GENERAL LEVEL GENERATION

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Ahmed Khalifa

September 2020

GENERAL LEVEL GENERATION

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Ahmed Khalifa

September 2020

Approved:

Department Chair Signature

09/11/2020

Date

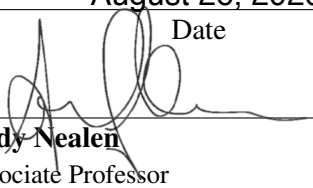
University ID: N15208753
Net ID: aak538

Approved by the Guidance Committee:

Major: Computer Science

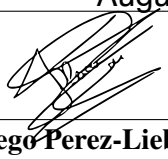
Julian Togelius
Associate Professor
Tandon School of Engineering
New York University

August 26, 2020
Date



Andy Nealen
Associate Professor
Cinematic Arts and Computer Science
University of Southern California

August 27, 2020
Date



Diego Perez-Liebana
Lecturer
School of Electronic Engineering and Computer Science
Queen Mary University of London

August 27, 2020
Date

Michael Cook
Research Fellow
School of Electronic Engineering and Computer Science
Queen Mary University of London

August 27, 2020
Date

Santiago Ontañón
Associate Professor
College of Computing and Informatics
Drexel University

August 27, 2020
Date

Microfilm/Publishing

Microfilm or copies of this dissertation may be obtained from:

UMI Dissertation Publishing
ProQuest CSA
789 E. Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Vita

Ahmed Khalifa

Education

PhD in Computer Science Sep. 2015 - Sep. 2020
New York University, Tandon School of Engineering, Game Innovation Lab

M.S. in Computer Engineering Sep. 2010 - May. 2015
Cairo University, Faculty of Engineering, Computer Engineering Department

B.S. in Computer Science Sep. 2005 - May. 2010
Cairo University, Faculty of Engineering, Computer Engineering Department

Research and Funding

This research was performed at the NYU Game Innovation Lab. Funding for tuition and personal expenses was granted by Prof. Julian Togelius from his start-up funds for the first two years. Later, the research was funded by NSF Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection.". Funding for traveling to present the papers that this work is based on and to attend conferences was provided by Prof. Andy Nealen, Prof. Julian Togelius and the Game Innovation Lab.

Professional Experience

Ahmed has worked as a teaching assistant at Faculty of Engineering, Cairo University, teaching Image Processing, Microprocessors, and various classes about object oriented programming. In 2009, he worked at Microsoft Research Center in Cairo developing tools for testing and debugging Bing Arabic translation and the Arabic spell checker in Microsoft word. He also worked as a game designer and developer on various independent games, of which several were nominated for prizes or selected to be showcased at game events. During his time at NYU, he has been a teaching assistent for courses on AI and AI for Games.

Acknowledgments

I would like to thank my advisor, Julian Togelius for being an awesome supervisor and supporter for me. I also thank Andy Nealen, my not formal co-advisor for challenging me and pushing my work to the next level. And a big thanks to Michael Cook for being an awesome person without your encouragement and help I wouldn't have started my Ph.D. degree. I thank my dissertation committee: Julian Togelius, Andy Nealen, Diego-Perez-Liebana, Michael Cook, and Santiago Ontañón. I thank all my co-authors as they made my research more fun: Fernando Silva, Gabriella Barros, Dan Gopstein, Michael Cerny Green, Scott Lee, Ivan Bravi, Philip Bontrager, Tiago Machado, Diego Perez-Liebana, Megan Charity, Chang Ye, Debosmita Bhaumik, Christoffer Holmgøard, Ruben Rodriguez Torrado, Niels Justesen, Amy Hover, Sam Earle, Aaron Isaksen, Luvneesh Mugrai, Andre Mendes, Daniele Gravina, Antonios Liapis, Georgios N. Yannakakis, Matthew Stephenson, Damien Anderson, Christoph Salge, Raluca D. Gaina, Omar Delarosa, Sebastian Risi, Matt Fontaine, Jialin Liu, Simon M. Lucas, Mike Preuss, Arthur Juliani, Hongwei Zhou, Yichen Gong, Hang Dong, Mindy Ruan, John Levine, Jochen Renz, Athoug Alsoughayer, Divyesh Surana, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, and Danny Lange.

A big thanks to the rest of the official and non-official members of the Game Innovation Lab for making the lab a welcome and a fun place to work at: Christopher O'Halloran, my dog (Koopa Troopa), Chris DiMauro, Erica Ribeiro, Rodrigo Canaan, Bruna Oliveira, Ming Jin, Lisa Soros, Catalina Jaramillo, Rammesy Nasser, Kaho Abe, Aaron Dharna, Annette Vera, Valeria Bontrager, Tae Jong Choi, and Mike Karlesky. I thank all my friends and family back in Egypt and around the world: my parents, my brothers (Mohamed and Mostafa), my sister (Hoda), Mohamed Zaki Ghazi, Mahmoud Ismail, Shehab Anwer, Mohamed Daif, Mohamed Hesham Fadl, Omar Shoukry, Ahmed Saker, Omar Sameh Shehata, Aly Amin, Ahmed Salem, Dina Tantawy, Magda Fayek, Ezzat Ismail Ezzat, Shehab Wagih, Hany Elsiouffy, Talha Kaya, and Mahmoud Nageeb. Finally, I would like to thanks all the friends that I made during my Ph.D. time for making my life fun during this period: Marco Scirea, Anita Simonsen, Jeremy Terry, Bashar Basim Makhay, Haitham Enasser, Farzad Alikozai, Sandy Ahmed, Ahmed Bessar, Tony Khalaf, Khadija Belmoudden, Keegan Hakim, Wardeh C. Hattab, Nick Baitoo, Vanessa Volz, Bayan Mashat, Sam Snodgrass, Matthew Guzdial, Chris Bamford, Mike Tynes, Michael Perrie Jr., Mark Hanke, Ron Orlovsky, Jonathon Lynch, Gillian Smith, Laura Hamel, Matt SanGiovanni, and Nabeel Abdur Rehman.

Dedication

I dedicate this work to my younger self that has always felt anxious, skeptical, alone, unworthy, and unloved.

ABSTRACT

GENERAL LEVEL GENERATION

by

Ahmed Khalifa

Advisor: Julian Togelius

Submitted in partial fulfillment of the Requirements for the Degree of Doctor of Philosophy (Computer Science)

September 2020

This thesis explores techniques and metrics for creating game level generators that can work for different games with minimal changes. We focus on methods that generate content by searching and that generate generators by searching. For the first approach, we explore search-based techniques to find a set of playable and diverse levels. We introduce the Constrained MAP-Elites algorithm to generate levels using a set of generalizable diversity and quality metrics. We test the technique on five different games: Super Mario Bros and four games in General Video Game AI corpus (Zelda, Solarfox, Plants, and Portals). The results show that on simple games with few mechanics such as Zelda and Super Mario Bros, the algorithm is able to find a set of diverse and playable levels; less so for more complex games. Next, we approach the problem of general level generation from a different angle: instead of generating the level, we generate the level generator itself. We introduce two different representations to represent level generators: neural network weights and a generator description language called Marahel. Neural networks performed better than using Marahel scripts as all the trained generators were able to generate almost 100% playable levels, while the Marahel generators struggled to achieve playability. On the other hand, the Marahel generators are more understandable and can be easily modified by humans compared to the neural network ones. Overall, both approaches can help us to achieve our goal of general level generation from two different perspectives. Searching for content takes a long time but can produce a large corpus of diverse and playable levels. This makes this approach suitable for offline generation. Searching for generators takes a long time, but the found generators are very fast, although they do not guarantee playability. The speed of these generators allows for multiple re-sampling, making them suitable for online generation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Procedural Content Generation (PCG) is the field of using algorithms to generate game content with limited user interaction [116]. Content could be anything that is being used in a game such as textures [47, 29], models [86, 144], music [108, 50], levels [124, 54], rules [23, 100], and so on. Designers and developers use PCG for a variety of different purposes, such as tailoring game content to the player's taste, assisting creative content generation, reducing the time and cost for designing and developing games, exploring new types of games, or understanding the design space of games.

In this thesis, we are going to focus on one type of content which is game levels. Levels are virtual spaces of n dimensions (usually 2 or 3 but some games are in 4 (Miegakure by Marc ten Bosch), or 1 (The Linear RPG by Sophie Houlden) dimensions) where player interact with the different game elements [93]. Level generation is one of the oldest problems in Procedural content generation, it dates back to the early days of video games. Rogue (Glenn Wichman, 1980), Elite (David Braben and Ian Bell, 1984), and Beneath the Apple Manor (Don Worth, 1978) are all early example of games that uses level generation to enable a new type of games. Since that time, level generation is being used in video games such as Civilization (Sid Meier, 1991), Diablo (Blizzard, 1996), Spelunky (Derek Yu, 2008), The Binding of Isaac (Edmund McMillen and Florian Himsl, 2011), and Dead Cells (Motion Twin, 2017). With all this time passed and with all the new advancements in the Artificial Intelligence field, there is still no definite solution for level generation that can work between different games easily. Designers have to always build a new generator or extensively modify an existing generator to fit their game [119]. For example: Spelunky uses human-designed templates and random walks to generates maps that can be explored by going down [153].

Having techniques that can work smoothly between different games will free a lot of time for the designers to focus more on the game and incorporating the level generation system in the loop. The problem is that most of the used techniques focus on optimizing a

certain function which usually doesn't work with video game levels. Levels are a subjective domain, there is no way we can define a level that is good for everyone. For example: a level that is good for a precision platformer player might not be good for a general platformer player. During the level design phase of any game, designers usually explore multiple different ideas and iterate on them till they find what fit their game best [24]. Generation methods should take this in consideration. Instead of trying to optimize levels for certain metrics, they should generate a set of different levels that covers different areas of the level design space. For example: In Super Mario Bros (Nintendo, 1985), instead of trying to find levels that have certain number of jumps or enemies, designers could benefit from seeing all the possible levels with all the allowed number of jumps and enemies.

Finding a diverse set of levels is usually not enough to solve the general level generation problem as levels are functional content. Levels need to be playable so they can function as intended within the game. For example: In Sokoban (Thinking Rabbit, 1982), you can't use a level that doesn't have a player avatar in it. Some of these properties might be hard constraints that need to be satisfied while the rest of them are soft constraints that need to be optimized. To have a general level generation system, we need a way to combine diversity and quality at the same time. We also need general metrics that can be used in measuring quality and diversity between game levels.

The goal of this dissertation is to explore and discuss some of the possible techniques and metrics that can be used toward solving the general level generation problem. We look at the problem from two different points of views and we also explore techniques for creating bigger levels that induce a certain playthrough experience. The remaining of this chapter specifies this dissertation's thesis statement, main contributions, and structure. It also lists publications made during my Ph.D. studies and notes on the style chosen for writing this work.

1.1 Thesis Statement

This work focuses on generating levels for any video game. It discusses two different approaches for generating the levels: searching for levels and searching for level generators. It presents methods for tackling each approach and testing them against different games. This helps us to understand the advantages and disadvantages of each method. Bellow is the Thesis Statement that drives the core discussion of this dissertation:

We can computationally generate a set of diverse levels for any video game through searching either the level space or the generator space.

1.2 Contributions

This thesis uses artificial intelligence methods to generate levels for any game. Most of the work presented in the thesis has been peer-reviewed and published in several conferences and its main contributions are:

- **Different frameworks for general level generation:** This work represents two different frameworks for solving general level generation problem. The problem is either addressed as searching for levels that suit the current game or searching for level generators that can produce levels for the current game.

- **A new algorithm to search for levels:** Levels are subjective domain and a level that I enjoy playing, might not be enjoyable for other players to play. This makes it hard to formulate a good objective function to find a good level. We present a new variation on the MAP-Elites algorithm (called Constrained MAP-Elites) that can help to address this problem. Constrained MAP-Elites generates a set of diverse playable levels where the designers/players can pick from based on their taste.

- **General methods for measuring diversity and quality between levels:** To be able to use Constrained MAP-Elites to generate levels for any game, we need general metrics that can measure diversity and quality of the generated levels. We suggest using triggered game mechanics during a playthrough as our diversity metric. Having different levels with a different combination of triggered mechanics makes sure that no two levels need the same skills to be passed. This metric not only guarantees diversity but also helps the designer to design levels that have a certain type of interaction. For quality metrics, we use an automated player to check if the level can be won as a hard constraint and simplicity of the level as a soft one. The hard constraint makes sure that you can finish the level while the simplicity metric helps to make the levels easy to read and parse as it has fewer elements in them.

- **Method for generating a mechanics-driven experience:** This work introduced a method to concatenate small levels together to create a longer level such that the new level follows a certain game experience. We use the inverse of the proposed diversity metric to measure the similarity between the input target experience and the generated level experience.

- **Possible representations for level generators:** Level generators are hard to represent because the level generation space needs to incorporate all the possible modifications. We show two representations for the level generator space: as weights of neural

network and as a script of a description language called Marahel. These two representations have the power to represent any generator as both of them are Turing complete.

- **Techniques to find good level generators:** Different representations need different search methods. We use a reinforcement learning algorithm, Proximal Policy Optimization, to adjust the neural network weights such that the final network can generate levels that satisfy certain objectives. For Marahel, we use grammatical evolution using NSGA-II to find scripts that can generate levels for the same problems provided for the neural network generators.

1.3 Publications

The work in this dissertation originates from 8 published papers. The major portion of the work presented here originates from the following:

- Ahmed Khalifa, Diego Perez-Liebana, Simon M. Lucas, and Julian Togelius. General video game level generation. In the Proceedings of the Genetic and Evolutionary Computation Conference, 2016. (Chapter 5)

- Ahmed Khalifa, and Julian Togelius. Marahel: A language for constructive level generation. In the Proceeding of Artificial Intelligence and Interactive Digital Entertainment Conference through EXAG, 2017. (Chapter 10)

- Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. Talakat: Bullet hell generation through constrained map-elites. In Proceedings of The Genetic and Evolutionary Computation Conference, 2018. (Chapter 6)

- Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. In the Proceedings of The Genetic and Evolutionary Computation Conference, 2019. (Chapter 7)

- Megan Charity, Michael Cerny Green, Ahmed Khalifa, and Julian Togelius. Mech-Elites: Illuminating the Mechanic Space of GVGAI. In the Proceedings of Foundation of Digital Games, 2020. (Chapter 7)

- Michael Cerny Green, Luvneesh Mugrai, Ahmed Khalifa, and Julian Togelius. Mario Level Generation From Mechanics Using Scene Stitching. In the Proceedings of Conference on Games, 2020. (Chapter 8)

- Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. PCGRL: Procedural content generation via reinforcement learning. In the Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2020. (Chapter 9)

- Ahmed Khalifa and Julian Togelius. Multi-Objective level generator generation with Marahel. In the Proceedings of Foundation of Digital Games through PCG Workshop, 2020. (Chapter 10)

Although not present in this dissertation, I have also contributed to the following publications within the field of Artificial Intelligence:

- Ahmed Khalifa, Aaron Isaksen, Julian Togelius, and Andy Nealen. Modifying MCTS for Human-Like General Video Game Playing. In the Proceedings of International Joint Conference on Artificial Intelligence, 2016.

- Philip Bontrager, Ahmed Khalifa, Andre Mendes, and Julian Togelius. Matching games and algorithms for general video game playing. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2016.

- Ahmed Khalifa, Mike Preuss, and Julian Togelius. Multi-objective adaptation of a parameterized GVGAI agent towards several games. In the Proceedings of International Conference on Evolutionary Multi-Criterion Optimization, 2017.

- Ivan Bravi, Ahmed Khalifa, Christoffer Holmgård, and Julian Togelius. Evolving game-specific UCB alternatives for general video game playing. In the Proceedings of European Conference on the Applications of Evolutionary Computation, 2017.

- Ahmed Khalifa, Gabriella AB Barros, and Julian Togelius. Deeptingle. In the Proceedings of International Conference on Computational Creativity, 2017.

- Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. General video game rule generation. In the Proceedings of Computational Intelligence and Games, 2017.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, and Julian Togellius. "Press Space to Fire": Automatic Video Game Tutorial Generation." In the Proceedings of the Experimental AI for Games workshop at the Artificial Intelligence and Interactive Digital Entertainment Conference, 2017.

- Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. In NeurIPS Deep RL Workshop, 2018.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Andy Nealen, and Julian Togelius. Generating levels that teach mechanics. In the Proceedings of Foundations of Digital Games through Procedural Content Generation Workshop, 2018.
- Hongwei Zhou, Yichen Gong, Luvneesh Mugrai, Ahmed Khalifa, Andy Nealen, and Julian Togelius. A hybrid search agent in pommerman. In the Proceedings of Foundations of Digital Games, 2018.
- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, Andy Nealen, and Julian Togelius. "AtDELFI: automatically designing legible, full instructions for games." In the Proceedings of Foundations of Digital Games, 2018.
- Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. In the Proceedings of International Joint Conference on Artificial Intelligence, 2019.
- Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. Procedural content generation through quality diversity. In the Proceedings of Conference on Games, 2019.
- Ahmed Khalifa, Fernando de Mesentier Silva, and Julian Togelius. Level design patterns in 2D games. In the Proceedings of Conference on Games, 2019.
- Ahmed Khalifa, Dan Gopstein, and Julian Togelius. ELIMINATION from Design to Analysis. In the Proceedings of Conference on Games, 2019.
- Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. Two-step constructive approaches for dungeon generation. In the Proceedings of Foundations of Digital Games through Procedural Content Generation Workshop, 2019.
- Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D. Gaina, Julian Togelius, and Simon M. Lucas. General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms. In IEEE Transactions on Games, 2019.
- Philip Bontrager, Ahmed Khalifa, Damien Anderson, Matthew Stephenson, Christoph Salge, and Julian Togelius. “Superstition” in the Network: Deep Reinforcement Learning Plays Deceptive Games. In the Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2019.

