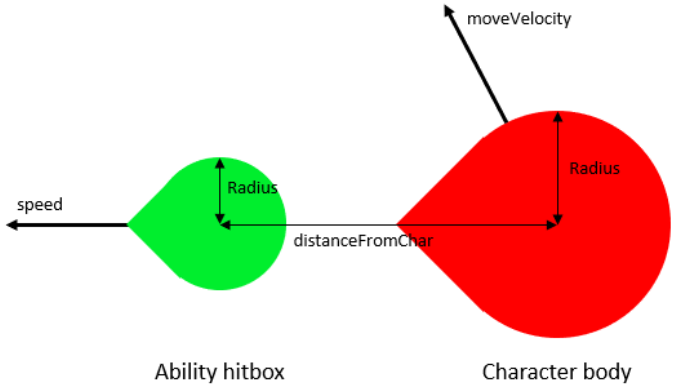


Figure 4.3: Physical properties of the Sol character representation. The red pointy circle corresponds to a character object, while the green pointy circle corresponds to the hitbox of an ability. Physical properties given in the character representation are shown. "speed" and "moveVelocity" influence, respectively, how fast a projectile hitbox and character can move

All characters also have a set of three abilities, which have their own set of properties. The "type" attribute of an ability has two possible values, namely *MELEE* and *PROJECTILE*. The properties "radius", "distanceFromChar", and "speed" determine physical attributes of an ability, and are shown in figure 4.3. They correspond to the green hitbox object. All ability hitboxes are represented by a circular physical shape. Recall the common fighting game design patterns, *spacing*, *timing* and *effectiveness* presented in section 2.3.1. The three physical properties are related to spacing. The properties "startupTime", "activeTime", "executionTime", "endlagTime" and "rechargeTime" are related to timing. The reminding properties; "damage", "baseKnockback", "knockbackRatio", "knockbackPoint" and "knockbackTowardsPoint" are related to effectiveness. The effectiveness properties determine how much damage an ability applies to an opponent and how knockback is applied.

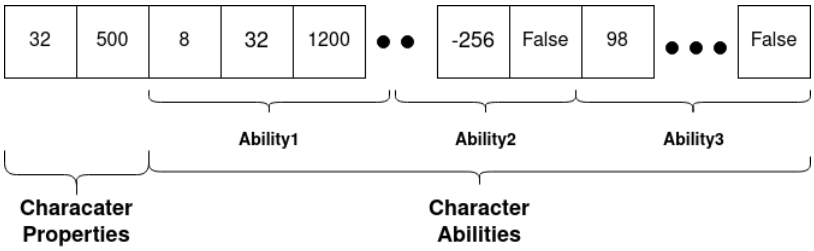


Figure 4.4: The genotype representation of the existing character Frank, seen as a list of numbers and booleans.

The representation can be thought of as a list of numbers and booleans, as seen in figure 4.4. This might be a more familiar representation in the context of evolutionary algorithms. However, the evolutionary operators defined in our method are based on metadata given by the representation structure and properties. Especially the distinction between the character body properties and the properties given for the three different abilities.

This representation can be loaded as a character in the Sol game fairly easily. Some value ranges of the individual properties are however not valid in the game. We have defined constraints on the search space based on what is considered valid properties in the game. Further, the properties can take unreasonable values even though they are allowed in the game. Property bounds were defined for all properties, where abilities of the *MELEE* type have different bounds than *PROJECTILE*. The property bounds were defined based on knowledge of the manifestation of each property in the game. This knowledge was obtained through playing experience. Although these values can be chosen more methodologically, it is not the scope of this study. The property bound constraints can be seen in Appendix B. These constraints are handled by our crossover and mutation operators, such that no character violating these constraints can be present in a population. Thus, these constraints are not used as explicit constraints in the search. With these constraints, the character properties are guaranteed to be valid when loaded into the game.

The objective of novelty search, as mentioned earlier, is diversity. The diversity is based on a measure of the distance between characters, as we will see later in section 4.3. The distance will be based directly on the character representation. Thus, this representation can be seen as direct, with the genotype-to-phenotype mapping being the identity function. For the the simulation-based feasibility evaluation, the phenotype of a character is the character object residing in the game. It is given by game defined rules and the properties given by our rep-

resentation. A genotype-to-phenotype conversion is handled by the game itself through how it loads the given representation.

The relation between the genotype and the phenotype, as a character in the Sol game, is not completely certain. There is an adjacency relationship, as small changes to a genotype property will yield a small change to the mechanic properties in the game. However, a small change in a property may be essential and can make or break a character. Take for example the *radius* property of a melee attack. If the radius is large, such that this attack can reach further than all opponent's attacks, it is a great advantage. Making the radius a little shorter, such that the opponent's attacks have a longer range can yield a great disadvantage. Thus the adjacency relationship between the genotype and phenotype might not be linear, and the exact relationship is unknown. Nevertheless, we argue that the representation is good enough for this work, but can be assessed further.

4.3 Evolution

As seen in chapter 2 and 3, evolutionary algorithms have been used in most previous successful examples of PCG of complex content. We have chosen a method to generate character mechanics with an evolutionary algorithm at its core, based on feasible-infeasible 2 population constrained novelty search (F-I 2pop novelty search).

4.3.1 Constrained novelty search variants: FINS and FI2NS

The Feasible-infeasible Novelty search (FINS) algorithm and the Feasible-infeasible dual Novelty search (FI2NS) algorithm were implemented by Liapis et al. [25] and are discussed in section 3.2. Because of the promising results to discover a diverse set of feasible individuals in the domain of game level generation, we tested these methods on our problem of generating characters, and will therefore expand on the methods in this subsection. FINS and FI2NS are two-population novelty search methods inspired by the FI-2pop genetic algorithm presented by Kimbrough et al. [20] in 2008.

Both FINS and FI2NS keep two separate populations that use different objectives for evolution. FINS and FI2NS split the population into a feasible population, and a infeasible population. The feasible population might produce infeasible offspring and vice versa. In both FINS and FI2NS, the objective of the feasible population is to evolve toward novelty alone. Novel individuals have a higher probability of being selected and reproduced.

To evaluate novelty globally, FINS and FI2NS keep track of previously discov-

ered individuals during evolution by using a *novel archive*, an archive in which previously selected individuals are stored. The m most novel individuals from each generation are put into the novel archive. The novelty of an individual is evaluated based on the average distance to the k closest individuals in the novel archive and the current population. This distance function between two characters is further discussed in section 4.3.2. For our experiments, we use $m = 5$ and $k = 15$. Liapis et al. used $m = 5$ and $k = 20$ in their work [25]. We use a smaller value of k, as the population size that was used in our experiments was smaller, seen later in chapter 5.

The difference between FINS and FI2NS is in the objective function of the infeasible population. In FINS, the objective of the infeasible population is to evolve toward feasibility. This is done by using a function to calculate the distance to feasibility. In FI2NS, both the feasible and the infeasible population evolve toward novelty by using their own independent novel archive. Thus in FI2NS, there is no objective in the search to ensure feasibility. The potential upside of FI2NS, is that as the infeasible population expands, it can produce feasible novel individuals that would be discouraged to be explored in FINS. In the rest of this section, we present how we calculate novelty and feasibility in our implementation of FINS and FI2NS. In addition, we present how we produce the initial population, and what selection mechanisms and evolutionary operators we use.

4.3.2 Novelty evaluation

Novelty in the context of novelty search is the property of an individual to be different from other seen individuals. It is based on the distance between pairs of individuals, where a domain-specific distance function must be provided. In our method for generating character mechanics, an individual is a character in the representation seen in section 4.2.

The distance between two characters is given by a function that takes two characters as input and yields a distance as a number. The distance between two characters is given by:

$$d(c_1, c_2) \rightarrow dist \tag{4.1}$$

We defined the distance function directly on the character representation seen in 4.2. The meta properties *name* for the character itself and the abilities are ignored. The *character body properties*, namely *radius* and *moveVelocity* are treated separately from the abilities.

The distance between two characters body properties are calculated by first normalizing the properties within their prior set bounds. The property bounds were discussed in section 4.2. This entails that the distance between the two outer

bounds of all properties have an equal distance. The distance between the characters body properties are then given by the euclidean distance as follows:

$$d_b(c_1, c_2) = \sqrt{(r_1 - r_2)^2 + (mv_1 - mv_2)^2} \tag{4.2}$$

Where c_1 and c_2 corresponds to the two characters and r_i and mv_i corresponds to the properties *radius* and *moveVelocity* for character i.

The distance between two abilities from two characters is calculated in a similar way, but with two exceptions. The property *knockbackTowardsPoint* is a boolean, so it must be converted to a number. The value is converted to 0 (false) or 1 (true), and the property range is also set to the range [0, 1]. This makes the two values far apart in euclidean space. This is reasonable, as changing the value of the property *knockbackTowardsPoint* greatly affects the ability. The other exception is the property *type*, that distinguishes between a *projectile* ability or a *melee* ability. These two types of abilities are radically different, so the distance is set to the maximum. If the two abilities are of the same type, the distance is given by:

$$d_a(a_1, a_2) = \sqrt{\sum_{i=1}^{n} (a_{1_i} - a_{2_i})^2} \tag{4.3}$$

Where a_1 and a_2 respectively corresponds to an ability from character 1 and character 2. a_{1_i} and a_{2_i} respectively corresponds to the i^{th} property of ability a_1 and a_2.

Which two abilities from the two characters should be compared is however not trivial. The order of abilities is unimportant, as our representation is based on a set of abilities. Our approach was to use the shortest distance of all combinations of two and two abilities from the two different characters, as follows:

$$d_{as}(c_1, c_2) = \sum_{i=1}^{3} d_a(a_{1_i}, a_{2_i}) \tag{4.4}$$

Where a_{1_i} and a_{2_i} corresponds to the two abilities from character c_1 and c_2 that overall yields the shortest distance. The intuition behind this comparison is that the two and two most similar abilities are compared. Abilities should have different purposes for a character, and thus the two most similar abilities will have the most similar purpose.

The character body distance and abilities distance are then combined to yield the character distance function:

$$d(c_1, c_2) = \frac{d_b(c_1, c_2) + d_{as}(c1, c2)}{n} \tag{4.5}$$

Where n is the total amount of properties for the character body and the three abilities. The distance is normalized based on the total number of properties that are taken into account. Thus all the properties impact the distance equally.

In novelty search, novelty is calculated by the average distance to the k closest individuals from the the combination of the given population where the individual resides and the novel archive. k is a hyperparameter, and will be issued later:

$$N(c) = \frac{\sum_{i=1}^{k} d(c, c_i)}{k} \tag{4.6}$$

Where c_i is one of the k closest characters in the current population and the novel archive.

Calculating character distance on the representation of a character is an approximation of the actual character distance. As discussed in section 4.2, the adjacency relationship between the representation and a character in the Sol game is not direct. It can be argued that the novelty should reside in the experienced novelty by a player. Thus the novelty could be evaluated based on criteria through simulation-based evaluation (see section 2.5). For a distance measure through simulation, one could use criteria that assess different playing styles that a character promotes. Inspiration could be taken from Konecný [21].

4.3.3 Initial population

Evolutionary algorithms need an initial population. The initial population is the starting point for the evolutionary search and is an import factor for the performance of an evolutionary algorithm. The diversity over generations in evolutionary search is greatly affected by the diversity of the initial population, as discussed in section 2.2. There is no guarantee for the crossover and mutation operators to explore the search space far outside the bounds of the initial population.

A rather trivial way to generate the initial population is to generate random individuals that are uniformly sampled from the search space. With a sufficiently sized population, diversity can be expected. Another way is to use domain-specific heuristics on desired solutions. The latter approach will usually promote faster convergence, but have a tendency to be less diverse.

We define two different ways of generating the initial population used for our evolution method; a random population generator, *random generator*, and a heuristic-based, *existing character-based generator*. Both methods will later be explored through experimentation in chapter 5. The two methods will be further presented.

Random generator

The random generator uniformly samples the character search space. The search space is constrained by the prior set character properties bounds, discussed in section 4.2.

Existing character-based generator

The existing character-based generator is based on the four existing characters in Sol (section 4.1). The existing characters have proven interesting to the authors and to others based on the authors' observations. Through our experience, they are fun and balanced to play, and seem diverse in the sense that the playing experience is rather different depending on the given character. Thus, they are adopted as heuristics on interesting characters. The generation process generates an equal amount of the four existing characters. To further promote diversity, a mutation operator is applied for each character, except one of each type. The same mutation operator used is the same one that is applied in the evolutionary algorithm and will be presented in the next section.

4.3.4 Parent selection and evolutionary operators

An evolutionary algorithm uses parent selection, crossover, and mutation, as discussed in section 2.2. The parent selection scheme chooses parents among individuals in a population. Parent selection is based on the fitness of each individual, where novelty search uses *novelty* instead of fitness. Parents produce children through a crossover scheme, and the children are issued for mutation.

Parent selection

In our method, we use novelty proportionate selection with replacement for parent selection. This scheme was also used by Liapis et al. [25]. Two and two parents are selected from a fitness proportionate distribution of a population. Replacement of the chosen parents lets them be chosen to be paired with other individuals later.

Crossover

The chosen parent pairs are mated through a crossover function. We apply a crossover on two characters that is based on swapping abilities between the two parents to produce two offspring. We call the parents cp_1 and cp_2, and the offspring co_1 and co_2. An ability index, a, is uniformly chosen in the range [1, 3]. co_1 inherits the character body properties (seen in section 4.2) of cp_1 and the abilities, ab_i, where $i \neq a$. The ability ab_a of co_1 is inherited from cp_2. The other

offspring, co_2, is constructed in a similar manner, but with the role of cp_1 and cp_2 swapped.