- Matthew Stephenson, Damien Anderson, Ahmed Khalifa, John Levine, Jochen Renz, Julian Togelius, and Christoph Salge. A continuous information gain measure to find the most discriminatory problems for ai benchmarking. In the Proceedings of Congress on Evolutionary Computation, 2020.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, and Julian Togelius. Automatic Critical Mechanic Discovery in Video Games. In the Proceedings of Foundation of Digital Games, 2020.

- Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. Bootstrapping Conditional GANs for Video Game Level Generation. In the Proceedings of Conference on Games, 2020.

- Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Rotation, Translation, and Cropping for Zero-Shot Generalization. In the Proceedings of IEEE Conference on Games 2020.

- Megan Charity, Ahmed Khalifa, and Julian Togelius. Baba is Y'all: Collaborative Mixed-Initiative Level Design. In the Proceedings of IEEE Conference on Games, 2020.

- Debosmita Bhaumik, Ahmed Khalifa, Michael Cerny Green, and Julian Togelius. Tree Search vs Optimization Approaches for Map Generation. In the Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2020.

- Omar Delarosa, Hang Dong, Mindy Ruan, Ahmed Khalifa, and Julian Togelius. Mixed-Initiative Level Design with RL Brush. Submitted to EXAG Workshop at the Artificial Intelligence and Interactive Digital Entertainment, 2020.

1.4 Outline

To satisfy our thesis statement, we divide the thesis into 3 parts: Context, Searching for Content, and Searching for Levels. The explanation and the outline of each part is as follows:

- **Part 1**: **Context** introduces the necessary techniques, domains, and representations used in this work.

  - **Chapter 2: Background** presents the needed background towards understanding this thesis.

  - **Chapter 3: Problems** introduces the different game problems that are used as our test bed for general level generation.
  - **Chapter 4: Algorithms** explains the different algorithms used during the work of the thesis. It also introduces a new variation of an algorithm that was invented during our work.

- **Part 2**: **Searching for Content** describes different approaches and techniques toward general level generation.
  - **Chapter 5: Search-Based General Level Generation** introduces a framework for general level generation in General Video Game Framework. It also describes our first technique for generating levels using a search based approach.
  - **Chapter 6: Quality Diversity Level Generation** approaches the problems of diversity and generation time introduced in the previous chapter by using quality diversity techniques. We use the new technique to generate bullet hell levels for various player skills.
  - **Chapter 7: Mechanical Diversity Level Generation** introduces a general measure of diversity that can be used in any game. It uses the fired game mechanics as a metric to find a diverse set of levels. It also focus on generating small sized playable levels that are easy to read. The technique was applied to Super Mario Bros and the General Video Game AI Framework.
  - **Chapter 8: Stitching For Mechanical Experience** finishes Part 2 with a search based technique that can combine the small generated levels from the previous chapter into a longer levels. Our technique focus on generating levels that follow a certain mechanic experience that the designer/player wants. We apply the technique on Super Mario Bros to generate levels that have same experience as Level 1-1.

- **Part 3**: **Searching for Generators** presents a different approach toward general level generation. Instead of finding a level, we try to find a generator that fits our needs.
  - **Chapter 9: Learning Level Generators through Reinforcements** uses reinforcement learning algorithms to learn a level generator that can be used to generate multiple levels later. We applied this technique on 3 different problems: Binary, Zelda, and Sokoban (which are explained in Chapter 3).
  - **Chapter 10: Constructive Level Generator through Multi-Objective Evolution** shows a different way to represent level generators using Marahel descrip-

tion language. In this chapter, we use multi-objective optimization approaches to search Marahel scripts for ones that can perform similar to the reinforcement learning generators presented in the previous chapter.

- **Chapter 11: Conclusion** summarize this thesis and discusses its contributions. We also talk about future work and possible future projects that can extend on this work.

1.5 Notes on Pronoun

The work presented in this dissertation could not be achieved without the collaboration of my co-authors. Although I'm the first author in all the work discussed here, throughout the rest of this work the pronoun "we" will be used in favor of "I", to refer to the effort of all researchers involved. When referring to a player, this dissertation chooses to use the pronouns "they" and "their" as to respect a gender inclusive speech.

Part 1

Context

Chapter 2

Background

This chapter provides the background needed to fully understand this work. We focus on the different techniques used during this thesis such as evolutionary computation and reinforcement learning. We also talk about the related work in level generation research (searching for content) and level generator generation research (searching for generators).

2.1 Evolutionary Computation

Evolutionary computation [9] is an AI optimization algorithms inspired by Darwin's theory of evolution [37]. The idea is based on survival for the fittest, where the best individuals in a population survive between different generations, and their traits get reproduced to create new individuals. The individuals are usually called *chromosomes*, we use *genetic operators* to produce new chromosomes, and their performance gets measured using a *fitness function*. Evolutionary algorithms usually represent individuals using an array of numbers. This array is called the *genotype* and it is usually get transformed to its corresponding solution called the *phenotype*. The genetic operators are applied to the genotype, while the fitness function is applied to the phenotype.

Algorithm 1 shows a pseudo-code for an evolutionary algorithm. It is an iterative algorithm where at each iteration (called generation), the algorithm will sample a group of chromosomes from a pool called population. The sampling process usually depends on the chromosome fitness. These selected chromosomes produce a chromosome that will be inserted back in the population. In the end, the population gets reduced to its original size to survive for the next generation. Usually, evolutionary algorithms run until they converge (most of the chromosomes are very similar) or we run out of time (usually the case in hard problems). In the following subsections, we will explore the different variants of evolutionary algorithms based on their optimization target.

Algorithm 1: Evolutionary Algorithm Pseudo Code

Initialize the population with random solutions
while ***Termination Condition*** *not met* **do**
    Select parents
    Recombine the parents to produce a child
    Mutate the resulting children
    Evaluate the children
    Select solutions for the next generation
    Replace the population with selected children
end

2.1.1 Objective-Based Algorithms

Objective-Based Algorithms [8] are the traditional optimization algorithms where there is a fitness function that measures how well each chromosome is preforming. If the problem has more than one fitness value, these functions are usually combined into a single fitness function that is being used. For example, we want to optimize two students' schedules such that they have the least gaps. In this problem, our fitness is the total number of gaps for each student. We could combine these two values either by using summation or multiplication. Objective-based algorithms work best in problems with smooth and granular solution space so any change in the chromosome will be reflected directly in the fitness function.

Genetic Algorithm (GA) is one of the traditional evolutionary algorithms that you can find in any evolutionary computation book. At each generation, the genetic algorithm uses crossover and mutation to create a new population that will swap the current population. Crossover allows the algorithm to combine 2 chromosomes together by swapping some parts of the chromosomes. For example: one point crossover picks a point in the chromosome and swaps both parent values around it. Mutation allows the algorithm to change part of the chromosome using random probabilities. For example: one point mutation picks a random value from the chromosome and switches it with another random value. Elitism is usually used with GA to make sure that the fittest individuals survive between different generations.

2.1.2 Multi-Objective Algorithms

As discussed in the previous subsection, some problems might have more than one fitness function. These fitness functions sometimes are not directly proportional to each other which makes them hard to be combined. For example: If you want to optimize the number of organic ingredients you are buying from a list but at the same time save the most amount of money. The problem usually is organic ingredients are expensive which leads to less money

saved. In these problems, we use multi-objective algorithms [38] to calculate the Pareto front. The Pareto front is all the chromosomes that can't be dominated by other chromosomes in the population. A chromosome is dominated by another if the other chromosome has at least one fitness function better than the original chromosome and the rest of the functions are at least the same. There is several different known algorithms that can calculate the Pareto front such as NSGA-II [39], SPEA2 [154], NSPSO [90], etc.

2.1.3 Divergent Search Algorithms

Different from the previous paradigm, divergent search algorithm [87] optimize toward having a diverse set of population and not towards a certain fitness function. Although this doesn't look like it will work, divergent search usually finds the optimal solution [87]. Divergent search is usually used in domains where the fitness function is deceptive and/or not easy to navigate. Divergent search is also a good method to be used in subjective domains (like image generation) where there is no definite way to identify what is good and what is bad. The different algorithms in Divergent search usually differ in the way they calculate the diversity between chromosomes. For example, surprise search [54] calculate the diversity by measuring the distance between the current solution and the expected solution using a predictor.

2.1.4 Quality Diversity Algorithms

Quality diversity algorithms [110] are a new group of algorithms in Evolutionary Computations. The idea is to combine objective search algorithms with divergent search algorithms. The algorithm tries to find the best set of different solutions in the solution space. These algorithms find the best solution for every group of solutions that are similar to each other. These algorithms work well for functional solutions. Functional solutions are solutions that need minimum criteria to be used but at the same time, there is more than one correct solution for them. For example, agent controllers that can move a creature on a straight line with minimum speed is a functional solution. As we need a certain minimum criterion to use that solution but at the same time there is a lot of different ways to achieve, the agent could be jumping, crawling, or walking normally.

One of the most interesting new algorithms is MAP-Elites [96]. Instead of keeping a population of solutions, it saves the solutions into a multidimensional array called *MAP*. The dimensions of the array are called *Behavioral Characteristics* and they divide the solution space into several different niches where similar chromosomes compete with each other and the highest quality survives which is called *Elite*. Algorithm 2 shows a pseudo-code

Algorithm 2: MAP-Elites Pseudo Code

```
Initialize the map with a group of random solutions
while Termination Condition not met do
    Select parents randomly from the map
    Recombine the parent to produce a child
    Mutate the resulting child
    Calculate the child's behavior characteristics
    Evaluate the child's fitness
    if Corresponding Cell is Empty or Child Fitness > Elite Fitness then
        Replace elite with the child
    end
end
```

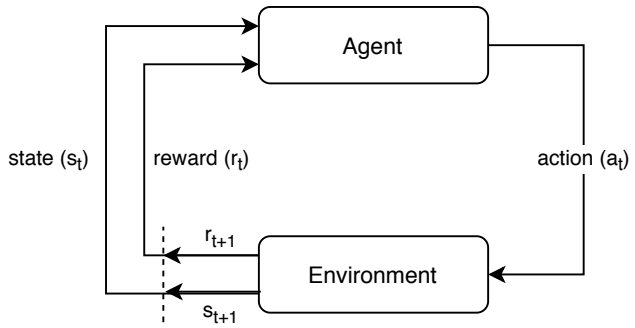


Figure 2.1: Agent Environment interaction.

for the MAP-Elites algorithm. At each step, MAP-Elites randomly samples chromosomes from the MAP and generate a new offspring using mutation and/or crossover. This new chromosome gets evaluated with respect to its Behavioral Characteristics and its fitness function. The chromosome will be inserted back into the correct location in the MAP based on its behavioral characteristics. If the location is empty or the new chromosome fitness is better than the previous elite, it replaces the current elite, otherwise, it gets discarded. The algorithm is popular in the research field due to its simplicity and its great performance in various research fields [96, 53, 140]. There are various variants of this algorithm to tackle different problems such as Constrained MAP-Elites [79], MAP-Elites with Sliding Boundaries [53], BOP-Elties [73], etc.

2.2 Reinforcement Learning

Reinforcement learning[132] focuses on learning an optimal policy for a certain task by interacting with the environment. A policy is a function that can map any given environment

state to an action. An optimal policy is a policy that maximizes the total reward coming from the environment by taking the optimal action at each given step. For example: an optimal policy playing chess will always play moves that will lead to winning (if possible) from any board state. An agent is an algorithm that applies a certain policy. Figure 2.1 shows an agent interacting with an environment. At each time step the agent gets the current state and based on it, it takes an action that updates the state of the environment to return the new state and a reward value based on how good is that action and the loop continues.

This system can be mathematically modeled as a Markov Decision Process. A Markov Decision Process (MDP) [12] is a mathematical framework that can model the decision-making process with a random outcome. MDP consists of four sets: a set of possible states (S), a set of finite actions (A), transition probabilities that tells us the probability to transfer from one state to another given an action ($P(S_{t+1}|S_t, A_t)$), and a reward function that return positive or negative value based on transition from one state to another ($R(S_{t+1}|St, At)$).

Reinforcement learning algorithms solve MDP by finding the optimal policy that can maximize the total rewards. Reinforcement learning until recently was only possible on small problems where the number of states is small. Most of the classic reinforcement learning algorithms, such as Q-Learning, models the policy as a table. When the number of states increases, the probability to visit the same state again becomes harder and the need for better representation is required. An easy solution is to use a function approximator such as neural networks instead of using the table. With the recent advancement of Deep Learning, Reinforcement learning start booming again where it can perform a lot of different tasks on a superhuman performance level [95].

2.3 Procedural Content Generation

Procedural Content Generation (PCG) [116] refers to the use of computer algorithms to produce content. These techniques have played an important role in video games since the early eighties, with such examples as Rogue (Glenn Wichman, 1980), Elite (David Braben and Ian Bell, 1984), and Beneath the Apple Manor (Don Worth, 1978). PCG is primarily used in games because of its ability to produce large amounts of content with a negligible memory cost, fitting on a small floppy disk [72]. Although the scarcity of memory is less of a concern today, PCG is widely used in game design and development such as Spelunky (Derek Yu, 2008), The Binding of Isaac (Edmund McMillen and Florian Himsl, 2011), or No Mans Sky (Hello Games, 2016). The game industry and the research have developed and presented a number of different methods that can be used to generate content in games. These methods can be divided into three main categories: constructive methods,

search-Based methods, and machine learning methods.

2.3.1 Constructive Methods

Constructive methods [116] are widely used in the video game industry due to many algorithms being very fast and being relatively easy to implement and debug. These techniques have been used in video games since the early days. For example, *Rogue*[1] generates a new dungeon for every playthrough by placing non intersecting rectangles (rooms) and connecting them using lines (hallways). While constructive generation methods differ widely among each other, the defining feature of constructive generation is that there is no regeneration of the output based on testing; generation happens only once. Due to this limitation, the algorithm should guarantee that the generated levels have the required features during construction. Not all algorithms can guarantee playable levels 100% of the time, so developers use repair techniques to fix the generated content. Since constructive algorithms are fast, developers sometimes wrap the algorithm in a generate and test algorithm where it keeps generating levels until a suitable one is found.

Depending on the requirements of the particular game and the desired type of content, different constructive methods are used such as template-based generation, grammar-based generation, cellular automata, etc. Template-based level generation uses hand authored content to generate the content. The algorithm combines different authored pieces that fit together according to certain constraints. In some cases, the algorithm alters the generated content by adding noise to it. You can find this technique used in a lot of games for different types of content. For example, level generation in Spelunky (Derek Yu, 2008), creature generation in Spore (Maxis, 2008), weapon generation in Borderlands (Gearbox Software, 2009), etc.

Grammar-based generation uses a hand authored rules. Rules are defined as a set of transformations of non-terminal symbols to a group of terminal and/or non-terminal symbols. These rules are called production rules. Applying these transformations are usually called grammar expansions because it transforms a nonterminal symbol to a set of other symbols. Grammar starts by expanding a starting symbol. Grammar-based generation is usually defined using a context insensitive grammar. Context insensitive grammar means the all the production rules only care about the nonterminal symbol and not what surrounds it. Grammar-based generation is famous to be used in several domains such as tree generation [109], story generation [19], level generation [141, 41], etc. Tracery [30, 31] is an open-source grammar library that is easy to be used and powerful in generation. People

[1] https://en.wikipedia.org/wiki/Rogue_(video_game)

not only use it to generate text [142], but also SVG art [103, 32], game levels [79], level events [52], level generators [82], etc.

Cellular automata are a mathematical technique of great generality [149], this technique is usually applied on a fixed size multi-dimensional array of cells. Similar to grammar-based techniques it transforms the input to a new output by applying a group of rules. The rules are usually defined as the change in the state of the cell based on the neighboring cells. This technique generates content with organic-like structures so it is famously used in generating cave-like levels [69]. For example: Galak Z [2] uses cellular automata to generate separate rooms. Tomb of Tomeria [33] not only uses cellular automata to generate the whole level but also utilizes it as a game mechanic.

2.3.2 Search-Based Methods

Search-based Methods [137] (SBPCG) are techniques that explore the solution space using a search algorithm (such as evolutionary algorithms [23, 1, 76], tree search algorithms [14, 22], etc). Search-based methods can control the quality of the generated content as it explores the space by maximizing a fitness function. Search-based methods are more commonly seen in research than the industry due to their long generation time (might not be suitable to be used during a game) and their indirect control on the generated content (the main control is using the fitness function). On the other hand, search methods are versatile as they only need a content representation and a fitness function to generate any type of content. They have been used in various research projects to generate bullet patterns [65, 79], game levels [5, 7, 76], board games [23], flowers [111], level generators [74, 82], etc.

If we look at the large body of research, we can see that value and novelty are vital criteria to evaluate creative artifacts[112]. While value is commonly targeted in traditional PCG, novelty—or more broadly, game content diversity—is not. Recently, quality diversity started to be more and more used in the PCG community. MAP-Elites was used to generate 2D images [98] and 3D objects [86]. Constrained MAP-Elites was used to generate levels for bullet hell games [79], Super Mario Bros [77], dungeon levels similar to The Legend of Zelda (Nintendo, 1986) [3], and 4 different games from the General Video Game Framework [26]. MAP-Elites with Sliding Boundaries [53] was used to generate decks for the online card game *Hearthstone* (Blizzard, 2014).