From a design perspective, this crossover function can be seen as combining character concepts. Each ability of a character should have a unique purpose for the character. However, a character might have several abilities that conform to the same purpose. Thus, combining abilities from different characters might yield a set of complementing abilities.

Mutation

Mutation is applied to the offspring. The offspring has a probability of p_m to be mutated. A property is mutated according to the following function:

$$m_p(v) = max(min(v * r,\ v_{b_u}), v_{b_l}) \tag{4.7}$$

Where v is the value of a given property, and r is a sample from a uniform distribution in the range $[1 - r_b, 1 + r_b]$. v_{b_u} and v_{b_l} are respectively the upper and lower bounds for the given property, as discussed in section 4.2. The value of r_b is distinct for character body properties and ability properties, respectively denoted r_{b_b} and r_{b_a}. $m_p(v)$ is applied to all properties of a character, with a probability of p_m, and constitutes the complete mutation function.

There are a few exceptions regarding some of the properties in the representation. There is one boolean property in our representation (section 4.2) that is mutated differently. In this case, the value is inverted with a probability p_{m_b}. The *type* property cannot be changed through the mutation scheme.

For our experiments, the parameters have been chosen as follows:

$$p_m = 0.3$$

$$p_{m_b} = 0.1$$

$$r_{b_b} = 0.2$$

$$r_{b_a} = 0.5$$

The parameters were set based on intuition and empirical testing. From the properties of a single ability, about 4 are mutated with this scheme on average, with $p_m = 0.3$. That seems reasonable from a design perspective, as a designer would try to change a limited subset of properties to assess the outcome in the game. The character body properties are applied a less significant mutation, with $r_{b_b} = 0.2$, compared to the properties of an ability. As each of these properties has a greater effect on a character, and thus should be issued with smaller changes.

Feasible offspring boost

To avoid emptying the feasible population of FINS and FI2NS during evolution, a technique called *feasible offspring boosting* was performed. Feasible offspring boost ensures that the feasible population produces more offspring if the feasible population gets too small. Specifically, the feasible population is forced to produce a number of offspring equal to 50% of the total size of both populations. This applies if the feasible population contains at least two characters. This technique enhanced the performance of Constrained Novelty Search performed by Liapis et al. [25]. This technique was used for all experiments with evolution, seen later in chapter 5.

4.4 Simulation based feasibility evaluation

Constrained novelty search by the FINS and FI2NS methods are both based on a feasible-infeasible 2 population evolutionary algorithm. Two populations are kept at all times, a feasible and infeasible population. We define feasibility based on player experience through simulation-based evaluation. Generated characters are evaluated for feasibility by simulation-based evaluation. A character is either classified as feasible or infeasible, based on the expected player experience of the character.

We described our notion of interestingness in the context of fighting game characters in section 2.4, as novel, fun, and balanced. The search process itself has the objective of novelty. Thus balance and fun will be the important aspects of the player experience to evaluate. Five criteria on characters that are played in *simulation* are defined for this purpose and will be presented in section 4.4.1.

The five criteria are based on measurements of games played by computer players. This is a common pattern for simulation-based evaluation in games, seen in section 2.5. We use the term *simulation* to denote a single game being played with two characters, where the output is a set of measurements. Each criterion uses a metric that is measured within each simulation, such that a simulation outputs five measurements for our five criteria. Criteria will either be fulfilled or violated based on the given measurements and can be seen as constraints in the search space.

An overview of our simulation-based feasibility evaluation can be seen in figure 4.5. An input character is simulated with each of the four existing characters, presented in section 4.1. The games are played by two computer players, and measurements are taken. It can further be assumed that a simulation takes one character to be evaluated alongside an existing character. For each simulation, 5 metrics are measured, according to 5 criteria. For each metric, the average

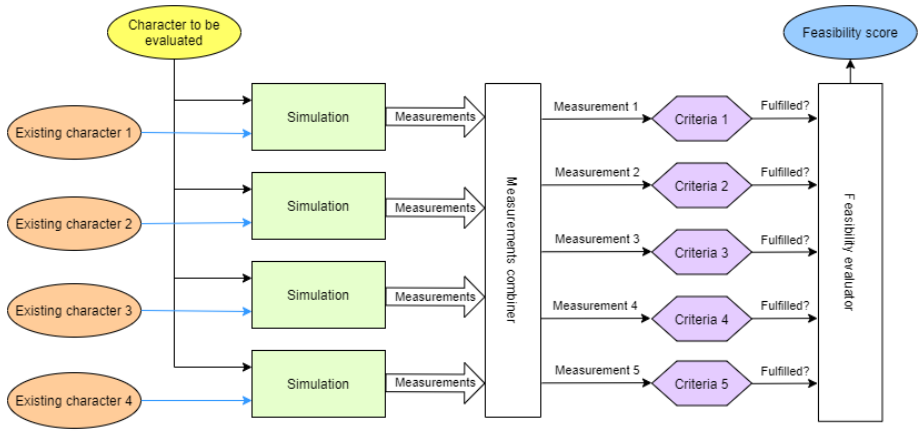


Figure 4.5: Simulation based feasibility evaluation in Sol.

is taken of all the corresponding measurements from the individual simulations, which happens in the *measurements combiner* seen in figure 4.5. In other words, a criterion is based on a single metric that is measured multiple times over multiple simulations. The criterion individually outputs whether it is fulfilled or violated. A feasibility score is given to the character based on how many of the criteria were fulfilled. The feasibility score is given by the following equation:

$$fs = \sum_{i=1}^{5} \begin{cases} \frac{1}{5} & \text{if } c_i \text{ is fulfilled} \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Where fs is the feasibility score and c_i is the i^{th} criteria. The feasibility score is thus given in the range $[0, 1]$. A feasible character is defined to have a feasibility score of 1. An infeasible character will have a feasibility score in the range $[0, 0.8]$ with a step size of 0.2. The feasibility score of infeasible characters can be seen as the distance from feasibility. This score is used as the objective for the infeasible population in the FINS method discussed in section 4.3.1.

The individual simulations can be repeated several times. This might be necessary as the player controller is non-deterministic, such that the outcome of multiple games played with the same characters and the same player controllers might yield different measurements. The number of repeated simulations is controlled by a parameter, rs, and is the number of times an input character is simulated repeatedly with each of the existing characters. Thus the input character is played an equal number of times against all the existing characters. The

value of this parameter will be further investigated through experimentation in section 5.1.

A simulation could in theory last forever, or for an unnecessarily long time. This can happen in cases where the characters are too weak so that neither player can win the game. This is not desirable as a single, long, simulation may slow down the whole system. To eliminate this issue, a fixed max length of a simulation is defined. A parameter is defined to give a max simulation length, *msl*. It defines the max length that a simulation can run, and is given as a number of game updates. Recall that an update is $\frac{1}{60}^{\text{th}}$ second when the game is played in human speed. If a simulation exceeds this length, it is terminated. The value of *msl* will be chosen based on an experiment in section 5.1.

The validity of this approach is based on the design and appeal of the four existing characters with respect to them being balanced and fun to play. We argue that the existing characters are suitable for this method, but can be further assessed.

We will further in this section present the criteria that were used to evaluate generated characters for the Sol game, followed by the player controller that is used.

4.4.1 Player experience criteria

The goal of our criteria is to capture the experience that would be perceived by a human player. The experience will be evaluated according to what a player finds fun and balanced. Five criteria are defined for the purpose, that are heuristics on the player experience.

We have seen that quantitatively measuring human player experience is hard without a human subject, from section 2.5. Successful approaches are based on a set of criteria of a simulated game that act as heuristics for player experience, as seen in section 3.3. However, we have not seen any attempts to do this in the domain of fighting games. The criteria used in related works are designed for their given domain, so a direct mapping to fighting games is not possible. We have constructed criteria inspired by the work presented in 3.3 to create a heuristic for player experience for the Sol game. The five criteria are *game balance*, *game length*, *stage coverage*, *character balance* and *lead change*.

It should be noted that these criteria are not exhaustive for the actual player experience model. They are heuristics, and will later be evaluated through user studies in section 5.3.

A criterion in this work consists of a *metric* that can be measured through simulations and is evaluated according to a *feasibility range* for the given criteria.

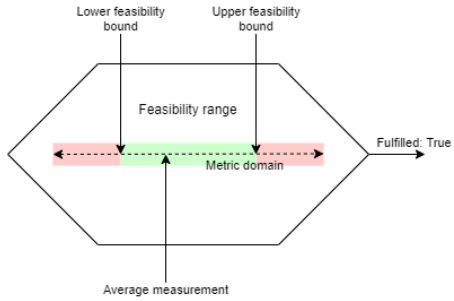


Figure 4.6: A fulfilled criteria for simulation based evaluation

The feasibility range is given by an upper and lower value bound. The criteria is fulfilled if the average of its measurements fall within the feasibility range, as seen in figure 4.6. Thus a criteria is a function given by the following equation:

$$c_i(m_i) = \begin{cases} fulfilled & \text{if } fr_0 \leq m_i \leq fr_1 \\ violated & \text{otherwise} \end{cases} \tag{4.9}$$

Where fr_0 and fr_1 is the lower and upper feasibility bound, respectively. m_i is the average measurement over multiple simulations, of the metric corresponding to criteria c_i. fr_0 and fr_1 are left as hyperparameters. They will be further investigated in section 5.1.2.

The criteria in this work are formulated as constraints, given a feasibility range. This differs from the work we have previously seen. Criteria have priorly been defined as objectives issued for optimization, such as in the work of Brown and Maire [4]. In their domain, the criteria output a number instead of acting as constraint, with a feasibility range. The criteria are then maximized or minimized. This is not trivial for criteria related to fighting game characters. The characters should preferably have different strengths and weaknesses. An example is the *stage coverage* criterion, which evaluates how much of the game stage is used. Some characters might be strong and slow, and others may be fast and weak. The slow and strong character would naturally have less stage coverage, without that necessarily being negative. An approach could be to optimize for multiple objectives, where each criterion is an objective. This approach could be taken with a multi-objective evolutionary algorithm, as seen in section 2.2. The feasibility ranges we use can be seen as outer bounds of preferred player experience within the respective criteria. Viewing the criteria as constraints also lets us prioritize the search for novel characters through constrained novelty search.

Browne and Marie presented distinctions between criteria based on how their measurements are taken [4], as was discussed in section 2.5. Game balance and game length are given by the outcome over several simulations. They correspond to the *viability* criterion. The metric of a viability criterion can be seen as a function of the outcome of a simulation, sim, yielding a measurement, m:

$$M_v(sim) \rightarrow m \tag{4.10}$$

Stage coverage, character balance, and lead change are on the other hand directly related to the actions and events within a simulation. They correspond to *quality* criterion. A metric corresponding to a quality criterion can be seen as a function of the game states, sim_{s_i}, during a simulation, sim:

$$M_q(sim_{s_1}, sim_{s_2}, ..., sim_{s_i}) \rightarrow m \tag{4.11}$$

The game states sim_{s_i} are programmatic perceptions of the game state, give by the Sol game. We will further present the five criteria.

Game balance

The balance criterion is given by the average outcome over all simulations. This criteria is based on the metric *characterWon*, that measures weather or not the evaluated character won a simulated game, denoted w_i. It takes either the value 1 or 0, where 1 indicates that the evaluated character won. The average measurements of *characterWon* is given as follows:

$$M_{cw} = \frac{\sum_{i=1}^{n} w_i}{n} \tag{4.12}$$

Where n is the number of simulations. The feasibility range for v_{gb} should be close to 0.5, with the intuition that a character should have a fair chance of winning a game.

Game length

The game length criterion is given by the metric *gameLength*, which measures the length of a simulation. The average measurement is given by:

$$M_{gl} = \frac{\sum_{n=1}^{n} l_i}{n} \tag{4.13}$$

Where n is the number of simulations, and l_i is the length of simulation i. The feasibility range of v_{gl} should be given by the desired length of a game. This is an important criterion that might be dependent on the intentions of the game. This criterion will also influence the overall strength of the characters. A low game length might lead to characters eliminating each other with a few hits, even though they are balanced.

Stage coverage

The stage coverage criterion is based on how much of the stage is covered on average by the evaluated character during a simulation. It is based on the metric *stageCoverage*, and is measured by:

$$M_{sc} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{k} \begin{cases} 1 & \text{if } c_j \text{ visited} \\ 0 & \text{otherwise} \end{cases}}{n} \tag{4.14}$$

Where n is the number of simulations. The game stage is divided into 1400 discrete cells, c_j. The lower bound of the feasibility range should be defined to eliminate extreme camping, meaning characters that can win without moving. This may happen with powerful long-ranged abilities.

Character balance

The character balance criterion is defined to ensure a balance within a character with regards to its abilities. The corresponding metric, *leastInteractionType*, is based on how many times the three different abilities hit the opponent during a simulation. The amount of times an ability hit the opponent during a simulation is denoted ah_i, where i corresponds to ability i. The amount of hits by the ability with least hits is denoted ah_l. The average measurements of the metric is given by:

$$M_{lit} = \frac{\sum_{i=1}^{n} \frac{ah_l}{\sum_{j=1}^{3} ah_j}}{n} \tag{4.15}$$

Where n is the number of simulations. M_{lit} measures the share of hits by the least hit ability to the sum of hits by all abilities in simulation i. The average is taken over all n simulations.

The feasibility bounds should be defined such that all abilities are required to be used. Otherwise, an ability will probably be useless. It is not reasonable, however, to assume that all abilities are used equally frequently. Some abilities might be useful in special situations, and it is natural to assume some abilities are used more than others.

Lead change

This criterion is based on the metric *leadChange*, which measures the number of changes in lead between the two players during a simulation. The lead is given by a player's character being in a better state than the other character. The value of the state of a character, c, is given by a *state evaluation function*, that yields

a evaluation e as a number for a given state, s: $ev_c(s) \rightarrow e_c$. The character with the highest e_c has the lead. It is measured by:

$$M_{lc} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{k_i - 1} \begin{cases} 1 & \text{if} \frac{ev_1(s_j) - ev_2(s_j)}{|ev_1(s_j) - ev_2(s_j)|} \neq \frac{ev_1(s_{j+1}) - ev_2(s_{j+1})}{|ev_1(s_{j+1}) - ev_2(s_{j+1})|} \\ 0 & \text{otherwise} \end{cases}}{n} \tag{4.16}$$

Where n is the number of simulations, and k_i is the number of states in simulation i. s_j is the j^{th} state of simulation i. M_lc measures the number of game states where the character in the lead changes for the next game state, according to $ev_c(s)$.

The state evaluation function is based on the programmatic game state perception of Sol, and is given by:

$$eval_c(s) = 0.67 * \frac{st_c - 1}{st_m - 1} + 0.33 * min(\frac{d_m - d_c}{d_m}, 1) \tag{4.17}$$

Where st_c is the current amount of stocks left for the character c in state s, and st_m is the maximum amount of stocks. The maximum amount of stocks in Sol is 3. Recall that stocks are the number of lives a character has before the game is lost. d_c is the current accumulated damage taken by character c in state s and d_m is the maximum damage that a character can take, and is set to 5000. In Sol, the real maximum damage that can be taken is arbitrarily set to 9999. However, damage above 5000 is negligible when evaluating the state.

The intuition behind the state evaluation function is that a character with more stocks than the other character will always be in a better state. If both characters have the same amount of stocks remaining, the character with the least amount of damage will be in a better state.

This state evaluation is rather simple and does not capture every aspect of a state. More complex state evaluations have been seen in the literature, for example by Brown and Maire [4]. In their work, the state evaluation function was learned for the evaluated content. This could be used for Sol as well, but given that the game rules are known we can simplify the state evaluation function based on domain knowledge.

4.4.2 Computer player

We will further present the computer player that we use in the simulations. Our simulation criteria are heuristics on the fun and balance perceived by a human player and are based on measurements on simulation. These heuristics are only

valid if the computer player can play similarly to a human player. The goal for our character generation system is to be used without human interaction, and amounts of human play data is not available. The computer player must also be able to play any character that can be expressed in our representation, and are unknown during the design of the computer player. We will further discuss the rationale for the design of our computer player along with an overview of how it works. The player we present is used for both characters that are played in a simulation. That is, both the character to be evaluated and the existing character it is played against.

There are no existing AI agents that can be directly adapted to Sol. We have seen game-playing AI created for fighting games in section 2.3.2. The superior methods seen over the last couple of years in the Fighting game AI (FTGAI) competition are based on Monte Carlo tree search (MCTS) or reinforcement learning (RL) variants. These methods are rather complicated and computationally demanding. Thus, simulations would take a long time to run. MCTS is dependent on a search through the state space of the game while playing. This requires a forward model of the game, a parallel version of the game that can be played forward in time from a given state in the actual game. Sol does not have such a forward model at this time. RL methods need a lot of training time to be able to play characters. As the computer player is frequently exposed to different characters, a training process would need to be performed for every character. As such, MCTS and RL will not be in scope for this work. It should also be noted that the goal of the FTGAI competition is to appoint the best performing AI agent. For a computer player evaluating player experience in simulation, the player should optimally act like a human player. This has not been an objective for the agents we have seen in the fighting game domain, neither for the FTGAI competition nor other fighting games.

The focus of this work is not to create an advanced computer player. We have chosen to create our computer player based on classical rule-based agents, seen in the context of fighting games in section 2.3.2. The rather simple player can have an impact on the quality of our system and can be further investigated.

The rule-based agents previously seen have been tailored to the character it controls, and are optimized for the behavior of a single character. A final state machine (FSM) has usually been adopted at the core of this approach [1]. A computer player that can play any character in Sol is based on knowledge of the game rules, and partially on the properties of our character representation (seen in section 4.2). It is based on independent rules, instead of an explicit FSM. The

[1] A rule-based agent based on a final state machine can be seen by the Giant team's entry in the 2015 FTGAI competition. See slide 60 - 66 in the slides at `https://www.slideshare.net/ftgaic/2015-fighting-game-artificial-intelligence-competition`

rules are game-specific heuristics of how a player should play Sol.

The computer player is a reflex agent that can be seen as a function of a given game state that yields inputs to the game for the character it controls:

$$a(s) \rightarrow INP \tag{4.18}$$

Where s is the programmatic game state perception of Sol, in a given state. INP is the set of inputs that the game accepts, where the individual inputs are denoted inp_i. Both s and INP was discussed in section 4.1.1.

The agent is based on individual rules that suggest different values for INP. We divide INP into three categories; inp_{m_i} (movement input), inp_{a_i} (ability input) and inp_{aim_i} (aim input). inp_{aim_i} is handled by a single rule such that the agent's character always aims at the opponent character. Input within the two remaining categories, or just a single category, might be suggested by each rule. A rule also outputs an urgency, u, according to the importance of the suggested game inputs. A rule can thus be seen as a function as follows:

$$r(s) \rightarrow inp_{m_i}, inp_{aim_i}, u \tag{4.19}$$

Each rule is given a static weight, w_i. The weight determines the global relative importance of a rule, $r_i(s)$, independent of the state, s. The agent based on a set of such rules with weights is given by the following function:

$$a(s) = f(r_1(s), w_1,\ r_2(s), w_2 \ldots, r_i(s), w_i) \tag{4.20}$$

Where $r_i(s)$ is a rule as a function of the given state s. The function f combines the inp_{m_i} and inp_{aim_i} suggestions of each rule $r_i(s)$, with given weights, w_i. f yields a final INP that will be the actual input to the game.

We have defined 7 rules that will be briefly described, where the character that the computer player controls is denoted *character* and the character controlled by the opponent is denoted *opponent*:

- **Avoid holes rule**, r_{aho}: This rule prefers movement away from holes through suggesting inp_m to accomplish this task. This is accomplished by suggesting inp_a to not input any abilities. The urgency, u is given by the distance between the controller's character and the closest hole.
- **Move random rule**, r_{mr}: This rule promotes random movement, such that inp_m is randomly set. The urgency, u, is constant.
- **Retreat rule**, r_{ret}: This rule prefers character movement away for the opponent if they are close. inp_m is given to move away from the opponent character. The urgency, u, is based on how close the character is to the opponent character.

- **Approach rule**, r_{app}: This rule has the opposite effect of the retreat rule. imp_m is given to move towards the opponent character. The urgency, u, is based on the distance from the other character.
- **Avoid hitboxes rule**, r_{ahi}: This rule promotes movement to avoid opponent hitboxes close to the character. Recall that a hitbox is the area of an ability where it will hit an opponent character. The urgency, u, is given by the distance from the character to the closest opponent hitbox.
- **Character attack rule**, r_{cat}: This rule promotes the use of abilities that may hit the opponent. It is based on the distance to the opponent character, d_{oc}. A random ability is chosen among the abilities that may reach the opponent in the given state. The preferred ability is given The reach of an ability is given by:

$$reach(ab_i) = \frac{s_i * at_i}{60} + dfc_i + r_i - r_c \tag{4.21}$$

  Where ab_i is the i^{th} ability of the character. s_i and at_i is respectively the *speed* and *activeTime* of the i^{th} ability, given in the character representation (see section 4.2). The first addend of the function is only relevant for projectiles, as any melee abilities have $s_i = 0$. dfc_i and r_i correspond to the properties *distanceFromChar* and *radius* of the ability, while r_c is the *radius* property of the character itself. These properties are also given in the character representation. The urgency, u, is dependent on the distance between the character and the opponent character. If the distance is short, the urgency escalates.
- **Random attack rule**, r_{rat}: This rule suggests a random ability. The urgency is given by the distance from the character to the opponent, similarly to r_{cat}. This one rule is not used for the computer player in simulation-based testing but used for our user tests, seen later in section 5.3.

The rules and their corresponding weights were defined based on game knowledge and manual testing. The *avoid holes rule* is based on the intuition that a character being in a position close to a hole (the outside of the stage) is not preferred as it is eliminated if it falls into the hole. This rule keeps the agent from eliminating its own character as well as recovering from a bad position if the opponent pushes the character close to a hole. The rule will also promote not using any abilities, as abilities usually have a time delay where the character can't be controlled. The three movement rules, *move random rule*, *retreat rule* and *approach rule*, promote human-like movement. The character moves toward the opponent if it is too far away, and move away if it is too close. The induced random movement yields unpredictability and promotes exploration of the game stage. The *avoid*

hitboxes rule lets the character have the opportunity to avoid enemy attacks. This is especially important for long-ranged projectile attacks that would easily hit the opponent without this rule. Only one of the two attack rules, *character attack rule* and *random attack rule* is used for a single agent.

Two computer players are defined, cp_{sim} and cp_{test}, both using the rule-based agent described. For our simulation based feasibility evaluation, cp_{sim} is applied for both characters that are played in the simulation. It is given by the rules and corresponding weights:

$$\begin{aligned} cp_{sim}(s) = f(&r_{aho}(s), 0.31,\ r_{mr}(s), 0.1,\ r_{ret}(s), 0.06,\\ &r_{app}(s), 0.14,\ r_{ahi}(s), 0.27,\ r_{cat}(s), 0.12) \end{aligned} \tag{4.22}$$

Where f is the same function as defined in equation 4.20. The computer player, cp_{test} is used for our user studies. It act as the opponent when human testers play the game. It is similar to cp_{sim}, but the attack rule r_{cat} is replaced with r_{rat}. The r_{cat} rule promotes better estimation of when an ability should be used, and approximates a player with intermediate to good skills. r_{rat} promotes the selection of random abilities, and approximates a beginner.

Real-time video games let players apply game input at a high rate, usually 60 times a second. The input rate is effectively reduced when humans play, due to perception time and input inaccuracy. Our defined computer players can react instantaneously, and can thus react much faster than a human and with greater accuracy. This is undesired, as the computer player should play similarly to a human player, as discussed in 2.5.2.

To make the computer players more human-like, we apply two limitations; *game state delay* and *input fuzziness*. We delay the state perception from Sol that the computer player can "see" by 0.2s. That corresponds to 12 game state updates at 60 frames per second. We introduced noise on the input of our computer player, INP. The four movement inputs, inp_{m_i} , and the three ability input, inp_{a_i} could be flipped with a probability of respectively p_m and p_a. The aim input, inp_{aim_i}, consist of an x and y coordinate that are both represented by numbers. We denote the aim input after noise is applied as $ninp_{aim_i}$. Noise were applied to each coordinate individually. $ninp_{aim_i}$ is given by a sample from a normal distribution with mean equal to inp_{aim_i}, and a variance of v_{aim}. The noise parameters were set based on evaluation through observation: $p_m = 0.1$, $p_a = 0.001$ and $c_{aim} = 30$. The value of c_{aim} represent distance units in the game.

The state perception delay had a large impact on the computer player. Abilities had a large chance to miss the opponent, as our computer player aims at the position of the opponent. With the state delay, the player aimed at where the

opponent was 0.2 seconds ago. To cope with this challenge, we introduced linear extrapolation of the opponent's position, inspired by Asayama et al. [1].

Chapter 5

Experiments and results

We will in this chapter present experiments that we have constructed to examine RQ1, RQ2, and RQ3. The experiments are conducted based on our applied system of generating characters, from chapter 4.

We have defined four experiments. Experiment 1 is conducted to define the parameters of our simulation-based feasibility evaluation (section 4.4) that are left to explore.

Experiment 2 is conducted to answer RQ1. Four variations of constrained novelty search are compared. The FINS method and FI2NS method are (presented in section 4.3.1) both tested with a random initial population and an initial population based on the four existing characters (presented in section 4.3.3).

Experiment 3 and experiment 4 are based on user studies to evaluate our heuristics of "interesting" characters according to perceived experience by human players. Experiment 3 evaluates our feasibility criteria in simulation-based evaluation. Thus the experiments target RQ2 to evaluate our quantification of "balance" and "fun" according to human experience. Experiment 4 is concerned with RQ3 and aims to evaluate the quality of generated characters from our system by comparing the generated characters to existing characters

The chapter is structured according to each experiment. The experiments will be presented along with their results and analysis of the results. Experiment 1 will be presented in section 5.1 and experiment 2 in section 5.2. A general overview of the user studies will be presented along with Experiments 3 and 4 in section 5.3.

5.1 Experiment 1: Parameters of simulation based feasibility evaluation

We presented the simulation-based feasibility evaluation method for our character generation system in section 4.4. In this section, three of the parameters will be explored through experimentation. The parameters are:

- The number of repeated simulations between a character to be evaluated, and each of the existing characters, rs.
- The feasibility ranges for each of the five defined criteria. The ranges are given by a lower and upper bound, respectively fr_{0_i} and fr_{1_i}, for each criteria, c_i.
- The max length of a simulation, msl.

Two separate experiments were performed to deduce a suitable value for the three parameters. The parameter rs will first be presented. Then fr_{0_i} and fr_{1_i} will be discussed, where the max simulation length, msl, is given based on the feasibility ranges. The parameter values given in this section will be used for the other experiments.

5.1.1 Repeated simulations

The simulation-based feasibility evaluation that we use is non-deterministic, due to the computer player being non-deterministic. This affects the measurements from a simulation and may vary between several simulations with the two same characters. We will look at the effect on the metrics for our criteria. The criteria with their corresponding metrics are shown in table 5.1.