2.3.3 Machine Learning Methods

Machine learning methods [130] (PCGML) uses a machine-learning algorithm to generate the content. These methods usually learn the generation process from datasets but they

could also learn from interaction with a game environment. PCGML is rarely used outside of the research community as they usually need a big amount of data, require long training time, and designers have less control over the generation. Because of these problems, the industry usually uses Machine Learning to generate aesthetic elements such as the books in Caves of Quds [59]. On the other hand, most of the advancement and usage of PCGML is happening on the research side. They use several different techniques such as Markov Chains [126], N-Grams [36], GANs [143], Autoencoders [68], LSTMs [129], etc.

Most of the previous examples have been using either supervised or unsupervised learning techniques. Reinforcement learning, though, was rarely applied to PCG, even after its success in playing video games [95]. This is not an easy task, because it is unclear how to transform the content generation process into a reinforcement learning problem. One solution is to frame the content generation as an iterative process, where at each step the agent is trying to modify a small part of the content, similar to the idea from Guzdail et al.'s [63, 62] work. Several researchers have explored the idea of iterative generation using supervised sequence learning methods, such as Markov Chains [125], N-Grams [36], and LSTMs [129]. These attempts all built the level in a fixed order (e.g. left-to-right for a platformer level).

Looking at level generation from a sequential perspective makes it possible to formulate it as a Markov Decision Process (MDP) where the agent is making small, iterative changes to improve the current level. For example, Mcdonald [92] formulates the process of 3D building-generation as a MDP. The only work we could find that used reinforcement learning to directly generate game content was that of Chen el al. [27] and Guzdial et al. [63, 62]. In Chen et al.'s work, Q-Learning [146] is used to train a deck-building system that outperforms search-based methods. In Guzdial et al.'s work, they proposed a mixed-initiative tool to design levels for Super Mario Bros (Nintendo, 1985) that uses active learning to update the trained models to adapt to the user choices. Although the paper didn't use RL to train a model from scratch, the system was successfully adapting to the users' choices.

A recent approach is combining both machine learning and search-based approaches to generate content. In these hybrid approaches, machine learning is used to learn a representation for the solution space, while the search based algorithm explores that space to find the needed content. This process is called Latent Variable Evolution (LVE) which was introduced by Bontrager et al. to evolve fingerprints [18]. These methods are usually being used with GANs and AutoEncoders and the searching usually uses CMA-ES. One notable example is Volz et al.'s work [143] in generating Super Mario Bros level using a GAN trained using a single level and CMA-ES to search the space for a playable level.

2.4 Procedural Level Generation

Most known level generation algorithms are tailored to generating content for a particular game [133, 51] using a significant amount of game or genre-specific knowledge to make sure the generated levels are playable and enjoyable.

In this work, we will be investigating search based methods as they are very flexible and can be adapted easily between different games. Search-based level generation has been used in various different games such Cut the Rope (ZeptoLab 2010) [115], strategy maps [88], FPS levels [55], etc. A big body of this research was done over Super Mario Bros. Super Mario Bros is considered the traditional benchmark in the game AI research community. The *Pattern-based* generator uses *slices* taken from the original *Super Mario Bros* (Nintendo 1985) to evolve levels, where the fitness function counts the number of occurrences of specified sections of slices, with the objective of maximizing the number of different slices [35]. The *Grammatical Evolution* generator evolves levels with design grammars. Levels are represented as instructions for expanding design patterns, and the fitness function measures the number of items in the level and the number of conflicts between the placement of these items. Green et al. [57] generated small level "scenes" that try to teach the player-specific mechanics, like high jumping or stomping on enemies. Volz et al. [143] combined search based algorithm with GANs to search the latent space of GANs for good levels.

For general level generation, not a lot of work has been done in that field due to the difficulty of defining a general representation that can work between different games and the difficulty to define fitness metrics. Khalifa and Fayek [76] generated puzzle levels to any puzzle script game using a genetic algorithm. Beaupre [11] used design pattern approach to generate playable levels for the general video game framework. Neufeld [97] used answer set programming to generate playable levels for the general video game framework.

2.5 Procedural Procedural Level Generation Generation

Another way to reach generality is to generate a level generator instead of a level. This problem is under-explored in both academia and industry due to its high complexity. Procedural Procedural Level Generation Generation is the problem of using a procedural generation method to find a level generator. A key requirement of most procedural generators is a reasonable and workable representation of the generated content. For example: Browne and Maire [23] represented board games using Game Description Language (GDL) to be able to evolve new board games like Yavalath [147]. Finding a representation for a level generator

is a much higher problem as the representation should be able to represent a lot of different types of generators that can produce a different type of levels.

One of the early work in this field [134] searched the parameter space of ASP programs to generate a dungeon crawler level generator that is challenging for an automated agent. Later, Kerssemakers et al. [74] designed a meta-generator for Super Mario Bros (Nintendo, 1985) and used an evolutionary algorithm to search the space for diverse generators. On a similar note, Drageset et al [42] defined a meta-generator space where each generator is defined as a set of parameters for a design generator. They used an optimization algorithm to search that space and returns the best-found level so far.

Cellular Automata can be considered as level generators as they can modify the input to a new output that follows certain rules. These rules can be designed in a way to generate organic like levels [69]. Ashlock [6] evolved cellular automata rules to find different generators that generate black and white maps with full connectivity. Similarly, Pech et al [104] and Adams and Louis [1] generate cellular automata rules to generate mazes with certain features.

Another way is to represent the generator in the form of a neural network. Earle [44] trained fractal neural networks using A2C [94] to play SimCity (Will Wright, 1989). This might not look like level generation but if you look at SimCity as a city planning problem, then the agent is generating cities. Later, in Part 3, we explore different representations for level generator based on a neural network similar to Earle's work and a level description language called Marahel [81]. We use them on three different problems to find a suitable generator.

Chapter 3

Problems

This chapter introduces all the different domains that we use in this work. Instead of focusing on one domain, we decided to use four different ones: General Video Game Playing, Talakat, Mario, and PCGRL. For each project, we decided to pick the domain that showcases the results best and highlights their advantages. In all the domains except Talakat, the levels can be represented as 2D tilemap. Tilemaps are a multidimensional matrix where each cell holds an integer value that represents a different game object. For Talakat, the levels are represented as a script that is described later in this chapter.

3.1 General Video Game Playing

The General Video Game Playing [105] (GVGAI) is a framework built to run games written in the Video Game Description Language (VGDL) [45]. GVGAI was originally developed to help researchers have a benchmark to compare general AI planning agents. General AI planning agents are usually search-based agents that can play more than one game provided a forward model. To be well-equipped to play different games, an agent must hold a varying set of skills, such as reacting to the system, agile decision making and long-term planning. With time more researchers adopt the framework for different types of research such as rule generation [100, 78], tutorial generation [58, 56], learning [152, 70], etc.

The Video Game Description Language (VGDL) [45] is a game description language used to represent 2D games of the arcade (Pac-man), action (Space Invaders), and/or puzzle game (Sokoban) genres. VGDL games consist of 2 parts: a game description and a level description. The game description is responsible for holding information about game objects, game mechanics and termination conditions. Figure 3.1 shows a simple Sokoban game and its game description. The game description consists of 4 subparts:

```
BasicGame frame_rate=30
    SpriteSet
        hole   > Immovable color=DARKBLUE img=hole
        avatar > MovingAvatar #cooldown=4
        box    > Passive img=box
    LevelMapping
        0 > hole
        1 > box
    InteractionSet
        avatar wall > stepBack
        box avatar  > bounceForward
        box wall    > undoAll
        box box     > undoAll
        box hole    > killSprite scoreChange=1
    TerminationSet
        SpriteCounter stype=box    limit=0 win=True
```

Figure 3.1: A definition of a simple game (a version of Sokoban) in VGDL, and a screenshot of part of the game in-engine.

- **Sprite Set:** a hierarchical list of all game objects, called game sprites. Similar game sprites can be grouped under a common name in the hierarchy. This name is considered a parent of these similar game sprites. For example, in Pac-man, there are two sprites that are grouped into the "Pac-man" parent type: *hungry* and *powered*. Each type shares some game rules with the other while also having different game rules associated with it: Whereas *hungry* can be destroyed by ghosts, *powered* can destroy ghosts instead. Each sprite has a type, orientation, and image. The sprite type defines its behavior, e.g. in Pac-man, a ghost is a *RandomPathAltChaser*, which means it chases a "hungry" Pac-man but flees from it after Pac-man eats a power pellet.

- **Interaction Set:** a list of all game interactions. Interactions occur upon collision between two sprites. For example, In Pac-man, if the player collides with a pellet, the latter will be destroyed and the score increases.

- **Termination Set:** a list of conditions, which define how to win or lose the game. These conditions can be dependent on sprites or on a countdown timer. For example, in Pac-man, if the player eats all the pellets, the player wins the game.

- **Level Mapping:** a table of characters and sprite names that is used to decode the level description.

The level description contains a 2D matrix of characters, and each character can be decoded using the Level Mapping. Each character maps to game sprites' starting location for that level.

In this work, we are focusing on level generation in the GVGAI framework. We don't use all the games from the framework but a subset of games. We pick a subset that cover different type of games and different experiences based on the analysis done by Bontrager et al. [17]. This is a full list of the used games in this work:

- **Frogs:** is a port of *Frogger* (Konami, 1981). The player controls a frog where they try to reach the goal while avoiding getting hit by a car in the streets or drowning in the water if not land on the wood logs.

- **PacMan:** is a port of *Pac-Man* (Namco, 1980). The player controls PacMan character where they aim to collect all the pills in the maze while avoiding the chasing ghosts. The player can eat a power pill to make the ghosts stop chasing them and allow the player to be able to eat them for a fixed amount of time. Eating ghosts gives more score than collecting pills and power pills.

- **Zelda:** is a port of the dungeon system in *The Legend of Zelda* (Nintendo 1986). The player must pick up a key and unlock a door in order to beat a level. Monsters populate the level and can kill the player, causing them to lose. The player can swing a sword, which can destroy monsters and grant points.

- **Solarfox:** is a port of *Solar Fox* (Bally/Midway Mfg. Co 1981). The player must dodge both enemies and their flaming projectiles in order to collect all the "blibs" in the level. The player gains a point for each blib collected, and victory is granted after collecting all blibs on the level. Several levels contain "powered blibs," which are worth no points. If a player collides with a powered blib, it will spawn a "blib generator," which as the name implies, can spawn more blibs to collect and gain more points. If a player touches a blib generator, however, the generator will be destroyed and no longer generate any more blibs.

- **Plants:** is a port of *Plants vs. Zombies* (PopCap Games 2009), a tower defense style game. If the player survives for 1000 game ticks, they win. Zombies spawn on the right side of the screen and move left, and the player loses if a zombie reaches the left side. Plants, which the player must grow in specific "marsh" tiles, can destroy zombies by automatically firing zombie-killing peas. Each zombie killed is worth a point. Occasionally, zombies will throw axes, which destroy plants.

- **RealPortals:** is a 2D port of *Portal* (Valve 2007). The player must reach the goal, which sometimes is behind a locked door that needs a key. Movement is restricted by water, which kills the player if they touch it. To succeed, players need to be creative in overcoming this hazard by using portals that can teleport them across the map. Players need to pick up wands, which allow them to toggle between the ability to create *portal entrances* and *portal exits*. There are also boulders on some levels, which the player can push into the water to transform the water into solid ground, creating land-bridges on which they can walk.

⟨*script*⟩ ::= ⟨*spawners*⟩ ⟨*boss*⟩

⟨*spawners*⟩ ::= ⟨*spawner*⟩ | ⟨*spawner*⟩ ⟨*spawners*⟩

⟨*spawner*⟩ ::= id ⟨*spawnerParameter*⟩ ⟨*spawnedParameter*⟩ ⟨*bulletParameter*⟩

⟨*spawnerParameter*⟩ ::= ⟨*spawnPattern*⟩ patternTime patternRepeat ⟨*angleCSV*⟩ ⟨*radiusCSV*⟩

⟨*spawnPattern*⟩ ::= 'bullet' ⟨*spawnPattern*⟩ | 'wait' ⟨*spawnPattern*⟩ | id ⟨*spawnPattern*⟩ | ϵ

⟨*spawnedParameter*⟩ ::= ⟨*numberCSV*⟩ ⟨*angleCSV*⟩ ⟨*speedCSV*⟩

⟨*bulletParameter*⟩ ::= ⟨*radiusCSV*⟩ ⟨*colorCSV*⟩

⟨*numberCSV*⟩ ::= minValue maxValue rate interval ⟨*type*⟩

⟨*angleCSV*⟩ ::= minValue maxValue rate interval ⟨*type*⟩

⟨*speedCSV*⟩ ::= minValue maxValue rate interval ⟨*type*⟩

⟨*radiusCSV*⟩ ::= minValue maxValue rate interval ⟨*type*⟩

⟨*colorCSV*⟩ ::= minValue maxValue rate interval ⟨*type*⟩

⟨*type*⟩ ::= 'circle' | 'inverse'

⟨*boss*⟩ ::= bossPosition bossHealth ⟨*script*⟩

⟨*script*⟩ ::= ⟨*scriptEvent*⟩ | ⟨*scriptEvent*⟩ ⟨*script*⟩

⟨*scriptEvent*⟩ ::= trigger ⟨*events*⟩

⟨*events*⟩ ::= ⟨*event*⟩ | ⟨*event*⟩ ⟨*events*⟩

⟨*event*⟩ ::= ⟨*spawnEvent*⟩ | ⟨*clearEvent*⟩

⟨*spawnEvent*⟩ ::= 'spawn' id speed angle | 'spawn' 'bullet' speed angle

⟨*clearEvent*⟩ ::= 'clear' id | 'clear' 'bullets' | 'clear' 'spawners'

(a)

```
{
  spawners:{
    one:{
      pattern:["two"],
      patternTime:"4",
      spawnerAngle:"0,360,10,12,circle",
      spawnedSpeed:"0",
      spawnedNumber:"4",
      spawnedAngle:"360"
    },
    two:{
      pattern:["bullet"],
      patternRepeat:"1",
      spawnedAngle:"30",
      spawnedNumber:"3",
      spawnedSpeed: "4"
    }
    three:{
      pattern:["bullet"],
      patternTime:"4",
      spawnerAngle:"0,180,2,0,reverse",
      spawnedSpeed:"2",
      spawnedNumber:"2",
      spawnedAngle:"360"
    }
  },
  boss:{
    bossHealth: 3000,
    bossPosition: "0.5, 0.2",
    script:[
      {
        health:1,
        events:["spawn,one"]
      },
      {
        health:0.5,
        events:["clear,spawners", "spawn,
            three"]
      }
    ]
  }
}
```

(b)

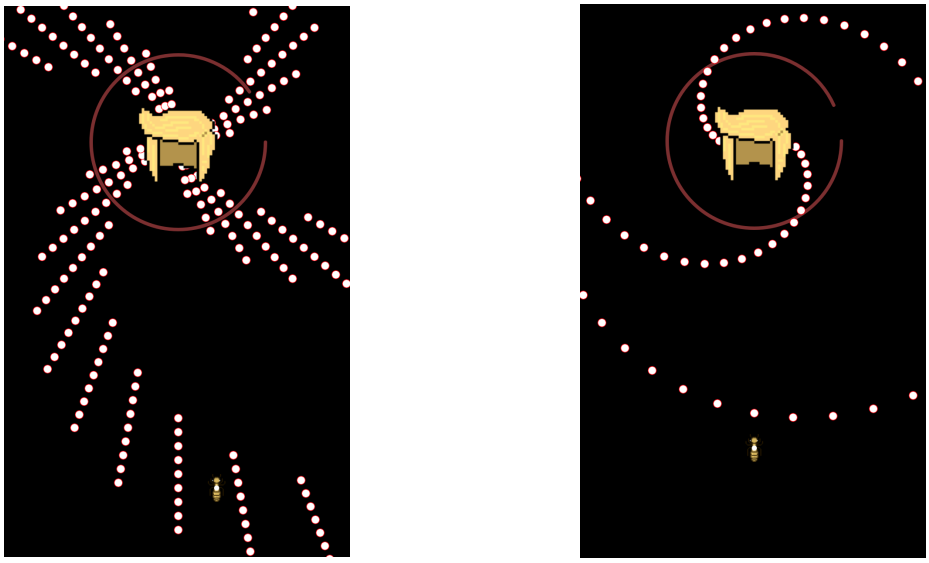
Figure 3.2: The left figure shows *Talakat* language as a context free grammar. Angular brackets values such as <spawners> are non terminal, quoted values such as 'bullet' are string terminals, while other values such as minValue are number terminals. The right figure shows an example of a full Talakat script.

The problem of level generation was introduced by us in 2016 [80] and we built an interface and framework around which will be explained in detail later in Chapter 5. Beside Chapter 5, we use the framework in Chapter 7 for testing quality diversity generation in a more general domain.

3.2 Talakat

Talakat (means bullets in Arabic) [1] is a description language that describes bullet hell levels. A Talakat script constitutes a single bullet hell level. Figure 3.2 shows the full grammar of Talakat and an example of a Talakat script. A single script is divided into two parts: the Spawner section and the Boss section.

[1] Detailed documentation of the grammar can be found at https://github.com/amidos2006/Talakat/wiki/Scripting-Language



(a) (b)

Figure 3.3: The visual representation of the spawners specified in Figure 3.2b. Figure 3.3a shows the output of spawners “one” and “two”. Figure 3.3b shows the output of spawner “three”. The player is the small bug in the bottom of the screen, while the boss is the yellow wig from the top.