Criteria	**Metric**
Game balance	characterWon
Game length	gameLength
Stage coverage	stageCoverage
Character balance	leastInteractionType
Lead change	leadChange

Table 5.1: The simulation criteria presented in section 4.4, with their corresponding metrics.

Metric	$rs = 1$	$rs = 10$	$rs = 50$
leadChange	$\overline{x} = 4.78$ $s = 1.758$	$\overline{x} = 4.93$ $s = 0.628$	$\overline{x} = 4.90$ $s = 0.314$
characterWon	$\overline{x} = 0.483$ $s = 0.255$	$\overline{x} = 0.490$ $s = 0.111$	$\overline{x} = 0.482$ $s = 0.082$
stageCoverage	$\overline{x} = 0.314$ $s = 0.053$	$\overline{x} = 0.313$ $s = 0.048$	$\overline{x} = 0.313$ $s = 0.048$
gameLength	$\overline{x} = 5764$ $s = 831$	$\overline{x} = 5762$ $s = 632$	$\overline{x} = 5764$ $s = 589$
leastInteractionType	$\overline{x} = 0.060$ $s = 0.058$	$\overline{x} = 0.060$ $s = 0.055$	$\overline{x} = 0.060$ $s = 0.055$

Table 5.2: Variance in repeated simulations. The table shows the five metrics measured by rs repeated simulations against the four existing characters. The measurements are taken over 100 evaluations, and for each of the four existing characters that are evaluated. $\overline{x}$ is the sample mean, and s is the standard deviation.

The four existing characters were used to assess the effect of the number of repeated simulations, where measurements by the four characters are combined. Three separate runs of evaluation of the existing characters where performed. For each run, the number of repeated simulations, rs, was set to 1, 10, and 50. Each of the three runs is performed with rs simulations and the evaluation is repeated 100 times. The results can be seen in table 5.2.

It can be seen that the standard deviation, s, of the measurements corresponding to each metric is reduced (or equal) when rs grows. Preferably, a high rs is used for precise measurements, but it greatly impacts the speed of the generation method. The performance speed of the system is greatly dependent on the number of simulations, so a compromise has to be made. Seeing that $rs = 1$ yields a rather large standard deviation in the measurements, it is undesirable. The difference between $rs = 10$ and $rs = 50$ is negligible for some of the metrics, *stageCoverage* and *leastInteractionType*. The *leadChange* metric is especially benefiting from a higher rs. Given that the evaluation speed is approximately increased by five times with $rs = 50$ compared to $rs = 10$, we do not consider the benefit great enough. For the following experiments, we use $rs = 10$.

5.1.2 Feasibility ranges of criteria

The simulation-based feasibility evaluation described in section 4.4 is being used in all evolution strategies we tested. A criterion was represented as a metric

to be measured through simulation-based evaluation and a *feasibility range*. If the average measurement of the metric over a set of simulations falls within the feasibility range, the criterion is fulfilled.

We will further present the experiment conducted to determine the feasibility ranges of the five criteria. The criteria and their corresponding metrics are shown in table 5.1.

The feasibility ranges for our criteria are given by a lower and upper bounds, respectively fr_0 and fr_1. Our rationale for choosing these parameters was to use the set of existing characters, seen in section 4.1. They are human-designed characters that act as heuristics of feasible characters. We consider our own characters fun and balanced through playing experience. Thus, the feasibility bounds will be quantified, based on this set of existing characters.

The four existing characters were measured in simulation according to the metric of each criterion. The simulations were conducted in the same way as presented in section 4.4, such that each of the four existing characters to be evaluated was simulated against each of the four existing characters. For each evaluated character, 100 runs were performed. For each run, the simulation repeat was set to $sr = 10$. Average measurements for each metric and character over the 100 runs can be seen in figure 5.1. We define the feasibility range of each criterion such that measurements for all four existing characters from a major portion of the runs reside within the ranges. For each criterion, the lowest average measurements from all the characters were as follows:

- *leadChange*: 4.483
- *characterWon*: 0.398
- *stageCoverage*: 0.231
- *gameLength*: 5105.37
- *leastInteractionType*: 0.0234

whereas the highest average results were:

- *leadChange*: 4.99
- *characterWon*: 0.6175
- *stageCoverage*: 0.353
- *gameLength*: 6598.81
- *leastInteractionType*: 0.154

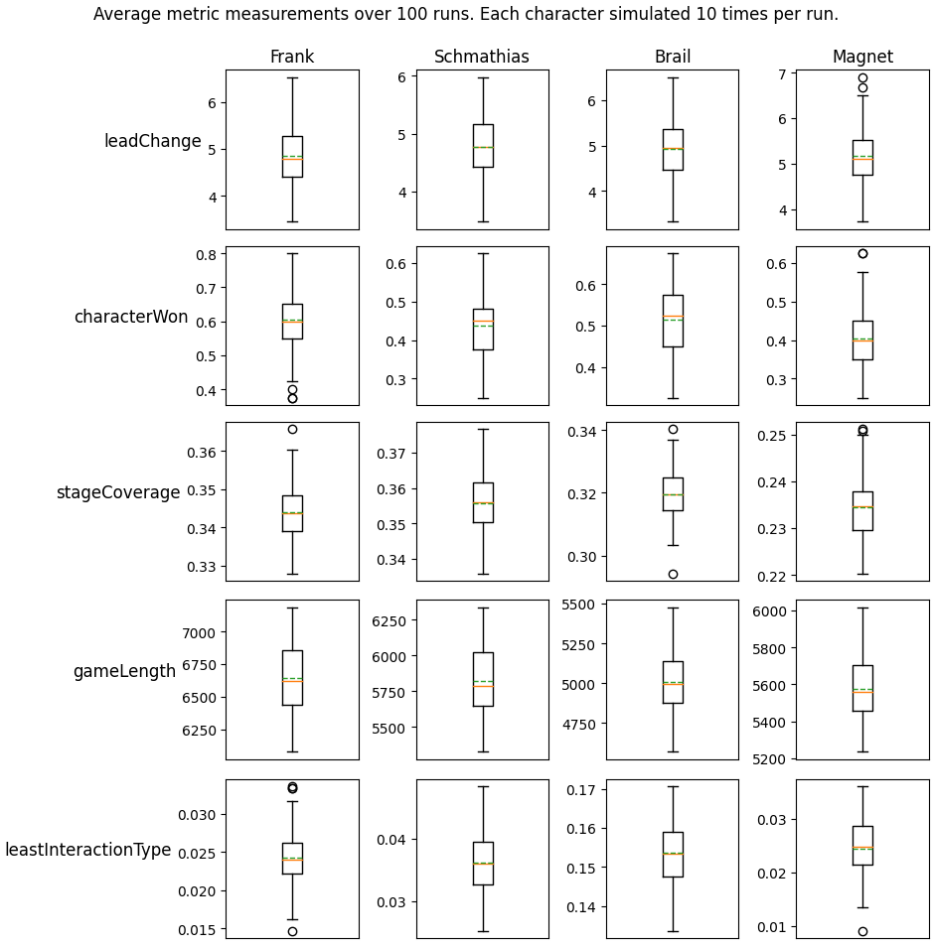


Figure 5.1: Box plots of all measured metric values of a simulation of the existing characters in Sol, namely Frank, Schmathias, Brail, and Magnet. These values are used as a baseline for the constraints given on the characters used in simulation-based testing. Generated characters should perform as well, or better than the existing characters to be classified as feasible.

```
feasible_metric_result_ranges = {
    "leadChange": (4, INF),
    "characterWon": (0.3, 0.7),
    "stageCoverage": (0.2, 1),
    "gameLength": (3600, 7200),
    "leastInteractionType": (0.02, 1)
}
```

Figure 5.2: Our chosen lower and upper bounds of feasible metric results. A character is considered feasible if all measured metrics fall within their feasible metric result bounds. The fewer measured metrics within these bounds, the lower the feasibility score a character gets.

For the *characterWon* and *gameLength* metrics, we need both lower and upper bounds. We do not want a character to win too little or too much, and we not want a game to be too short or too long. For the other three metrics, we only need lower bounds, but no upper bounds (the more the merrier!). Since *stageCoverage* and *leastIneractionType* are metrics based on percentage, 1 is the maximum value. For *leadChange*, there is in no upper boundary needed. We now have a good basis for deciding feasible metric result ranges. The chosen values are shown in figure 5.2.

The max simulation length parameter, msl, is determined based on the feasibility range of *gameLength*. An evaluated character is defined to be infeasible if the average simulation length is above 7200. Based on this, the max simulation length is set at $msl = 10000$. The maximum length should be higher than 7200 because the measurement is given as an average over multiple simulations.

5.2 Experiment 2: Comparing evolution variants

Experiment 2 was conducted by comparing four different strategies of constrained novelty search. This experiment is related to RQ1:

RQ1 How can constrained novelty search be utilized to evolve interesting character mechanics?

We have seen that constrained novelty search is a promising method for evolving novel content. Through our research, two variants of this method seem promising for the goal of generating novel character mechanics. Thus we constrain this experiment to evaluate the two feasible-infeasible 2 population constrained novelty search variants, FINS, and FI2NS, presented in section 3.3. Each of the evolutionary algorithms was evaluated with two different strategies for generating

the initial population, namely the *random generator* and *existing character-based generator*, as presented in section 4.3.3. The character generation system was run 20 times with each of the four variants. All runs operated with a total population size of 40 individuals and were terminated after 30 generations.

Box plots of the evolution from one of the runs of the FI2NS algorithm with the existing character-based initialization is displayed in figure 5.3. It can be seen that the Novelty of characters drops steadily over generations. That was a trend through all runs of all the variants. This is expected, since the number of individuals in the novel archive increases over time, and thus the likelihood of individuals being unique decreases. The number of feasible individuals decreases initially. The methods that used the existing character-based initialization had a trend of starting with a rather large feasible population that initially decreased. The *feasible offspring boost* mechanism (see section 4.3.4) ensures that the size of the feasible population is somewhat sustained.

To evaluate the performance of a run, we used two different metrics. The number of feasible individuals in the last generation is the first metric. The second metric is a *diversity* function of applied to the last generation. The diversity function is defined by

$$D(p) = \frac{1}{\frac{N!}{2!(N-2)!}} \sum_{(c_1,c_2)\in pairs(p)} d(c_1, c_2) \tag{5.1}$$

where p is the population, N is the population size and d is the distance function between two characters. The sum of the distance between all pairs is divided by the number of pairs. An overview of the outcomes of the different runs is found in table 5.3.

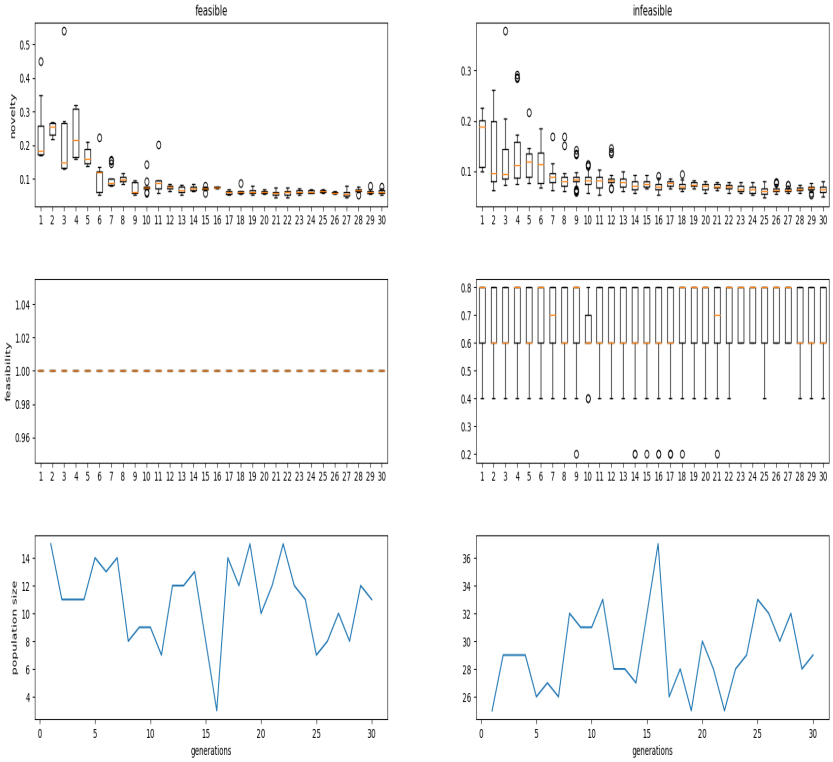


Figure 5.3: An example of a run of FI2NS with an initial population based on the existing characters. The left side plots show the novelty, feasibility, and population size of the feasible population, whereas the right side plots show the infeasible population. The feasibility score of the feasible population is constant at 1 for all individuals, which is expected by the very definition of feasibility. As we see in the upper plots, the novelty of both populations is increasing the first couple of generations but decreases as the novel archive expands. This is also expected. The number of individuals in the feasible population is between the 5-15.

Method	Init. pop	No. of runs	Feasible no.	Diversity	Time taken (m)
FINS	random	20	$\overline{x} = 11.4$ $s = 3.87$	$\overline{x} = 0.203$ $s = 0.107$	$\overline{x} = 90.09$ $s = 41.48$
FINS	existing chars	20	$\overline{x} = 11.2$ $s = 3.70$	$\overline{x} = 0.251$ $s = 0.100$	$\overline{x} = 82.17$ $s = 25.26$
FI2NS	random	20	$\overline{x} = 10.8$ $s = 2.44$	$\overline{x} = 0.251$ $s = 0.076$	$\overline{x} = 99.11$ $s = 49.55$
FI2NS	existing chars	20	$\overline{x} = 9.65$ $s = 2.54$	$\overline{x} = 0.268$ $s = 0.079$	$\overline{x} = 92.46$ $s = 41.03$

Table 5.3: An overview of the results of the four evolution strategies. All methods had a total population size (feasible + infeasible population) of 40 individuals. $\overline{x}$ is the sample mean, and s is the standard deviation. Feasible number and diversity refers to the number of feasible individuals and the diversity of the last generation of the evolution, respectively. The time taken is measured in minutes. Evolutions were run using a Intel i7-8550U processor.

As we can see in table 5.3, FI2NS with an initial population based on existing characters had the best performance with regards to diversity. FI2NS with a random initial population was almost identical to FINS with existing characters as with the initial population on diversity performance. The only notable difference in diversity was FINS with a random initial population, having approximately 0.05 lower diversity than the three others. FINS with a randomly generated initial population had 0.2 more feasible individuals in the last generation compared to FINS based on existing characters. This difference is rather small however and falls within the boundaries of the standard errors of the means of the feasibility number ($\frac{s}{\sqrt{n}} = 0.865$ for FINS with random, and 0.827 for FINS with existing).

In general, FINS produced slightly more feasible individuals than FI2NS on average, whereas FI2NS produced a slightly more diverse population. This is coherent with the argumentation laid out by Liapis et al. [25]. Since FINS' infeasible population evolves towards feasibility, more individuals are likely to end up in the feasible population. The infeasible population of FI2NS evolves toward novelty, and some of these individuals will eventually be feasible. It is therefore to be expected that the diversity of feasible individuals is higher than in FINS.

It should, however, be noted that all there is only a small variance between the methods on both the feasible number and diversity. There are also uncertainties in the utility of both the feasible number as a metric and the diversity as a metric. Taken the small variance and these uncertainties into account, we cannot make convincing conclusions that one method outperforms others on any of the

metrics, only that our evolution experiments show a slight favor of FINS on evolving a larger feasible population, and a slight favor of FI2NS for a more diverse population.

5.3 Experiment 3 and 4: User study

In experiments 3 and 4, we used human test subjects to evaluate characters that were generated by our system. Since the goal of the thesis is to explore the value of generating characters, we need a way to determine the value of generated characters. Ultimately, humans are the best judge of what humans consider interesting. We therefore recruited human test subjects to judge our constraints and compare the generated characters to the human-designed characters, with regards to interestingness. We conducted two different experiments based on user studies. Experiment 3 is related to RQ2, and Experiment 4 is related to RQ3.

5.3.1 User study overview

The user test was based on a questionnaire, seen in Appendix C. The questionnaire consists of two tests; *Test1* and *Test2* that corresponds to Experiment 3 and Experiment 4, respectively.

There were a total of eight test subjects used for Experiment 3, and seven test subjects used for Experiment 4. The test subjects were students from 20 to 25 years old, and with varying prior experience with fighting games. Some of the tests were performed with instructors present, and some were performed without. Several sessions were organized, where one to three subjects participated in each session. The authors acted as instructors.

The version of the game Sol that was used for the studies had minimal graphics. All distinguishable elements of visual character aesthetics were left out, since we wanted to isolate the character mechanics, without the interference of other aspects of the game. A screenshot of the game used for playtesting is found in figure 5.4.

Both user studies are based on comparisons between two or four characters. Comparisons are made by plying the characters, one after another. Each character was allowed to be played multiple times. The opponent of the evaluated characters was not a focus. The opponent character was in all test cases controlled by the computer player, cp_{test}, seen in section 4.4.2. This computer player is not as good as the player used for the simulation-based feasibility evaluations. It is expected that the test subjects are not familiar with the game Sol, and only a limited time to learn the game is given. Thus, it is reasonable to use an opponent

player that is rather easy to play against. If a test subject is in a stressed situation, it will probably be harder to explore a character, and it would probably not seem interesting.

The studies started with a test part, lasting for about 20 minutes, to make the subjects comfortable with the game. The subjects were presented with the rules of the game and how it was played. Then Test 1 (related to Experiment 3) was performed and took about 30 minutes. Lastly, Test 2 was done in about 30 minutes. The subjects were asked to test all three abilities of each character that they played and to explore each character.

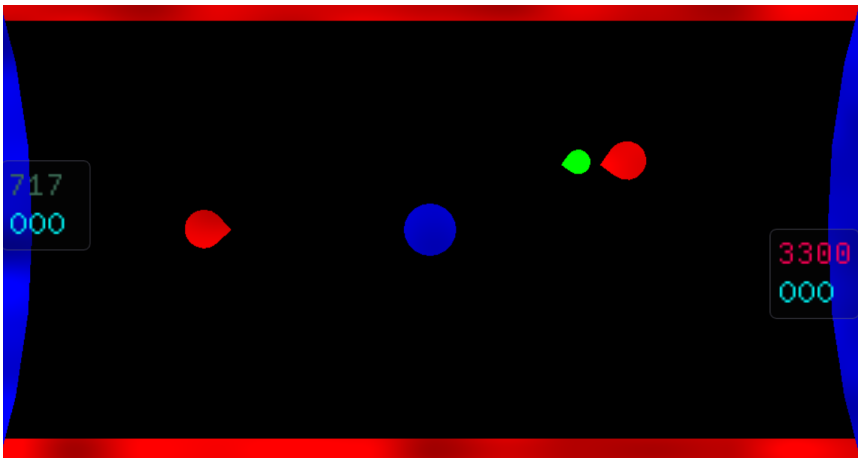


Figure 5.4: A screenshot of the Sol game used for user studies. The graphics are simplified to focus the attention towards the mechanics of the characters.

5.3.2 Experiment 3: Feasibility evaluation

This experiment targets RQ2: *Can we measure fun and balance of character mechanics in a simulated game?*

This question regards the utility of the constraints we have defined on characters based on balance and fun. The constraints are defined by criteria through simulation-based evaluation, section 4.5. To answer the question, we evaluated the correspondence between the feasibility score given by our criteria and the enjoyment of human players. Recall that the feasibility score is given based on a character's distance from feasibility. Our hypothesis entails that the lower feasibility score a character gets, the less likely this character is to be considered interesting by humans.

Match number	Character pairs
1	1.1 vs 2.1
2	1.2 vs 2.2
3	1.3 vs 2.3
4	1.4 vs 2.4

Table 5.5: The characters that were compared to each other.

Feasible set s_1

Character number	Feasibility score
1.1	1
1.2	1
1.3	1
1.4	1

Infeasible set s_2

Character number	Feasibility score
2.1	0.2
2.2	0.4
2.3	0.6
2.4	0.8

Table 5.4: The two sets of characters compared to each other. Set s_1 only contains feasible characters. Set s_2 contains infeasible characters of different feasibility scores.

To test the hypothesis, we randomly sampled two sets of four generated characters from the evolution runs. The first set, s_1, only contained feasible characters. The second set s_2 contained infeasible characters of different feasibility scores. The feasibility scores of s_1 and s_2 are found in table 5.4.

All these characters were played and compared to each other by eight human playtesters, P_1-P_8. None of the testers knew anything about the feasibility score of any of the characters. To compare two characters, each playtester was instructed to use both characters to play four matches versus two of the existing characters from the Sol game. After playthrough, the playtester was asked the question "Which of the two characters you played was more interesting to play?", where "fun, balanced and perceived creative" was the working definition of the word "interesting". The pairs of characters tested against each other is found in table 5.5.

We will further present the quantitative results given for the character comparison, followed by qualitative results based on comments and feedback from the test subjects.

Results of Experiment 3

Summary of the responses from the playtesters is found in table 5.6. It can be seen that the feasible characters outperformed the infeasible approximately 80%

Match number	Win % character 1 (feasible character)	Win % character 2 (infeasible character)	Character 2 Feasibility score
1	93.75%	6.25%	0.2
2	93.75%	6.25%	0.4
3	87.5%	12.5%	0.6
4	43.75%	56.25%	0.8
Average	79.6875%	20.315%	0.5

Table 5.6: The results based on the answers from P_1-P_8.

of the tested cases. Furthermore, infeasible characters of higher feasibility scores were more likely to outperform feasible characters than those of lower feasibility scores, which is in harmony with our hypothesis. The character with a feasibility score of 0.8 outperformed its feasible counterpart in 9 out of 16 comparisons. This suggests that the feasibility score function is flawed in certain cases.

As the random sample size of feasible and infeasible characters is relatively small, as well as the number of playtesters, we cannot unambiguously draw conclusions about the utility of the feasibility score function. However, the data we have suggests that the feasibility score is rather accurate. Each pair was tested a total of 16 times, so with 4 pairs tested, the total number of tests between feasible and infeasible pairs is 64. In 51 out of these 64 comparisons, the feasible character outperformed the infeasible. Applying the null hypotheses to this result gives us a probability of

$$P(X \geq 51) = \sum_{k=51}^{64} \frac{64!}{(64-k)!k!} \cdot 0.5^k \cdot 0.5^{64-k} < 0.000001 = 0.0001\% \qquad (5.2)$$

which makes the result statistically significant. It is very unlikely that the feasibility function has no value concerning the interestingness of characters. The responses suggest that the feasibility function has a harder time distinguishing close to feasible characters from feasible characters in accordance with what humans find interesting. It should, however, be pointed out that the character sample set is limited, and that to be able to draw stronger conclusions, we would need to perform several more samples of characters.

As can be seen in table 5.6, character 2.4 was preferred to character 1.4 in 56.25% of the cases. The most common reason participants preferred character 2.4 over character 1.4 (seen in table 5.4) was that they perceived character 1.4 to be overpowered, that it was too easy to beat the computer player. The character

was equipped with large and strong abilities. This points towards a weakness in the *character balance* criteria seen in section 4.4.1. Thus, the feasibility ranges defined for the criteria might be inaccurate. The criteria are also dependent on the feasibility of the existing characters, as they are used as a basis for the simulation-based feasibility evaluation, and thus act as heuristics on interesting characters. This observation can also indicate that this heuristic has some downsides.

Another observation was made regarding the overall experience. In a few of the comparisons, none of the characters were experienced as especially enjoyable. This was discovered through feedback comments. This observation shows that feasible characters are not guaranteed to be interesting. Our findings presented earlier still point toward a meaningful distinction of characters based on the feasibility score.

5.3.3 Experiment 4: Evaluating generated characters

This experiment is related to RQ3: *Can constrained novelty search yield character mechanics that humans find interesting?*

To answer RQ3 through human testing, we need a reference point of what is considered an interesting character. Therefore, we compared generated characters from evolution to the set of existing human-designed characters we consider interesting, presented in section 4.1.

First, three runs of each of the four evolution variants, discussed in experiment 2 (section 5.2), were performed. The six most diverse and feasible characters from the final population of any of the evolution variants where chosen. The diversity was evaluated by the permutation of six characters that were most divers relative to each other. The diversity was given by the average distance between all six characters, and the distance function discussed in section 4.3.2 was used. Six characters were taken from a single population because the generation method promotes diversity within a single run. Thus, there is no assumption of the novelty of characters across different runs.

For the user studies, the six chosen generated characters were compared to six human-designed characters. Each of the seven test subjects (one test subject from Experiment 3 could not participant in this experiment) played three batches of four different characters. They were then asked to rank the four characters of each batch from 1 to 4 based on how they enjoyed playing them, where 1 is most enjoyable and 4 is least enjoyable. We further asked them to comment on any unique or especially interesting aspects of a character, to get feedback corresponding to the interestingness of the characters. First, we will present the ordering feedback, then some comments that were given to the characters.

Table 5.7 shows the character reference number and status of each character used in Experiment 4. Each of the three batches contained two generated characters, and two human-designed characters, as displayed in table 5.8.

Character	Status
C1	Generated
C2	Human designed
C3	Generated
C4	Human designed
C5	Generated
C6	Human designed
C7	Generated
C8	Human designed
C9	Generated
C10	Human designed
C11	Generated
C12	Human designed

Table 5.7: The 12 characters used in Experiment 4. The *Status* column shows if the character is generated through our system or human designed

Batch 1

Character	Status
C1	G
C2	Hd
C3	G
C4	Hd

Batch 2

Character	Status
C5	G
C6	Hd
C7	G
C8	Hd

Batch 3

Character	Status
C9	G
C10	Hd
C11	G
C12	Hd

Table 5.8: The three character batches used in Experiment 4. The *Status* column shows if the character is generated (G), or human designed (Hd)

Results of Experiment 4

The character rankings according to our test subjects P_1-P_7 are displayed in table 5.9. The *Overall* column displays the overall best ranked character, based on a ranking score, and the *Score* column shows the ranking score of the best ranked character. The ranking score is given to each character by:

$$score_{C_j} = \sum_{i=1}^{7} (rank_{C_j}(P_i) - 1)$$

Where $score_{C_j}$ is the ranking score of the j^{th} character. $rank_{C_j}(P_i)$ is the ranking of the j^{th} character by subject P_i.

Batch 1 character rankings

Ranking	P_1	P_2	P_3	P_4	P_5	P_6	P_7	Overall	Score
1	C2	C3	C1	C2	C3	C4	C3	C3	14
2	C3	C1	C4	C4	C1	C3	C1	C1	12
3	C1	C2	C2	C3	C4	C1	C2	C2	9
4	C4	C4	C3	C1	C4	C2	C4	C4	8

Batch 2 character rankings

Ranking	P_1	P_2	P_3	P_4	P_5	P_6	P_7	Overall	Score
1	C6	C7	C8	C7	C5	C7	C6	C8	12
2	C8	C8	C5	C8	C6	C8	C5	C7	11
3	C7	C5	C6	C5	C7	C5	C8	C5	10
4	C5	C6	C7	C6	C8	C6	C7	C6	9

Batch 3 character rankings

Ranking	P_1	P_2	P_3	P_4	P_5	P_6	P_7	Overall	Score
1	C11	C9	C10	C9	C11	C9	C11	C11	13
2	C12	C10	C12	C10	C12	C12	C10	C9	11
3	C9	C11	C11	C11	C10	C11	C9	C10	10
4	C10	C12	C9	C12	C9	C10	C12	C12	8

Table 5.9: The character rankings by the seven test subjects, P_1-P_7, for the three batches in experiment 4. The *Overall* column for each batch shows the character with the best ranking score. The *Score* column shows the score of the best ranked character.