3.2.1 Spawners Section

The spawners section contains information about the bullet spawners. Bullet spawners are invisible objects that are responsible for producing either bullets or additional spawners. The spawners section consists of an array of spawner definitions. Each spawner has a unique ‘id’ to identify it, as well as parameters that define its spawning behavior. These parameters can include angle, speed, number of spawned objects, etc. Complex patterns can be generated by overlaying different spawners on top of one another. For example, Figure 3.2b utilizes three spawners: “one”, “two”, and “three”. Spawner “one” generates 4 instances of spawner “two” evenly over an arc of $360°$ (4 spawners at $90°$ intervals) every 4 frames. Every 12 frames, the spawner rotates 10 degrees. Spawner “two” spawns 3 bullets evenly over an arc of $30°$, each of which move at a speed of 4 pixels/frame. The end result of this pattern is a boss that fires 3 bullets in a small arc in four directions three times, rotates 10 degrees, and repeats. Spawner “three” spawns 2 bullets over an angle of $360°$ while rotating 2 degrees per frame and changing direction once the spawner has rotated $180°$, creating a pattern that fires bullets in a sweeping motion. Figure 3.3 shows this spawner configuration in action.

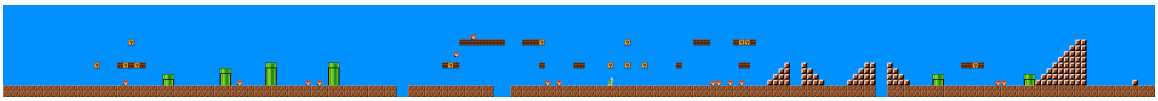


Figure 3.4: Level 1-1 From Super Mario Bros.

3.2.2 Boss Section

The boss section contains information about the level. It defines boss health, boss position, and contains the level script which details boss behavior. Figure 3.2b contains an example of a simple boss section. Boss health controls the length of the level. In figure 3.2b, the length of the level is specified to be 3000 frames. For this version of Talakat, one point of boss health is depleted per frame regardless of player action, making health and duration one and the same. Boss position controls the placement of the boss in the level. In figure 3.2b, the boss will be in the upper center part of the level. The level script describes events that trigger when the boss' health reaches certain thresholds. In figure 3.2b, the boss has two events: the first event spawns spawner "one" and triggers when boss health is at 100% (that is, the boss opens with this event), and the second event clears all of the previous spawners and spawns spawner "three" when the boss' health reaches 50%.

3.3 Mario

Infinite Mario Bros. (IMB) was developed by Markus Persson [107] as a public domain clone of *Super Mario Bros*(Nintendo 1985). Much like the original, IMB consists of Mario (the player's avatar) moving horizontally on a two-dimensional level towards a goal. Mario can be in one of three possible states: small, big, and fire. Each state increases the number of times Mario can take damage without failing the level and also give Mario special abilities. Mario can move left and right, jump, run, and shoot fireballs when in the fire state. The player returns to the previous state if they take damage and dies when taking damage if in the small state. They can also die when falling down a gap in any state. Additionally, unlike the original game, IMB allows for automatic generation of levels.

The Mario AI framework is a popular benchmark for research on artificial intelligence built on top of IMB [71], having been used in AI competitions in the past [71, 136]. It improved on limitations of IMB's level generator, and several techniques have been applied to automatically play [71] or create levels [117, 128], some of which are search-based methods. The framework comes with all the original levels from SMB and SMB Lost world, also with multiple different constructive generators and thousands of generated levels by them. The framework comes also with different AI agents that can play the game with

1	1	1	1	1	1	1
1	4	0	0	0	0	1
1	0	3	1	3	0	1
1	1	0	1	0	1	1
1	0	2	0	3	0	1
1	4	0	0	0	4	1
1	1	1	1	1	1	1

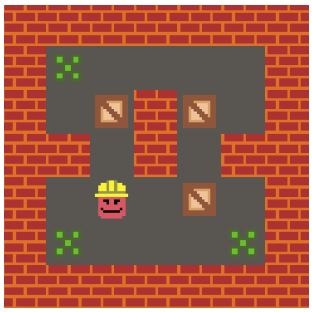


Figure 3.5: Sokoban level as 2D tile map.

different skills Including Robin Baumgarten's A* agent, the winner of the first Mario AI competition [135]. Figure 3.4 shows the first level from SMB that comes with the framework. We use the Mario framework in our work during Chapters 7 and 8.

3.4 PCGRL

PCGRL Framework is a framework that is introduced by us in [75] to explore using Reinforcement Learning to learn a level generator. We preferred to implement our own framework instead of using games from other frameworks as the problem of finding a level generator is harder than searching for level as it requires much more computation time and better reward signals. The PCGRL framework is built from the ground up with these requirements in our mind so the games are stripped to their bare minimum and only the required parts are being evaluated for the reward functions.

The PCGRL framework provides a lot of different problems as long as they can be expressed as tilemaps. Figure 3.5 shows a 2D tilemap of the game of Sokoban. We will only focus on 3 different problems from the framework:

- **Binary:** the easiest task, the goal is to modify a 2D map of solid and empty spaces such that the longest shortest path between any two points in the map increases by at least X tiles and all the empty spaces are connected.

- **Zelda:** the goal is to modify a 2D level for the Zelda game from the GVGAI domain [105]. Zelda is a port of the dungeon system of The Legend of Zelda (Nintendo, 1986) for the GVGAI framework [105]. To guarantee playable levels, the level has to have exactly 1 player, 1 door, 1 key, a few enemies, and the player has to be able to reach the door and the key in at least X steps while making sure enemies are not too close.

- **Sokoban:** the goal is to generate a 2D level for the puzzle game Sokoban (Thinking Rabbit, 1982). Sokoban is a Japanese puzzle game where the player needs to push all the crates towards certain locations on the map called targets while avoiding getting stuck by walls. To achieve the generation task, the level has to have exactly 1 player and the number of crates and targets has to be equal and reachable from the player location. If all constraints are satisfied, the game uses a Sokoban solver (a tree search algorithm with limited depth) to check the playability of the level and make sure levels can be solved in at least X steps.

The PCGRL framework provides signals on how close any tilemap to a playable tilemap (satisfying all the constraints) which can be used as a fitness function or a reward signal. We use the PCGRL framework problems in Chapters 9 and 10.

Chapter 4

Algorithms

This chapter summarizes the different algorithms that we are using during this thesis. This chapter is divided into two main sections similar to the parts of the book. Section 4.1 focuses on the algorithms used during Part 2 of the thesis, while section 4.2 explains the algorithms used during Part 3.

4.1 Content Generation Algorithms

In this section, we are focusing on the algorithms that we are using during the process of searching for levels. The goal is to generate levels for our different domains that satisfy certain constraints (such as playability, certain playstyle, etc) and different from each other. We use search-based algorithm with abilities to satisfy constrained such as FI2Pop [83] and Constrained MAP-Elites [79].

4.1.1 Feasible Infeasible 2 Populations (FI2Pop)

Feasible Infeasible 2-Populations [83] is a genetic algorithm that uses 2 populations at the same time one for feasible chromosomes and the other for infeasible chromosomes. The feasible population tries to improve the fitness of the overall chromosomes, while the infeasible population tries to decrease the number of chromosomes violating the problem constraints. Each population evolves on its own, where the children can transfer between the two populations. This algorithm fits well in the work of search-based generation. It always guarantees that our levels satisfy some hard constraints (such as playability) before optimizing them towards a certain fitness. This algorithm is being used in Chapter 5 and Chapter 8.

4.1.2 Constrained Map Elites (CME)

We introduced this algorithm during our work in bullet hell level generation [79]. The problem is that judging levels is subjective to the player or the designer taste and usually there is no certain value to that we can optimize towards it. For example: having more bullets on the screen in a bullet hell level doesn't make the level better nor worse than before. Constrained Map Elites helps us to find multiple playable levels that are different from each other. Constrained MAP-Elites is a hybrid algorithm that combines the illuminating functionality of MAP-Elites with the constraint-solving abilities of FI2Pop. As in standard MAP-Elites, this algorithm maintains a map of n-dimensions where each dimension is divided into regions. However, rather than a single chromosome, each cell stores two populations. One represents the feasible population that aims to maximize its fitness (measured as a function of playability), while the other is the infeasible population which attempts to satisfy a set of constraints. Every chromosome is located at a (cell, population) combination, and moves on those two levels: a cell level and a population level. A chromosome can move between cells if its behavior characteristics change along with one of the map's dimensions. A chromosome can also move between populations if it satisfies or fails to satisfy its feasibility constraints. This algorithm is being used in Chapter 6 and Chapter 7.

4.2 Generator Algorithms

In this section, we are focusing on the algorithms that we use in the process of searching for level generators. The goal is to generate a level generator that can produce playable and diverse set of levels. Searching for a generator is not an easy task so we introduced, in Part 3, two new representations that allows for a big enough search space to explore. We use the following algorithms to search these spaces for a fit generator.

4.2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization [114] is a reinforcement learning training algorithm introduced by OpenAI in 2017 and considered one of the most standard used algorithms in Deep Reinforcement Learning. The main difference that PPO introduces is that it doesn't allow for big changes during updating our policy network. PPO is used with Actor Critic [84] models. Actor-Critic is a model that consists of two networks, one is called Actor and the other is called Critic. The actor-network is our policy and it generates the probabilities of the next actions. The critic network is responsible to evaluate the current state (Value function). The algorithm usually runs for a couple of steps and records all the interactions with the

environment, then it uses these data to update both networks such that the critic can predict a better value function and the actor can minimize its entropy and increase its performance. PPO algorithm scales these updates such that the policy network doesn't change much which help towards stable training. This algorithm is being used in Chapter 9.

4.2.2 Nondominated Sorting Genetic Algorithm (NSGA-II)

Level objectives sometimes contradict with other objectives in a lot of cases. For example: having a fully connected level might cause the longest shortest path in a level to be small for a lot of different levels except for a small portion. In these cases, a multi-objective optimization algorithm will be better as it will find all the levels that lie on the Pareto front for these objectives. Non-dominated Sorting Genetic Algorithm [39] is a multi-objective optimization algorithm that can try to find the Pareto front of multiple input objectives. At each generation, new chromosomes are generated using crossover and/or mutations and added to the population. The algorithm sorts these chromosomes based on their domination on other chromosomes. A chromosome dominates another chromosome if it is better than or equal to the other chromosome in all its fitness dimensions. The sorting happens in ascending order such that chromosomes that are not dominated by any other are first then followed by the rest in increasing order. The population gets truncated to its original size and we use crowding distance to break the ties between chromosomes with the same domination count. The crowding distance measures the diversity between the chromosomes based on their fitness values and gives higher values to the diverse ones in the population. Using crowding distance guarantees that the evolution won't collapse very quickly. This algorithm is being used in Chapter 10.

Part 2

Searching for Content

Chapter 5

Search-Based General Level Generation

We start our journey towards the general level generation problem by investigating the search-based procedural content generation [137] approach applied to multiple different games. We decided to focus on search-based approaches due to their ability to guarantee that the generated content adheres to certain criteria. To save time and maintain generality from the ground up, we extended the General Video Game Playing Framework [105] to include a level generation interface. The general video game playing framework not only provides us with more than 100 games but also provides us with AI controllers that were constructed for the planning competition [106].

In this chapter, we explain our new framework, our search based generator, then we empirically compare it to two baseline constructive algorithms [116] on three different games from the GVG-AI framework (Frogs, PacMan, and Zelda).

5.1 General Video Game Level Generation Framework

The General Video Game Level Generation framework [80] (GVGLG) is an extension to the General Video Game Playing framework [106]. It allows competitors to integrate their level

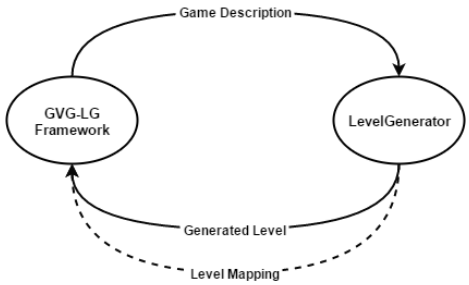


Figure 5.1: Relationship between the GVG-LG Framework and the Level Generator

generators and test them against a variety of different games from the framework. Figure 5.1 shows the relationship between a supplied level generator and the GVG-LG framework. The framework provides the generator with a game description object and in return, it provides a generated level in form of a 2D array of characters. The generator can also return a level mapping to help the framework decipher the generated level. If the generator didn't provide the level mapping, the system will use the default one from the game description.

The game description object encapsulates all the information about the current game. It provides the generator with different functions that can retrieve information about the different game sprites, the termination conditions, the interactions, and the default level mapping.

The sprites are divided into six different categories: Avatar, NPC, Resource, Portal, Static, and Moving. Generators can get a list of all sprites or for a specific sprite category. For example: In *Space Invaders*, we have one avatar which the player, one NPC which is the aliens, two moving which are the player and aliens bullets, etc.

Generators can get a list of all the termination conditions for winning and losing through the GVGLG interface. For example: In *PacMan*, the winning condition is the number of pellets sprites has to be equal to zero, while the losing condition is the number of the avatar sprite is equal to 0.

Generators can get a list of all the interactions that can happen between any two defined sprites using the GVGLG interface. The list contains information about the collision effect and the score change due to this interaction. For example: In *Frogger*, when a frog collides with a car, the frog dies and the player score is reduced by 1.

A Level Mapping is a Hash Table which maps each character to a list of sprites. The generator can either use the default hash table (provided through the interface) or provide a new one to the framework. For example: In *PacMan*, *p* refers to a *pill*, while *a* refers to *PacMan* himself.

The generated level must follow two hard constraints. The first constraint is that the level string should not contain any unknown character that is neither in the original nor the new level mapping. The second constraint is that the level must only have one avatar. Violating any of these constraints stops the framework from running the provided generator.

5.2 Description of generators

This section explains three sample level generators that are provided with the GVG-LG Framework, and which are also compared empirically in section 5.3.

5.2.1 Random Level Generator

This is a very simple level generator. It generates levels by placing the defined sprites at random empty positions, then surround the borders with solid tiles. Each tile position has a probability (set to 10% to give the best playable and visually appealing) to be filled with a random sprite picked using a uniform distribution. This probability can be adjusted to generate more or less cluttered levels. The generated levels have a size proportional to the number of game sprites defined. The generator ensures that the produced levels have at least one of every sprite and only one avatar.

5.2.2 Constructive Level Generator

This level generator utilizes the information presented in the game description object to generate better-designed levels. For example: if the *avatar* sprite is of type *FlakAvatar* or *HorizontalAvatar* (a type of avatar which can only move horizontally), it should be placed either at the top or the bottom of the level. The generator analyzes the game description object and divides the game sprites into five different categories:

- **Avatar Sprites:** the player-controlled sprite as defined by the game description object.
- **Solid Sprites:** block the movement of the avatar and do not have any other interaction with the avatar.
- **Harmful Sprites:** kill the avatar upon interaction (or spawn sprites that do this).
- **Collectible Sprites:** are destroyed upon interaction with the avatar and are not harmful sprites.
- **Other Sprites:** any sprites that do not fit into the above categories.

The generator also provides two lists: the first list keeps track of game sprites created from other sprites (i.e. bullets), while the second list contains sprites that appear in the termination set. It also calculates a priority value for each sprite, based on how many interaction rules it occurs in. All this information is used to generate a well-formed level. Figure 5.2 summarizes the core steps done by this approach. The generation procedure is divided into four core steps with a pre-processing step and a post-processing step:

1. **(Pre-processing) Calculate Cover Percentages:** This step calculates the percentage of tiles in the generated level that should be covered with sprites. It also calculates that percentage for each different sprite category. The total cover percentage is directly

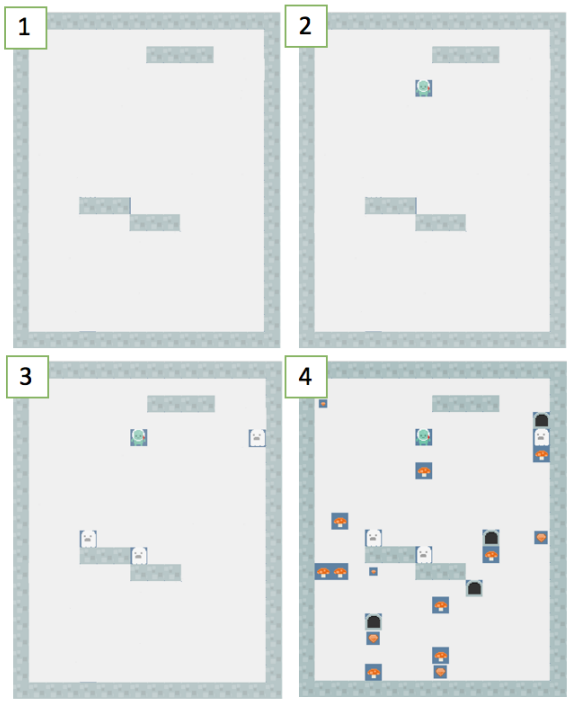


Figure 5.2: Steps applied in the Constructive generator for Pacman: (1) Build a Level Layout (2) Add an Avatar Sprite (3) Add Harmful Sprites (4) Add Collectible and Other Sprites

proportional to the number of collectible sprites, and it is inversely proportional to the number of harmful sprites and the number of sprites that are created by other sprites. All categories have a percentage directly proportional to the sum of the priority values of each sprite in each category.

2. **Build a Level Layout:** This step only takes place when there are solid sprites. It picks a random solid sprite and surrounds the level with that sprite. Based on the calculated solid percentage it fills the internal level with solid sprites that are connected to each other without blocking any area.

3. **Add an Avatar Sprite:** This step places a randomly picked avatar sprite to a random free location.

4. **Add Harmful Sprites:** This step adds harmful sprites to the game based on the calculated harmful percentage. If the harmful sprite is a moving sprite, the generator chooses a free location away from the avatar sprite but if the sprite is a static sprite, the generator chooses any random free location.