The rankings show a slight positive correlation between high ranking and generated characters. The aggregated score of the generated characters was 71, whereas the aggregated score of human-designed characters was 56.

We will mention two characters that were pointed out by several subjects to be different from the other played characters, namely C7 and C11 seen in table 5.7. C7 had one ability that was especially interesting, pointed out by the subjects. It was a projectile ability with a relatively large size, really slow speed and persisted for a long time (respectively corresponding to the ability's *radius*, *speed* and *activeTime* properties, seen in section 4.2). The subjects noted that this ability could be used to "lay mines" that could be used to trap the opponent.

Character C11 was pointed out because it had an intriguing mechanic. C11 also had the best score in *batch 3*. The character featured a melee ability (the hitbox did not move) that was distanced rather far from the character, along with above average movement speed. It could be used fairly frequently, without being too strong. The comments stated that character *C11* was unique relative to the other played characters.

Two character types were noted for several of the generated characters. Namely the *mosquito* and the *bully* type. A mosquito character is small, moves and attacks fast, but with low damage. The bully type on the other hand refers to characters that are large and slow, with powerful attacks.

The *composition* of the characters was commented heavily throughout the user studies. The composition is the relation between a character's three abilities, movement speed, and size. In most of the cases, comments were made by the lack of composition. Some abilities were rendered useless as another ability was always a better choice. This led to worse playing experience. Some characters were, on the other hand, pointed out to have good composition. Another point mentioned was the relation between the inputs to perform an ability. Each ability corresponds to a single key on the keyboard, given by the order of the abilities in the representation (section 4.2). It was assumed that the *effectiveness* (mentioned in section 4.2) of abilities corresponding to each respective input, the *key mapping*, was consistent across different characters. This was not the case for most of the generated characters. The two character aspects, composition, and key mapping were not directly handled by our criteria. However, the user study shows that they might be valuable heuristics on fighting game character mechanics.

The results from Experiment 4 shows that characters generated with our applied method can compete with, or even outperform, human-designed characters on with regards to interestingness. Given that we consider our characters interesting, the rankings according to the test subjects are promising to confirm RQ2.

The qualitative results based on feedback from test subjects further strengthens the assumption that our method can generate interesting characters. Different character types were noted along with unique, intriguing abilities. This study also shows that our method can not guarantee high quality for all the generated characters. However, out of the six generated characters that were evaluated, several were confirmed interesting.

5.3.4 Discussion of the user study

There are several uncertainties related to performing user studies based on human playtesting in the Sol game. To properly evaluate the interestingness of

a character, one should preferably have much more time familiarizing with the game and each of the characters. That would give more certain answers than can be captured by one session of playtesting in a little more than one hour.

In Experiment 3 (section 5.3.2), test subjects are told to compare two characters. With this feedback method, there is not much room for nuance. A third option stating "equally enjoyable" could be provided. The characters could also be compared on a scale stating to what degree one character is preferred over another. Comparing on a scale might on the other hand yield inaccurate results, as it is easier to be certain about a relative ordering.

The number of test subjects is rather small, with respectively 8 and 7 participating in experiments 3 and 4. To reach a higher precision of the presented results, more subjects are needed. Representation from a wider demographical distribution of subjects would also be beneficial.

The study sessions were rather long, close to one and a half hours. A person's responses can vary on the person's emotional state, which varies over the course of a session.

Furthermore, there is no doubt that the visual and audio aspects of playing a game play a huge role in interestingness and enjoyment. Though this is true, the subjects seemed to appreciate the testing. Most of the subjects had prior experience with fighting games or similar games, such that they showed appreciation for the mechanics even though the graphics were limited.

A common theme among many of the participants we observed for experiment 4 (section 5.3.3) was that ranking characters from 1^{st} to 4^{th} was not easy. Some of the players said they were not comfortable enough giving a definitive ranking, as they did not get to know some characters well enough to discover their potential. In some of the cases of ranking characters, the participants told us that two or more characters were hard to distinguish, so a ranking would be misleading. Since they were instructed to make a choice regardless, the final rankings contain uncertainties.

Some of the test subjects pointed out that the computer player was too skilled, and that this made character evaluation harder. Mainly the subjects that were less experienced in the fighting game genre. Another comment expressed that the computer player seemed relatively human-like, except for the fact that it was always moving and never stopped to think. Comments on the balance between the played character and the opponent character were made. The comments suggested that the perceived balance of a played character was relative to the opponent, the computer-controlled player. The balance was observed to be an important factor, as a character that made it too easy or too hard to win affected

the enjoyment. This suggests that the results are related to the skill of the individual test subjects relative to the computer player.

Other recurring comments was that characters with slow abilities and low movement speed were frustrating to play. They often did not feel "smooth" in the sense that the character felt like it was not responding to the user input. This comment can arguably be partly explained by the lack of graphics and animations associated with the characters. In a full-featured fighting game, the character animation often indicates the strength and speed of an attack. Larger and slower characters tend to have stronger, but slower moves. In the Sol game, animation or graphical aspects could not properly communicate the speed or power of an attack to the players. The lack of smooth control could also be since faster character movement and attacks tend to be preferable to slower moves.

Chapter 6

Conclusion

In this thesis, we have presented a system for generating interesting characters in a fighting game, Sol. The research is based on the field of procedural content generation, where evolutionary algorithms are often applied. Simulation-based evaluation is an approach that has been used to estimate the player experience of generated content in PCG. These concepts were presented in chapter 2, along with the fighting game domain and theoretical background on theories of fun and creativity in games. The system presented in this thesis is inspired by work conducted to generate similar content in similar game domains. This was presented in chapter 3, along with the algorithms Novelty search from the class of evolutionary algorithms.

We presented a method for generating interesting character mechanics in fighting games in chapter 4. The method was based on feasible-infeasible 2 population constrained novelty search with the objective of novelty. Constraints on the search space were given through simulation-based evaluation. Criteria were defined to evaluate the feasibility of a character. Criteria were based on heuristics of what constitutes a fun and balanced character in fighting games. In chapter 5, the proposed method was applied, and experiments were conducted to tune the method and to examine the three research questions, RQ1, RQ2, and RQ3.

Further in this chapter, we discuss our findings with respect to the research goal and research questions in section 6.1. The implications and value of our work within the field of PCG and AI for fighting games is discussed in section 6.2. In the last two sections, we reflect on the limitations of our research and suggest how future work can improve and expand on the knowledge gained.

6.1 Discussion

Our research was an ambitious attempt at exploring ways to automate the design process of game mechanics. The Research goal was stated as follows: "Explore generation of game mechanics in fighting games through the use of evolutionary algorithms". Through our research, we came across different evolutionary algorithms (EAs) that could be applied in our method. Through trial and error, we ended up with Constrained Novelty evolution. We believed the algorithm would be valuable for generating interesting character mechanics. We found Novelty search to be appropriate as it values diversity more than other variants of EAs with diversity maintenance, and we considered novelty an important aspect of what is perceived interesting (presented in section 2.4). Evaluation of player experience, i.e., what is considered balanced and fun had promising applications in related work through simulation-based evaluation. We wanted to end up with a set of diverse and interesting characters. Encouraging diversity to a high degree through constrained novelty search with constraints on player experience was therefore believed to be appropriate for our proposed method.

To further examine the research goal, three research questions where defined. Given our system to generate interesting fighting game mechanics, the research questions were defined to evaluate the system. The system could then indicate that interesting character mechanics can be generated for fighting games, based on obtained results. First, an experiment was conducted to define parameter values of the generation method in which the four existing characters in the Sol game were used as heuristics on interesting characters. This experiment was necessary to tune the method.

The first research question was examined through Experiment 2, and was stated as follows.
RQ1: How can constrained novelty search be utilized to evolve interesting character mechanics?
Through our research, two methods of constrained novelty search showed the most promise, namely the *FINS* and *FI2NS* algorithms, discussed in section 4.3.1. An assumption was made, that the initial population used in the generation method could have a great impact. Thus, the two initial population variants discussed in section 4.3.3 were evaluated with each of the two search algorithms and forms four variants that were compared. The variants were compared on *diversity* and *feasible number* (number of feasible characters) in the final population. The results of the experiments showed that FI2NS with an existing character-based initial population had the best performance on diversity. FINS with a random initial population and FINS with an existing character-based initial population showed the highest number of feasible individuals. However, none of the four variants

performed remarkably better than the others on any metrics and no statistical significance could be proven.

We refer to the result of experiment 3 (section 5.3.2) to examine RQ2.
RQ2: Can we measure fun and balance of character mechanics in a simulated game?
Based on the responses of the test subjects, the feasible characters from the simulation-based evaluation outperformed the infeasible characters in approximately 80% of tested cases. This suggests that it is possible to quantify fun and balance of characters using simulation-based evaluation somewhat consistently. The quantification was based on constraints in the search, rather than a fitness function that could be optimized for. The constraints were given by a feasibility evaluation, defined by criteria on simulated games. The feasibility evaluation function we used had many similarities with more traditional fitness functions. One of the differences between them, however, are how the functions typically are used in EAs. Fitness functions are used as the objective for evolution, whereas our feasibility evaluation function is used as a constraint on individuals. Another common difference is that fitness functions often return a continuous range of values, in contrast to the discrete values assigned by our feasibility evaluation function. We believe that continuous fitness functions require a more precise evaluation of our criteria. Quantifying a fine-grained fitness function based on interestingness is not a trivial task.

RQ3 was formulated as follows.
RQ3: Can constrained novelty search yield character mechanics that humans find interesting?
Experiment 4 in section 5.3.3 was conducted to examine RQ3 by comparing the interestingness of generated characters to the existing, human-designed characters, based on the experience of test subjects. The results of experiment 4 show that the applied generation system is capable of producing interesting characters, but not consistently, however.

The results from our experiments indicate that the generation method is capable of generating interesting fighting game character mechanics to a certain degree. The results also showed that there is room for improvement in the method. The constraints on player experience, given through the criteria are not capable to distinguish fun and balanced characters in all cases. This is further shown in experiment 4, where some of the generated characters seemed uninteresting by the test subjects. There are several reasons why this may be. Possible reasons will be further presented in section 6.3.

6.2 Implications

The research conducted in this thesis has shown that interesting character mechanics can to some degree be generated for fighting games. The work presented can be seen as an initial attempt to combine content generation through constraint novelty search with constraints based on simulation-based evaluation in the domain of fighting games. To our knowledge, this is also a first attempt at generating character mechanics in a fighting game.

In the field of automating design for video games, this work can be seen as a contribution to expanding the application of existing methods. As discussed in chapter 3, several other researchers have succeeded in developing algorithms and systems for generating game levels and game rules through the use of fully autonomous PCG. This work show results in applying a similar approach to fighting game mechanics.

Mixed-initiative systems (explained in section 2.1) have been created for generating character weapons and character abilities using interactive fitness evaluation in role-playing games. We have shown a method for generating character mechanics through autonomous PCG. Simulation-based evaluation was used, and thus removed the need for human involvement in the generation process.

Our results indicate that the generation method can not guarantee quality for all generated characters. Thus, the approach is not suitable to be used for generating characters that can be used directly in a game. The method could be suitable as a design tool that a game designer could use for inspiration. This could promote better character designs and a more efficient design process.

The generation method presented can be applied to other fighting games, as the Sol game is similar to other fighting games (shown in section 4.1). However, our system is heavily based on technical aspects of the Sol game. Several heuristics in the simulation-based evaluation and especially the computer player are also based on this game. The presented method is dependent on four priorly existing characters that were human designed for the Sol game. They were used in the game simulations and for generating the initial population for the novelty search. This entails dependence on human-designed characters that must be present in a game. The main takeaway of our applied method is the promising concept of constrained novelty search with simulation-based feasibility evaluation, with the presented criteria.

6.3 Limitations

There are several limitations to our work. We will discuss those we consider the most relevant.

Only a few variants of evolutionary algorithms were tested for the proposed method. Testing and comparing more evolution strategies could produce better results. There are several different evolutionary search strategies that could be used. If the simulation-based criteria (from section 4.4) that are used as heuristics on *fun* and *balance* were formulated as objectives, several other evolutionary search approaches could be tested. Novelty could be achieved by methods such as niching (presented in section 2.2). Multi-objective evolutionary algorithms could also be tested, where criteria could be used as different objectives. Further experimentation is needed to examine the presented approach. Furthermore, we could also research the effect of the different parameters used in for the evolution to determine their effect on the outcome.

The results from Experiment 4 (section 5.3.3) suggest that several generated characters are perceived as similar. There are several reasons that may explain this effect. The evolutionary operators, presented in section 4.3.4, can be limited in their capability of generating character offspring that are sufficiently diverse. The four variants of the novelty search algorithms that were evaluated in Experiment 2 (section 5.2) showed little variance in the possibility of generating diverse characters. This result might be affected by the potential limited capabilities of the mutational operators to produce diverse offspring.

The distance function that was applied (seen in section 4.3.2) might also affect the novelty of generated characters. The distance function was applied directly to the character genotype. Thus, the distance may not reflect the actual distance of two characters perceived by a human player. The relation between the genotype of a character and a character presented in the Sol game was discussed in section 4.2.