5. **Add Collectible and Other Sprites:** This step places randomly picked sprites to randomly free locations. The number of added sprites depends on their cover percent-

ages.

6. **(Post-processing) Fix Goal Sprites:** This step makes sure that the number of goal sprites is greater than the number specified in the termination set for that sprite type; sprites are added until this is the case.

5.2.3 Search-based Level Generator

This is a search-based level generator based on the Feasible Infeasible 2 Population Genetic Algorithm (FI2Pop) [83]. The initial population is generated using the Constructive level generator described in Section 5.2.2. The levels are represented as a 2D array of tiles. Each tile consists of an array of strings representing all sprites at that tile. The generator uses a one-point crossover which swaps the 2 chromosomes around a random tile. For mutation it uses 3 different operators:

- **Create:** creates a random sprite to any random tile position.
- **Destroy:** clears all sprites from a random tile position.
- **Swap:** swaps the sprites in two random tile positions.

The level generator uses an altered version of the provided *OLETS* controller for simulation-based fitness and constraint evaluation. The provided *OLETS* controller is a well-performing controller that was designed by Adrien Couëtoux, which won the 2014 edition of the GVGAI competition [106]. The controller has certain super-human skills (i.e. fast reaction time), and it was therefore altered to make its playing style somewhat more human-like. This is achieved through two modifications: adding action repetitions so that the controller has a tendency to repeat the last action for few time steps, and adding *NIL* repetition so that the controller has a tendency to add *NIL* values between changing actions. These modifications make sure that the controller cannot handle situations that require extremely fast reactions, which in turn discourages the generation of levels that include such situations.

Some of the fitness and constraint evaluations are compared with the *OneStepLookAhead* or *DoNothing* players. The *OneStepLookAhead* player plays by greedily choosing among the immediate next actions. The *DoNothing* player simply applies the *NIL* action. Both players play for the same amount of steps as the altered *OLETS* controller. The feasible population is subjected to two different heuristic functions:

- **Score Difference Fitness:** difference between score achieved by *OLETS* and the best score achieved by *OneStepLookAhead* (over 50 runs), as suggested by Nielsen et

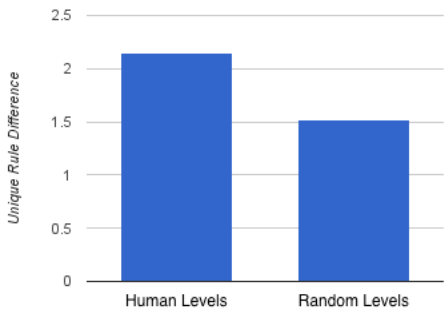


Figure 5.3: Average number of unique rules used by OLETS when playing human-designed levels and randomly generated levels for all VGDL games

al. [99]. That paper suggests that the relative difference between algorithms can be used to differentiate between well-designed and badly designed games. We compared with the score achieved by *OneStepLookAhead*, a weak player, to make sure to generate levels that require more skill to get a high score.

- **Unique Rule Fitness:** the number of unique events that happened in the level due to the avatar or any sprite spawned by it. We hypothesized that well-designed levels try to utilize almost all the different game rules while badly designed levels don't. To investigate this hypothesis, we played all levels of the publicly available VGDL games using *OLETS* and calculated the number of unique rules used; we then compared this to the number of unique rules used when playing randomly generated levels using the same algorithm. Figure 5.3 shows that well-designed levels tend to have a higher average of unique rules than random designed levels.

The fitness functions are treated as an average value of both the *Score Difference Fitness* and the *Unique Rule Fitness* as show in equation 5.1.

$$f_{feasible} = \frac{f_{score} + f_{rule}}{2} \tag{5.1}$$

where f_{score} is the *Score Difference Fitness* and f_{rule} is the *Unique Rule Fitness*. The Infeasible population is subject to seven different constraints:

- **Avatar Number:** Each level must have one avatar.
- **Sprite Number:** Each level must have at least one of each sprite which is not spawned by other sprites.

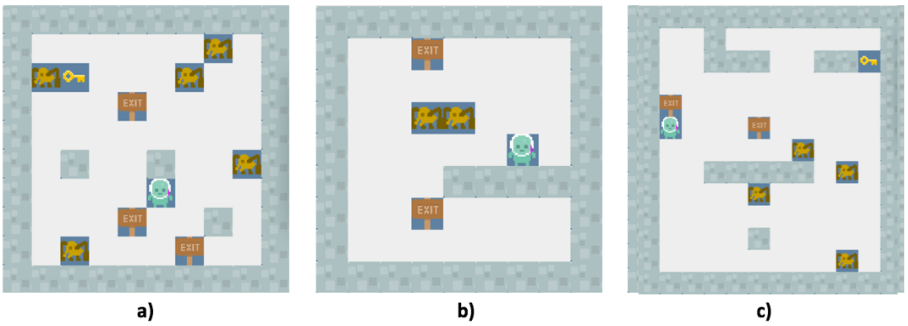


Figure 5.4: Examples of generated levels from all the algorithms for Zelda: (a) Random Level Generator (b) Constructive Level Generator (c) Search-Based Level Generator

- **Goal Number:** The number of goal sprites must be greater than the limit of the termination rule associated with these sprites.
- **Cover Percentage:** Between 5% and 30% of level tiles must be covered by sprites.
- **Solution Length:** Levels must not be solved (by any of the players) in less than 200 steps.
- **Win:** The *OLETS* player must win the generated level during evaluation.
- **Death:** The *DoNothing* player must not die for at least 40 steps over 50 different runs. Also, it must not win after the same amount of time steps as *OLETS*.

5.3 Pilot Study

We performed a small pilot study to test the performance of the proposed algorithms. The study used three different games, Frogger, PacMan, and Zelda. The study did not incorporate any puzzle games because most existing agents do not play puzzle games well. Each generator supplies five levels for each game and is allowed up to five hours to generate each level. Figure 5.4 shows an example level for *Zelda* from each generator.

The generated levels were tested by a group of human players. For that purpose, we created a program to help the players to understand the purpose of the study and get them familiar with the games. After that, the system picked two random levels from any of the generators and showed them to the human player. After playing both of these levels, the system showed a poll question as shown in Figure 5.5 where the player indicated which level they preferred, if both levels are equally preferred, or if none of them was preferred. We use preference indication (ranking) rather than a rating scheme such as Likert scales and rankings have been shown to be more consistent and reliable [151]. Rating systems

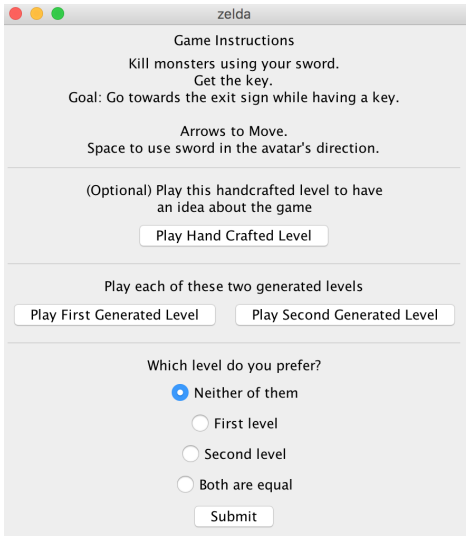


Figure 5.5: Poll question comparing quality of the generated levels.

	Preferred	Non-preferred	Total	Binomial p-value
Search-Based vs Constructive	23	12	35	0.0447
Search-Based vs Random	21	10	31	0.0354
Constructive vs Random	17	24	41	0.8945

Table 5.1: This table shows players' preferences between different generators, aggregated over all three games.

violate two basic assumptions (ratings are ordinal data and they are not linear). On the other hand, ranking systems treat data as ordinal data while minimizing subjectivity bias. Data was collected from 25 players where each player played five pairs of games on average. The preference data was submitted automatically to a database.

Table 5.1 shows the results of our pilot study for all the three level generators. To simplify our statistical test, the *Equal* and *Neither* answer to the survey are discarded in this study, only the preference choices are taken into account. For this study, we used three two-tailed binomial tests to test three null hypotheses about our generators, namely that there are no differences in preference between constructive and search-based, between random and search-based, and between random and constructive.

From the preference numbers in the table, it is clear that Search-Based is better than both the Constructive and the Random, but the Constructive is voted far less than random which is not expected. The calculated p-values support this conclusion, as they reject only two null hypotheses (Search-Based and Random; Search-Based and Constructive) while showing no significant difference between the Constructive and Random level generators.

5.4 Discussion

We had expected the search-based generator to produce levels that would be significantly better (more often preferred) than the Constructive generator; this was borne out in our pilot study. The main reason for this is likely the simulation-based evaluation function including constraints which ensured having a good playable level. In particular, using an altered version of *OLETS* ensured that the levels would not have too many enemies too close together, making the level playable for humans operating on human timescales.

The same cannot be said about the Constructive versus Random generator levels. The Random generator was preferred marginally more often than the Constructive generator. While the effect is not statistically significant, the likely main reason for any such effect is that the Constructive algorithm does not guarantee that each game object is generated at least once like in the case of the Random generator. For example: the Zelda level in Figure 5.4 looks better than the random level but since it does not include a key, people tend to prefer the random generator over the Constructive generator. Not having this constraint gives more flexibility to Constructive generator over Random generator. For example: In almost all VGDL games there is multiple different kinds of enemies where they are used to construct different levels. For example in Zelda, you can have slow enemies, normal enemies, and fast enemies. You don't need to place one of each type of enemy in the generated levels. Based on this fact, the Constructive generator tries to generate at least one sprite of each different category.

General Level Generation is a very hard problem to tackle. In order to generate playable levels for any game, we need a language to describe all different kinds of games. The only languages that could represent all games would be Turing-complete programming languages. However, programming languages are not a good fit to describe games as they are simply too general and can be used to describe any kind of software (not only video games); a random string in a standard programming language would not be likely to be a functioning game or even a functioning program at all. This was the main reason for the development of Video Game Description Language (VGDL). VGDL is a description language that is readable, simple and can be used to describe a subset of different kinds of video games (real-time 2D games). Although VGDL looks limited from outside, it can be used to describe different genres (such as puzzle, arcade, and shooter games). Even if the language is limited, the range of possible contributions is very large. For example: Ludi language was developed by Browne and Marie [23] is only used to describe Combinatorial games, they managed to evolve a very interesting board game (Yavalath) which is ranked in the 100 top best abstract games by BoardGameGeek [147].

5.5 Summary

In this chapter, we build a framework to tackle the general level generation problem. The framework was designed as an extension for the general video game playing framework. This allowed us to access a wide range of games and AI controllers. We also introduced three different generators: random, constructive, and search-based. We also did a small pilot study to compare the levels produced by these generators. We had conjectured that the search-based level generator would be better than the constructive level generator, which in turn would be better than the random level generator. Although the human players were unable to distinguish between the Constructive and Random generators, our search-based generator exceeded both of the Constructive and Random generators.

On the other hand, our search-based generator took five hours to generate one single level where most of the levels in the last population are the same due to convergence [48]. Also, there was no guarantee that the generated levels will be different between several runs. Our next step will be to improve the search-based level generator so it can generate a diverse amount of content in a fixed amount of time.

Chapter 6

Quality Diversity Level Generation

In the previous chapter, we introduced the general video game level generation framework. We used it to show that the search-based approach suits the general level generation problem better than constructive approaches. We noticed that search-based techniques take a long time to generate a single level where we could have sampled more levels using a constructive generator. Also, multiple runs of a search-based generator don't guarantee diversity between these runs.

In this chapter, we are tackling these problems by using a quality diversity algorithm [110]. Instead of trying to decrease the generation time, we decided to focus on providing a diverse set of levels during the same amount of time. To that end, we present Talakat[1], a description language designed to encapsulate and describe bullet-hell patterns, a variation upon the MAP-Elites algorithm [96] (a quality diversity algorithm) that generates Talakat descriptions, and a simulation evaluation method that guides evolution toward levels of specific challenge for different play styles.

6.1 Constrained MAP-Elites

Generating levels for a bullet hell game is a non-trivial problem. In addition to playability, one must consider difficulty, visual aesthetics, distribution, etc. Designing a fitness function to incorporate these dimensions is challenging. Additionally, there is more than one interesting level in the search space. For example: higher bullet count does not guarantee a more difficult level. Therefore, using a standard optimization algorithm or multi-objective optimization algorithm may be insufficient as it may fail to consider possible optima in remote reaches of the search space. It is therefore important to use Constrained MAP-Elites,

[1] Talakat means bullets in Arabic

as it can comprehensively search multiple subspaces and identify the optima contained in those areas.

6.1.1 Chromosome Placement

Upon generation, each chromosome is evaluated by an agent which is explained in section 6.1.3. The results of the evaluation determine the level's cell and population placements. The map used for the Constrained MAP-Elites has three dimensions: entropy, risk, and distribution. These dimensions were chosen based on the belief that they represent some aspect of difficulty in bullet hell levels. Entropy reflects the amount of input required of the player by calculating the information entropy using the first, second, and third derivatives of the agent's action sequence. This value is calculated by taking the average entropy over: the actions' probabilities over the agent action sequence, the probability of changing actions over the agent action sequence (first derivative), the probability of changing the values of the agent first derivative (second derivative), and the probability of changing the values of the agent second derivative (third derivative). In practice, entropy is correlated to the number of times the player changed direction, stopped while moving, or began moving while stopped. Risk reflects the presence of bullets in close proximity to the player. It is calculated by dividing the screen into a grid and counting the number of squares around the player that contain bullets. Distribution represents the amount of space occupied by bullets. The distribution is calculated in a similar fashion to the risk, by dividing the screen into a grid and calculating the number of squares occupied by at least one bullet. These three values are calculated by the agent during its evaluation and are used to place the level within its appropriate cell.

During the time of evaluation, the agent is also determining the level's population placement. A chromosome is placed in the feasible population if it satisfies the following two constraints:

- Number of spawners doesn't exceed a fixed maximum value.
- There are at least 10 bullets present for more than 50% of frames.

The fitness of chromosomes in the feasible population is inversely proportional to the remaining boss health (the inverse of remaining boss health is an analog for survival time). A chromosome that fails these constraints is placed in the infeasible population, where fitness is calculated by multiplying the inverse of remaining boss health by the percentage of frames that contain bullets.

6.1.2 Genetic Algorithm

The populations within the individual MAP-Elites cells are evolved using grammatical evolution [102]. Grammatical evolution is distinguished from genetic programming algorithms in that grammatical evolution uses a mapping between a set of numbers and the grammar and acts on the numerical representation, whereas genetic programming algorithms operate directly on expressions. Each chromosome is represented as 11 arrays, each of which consists of 23 integers between 0 and 99. The first 10 arrays are mapped to 10 different spawners using the grammar defined in figure 3.2a where each number correspond to a grammar expansion. The last array is mapped to the boss section in the same manner as the spawners.

To create new chromosomes, we use uniform crossover over the 11 arrays and uniform mutation on the integer values of one of the sequences. It is important to note that the parent chromosomes are retained, effectively implementing 100% elitism. Parent chromosome selection occurs by first selecting a random cell from the map, and then using rank selection to select its parent chromosome. Once a new chromosome is generated, it is evaluated by the agent and placed in its cell and population as outlined in section 6.1.1. This thesis will refer to a set of these mating events as a generation. If, after the creation of a generation, a cell's population exceeds its capacity, the lowest-performing chromosomes are removed starting from the infeasible population.

6.1.3 Agent

All chromosomes are tested using an A* [64] agent. The A* agent's heuristic function consists of 4 parts: progress, loss, safety, and future location. Equation 6.1 outlines the heuristic function used by the A* agent. $progress$ is the number of frames the agent has survived so far, $lose$ is equal to 1 if the agent dies and 0 otherwise, $safety$ corresponds to the number of frames a completely stationary agent would survive at its current position, up to a maximum of 10, and $future$ is the distance between the agent current location and the location with the fewest surrounding bullets. $future$ is weighted less heavily as it is less important than agent survival, which is reflected by the other values, while $lose$ has the highest weight as the agent's survival is its highest priority.

$$f(x) = 0.5 \cdot progress - loss + 0.5 \cdot safety - 0.25 \cdot future \tag{6.1}$$

Additionally, the agent's actions are constrained by setting two agent properties: dexterity error and strategy error [67]. Dexterity error forces the agent to repeat its actions for a

	Dexterity	Strategy
low	10	40
medium	6	60
high	2	80

Table 6.1: Values used for dexterity and strategy dimensions.

number of frames. The severity of dexterity error is modeled as a Gaussian distribution modeling the number of repeated frames. A high dexterity agent is forced to repeat fewer frames. The strategy error reduces the time allotted for the agent's decision-making process. A high strategy error can force the agent to make decisions before it arrives at an optimal choice. A high strategy agent has more time to explore its options. By using different dexterity-strategy configurations for the evaluating agent, it is possible to guide the evolution toward patterns that heavily favor higher dexterity, higher strategy, or some combination of the two. For this work, dexterity and strategy each have 3 (low, medium, high) possible values, for a total of 9 possible dexterity-strategy configurations.

6.2 Experiments

We ran 9 experiments, one for each agent configuration. These experiments were run in parallel with the expectation that each experiment would independently generate interesting and playable levels tailored to the evaluating agent's dexterity and strategy level. Table 6.1 shows the strategy and dexterity values used during the experiment. The dexterity values are the standard deviation of the Gaussian noise function representing the number of repeated frames. The strategy values correspond to the amount of decision-making time given to the agent in milliseconds for every frame.

In each experiment, we initialized the Constrained MAP-Elites with 100 random levels. The Constrained MAP-Elites use crossover with a 70% probability and mutation with a 30% probability. Each map dimension is divided into 11 values (from 0 to 10). Each cell in the map has a population capacity of 50 chromosomes, shared between the feasible and infeasible sub-populations. A generation consists of 100 mating events. Each experiment was run for 24 hours.