The existing characters in the Sol game were used as heuristics on interesting characters in multiple parts of our generation method. The feasibility ranges of our criteria were defined based on the performance of the existing characters, seen in Experiment 1 (section 5.1). The evaluation of generated characters was also made by playing them against the existing characters (seen in section 4.4). One of the initial population generators was also based on the existing characters. The strong dependence on the existing characters is not optimal, as we have no results proving that they are interesting, except for the authors' experience. Thus, this dependency on the existing characters should preferably be removed, or the characters should be further evaluated. For the feasibility ranges, user

studies could be applied to determine the values.

The criteria that were applied can not be assumed to evaluate all aspects of what constitutes an *interesting* character. Further exploration of criteria suitable for a fighting game is necessary to define more accurate heuristics of characters in fighting games. The capability of each criterion to estimate interestingness was not investigated but could be done to improve the accuracy. Inspiration could be taken from Brown and Maire, who performed user studies to evaluate the importance of their criteria for simulation-based evaluation [4]. Qualitative results from our user studies show aspects of character mechanics that were not accounted for in our criteria, seen in section 5.3.3. These aspects include *character composition* (the relation between a character's abilities, movement speed, and size) and the *key binding* (the relation between the effect of abilities and the player's input keys). These observations could be used to define further heuristics on interesting fighting game characters.

We want to point out that the Sol game used in this study is a relatively simple game. The representation of the generated character mechanics is limited relative to more complex fighting games seen in the industry. Many other games may have vastly different characters that might be hard to represent with a simple parameterized representation, as done in this research (see section 4.2). With larger and more complex representations, the search will be much larger. Performing a search on such domains can be very time consuming, such that more heuristics are needed for the search.

6.4 Future work

In section 6.3 we pointed out several limitations that can be considered for future work. Other evolution strategies and the effect of the different evolution parameters can be further researched. To evaluate the value of this work in the general field of fighting games and AI, the system should be implemented for a well known fighting game.

Other suggestions for improvement and future work is to utilize more sophisticated AI methods for the computer player used in the simulation-based evaluation. For example Monte Carlo tree search or variants of reinforcement learning, that have previously been used for game-playing AI in fighting games (see section 2.3.2). More focus on human-like play styles for the computer player could improve the value of the simulation-based evaluation.

Bibliography

[1] K. Asayama, K. Moriyama, K. Fukui, and M. Numao. "Prediction as faster perception in a real-time fighting video game". In: *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. 2015, pp. 517–522.

[2] Thomas Back. "Selective pressure in evolutionary algorithms: A characterization of selection mechanisms". In: *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*. IEEE. 1994, pp. 57–62.

[3] Tobias Blickle and Lothar Thiele. "A Comparison of Selection Schemes Used in Evolutionary Algorithms". In: *Evol. Comput.* 4.4 (Dec. 1996), pp. 361–394. ISSN: 1063-6560. DOI: 10.1162/evco.1996.4.4.361. URL: https://doi.org/10.1162/evco.1996.4.4.361.

[4] C. Browne and F. Maire. "Evolutionary Game Design". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.1 (Mar. 2010), pp. 1–16. ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2010.2041928.

[5] Cameron Bolitho Browne. "Automatic generation and evaluation of recombination games". PhD thesis. Queensland University of Technology, 2008. URL: https://eprints.qut.edu.au/17025/.

[6] Bobby Don Bryant. "Evolving Visibly Intelligent Behavior for Embedded Game Agents". PhD thesis. USA, 2006. ISBN: 9780549053514.

[7] Mihaly Csikszentmihalyi. "Flow: The Psychology of Optimal Experience". In: Jan. 1990.

[8] Gustavo Danzi, Andrade Hugo Pimentel Santana, André Wilson Brotto Furtado, André Roberto Gouveia, Amaral Leitao, and Geber Lisboa Ramalho. "Online adaptation of computer games agents: A reinforcement learning approach". In: *II Workshop de Jogos e Entretenimento Digital*. 2003, pp. 105–112.

[9] Kenneth Alan De Jong. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." AAI7609381. PhD thesis. USA, 1975.

[10] O. Drageset, M. H. M. Winands, R. D. Gaina, and D. Perez-Liebana. "Optimising Level Generators for General Video Game AI". In: *2019 IEEE Conference on Games (CoG)*. Aug. 2019, pp. 1–8. DOI: 10.1109/CIG.2019.8847961.

[11] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Springer, 2003.

[12] Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. "Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning". In: *CoRR* abs/1702.06230 (2017). arXiv: 1702.06230. URL: http://arxiv.org/abs/1702.06230.

[13] David E. Goldberg and Jon Richardson. "Genetic Algorithms with Sharing for Multimodal Function Optimization". In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, pp. 41–49. ISBN: 0805801588.

[14] E. J. Hastings, R. K. Guha, and K. O. Stanley. "Automatic Content Generation in the Galactic Arms Race Video Game". In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.4 (2009), pp. 245–263.

[15] Makoto Ishihara, Taichi Miyazaki, Chun Yin Chu, Tomohiro Harada, and Ruck Thawonmas. "Applying and Improving Monte-Carlo Tree Search in a Fighting Game AI". In: *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*. ACE '16. Osaka, Japan: ACM, 2016, 27:1–27:6. ISBN: 978-1-4503-4773-0. DOI: 10.1145/3001773.3001797. URL: http://doi.acm.org/10.1145/3001773.3001797.

[16] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popovic. "Evaluating competitive game balance with restricted play". In: *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012.

[17] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius. "General video game rule generation". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. Aug. 2017, pp. 170–177. DOI: 10.1109/CIG.2017.8080431.

[18] Ahmed Khalifa, Diego Perez-Liebana, Simon M. Lucas, and Julian Togelius. "General Video Game Level Generation". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: ACM, 2016, pp. 253–259. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908920. URL: http://doi.acm.org/10.1145/2908812.2908920.

[19] Man-Je Kim and Chang Wook Ahn. "Hybrid Fighting Game AI Using a Genetic Algorithm and Monte Carlo Tree Search". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO

'18. Kyoto, Japan: ACM, 2018, pp. 129–130. ISBN: 978-1-4503-5764-7. DOI: 10.1145/3205651.3205695. URL: http://doi.acm.org/10.1145/3205651.3205695.

[20] Steven Orla Kimbrough, Gary J. Koehler, Ming Lu, and David Harlan Wood. "On a Feasible–Infeasible Two-Population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch". In: *European Journal of Operational Research* 190.2 (2008), pp. 310–327. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2007.06.028. URL: http://www.sciencedirect.com/science/article/pii/S0377221707005668.

[21] R Konečn. "Modeling of fighting game players". MA thesis. 2016.

[22] J. Lehman and K. O. Stanley. "Abandoning Objectives: Evolution Through the Search for Novelty Alone". In: *Evolutionary Computation* 19.2 (2011), pp. 189–223.

[23] Joel Lehman and Kenneth Stanley. "Novelty Search and the Problem with Objectives". In: Nov. 2011, pp. 37–56. DOI: 10.1007/978-1-4614-1770-5_3.

[24] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. "Computational Game Creativity". In: *ICCC*. 2014.

[25] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. "Constrained Novelty Search: A Study on Game Content Generation". In: *Evol. Comput.* 23.1 (Mar. 2015), pp. 101–129. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00123. URL: https://doi.org/10.1162/EVCO_a_00123.

[26] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. "Sentient sketchbook: Computer-aided game level authoring". In: *In Proceedings of ACM Conference on Foundations of Digital Games, 2013. In Print.*

[27] Simon Liu, Li Chaoran, Li Yue, Ma Heng, Hou Xiao, Shen Yiming, Wang Licong, Chen Ze, Guo Xianghao, Lu Hengtong, Du Yu, and Tang Qinting. "Automatic Generation of Tower Defense Levels Using PCG". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. FDG '19. San Luis Obispo, California: ACM, 2019, 10:1–10:9. ISBN: 978-1-4503-7217-6. DOI: 10.1145/3337722.3337723. URL: http://doi.acm.org/10.1145/3337722.3337723.

[28] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas. "Fighting game artificial intelligence competition platform". In: *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*. Oct. 2013, pp. 320–323. DOI: 10.1109/GCCE.2013.6664844.

[29] Kevin Majchrzak, Jan Quadflieg, and Günter Rudolph. "Advanced Dynamic Scripting for Fighting Game AI". In: *Entertainment Computing - ICEC 2015*. Ed. by Konstantinos Chorianopoulos, Monica Divitini, Jan-

nicke Baalsrud Hauge, Letizia Jaccheri, and Rainer Malaka. Cham: Springer International Publishing, 2015, pp. 86–99. ISBN: 978-3-319-24589-8.

[30] Thomas W. Malone. "What Makes Things Fun to Learn? Heuristics for Designing Instructional Computer Games". In: *Proceedings of the 3rd ACM SIGSMALL Symposium and the First SIGPC Symposium on Small Systems*. SIGSMALL '80. Palo Alto, California, USA: ACM, 1980, pp. 162–169. ISBN: 0-89791-024-9. DOI: 10.1145/800088.802839. URL: http://doi.acm.org/10.1145/800088.802839.

[31] R. Manner, Samir Mahfoud, and Samir W. Mahfoud. "Crowding and Preselection Revisited". In: *Parallel Problem Solving From Nature*. North-Holland, 1992, pp. 27–36.

[32] Simón E. Ortiz B., Koichi Moriyama, Ken-ichi Fukui, Satoshi Kurihara, and Masayuki Numao. "Three-Subagent Adapting Architecture for Fighting Videogames". In: *PRICAI 2010: Trends in Artificial Intelligence*. Ed. by Byoung-Tak Zhang and Mehmet A. Orgun. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 649–654. ISBN: 978-3-642-15246-7.

[33] Alex Pantaleev. "In Search of Patterns: Disrupting RPG Classes Through Procedural Content Generation". In: *Proceedings of the The Third Workshop on Procedural Content Generation in Games*. PCG'12. Raleigh, NC, USA: ACM, 2012, 4:1–4:5. ISBN: 978-1-4503-1447-3. DOI: 10.1145/2538528.2538532. URL: http://doi.acm.org/10.1145/2538528.2538532.

[34] L. H. Phuc, K. Naoto, and I. Kokolo. "Learning human-like behaviors using neuroevolution with statistical penalties". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 207–214. ISBN: 978-1-5386-3233-8. DOI: 10.1109/CIG.2017.8080437.

[35] I. P. Pinto and L. R. Coutinho. "Hierarchical Reinforcement Learning With Monte Carlo Tree Search in Computer Fighting Game". In: *IEEE Transactions on Games* 11.3 (Sept. 2019), pp. 290–295. ISSN: 2475-1510. DOI: 10.1109/TG.2018.2846028.

[36] Graeme Ritchie. "Some Empirical Criteria for Attributing Creativity to a Computer Program". In: *Minds and Machines* 17.1 (2007), pp. 67–99. ISSN: 1572-8641. DOI: 10.1007/s11023-007-9066-2. URL: https://doi.org/10.1007/s11023-007-9066-2.

[37] Simardeep S. Saini. *Mimicking human player strategies in fighting games using game artificial intelligence techniques*. Jan. 2014. URL: https://hdl.handle.net/2134/16380.

[38] N. Sato, S. Temsiririrkkul, S. Sone, and K. Ikeda. "Adaptive Fighting Game Computer Player by Switching Multiple Rule-Based Controllers". In: *2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*. July 2015, pp. 52–59. DOI: 10.1109/ACIT-CSI.2015.18.

[39] Jacob Schrum, Igor V. Karpov, and Risto Miikkulainen. "Human-Like Combat Behaviour via Multiobjective Neuroevolution". In: *Believable Bots: Can Computers Play Like People?* Ed. by Philip Hingston. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 119–150. ISBN: 978-3-642-32323-2. DOI: 10.1007/978-3-642-32323-2_5. URL: https://doi.org/10.1007/978-3-642-32323-2_5.

[40] Penelope Sweetser and Peta Wyeth. "GameFlow: A Model for Evaluating Player Enjoyment in Games". In: *Comput. Entertain.* 3.3 (July 2005), pp. 3–3. ISSN: 1544-3574. DOI: 10.1145/1077246.1077253. URL: http://doi.acm.org/10.1145/1077246.1077253.

[41] Y. Takano, S. Ito, T. Harada, and R. Thawonmas. "Utilizing Multiple Agents for Decision Making in a Fighting Game". In: *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*. Oct. 2018, pp. 594–595. DOI: 10.1109/GCCE.2018.8574675.

[42] Y. Takano, W. Ouyang, S. Ito, T. Harada, and R. Thawonmas. "Applying Hybrid Reward Architecture to a Fighting Game AI". In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. Aug. 2018, pp. 1–4. DOI: 10.1109/CIG.2018.8490437.

[43] Chin Hiong Tan, Kay Chen Tan, and Vui Ann Shim. "Learning believable game agents using sensor noise and action histogram". In: *Memetic Computing* 6.4 (2014), pp. 215–232.

[44] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. "Search-Based Procedural Content Generation: A Taxonomy and Survey". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186.

[45] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. "Search-Based Procedural Content Generation: A Taxonomy and Survey". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.03 (July 2011), pp. 172–186. ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2011.2148116.

[46] K. Yamamoto, S. Mizuno, Chun Yin Chu, and R. Thawonmas. "Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator". In: *2014 IEEE Conference on Computational Intelligence and Games*. Aug. 2014, pp. 1–5. DOI: 10.1109/CIG.2014.6932915.

[47] Georgios N. Yannakakis, Antonios Liapis, and Constantine Alexopoulos. "Mixed-initiative co-creativity". In: *FDG*. 2014.

[48] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. http://gameaibook.org. Springer, 2018.

[49] Xiaodong Yin and Noël. Germay. "A Fast Genetic Algorithm with Sharing Scheme Using Cluster Analysis Methods in Multimodal Function Optimization". In: *Artificial Neural Nets and Genetic Algorithms*. Ed. by Rudolf F.

Albrecht, Colin R. Reeves, and Nigel C. Steele. Vienna: Springer Vienna, 1993, pp. 450–457.

[50] S. Yoon and K. Kim. "Deep Q networks for visual fighting game AI". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. Aug. 2017, pp. 306–308. DOI: 10.1109/CIG.2017.8080451.

[51] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada, and R. Thawonmas. "Application of Monte-Carlo tree search in a fighting game AI". In: *2016 IEEE 5th Global Conference on Consumer Electronics*. Oct. 2016, pp. 1–2. DOI: 10.1109/GCCE.2016.7800536.

[52] K. Yu and N. R. Sturtevant. "Application of Retrograde Analysis on Fighting Games". In: *2019 IEEE Conference on Games (CoG)*. Aug. 2019, pp. 1–8. DOI: 10.1109/CIG.2019.8848062.

[53] Adeel Zafar, Hasan Mujtaba, and Mirza Beg. "Search-based procedural content generation for GVG-LG". In: *Applied Soft Computing* 86 (Nov. 2019), p. 105909. DOI: 10.1016/j.asoc.2019.105909.

Appendix A: Representation of the existing characters

```
{
  "name": "Frank",
  "radius": 32,
  "moveVelocity": 500,
  "abilities": [
    {
      "name": "rapid shot",
      "type": "PROJECTILE",
      "radius": 8,
      "distanceFromChar": 32,
      "speed": 1200,
      "activeTime": 30,
      "startupTime": 2,
      "executionTime": 0,
      "endlagTime": 2,
      "rechargeTime": 30,
      "damage": 100,
      "baseKnockback": 200,
      "knockbackRatio": 0.5,
      "knockbackPoint": -128,
      "knockbackTowardPoint": false
    },
    {
      "name": "hyper beam",
      "type": "PROJECTILE",
      "radius": 20,
      "distanceFromChar": 32,
      "speed": 1500,
      "startupTime": 15,
      "activeTime": 120,
      "executionTime": 1,
      "endlagTime": 10,
      "rechargeTime": 120,
      "damage": 300,
      "baseKnockback": 400,
      "knockbackRatio": 0.8,
      "knockbackPoint": -256,
      "knockbackTowardPoint": false
    },
    {
      "name": "puffer",
      "type": "MELEE",
      "radius": 98,
      "distanceFromChar": 0,
      "speed": 0,
      "activeTime": 2,
      "startupTime": 8,
      "executionTime": 2,
      "endlagTime": 8,
      "rechargeTime": 180,
      "damage": 20,
      "baseKnockback": 1300,
      "knockbackRatio": 0.1,
      "knockbackPoint": 0,
      "knockbackTowardPoint": false
    }
  ]
}
```

```
{
  "name": "Schmathias",
  "radius": 32,
  "moveVelocity": 600,
  "abilities": [
    {
      "name": "Frog Punch",
      "type": "MELEE",
      "radius": 64,
      "distanceFromChar": 48,
      "speed": 0,
      "activeTime": 5,
      "startupTime": 3,
      "executionTime": 5,
      "endlagTime": 3,
      "rechargeTime": 20,
      "damage": 150,
      "baseKnockback": 700,
      "knockbackRatio": 0.8,
      "knockbackPoint": -48,
      "knockbackTowardPoint": false
    },
    {
      "name": "Hook",
      "type": "PROJECTILE",
      "radius": 24,
      "distanceFromChar": 32,
      "speed": 900,
      "startupTime": 5,
      "activeTime": 30,
      "executionTime": 0,
      "endlagTime": 18,
      "rechargeTime": 50,
      "damage": 200,
      "baseKnockback": 1400,
      "knockbackRatio": 0.2,
      "knockbackPoint": -128,
      "knockbackTowardPoint": true
    },
    {
      "name": "Meteor Punch",
      "type": "MELEE",
      "radius": 32,
      "distanceFromChar": 64,
      "speed": 0,
      "activeTime": 3,
      "startupTime": 15,
      "executionTime": 3,
      "endlagTime": 4,
      "rechargeTime": 60,
      "damage": 500,
      "baseKnockback": 1000,
      "knockbackRatio": 1.5,
      "knockbackPoint": -128,
      "knockbackTowardPoint": false
    }
  ]
}
```

```
{
  "name": "Brail",
  "radius": 44,
  "moveVelocity": 600,
  "abilities": [
    {
      "name": "Chagger",
      "type": "MELEE",
      "radius": 70,
      "distanceFromChar": 64,
      "speed": 0,
      "activeTime": 6,
      "startupTime": 6,
      "executionTime": 6,
      "endlagTime": 6,
      "rechargeTime": 30,
      "damage": 150,
      "baseKnockback": 600,
      "knockbackRatio": 1.2,
      "knockbackPoint": 400,
      "knockbackTowardPoint": true
    },
    {
      "name": "Light Force",
      "type": "PROJECTILE",
      "radius": 64,
      "distanceFromChar": 44,
      "speed": 650,
      "startupTime": 12,
      "activeTime": 30,
      "executionTime": 0,
      "endlagTime": 6,
      "rechargeTime": 120,
      "damage": 300,
      "baseKnockback": 400,
      "knockbackRatio": 0.8,
      "knockbackPoint": 64,
      "knockbackTowardPoint": false
    },
    {
      "name": "Merge",
      "type": "MELEE",
      "radius": 160,
      "distanceFromChar": 128,
      "speed": 0,
      "activeTime": 2,
      "startupTime": 10,
      "executionTime": 2,
      "endlagTime": 8,
      "rechargeTime": 60,
      "damage": 20,
      "baseKnockback": 800,
      "knockbackRatio": 0.4,
      "knockbackPoint": 0,
      "knockbackTowardPoint": true
    }
  ]
}
```

```json
{
  "name": "MagneT",
  "radius": 36,
  "moveVelocity": 600,
  "abilities": [
    {
      "name": "Spear Poke",
      "type": "MELEE",
      "radius": 30,
      "distanceFromChar": 100,
      "speed": 0,
      "activeTime": 6,
      "startupTime": 8,
      "executionTime": 3,
      "endlagTime": 0,
      "rechargeTime": 13,
      "damage": 150,
      "baseKnockback": 600,
      "knockbackRatio": 0.7,
      "knockbackPoint": -100,
      "knockbackTowardPoint": false
    },
    {
      "name": "Spear Throw",
      "type": "PROJECTILE",
      "radius": 32,
      "distanceFromChar": 36,
      "speed": 1500,
      "startupTime": 30,
      "activeTime": 40,
      "executionTime": 0,
      "endlagTime": 18,
      "rechargeTime": 50,
      "damage": 200,
      "baseKnockback": 250,
      "knockbackRatio": 1,
      "knockbackPoint": -64,
      "knockbackTowardPoint": false
    },
    {
      "name": "Spirit of the Wild",
      "type": "PROJECTILE",
      "radius": 64,
      "distanceFromChar": 0,
      "speed": 200,
      "activeTime": 240,
      "startupTime": 30,
      "executionTime": 0,
      "endlagTime": 12,
      "rechargeTime": 120,
      "damage": 450,
      "baseKnockback": 200,
      "knockbackRatio": 0.5,
      "knockbackPoint": 0,
      "knockbackTowardPoint": false
    }
  ]
}
```

Appendix B: Predefined character constraints

Character body properties constraints

```
character_properties_ranges = {
    "radius": (28.0, 50.0),
    "moveVelocity": (200.0, 800.0)
}
```

Melee ability properties constraints

```
melee_ability_ranges = {
    "radius": (16.0, 200.0),
    "distanceFromChar": (0.0, 200.0),
    "speed": (0.0, 0.0),
    "startupTime": (1, 30),
    "activeTime": (1, 60),
    "executionTime": (1, 30),
    "endlagTime": (1, 30),
    "rechargeTime": (10, 30),
    "damage": (100.0, 1000.0),
    "baseKnockback": (10.0, 1000.0),
    "knockbackRatio": (0.1, 1.0),
    "knockbackPoint": (-500.0, 500.0),
    "knockbackTowardPoint": (False, True)
}
```

Projectile ability properties constraints

```
projectile_ability_ranges = {
    "radius": (5, 50),
    "distanceFromChar": (0, 200),
    "speed": (100, 800),
    "startupTime": (1, 60),
    "activeTime": (20, 1000),
    "executionTime": (1, 30),
    "endlagTime": (1, 30),
    "rechargeTime": (10, 120),
    "damage": (15, 500),
    "baseKnockback": (50, 1000),
    "knockbackRatio": (0.1, 1.0),
    "knockbackPoint": (-500, 500),
    "knockbackTowardPoint": (False, True)
}
```

Appendix C: User study questionnaire

User testing of generated characters in the Sol game

Sol is a fighting game where you control a character and play against another character. The other character will always be computer controlled for this test. The goal is to use attacks to push (knockback) the other character off the stage.

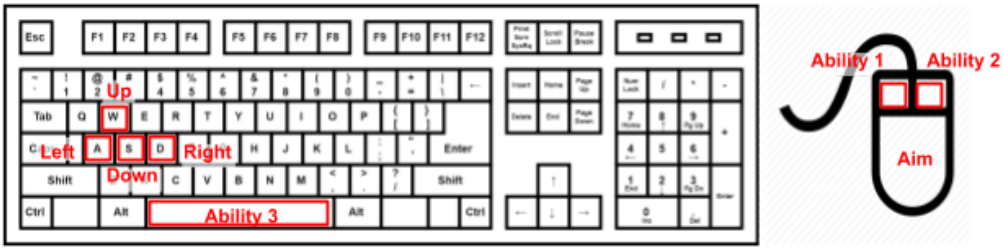
Set up

- First you need a java 11 runtime. It can be downloaded from here: https://adoptopenjdk.net/. Select the options *OpenJDK 11 (LTS)* and *HotSpot*.
- Then you need to download the game. You need git installed, then run `git clone --recurse-submodules https://github.com/sol-ai/solai_project.git` in a terminal.
- Then do `cd solai_project/solai_game_simulator`, you should now be in the game simulator directory.
- To play, you need to input the following command: `./gradlew bootRun --args="--experiment [experiment type],[experiment number]"` (no whitespace around the comma). [experiment type] and [experiment number] will be replaced as stated later.
- To test that everything works, run `./gradlew bootRun --args="--experiment test,0"` Hold the w button on the keyboard to die three times and let the game end.
- Note that in the terminal, you can use the arrow key up to get the last typed command.
- You should preferably have no other windows open on your pc when running the command, as the game will start behind the other windows

Game instructions

- The movement of your character is controlled by four directional keys: a (left), d (right), w (up) and s (down).
- A character has three attack abilities that may be executed by: left mouse button, right mouse button and the space bar.
- The character will always aim towards the mouse cursor.

The input buttons are seen in the following image:



Get familiar with the game

In the game you will see four kinds of objects. The pointy blue circles are the characters. You always control the character on the left. The other blue areas are walls that are impassable. The red area indicates where a character can fall off the stage. If a character's collides with a red area, it dies. If a character dies 3 times, the player that controls the character loses. When you perform abilities, they are displayed as blue pointy circles as well.

All abilities are used in the direction that the character aims (towards the mouse cursor). Abilities come in two variants, **melee attacks** and **projectile attacks**. Melee attacks do not move after they are used, while projectile attacks move forward. **Different attacks have varying damage, knockback, size speed among other varying properties**. Knockback can also be applied in the opposite direction of the attack. Abilities also have different **delays on your character**. That means that you might not be able to move for a brief period of time before and after an attack is used. No animations are present, but think of this as the time it takes for a character to perform a punch, kick or to shoot a bullet. There might also be some time before the same ability can be used in succession. Some abilities may be suitable to perform after another, so look for **ability combinations**.

The **computer player** is meant to approximate a pretty good player, so don't feel bad if you cannot defeat it, that is not the main goal of this test.

Test the game by running the command `--args="--experiment test,[experiment number]"` with the experiment number set to: 0, 1, 2 or 3. The different experiment numbers will let you play with different characters. Repeat each character if you'd like.

Test

You will be presented with different characters during the test. **Try out all the three abilities**, and look for **ability combinations** that might work. You can repeat the games with the same character multiple times if you would like to.

Please fill in the following:

Name (optional):	
Prior experience with fighting games from 1 (none) to 3 (much)	

Test 1

In test 1, we ask you to play two and two characters after each other. Then write down which one you enjoy the most. There is a total amount of 8 pairs. Enjoyment is subjective, so no answer is wrong. We would also like you to leave comments on each character, e.g. why you enjoy it why not.

Tu run the test, run the following command with experiment number as stated in the table below: `--args="--experiment e1,[experiment number]"` (“test” from the last section is changed to e1)

Experiment number	Comments on a character (optional)	Which one do you prefer?
0		
1		
2		
3		
4		
5		
6		
7		

8		
9		
10		
11		
12		
13		
14		
15		

Test 2

In this test we would like you to order 4 and four characters according to how you enjoy playing them. There are a total of 3 blocks of 4 characters. Comment if you found any **interesting or unique aspects of a character**.

Tu run the test, run the following command with experiment number as stated in the table below: `--args="--experiment e2,[experiment number]"` (e1 from the last test is changed to e2)

Experiment number	Comments on a character (optional)	Which one do you prefer? (0,1,2,3 for example)
0		
1		
2		
3		

Experiment number	Comments on a character (optional)	Which one do you prefer?
4		
5		
6		

7		

Experiment number	Comments on a character (optional)	Which one do you prefer?
8		
9		
10		
11		

Eirik H. Skjærseth and Harald Vinje

Evolutionary algorithms for generating interesting fighting game character mechanics

NTNU
Norwegian University of
Science and Technology