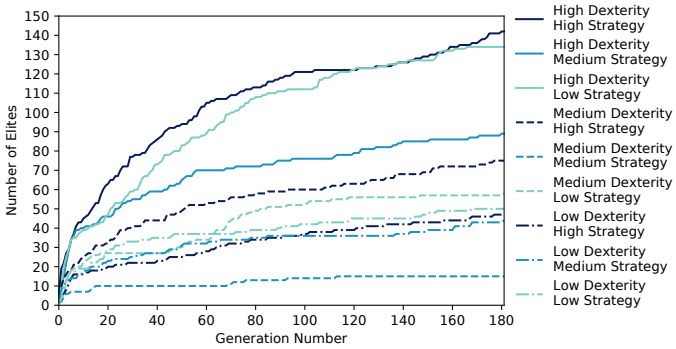


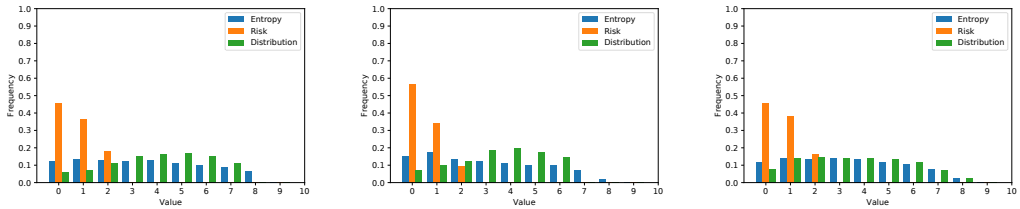
Figure 6.1: Number of elites with fitness of 100%.

6.3 Results

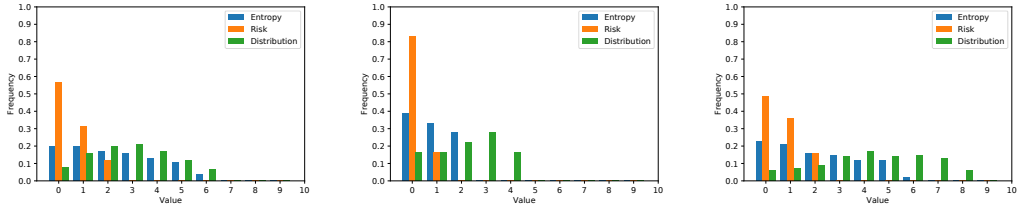
We analyzed the number of elites generated with each generation for the constrained MAP-Elites for every configuration. Figure 6.1 shows the number of elites with fitness of 100% with every generation. The figure reflects how successful the constrained MAP-Elites has been at finding new elites with every generation. Because feasible levels are evaluated based solely on survivability, it is reasonable to conclude that the majority of the initial population was unsurvivable, with survivability increasing between generations.

Due to the nature of the agent parameters, the high dexterity - high strategy experiment went through fewer generations (approximately 180) than the low dexterity - low strategy experiment (approximately 1700). One generation of high dexterity - high strategy takes substantially more time than a low dexterity - low dexterity generation. This discrepancy is offset by the fact that high dexterity and high strategy experiments will find survivable levels in fewer generations because the agents are better able to survive. It is also worth noting that the medium dexterity - medium strategy experiment has a highly anomalous graph. Upon investigation, this experiment was found to generate far fewer high-performing levels than the other experiments, which is especially surprising considering it occupies none of the extremes as far as dexterity and strategy parameters are concerned. It is possible that the population failed to mutate favorably over its run, but the exact cause is unknown.

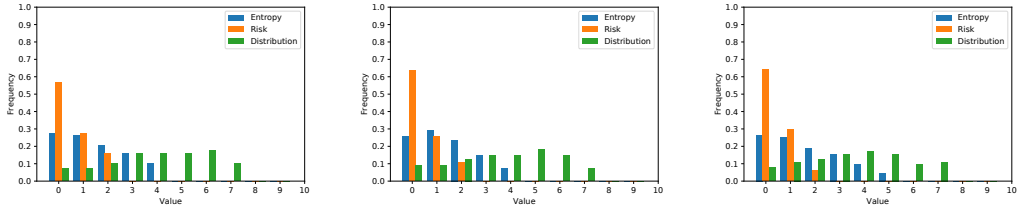
Figure 6.2 presents risk, entropy, and distribution (as discussed in section 6.1.1) of elite levels generated by each experiment. An elite level has a fitness of 1, the highest possible value, which indicates that the evaluating agent did not die when playing it. From figure 6.2, one can see that high dexterity experiments generally have more high entropy elite levels than low dexterity experiments. It seems obvious that high dexterity agents would be able to survive levels that would require more movement. However, this difference confirms that Constrained MAP-Elites is capable of generating a set of levels that are too demanding



(a) High Dexterity - Low Strategy (b) High Dexterity - Medium Strategy (c) High Dexterity - High Strategy



(d) Medium Dexterity - Low Strategy (e) Medium Dexterity - Medium Strategy (f) Medium Dexterity - High Strategy



(g) Low Dexterity - Low Strategy (h) Low Dexterity - Medium Strategy (i) Low Dexterity - High Strategy

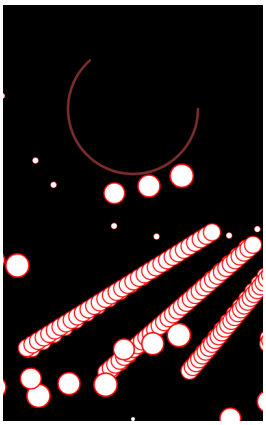
Figure 6.2: Histograms for all the different 9 experiments after 24 hours. The x-axis represents the value of entropy, risk, or distribution depending on which bar one looks at. The y-axis represents the frequency with which a value with the corresponding value appears. That is, an orange bar at x=1 and y=0.4 would indicate that 40% of the generated levels had a risk value of 1. The frequency values for a single evaluation dimension (entropy, risk, distribution) in a graph add up to 1.

for the low dexterity agent but are completable for the high dexterity agent, a set of levels which could be considered both non-trivial and playable. In this specific instance, any level generated by a high dexterity experiment with an entropy greater than 5 is will likely be too demanding for a low dexterity agent. There does not appear to be any noticeable difference in the three metrics with respect to strategy. It is possible that the difference between low and high strategy is too small to create any effect or that differences in strategy impact the levels in a way that cannot be expressed by entropy, risk, and distribution.

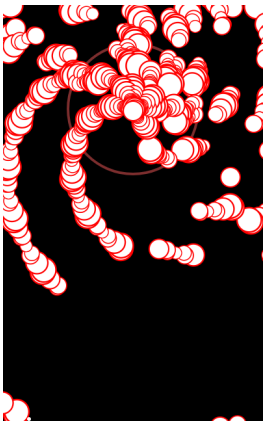
There are noticeable and notable flaws in the dimensions used by this implementation of Constrained MAP-Elites. For example, the distribution metric does not appear to be a

good analog for difficulty. One would normally expect a negative correlation between a difficulty analog and the number of survivable levels, but this is not the case for distribution. In every one of the histograms, the frequency of each value of distribution appears to be fairly close, with no indication of a trend with respect to value. On the other hand, the risk metric exhibits this behavior quite strongly. There are far fewer survivable levels with a risk value of 3 than there are levels with a risk value of 1. However, the range of values for risk appears to be largely invariant across all experiments. It is entirely possible that risk is an effective difficulty analog that is unaffected by dexterity and strategy. This renders it less useful than entropy as a means of identifying more or less demanding levels in this specific experiment.

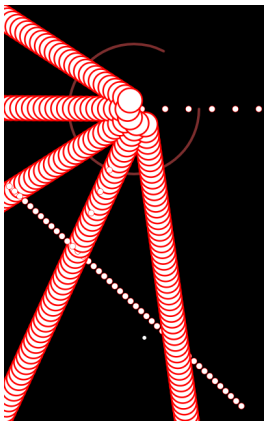
Figure 6.3 shows examples from the 9 experiments. We aimed to show levels with high entropy values to demonstrate the differences in difficulty generated by the experiments. The top row of images shows levels with entropy 8, the middle row shows images of levels with entropy 6, and the bottom row shows images of levels with entropy 4. Each of these values was chosen as they are higher than the highest entropy achieved by an elite in an experiment with a lower dexterity. Therefore, levels in the top row should be too dexterously demanding for medium-dexterity agents, and levels depicted in the middle row should be too hard for low dexterity agents. Although the same relation cannot be definitively stated for strategy, visual observation, and analysis shows distinctive differences in the amount of planning required for high strategy levels versus low strategy levels. For example, the level shown in figure 6.3i begins with an empty stage and quickly floods the left side with bullets after a short period of time. An agent without enough decision-making time to predict this will fail to move to the safe right side before it becomes closed off. Similar requirements are evident in the levels depicted by images 6.3f and 6.3c. Both levels open by splitting the level into sections and firing bullets into certain sections sometime later. A low strategy agent is less likely to be able to predict which sections will be safe in the time it is allotted, and inevitably die. This requirement is less pronounced but still present in medium strategy levels. Images 6.3e and 6.3f show levels with somewhat jagged walls of bullets. An agent can dodge the immediate threat by going between bullets, only to find itself trapped in the concave structure created by the jagged shape. From observing the images of levels created by the experiments, we believe that strategy did have some impact on the design of generated levels, even if the influence is not reflected in the statistics presented by figure 6.2.



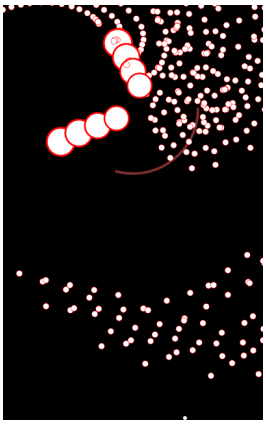
(a) High Dexterity - Low Strategy



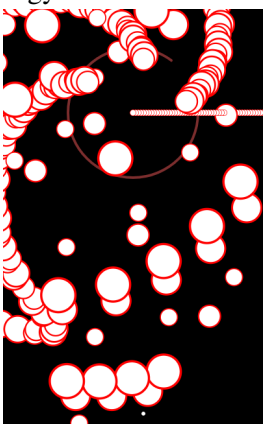
(b) High Dexterity - Medium Strategy



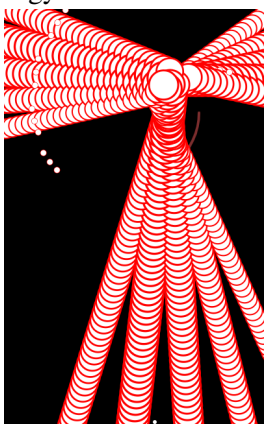
(c) High Dexterity - High Strategy



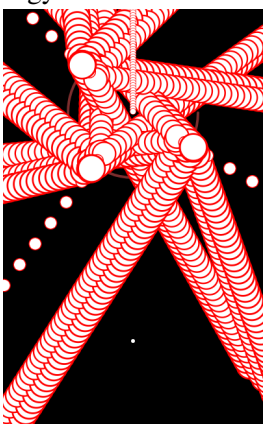
(d) Medium Dexterity - Low Strategy



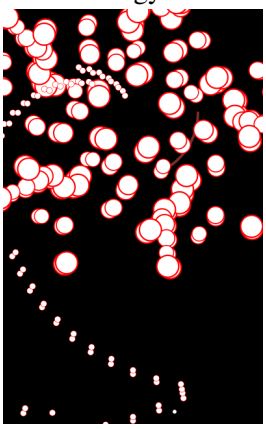
(e) Medium Dexterity - Medium Strategy



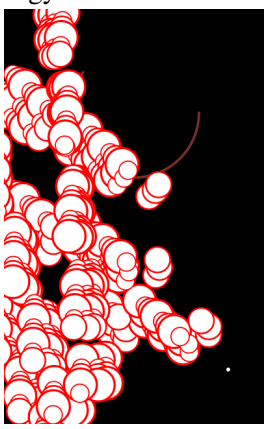
(f) Medium Dexterity - High Strategy



(g) Low Dexterity - Low Strategy



(h) Low Dexterity - Medium Strategy



(i) Low Dexterity - High Strategy

Figure 6.3: Examples of the generated levels for all the different 9 experiments. The high dexterity experiment images show levels with entropy 8, the medium dexterity experiment images show levels with entropy 6 (except image 6.3e due to an anomaly), the low dexterity images show levels with entropy 4.

6.4 Summary

In this chapter, we presented Talakat a new framework that can be used to describe bullet-hell levels. We also introduced a hybrid evolutionary algorithm called Constrained MAP-Elites that combines the MAP-Elites technique and the Feasible-Infeasible 2-Population genetic algorithm. We showed that the Constrained MAP-Elites can be used with Talakat to generate variations of levels. We suggest using Constrained MAP-Elites as a technique in level generation as game levels are subjective. Instead of trying to define a "good" level, one can use multiple metrics as different dimensions of the Constrained MAP-Elites and utilize only playability for the fitness function. From the analysis of the histograms in the 9 experiments as well as high-performing levels, we confirmed that it is possible to create levels of varying difficulty.

One of the problems in the current technique is the behavior characteristics (the MAP-Elite dimensions). It is not clear how to define a more generic metric for these behavior characteristics for general level generation. Another problem is the length of the level, having long levels causes the measurement for the behavior characteristic to be less descriptive as we only use mean but not considering standard deviation. In the next chapter, we will tackle these two problems by dividing the level into smaller sections that can be stitched together later while using a general behavior characteristics based on the triggered game mechanics in the winning playtrace.

Chapter 7

Mechanical Diversity Level Generation

Constrained map-elites (CME) can find a diverse set of generated levels in the same amount of time a FI2Pop algorithm takes. On the other hand, it introduces behavior characteristics that need to be defined to use the technique. Behavior characteristics are game defined metrics that differentiate between different levels. Although the behavior characteristics have been helpful, they make achieving generality harder. We need not only general fitness and constraints function but also general behavior characteristics.

In this chapter, we use game events/mechanics as the behavior characteristic (BCs) for our CME algorithm to generate small level sections called scenes. From a player's perspective, a game mechanic is "...everything that affords agency in the game world" [120]. For example, jumping over a gap in Super Mario Bros (Nintendo, 1985) is considered a game mechanic, as would destroying a monster in an action game. In this work, the used mechanics for each game are the same as the game rules. This doesn't need to be the case as our definition of mechanics involves more complex behavior than triggering a single game rule. For example: jumping over a Goomba without killing it is a game dynamic that is not specified in the game rules but according to our definition it is a game mechanic. Using game mechanics as BCs will allow the CME algorithm to generate scenes that are different from each other mechanically. For example: If we have two dimensions: player jumps over a gap (Jump) and player kills an enemy (Kill), the generated scenes suppose to cover all the variants of these two mechanics allowing for levels that have different interactions (No Jump/No Kill, Jump/No Kill, No Jump/Kill, and Jump/Kill). Generating scenes based on their mechanics is helpful as all games have events/mechanics happening during a playthrough which makes this behavioral characteristic general and can be easily transferred between different games. In this chapter, we assume these mechanics are defined by the designers but this work can be extended using machine learning algorithms to automatically detect these events/mechanics [61].

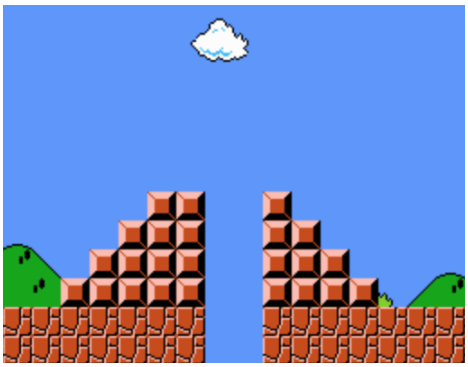


Figure 7.1: Super Mario Bros scene where Mario need to jump over a gap from the first pyramid to the second pyramid.

Lastly, we decided to generate scenes instead of full levels to avoid the problem of the behavior characteristic distribution over big levels discussed in the previous chapter. To get a full level, we can use the generated scenes as building blocks to more complex levels by stitching them together (which is discussed in the next chapter). We test the new behavior characteristics on two domains Super Mario Bros and General Video Game Framework.

7.1 Super Mario Bros

In the first half of this chapter, we generate scenes for the Mario AI Framework. A *scene* is a small section of the level that encapsulates a certain idea [4]. For example, figure 7.1 shows a scene from the first Super Mario Bros (Nintendo 1985) where the idea is to make sure the player learns about the jump mechanic by accurately jumping from the first pyramid to the second pyramid without falling in the gap. We use the *Constrained Map-Elites (CMElites)* algorithm [79] to generate passable scenes using a perfect agent and recording every mechanic that triggers during that playthrough. The algorithm places the chromosome in the appropriate cell in the CMElites' map depending on the triggered game mechanics.

7.1.1 Scene Representation

We assumed that a *Super Mario Bros* (Nintendo 1985) screen is equivalent to a scene. Therefore, chromosomes are represented as a sequence of vertical slices sampled from the first *Super Mario Bros*, similar to the representation used by Dahlskog and Togelius [35]. A scene consists of 14 vertical slices padded with three-floor slices surrounding the scene, as shown in Figure 7.2. This padding is necessary to ensure that there is a safe area where Mario starts and finishes the scene. Additionally, these slices were collected from the levels presented in the Video Game Level Corpus (VGLC) [131]. There are 3721 vertical slices in

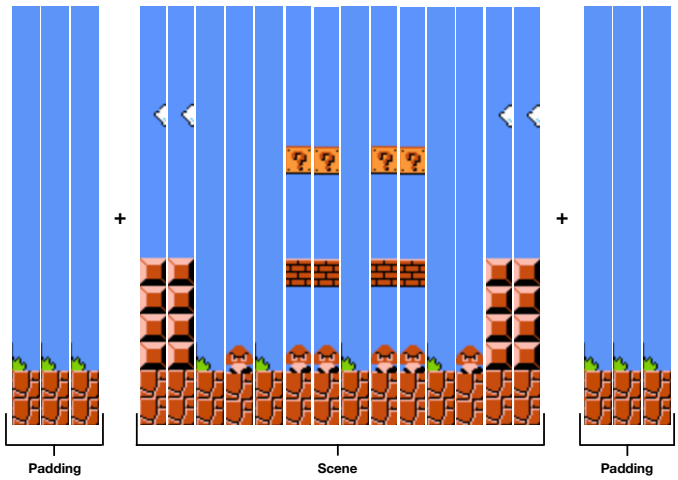


Figure 7.2: The chromosome consists of 14 vertical slices padded with 3 floor slices on both ends.

the corpus with 528 unique vertical slices. Slices are sampled based on their percentage of appearance in the VGLC. For example: a flat ground slice will have a higher chance to be picked compared to other slices.

7.1.2 Scene Evaluation

Scenes are evaluated in relation to the playability constraint. The constraint value is equal to 1 if a playing agent can beat the scene, and directly proportional to the traversed distance otherwise. The playability constraint is checked using Robin Baumgarten's A* agent, the winner of the first Mario AI competition [135]. Equation 7.1 shows the calculation of the playability constraint.

$$constraints = \begin{cases} 1 & \text{if } A_{perf} = 1 \\ \frac{d_{perf}}{d_{scene}} & \text{otherwise} \end{cases} \tag{7.1}$$

where A_{perf} is the result of the agent playing the scene, d_{perf} is the distance traversed by the agent, and d_{scene} is the scene length, used to normalize the result to $0 - -1$.

If the scene satisfies these constraints, it is also evaluated in terms of simplicity. The simplicity fitness tries to reduce the entropy in the scene. The entropy calculation makes sure that generated scenes place tile efficiently and have high horizontal consistency. Equation 7.2 shows the fitness calculation equation.

$$fitness = (w \cdot (1 - H(X))) + \left((1 - w) \cdot \left(1 - H\left(\bar{X}\right)\right)\right) \tag{7.2}$$

where w is a predefined weight to balance both terms of the equation, X is the set of

Dimension	Description
Jump	is 1 if the player jumped in the level and 0 otherwise.
High Jump	is 1 if the player jumped higher than a certain value and 0 otherwise.
Long Jump	is 1 if the player's horizontal traversed distance after landing is larger than a certain value and 0 otherwise.
Stomp	is 1 if the player stomped on an enemy and 0 otherwise.
Shell Kill	is 1 if the player killed an enemy using a koopa shell and 0 otherwise.
Fall Kill	is 1 if an enemy dies because of falling out of the scene and 0 otherwise.
Mushroom	is 1 if the player collected a mushroom during the scene and 0 otherwise.
Coin	is 1 if the player collected a coin during the scene and 0 otherwise.

Table 7.1: Constrained Map-Elites' dimensions.

different used tile types in the scene, and $\bar{X}$ is the set of tile changes in the scene (in our case, we are considering the horizontal changes only). $\bar{X}$ is calculated by running over the full level row by row and changing tiles to be 1 if they are different from the upcoming tile and 0 otherwise. This operation is similar to applying horizontal Sobel operator [127]. We used w equal to 0.2 to weighted the latter (i.e. the horizontal change entropy part) higher than the former, as the abrupt changes in tiles occur more often when there are many different types of objects, making the scene look noisy. For example, if there are two scenes with the same amount of objects, they will have the same entropy, but the order of their arrangement will affect the horizontal change entropy calculations.

7.1.3 Behavioral Characteristics

After calculating the constraints, the playthrough is analyzed to extract the different types of triggered mechanics during the playthrough. This extracted information is used to create binary dimensions for the constrained map-elites algorithm. Table 7.1 shows the different mechanics extracted from the playthrough. These mechanics were selected in order to differentiate between different types of jumps and kills in Super Mario Bros. For example, fall kill was added to help differentiate between scenes that require overcoming enemies through action and scenes where enemies will fall out of the scene if the player has waited or even played imperfectly (killing through inaction). Furthermore, the different jump types were added to differentiate between scenes that required (1) running and jumping, or (2) holding the jump button for a bit longer to high jump, or (3) just a small tap.

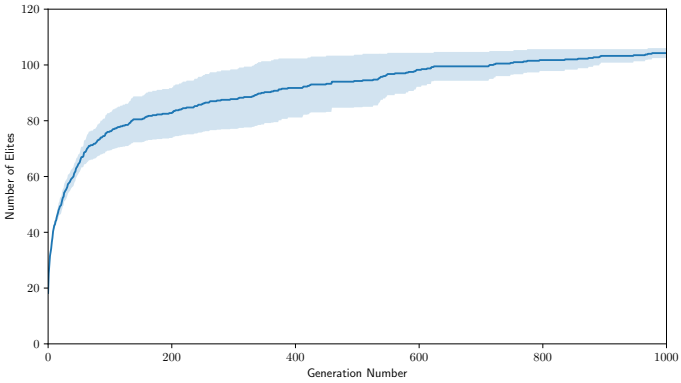


Figure 7.3: The average number of elites and the standard error in the Mechanics Dimensions approach over the 5 runs.

(a) No triggered mechancis (b) 50% of triggered mechanics (c) All triggered mechanics

Figure 7.4: Three generated scenes with various degrees of number of triggered mechanics.

7.2 Super Mario Bros Results

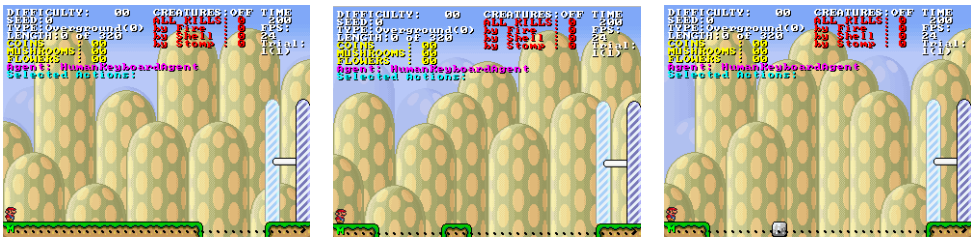
We ran our Constrained Map-Elites for 1000 iterations, where each iteration consists of 100 new chromosomes. We ran the experiment for 5 times. In the Constrained Map-Elites, the infeasible population size was fixed to twenty chromosomes, and the feasible population size was fixed to one chromosome. These values were selected based on our currently available resources and also to make sure that the generation algorithm will finish in under than 12 hours using 25 CPU cores. We used a two-point crossover that swaps any number of slices, ranging from a single slice to the full scene. The mutation replaces a single slice with a random slice sampled from slices in the original *Super Mario Bros*.

Figure 7.3 shows the average number of elites found during 1000 iterations. These numbers always increase, meaning that the algorithm can find more scenes as iterations pass. At the same time, they only populate around 40% of all the cells. The reason for this low coverage is some mechanics can't happen by itself. For example: you can't kill an enemy by stomping if you didn't jump.

The constrained map-elites algorithm was able to find a huge set of different combinations of mechanics that range from having all mechanics being triggered to none of them.

(a) Enemy stomping (b) Enemy stomping and fall kill (c) Enemy stomping, fall kill, and shell kill together

Figure 7.5: Three generated scenes with different ways to kill the enemies.



(a) Normal Jump. (b) Long Jump (c) Long High Jump

Figure 7.6: Three generated scenes with different ways to jump.

Figure 7.4 shows several generated scenes with various triggered mechanics. Figure 7.4a doesn't need any mechanic to trigger, as the floor is flat without game objects (i.e. enemies, coins, etc). Figure 7.4b shows a scene that triggers four different mechanics (jump, high jump, stomp, and coin) and figure 7.4c shows a scene that triggers all eight different mechanics. It has two enemies to guarantee shell kill, a question mark block for the mushroom, etc. However, one may notice that these scenes do not guarantee a mechanic will be triggered. For example, Figure 7.4c could be passed from the top of the blocks without interacting with enemies at all.

The final map (for the constrained map-elites) also contains multiple mechanics centered around how enemies can die. Figure 7.5 shows three scenes with different types of enemy killing. For example, in a fall kill scene, the generated scene contains goombas, as they can fall off the edge (Figure 7.5a). In a stomp kill scene, the generated scene contains a red Koopa Troopa since they cannot fall off the edge of the screen (Figure 7.5b). Lastly, in scenes that target every type of kill, the generated scene contains at least two enemies, where one of them can fall of the edge (green Koopa Troopa) (Figure 7.5c).

Finally, the final map (for the constrained map-elites) contains a variety of jump mechanics. Figure 7.6 shows three scenes targeting different ways to jump. The first scene targets the normal jump, which does not require holding the jump button for long nor running. Resulting scenes contain a small enough gap to cross (Figure 7.6a). The long jump, on the other hand, requires traversing long distances on the x-axis without the need of reaching a

higher point on the y-axis (thus not requiring to hold the jump button for long). The scene in Figure 7.6b ensures that the second jump is long enough for that x-axis traversal. Lastly, the long high jump requires a longer traversing distance, i.e. pressing the jump button longer (Figure 7.6c) with a larger second gap.

7.3 Super Mario Bros Discussion

Using Constrained Map-Elite does not guarantee that the player must trigger a specific mechanic to beat the scene, but it does guarantee that the mechanic *could* be used if the player desired. It was able to generate scenes with all various degrees of triggered mechanics. Furthermore, this method provides us with a set of scenes that contain the required targeted mechanic(s) which saves computation time. To modify this technique to guarantee that it generates scenes in which a targeted mechanic must occur, the playing agent could theoretically be swapped for an agent that is able to find *all possible solutions* in a scene. Then the target mechanic is recorded as triggered only if it is triggered in all possible solutions. This would however be impractical even for a game of Super Mario Bros' complexity.

Our simplicity fitness function was able to generate clean looking levels that might look appealing for humans. One might wonder if using this function might decrease the diversity between several runs. All the generated scenes from the 5 runs are almost different from each other. This difference in the scene's layouts is directly proportional to the number of triggered mechanics. For example, the no mechanics generated scene is the same between all 5 runs, while scenes with all mechanics being triggered are the most different. On the other hand, most of the scenes are flat with no much height difference, this is due to the simplicity of fitness and the selected mechanics during evolution. The simplicity forced the generated levels to have a small number of tiles, while the behavioral characteristics treated structurally different levels the same. For example: levels that have jumping over a gap is the same as levels with different height platform as both only trigger jump mechanics. This leads the algorithm to favor levels with a gap more than levels with different heights as the first uses fewer tiles compared to the second and at the same time both have the same behavioral characteristics.

7.4 General Video Game Framework

In the second half of this chapter, we apply the same ideas on the General Video Game Framework. The main reason is to showcase the ability of our method to work on more

general domains. To preserve time, we didn't run on all the games from GVGAI instead we picked 4 different games. The 4 GVGAI games used are Zelda, Solarfox, Plants, and RealPortals. Each was selected based on a previous work [17], which categorized GVGAI games based on how they were played. These four games contain a diverse array of mechanics, terminal conditions (time-based (Plants), lock-and-key (Zelda and RealPortals), and collection (SolarFox)), and incorporate ranging levels of complexity. For example, whereas Zelda is a relatively simple lock-and-key game, RealPortals requires complex problem-solving, and Plants contains relatively enormous maps to search. Thus, we selected these as a representative set of the GVGAI framework's games.

7.4.1 Scene Representation

The GVGAI levels are usually small on size which can act directly as scenes. Each scene is represented as a 2D array of tiles. The different tiles are extracted from the level mapping section in the VGDL game description. We decided to go with that representation because it can easily represent all games in the GVGAI and doesn't require any pre-processing or input data such as the input levels from the Super Mario Bros (Nintendo, 1985).

7.4.2 Scene Evaluation

Similar to Mario, each chromosome is being evaluated for playability using the constraints and for simplicity using the fitness function. Because the domain was different we adapted these functions to fit the new domain. For the constraints, we designed the constraints in the same vein to the constraints introduced in Chapter 5 which are very general and don't have any game-specific knowledge and can be applied to multiple different games.

The constraints consist of two main parts: survival/winning time constraint and the death constraint. For the first constraint, our system uses the OLETS algorithm that comes with the GVGAI framework [105]. If the agent successfully completed the level, the constraint value reflects how close the winning time to a certain ideal time. The closer to the ideal time, the better the constraint value. This is to ensure that the level isn't too easy to be finished very fast or too hard that it takes forever. In the other case, if the agent loses we want the agent to also survive for the longest amount of time (up to a certain point as we don't want levels where the agent is isolated from all the sprites that could lead to losing the game) before dying because the longer it survives the higher chance it can beat the level. Equation 7.3 shows the first two parts of the winning/survival constraint.

$$P = \frac{win}{|T_{win} - T_{ideal}| + 1} + \frac{(1 - win) * 0.25}{|T_{survival} - T_{ideal}| + 1} \tag{7.3}$$

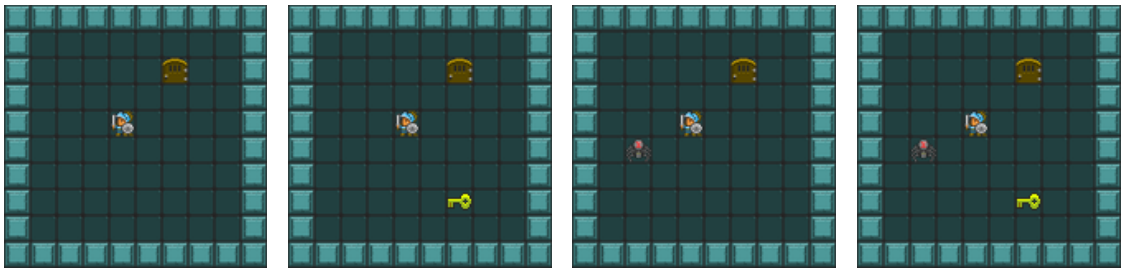
where win represents a 1 if the agent finished the level successfully and a 0 otherwise, T_{ideal} represents the ideal time pre-defined for the system, and T_{win} and $T_{survival}$ represent the finishing time of the agent before successful completion and unsuccessful completion respectively. We penalize the losing part by a factor of 0.25 to discourage losing.

The second constraint calculates the accessibility of the generated level for human users by making sure the player avatar doesn't die in the first few frames of the game before the human can react. To calculate this value, a DoNothing agent is run on the level for a certain number of times. This agent does not perform any user actions and remains idle in the level. If this agent dies before reaching T_{ideal}, the level fails the evaluation. If the agent does not survive for a majority of the evaluations tested, the ratio of successful idle agents over the number of times attempted is applied to the constraints. Equation 7.4 shows the second constraint where the constraint is satisfied if the DoNothing agent passes at least 50% of all the trails.

$$E = \begin{cases} 1 & \text{if } \frac{N_{pass}}{N_{total}} \geq 0.5 \\ \frac{N_{pass}}{N_{total}} & \text{otherwise} \end{cases} \tag{7.4}$$

where N_{pass} represents the number of times the idle agent survived to the ideal time, and N_{total} representing the total number of trials of the DoNothing agent. The DoNothing agent runs a total of 5 times on the level, of which it must survive 3 in order to pass the constraint test. Both agent's "ideal times" (T_{ideal}) were set to 70 timesteps for all experiments. In this domain, we decided to make the constraints loose by passing them if they passed a certain threshold because evolving GVGAI scenes are a harder problem than evolving Super Mario Bros scenes as we don't have any human knowledge injected in the form of representation.

If both constraints are passed, the level's fitness is evaluated according to the same equation used for Mario (Equation 7.2), where w is 0.25 in the Zelda and RealPortals experiments. Plants and Solarfox levels are more dependent on tile uniformity and open areas, as opposed to Zelda and RealPortals. Therefore, w was 0.2 for these experiments. In this domain, we are using a different representation so the calculation of the entropy derivative is also different. Instead of calculating the difference across the horizontal direction, we are comparing each tile to its all neighboring tiles in a 3x3 square. This will lead to simplistic scenes that tend to be more aesthetically pleasing and enjoyable than ones that are noisy and chaotic.



(a) No 'get key' | No 'kill enemy'

(b) 'get key' | No 'kill enemy'

(c) No 'get key' | 'kill enemy'

(d) 'get key' | 'kill enemy'

Figure 7.7: An example of an optimum MAP-Elites matrix for the Zelda game mechanics "get key" and "attack enemy".

7.4.3 Behavior Characteristics

The behavior characteristic of the CME cells consists of multiple binary dimensions that correspond to the game mechanics read from their VGDL description file. Each dimension represents whether or not a particular mechanic was performed by the agent (similar to the Super Mario experiment). For example, if the game mechanics for Zelda consisted of the list "get key" and "kill enemy", the CME behavioral characteristic will be 2 binary dimensions which will create 4 cells (No "get key"|No "kill enemy", No "get key"|"kill enemy", "get key"|No "kill enemy", and "get key"|"kill enemy"). Figure 7.7 shows some ideal levels for all these four possible cells. The behavior characteristic used in the system are shown in Tables 7.2, 7.3, 7.4, and 7.5. All of the games have a small amount of mechanics (around 8) except for RealPortals (has 35 mechanics). This makes the map size of RealPortals extremely big (2^{35}) compared to Super Mario Bros and the other games map (2^{8}). This huge size map will affect the generated scenes which we are going to discuss in the next section. Each scene is placed in its correct cell in the map based on its behavior characteristics. These characteristics are recorded during the constraint calculation using the triggered mechanics in the OLETS playthrough.

7.5 General Video Game Framework Results

As stated before, we tested our algorithm on 4 different GVGAI games (Zelda, Solarfox, Plants, and RealPortals). The initial chromosomes are generated randomly using GVG-AI's random level generator class [80]. Similar to Mario, a single MAP-Elite cell is allowed to store a maximum of 20 infeasible levels and 1 elite level. Each experiment ran for a total of 500 iterations where each iteration consists of 50 chromosomes. At each iteration, 10 scenes are randomly generated, while the rest 40 chromosomes are generated using the mutation

*	Dimension	Description
z1	space-nokey	Agent pressed the space bar when the avatar did not have a key
z2	space-withkey	Agent pressed the space bar when the avatar had a key
z3	stepback	A sprite ran into another sprite
z4	kill-nokey	A sprite killed the avatar when the avatar did not have a key
z5	kill-withkey	A sprite killed the avatar when the avatar had a key
z6	sword-kill	The agent killed an enemy sprite with a sword
z7	getkey	The agent picked up a key
z8	touchgoal	The agent touched a goal with the key and won the game

Table 7.2: Constrained MAP-Elites dimensions for the GVG-AI game Zelda

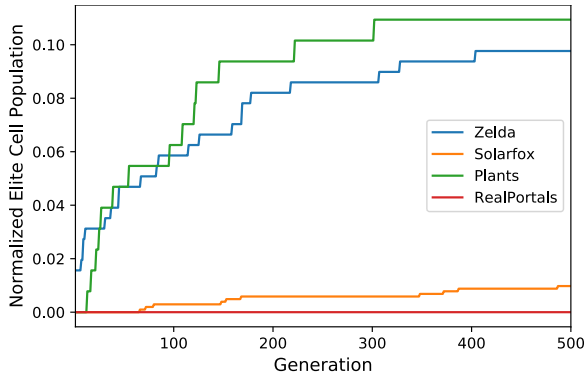


Figure 7.8: Number of Elite MAP Cells (Normalized) across generations. Although RealPortals is considerably lower than the other games, it's map contained 231 elites, nearly ten times more than any other game.

operator. This is to help to prevent the algorithm from potentially reaching a local optimum. For the Solarfox experiment, levels are much more dependent on having fewer empty tiles for functional gameplay and thus only 5 of Solarfox levels were randomly initialized each iteration. We only use mutation during evolution. Our mutation operator selects a random tile of the level and turns into a random new tile value. Then, based on some set probability, another tile from the level is randomly chosen and mutated. This process continues until the probability check fails.

Figure 7.8 shows the normalized elite counts across generations for all our experiments. Each experiment was normalized against its total possible elite cell count which is 2^n where n is the total number of mechanics of the corresponding game.

*	Dimension	Description
s1	hit-wall	Agent hit a wall
s2	hit-enemyground	Agent touched enemy ground
s3	hit-avatar	An enemy sprite hit the Agent
s4	touch-powerblib	The agent touched a powerblib
s5	spawn-more	A turning powerblib created a blib
s6	change-blib	A turning powerblib changed into a normal blib
s7	overlap-blib	A powerblib overlapped with a blib
s8	get-blib	The agent got a blib
s9	reverse-direction	A sprite hit a wall and reversed direction
s10	enemy-shoot	An enemy fired a missile at the avatar

Table 7.3: Constrained MAP-Elites dimensions for the GVG-AI game Solarfox

*	Dimension	Description
p1	space	Agent pressed the SpaceBar
p2	hit-wall	A sprite touched a wall
p3	kill-plant	A zombie destroyed a plant
p4	zombie-goal	A zombie sprite reaches the goal
p5	pea-hit	A pea sprite hits a zombie sprite
p6	tomb-block	A tomb sprite blocks a pea sprite
p7	make-plant	Agent placed a plant on a marsh tile

Table 7.4: Constrained MAP-Elites dimensions for the GVG-AI game Plants

7.5.1 Mechanical Frequency in Elites

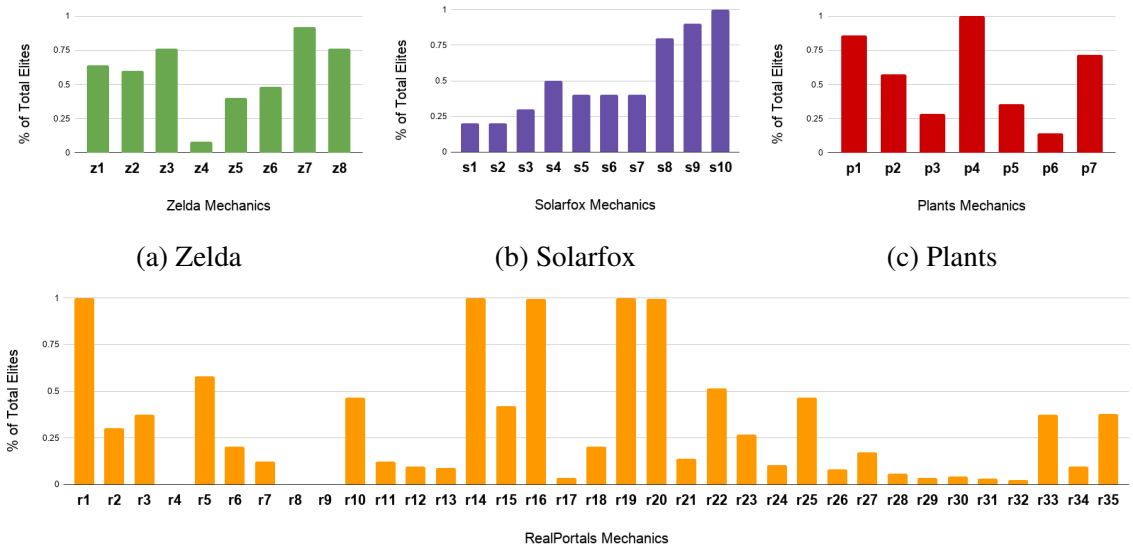
Figure 7.9 displays the symbolically represented mechanics present across all games and how prevalent each exists among that game's elite population. There are 3 mechanics (r4, r8, and r9) in RealPortals that are never expressed within any of the elites. These correspond to the "drown," "teleport-exit," and "no-moving-boulder" mechanics. The irony of the low activation of teleport mechanics ("teleport-entrance" (r13) $= 8\%$, "teleport-exit" (r8) $= 0\%$) in a game called "RealPortals" is not lost on us, and this is further represented when looking at the elites themselves, which do not require teleportation to win. However, no agent has ever been submitted to the GVGAI competition can reliably beat RealPortals levels. The system's constraint function, which drives evolution to produce beatable levels, causes the generator to develop levels simple enough for the agent to win. Because teleportation drastically expands the space that the agent needs to search, the simplest solution for the generator is to remove the need to teleport.

*	Dimension	Description
r1	space	Agent pressed the SpaceBar
r2	change-key-blue	changes blue avatar's current resource to a key
r3	hit-wall	A sprite touched a wall
r4	drown	destroy any sprite that falls in the water
r5	toggle-blue	avatar changes current portal shot to blue
r6	no-lock	any sprite tried to move through a lock
r7	no-portalexit	any sprite tried to move through an exit portal
r8	teleport-exit	orange avatar steps through the entrance portal
r9	no-moving-boulder	sprite tried to move through a moving boulder
r10	no-idle-boulder	sprite tried to move through an idle boulder
r11	change-key-orange	changes orange avatar's current resource to a key
r12	toggle-orange	avatar changes current portal shot to orange
r13	teleport-entrance	blue avatar steps through exit portal
r14	get-weapon	avatar picks up a weapon
r15	get-key	avatar picks up a key
r16	back-to-wall	portal turns back into a wall
r17	fill-water	moving boulder falls into water to fill it
r18	open-lock	avatar unlocks a lock
r19	touchgoal	The agent touched a goal and won the game
r20	make-portal	wall turns into a portal
r21	portal-missile-velocity	send a missile through a portal at the same velocity
r22	cover-goal	goal is covered by a missile
r23	blue-missile-in	send a blue missile thru a portal entrance
r24	orange-missile-in	send an orange missile thru a portal entrance
r25	portal-boulder	send a boulder through a portal at the same velocity
r26	stop-boulder-key	moving boulder stops after hitting a key
r27	stop-boulder-wall	moving boulder stops after hitting a wall
r28	stop-boulder-blue-toggle	moving boulder stops after hitting a blue portal toggle
r29	stop-boulder-orange-toggle	moving boulder stops after hitting an orange portal toggle
r30	stop-boulder-lock	moving boulder stops after hitting a lock
r31	teleport-boulder	sends a boulder to the other portal
r32	stop-boulder-boulder	moving boulder stops after hitting another boulder
r33	stop-boulder-avatar-blue	moving boulder stops after hitting the blue avatar
r34	stop-boulder-avatar-orange	moving boulder stops after hitting the orange avatar
r35	roll-boulder	boulder moved over a tile

Table 7.5: Constrained MAP-Elites dimensions for the GVG-AI game RealPortals

7.5.2 In-Depth Game Analysis

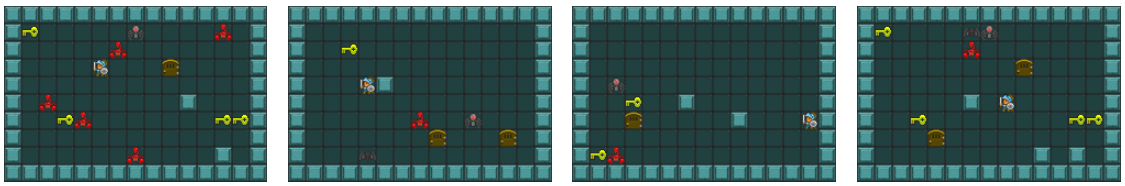
In the following subsections, we present a representative subset of each game's generated levels. The *mean* and *mode* levels correspond to the mean and mode number of mechanics triggered across all elites. When multiple elites contained the identical amount of mechanics for either mean or mode, we randomly sampled among these elites to display one of them. We realize that this is only a subset of the possible elites, and that dimensional similarity does not necessarily equate to structural similarity.



(a) Zelda (b) Solarfox (c) Plants

(d) RealPortals

Figure 7.9: The percentage of elites that contain a specific mechanic for each game. The lettering of a mechanic corresponds to that games mechanic table. Zelda: Table 7.2; Solarfox: Table 7.3; Plants: Table 7.4; RealPortals: Table 7.5.



(a) mean: z1, z2, z3, z5, z7 (b) mode: z1, z2, z3, z7, z8 (c) least: z7, z8 (d) most: z1, z2, z3, z5, z6, z7, z8

Figure 7.10: A subset of generated elite levels for Zelda. Their string representation corresponds to their showcased mechanics detailed in Table 7.2.

7.5.2.1 Zelda

After 500 iterations, 55 out of 256 possible cells were populated for the Constrained MAP-Elites matrix of Zelda, with 25 cells containing an elite map. The average fitness for these cells was 0.5186. Figure 7.10 displays 4 elites at opposite ends of the dimensional spectrum. The mean and mode elites are calculated to have 4.64 and 5 dimensions. respectively. The map with the least dimensionality (Figure 7.10c) showcased 2 mechanics, out of a possible 8. The map with the most mechanic-dimensionality 7.10d contained 7.

The representative least dimensional elite has multiple game sprites that don't trigger their corresponding mechanics, such as killing monsters and bumping into walls. Most likely

due to the wide-openness of the space, the agent failed to bump into any of these objects on its bee-line route to the key and the door. On the other hand, the most dimensional map showcases nearly all possible mechanics, only missing kill-nokey. We can interpret this to mean the agent went for the key first, before dealing with monsters. At a glance, it would be possible to also trigger the kill-nokey mechanic, depending on agent priority.

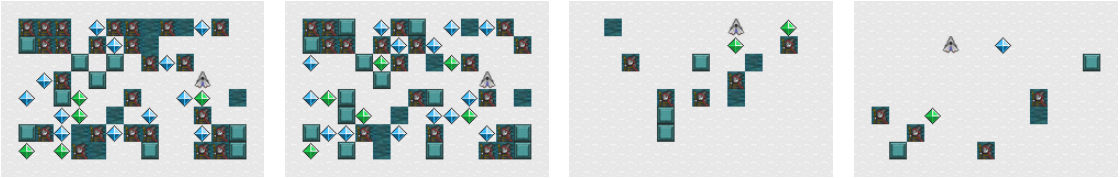
7.5.2.2 Solarfox

After 500 iterations, 52 out of 1024 possible cells were populated for the Constrained MAP-Elites matrix of Solarfox game, with 10 cells containing an elite map. The average fitness across all cells was 0.4311. Figure 7.11 displays 4 elites at opposite ends of the dimensional spectrum. The mean and mode are calculated to be 5.1 and 3 respectively. The least mechanic-dimensional map contained only 3 mechanics (Figure 7.11c), whereas the most (Figure 7.11d) contained 8 out of the possible 10 mechanics.

The representative least dimensional elite (Figure 7.11c presents a lightly populated level containing just a few enemies, blibs, and walls. It contains no powerblib-generators, only normal blibs, which are placed incredibly close to the player at the start for an easy win. The most dimensional elite contains nearly every mechanic in the game except for 2: the agent hitting a wall (which are placed far from the player's initial start), and the agent touching and dying on the enemy ground tile.

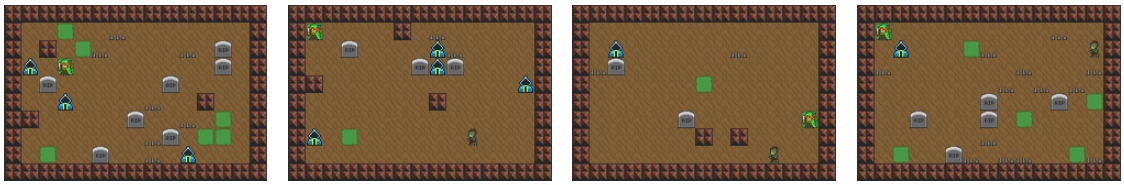
7.5.2.3 Plants

After 500 iterations, 31 out of 128 possible cells were filled for the Constrained MAP-Elites matrix of Plants, with 14 cells containing an elite map. The average fitness across all cells was 0.3993. Figure 7.12 displays 4 elites of varying dimensions. The mean and mode are calculated to be 3.93 and 5 respectively. The map with the least dimensionality showcased just 1 mechanic. The map with the most mechanic-dimensionality contained 6 out of the 7



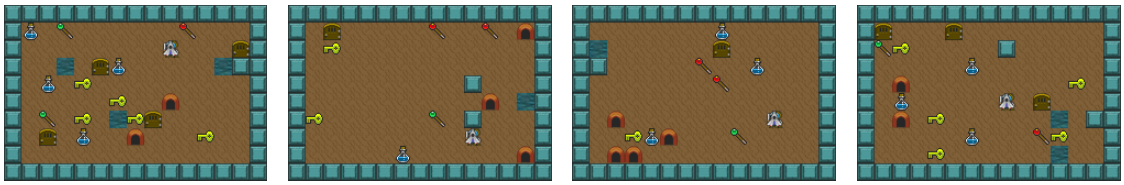
(a) mean: s2, s4, s8, s9, 10 (b) mode: s3, s9, s10 (c) least: s8, s9, s10 (d) most: s3, s4, s5, s6, s7, s8, s9, s10

Figure 7.11: A subset of generated elite levels for Solarfox. The string representation corresponds to their showcased mechanics detailed in Table 7.3.



(a) mean: p1, p3, p4, p5, p7 (b) mode: p5, p6, p7 (c) least: p4 (d) most: p1, p2, p3, p4, p5, p7

Figure 7.12: A subset of generated elite levels for Plants. Their string representation corresponds to their showcased mechanics detailed in Table 7.4.



(a) mean: r1, r3, r5, r11, r14, 15, r16, r19, r20, r21, r24 (b) mode: r1, r3, r5, r13, r15, r17, r18, r19, r20, r21, r22 (c) least: r1, r3, r14, r19, r22 (d) most: r1, r5, r7, r10, r11, r13, r14, r15, r16, r19, r20, r23, r25, r26, r34, r35

Figure 7.13: A subset of generated elite levels for RealPortals. Their string representation corresponds to their showcased mechanics detailed in Table 7.5.

possible mechanics.

Unlike any of the other elites of any other game, the representative least dimensional elite of Plants contains a single mechanic, which happens to be one that causes the player to lose the game. Based on the game rules, it is not possible to win this level no matter what actions the player does, as the zombies will spawn several tiles to the right of the villager and inevitably collide with it. We think that the zombie comes out really late which gives high survival time which causes the scene to pass the minimum threshold even if it was unplayable. The elite with the most activated dimensions, on the other hand, was possible to win. The triggered mechanics guarantee that a player could encounter most of the mechanics in the game during play.

7.5.2.4 RealPortals

After 500 iterations, 6966 cells were filled for the Constrained MAP-Elites matrix of RealPortals, with 231 cells containing an elite map. The average fitness across all cells was 0.4257. Figure 7.13 displays 4 elites of varying dimensions. The mean and mode are calculated to be 10.78 and 10 respectively. The least mechanic-dimensional map contained

5 mechanics, and the most contained 16 out of a possible 35.

In contrast to the other games described above, RealPortals is extremely complex, echoed in the sheer amount of elites. Ironically, the generated levels are all extremely simple to solve, unlike any of the GVGAI included levels. Without water dividing the map and requiring the player to use portals, the levels are transformed into a simple find-the-goal game even simpler than Zelda. Even if the agent uses a portal, there is no need to do so or to use any of the other game mechanics which are normally required by the framework game levels (pushing boulders into water, unlocking the lock with a key, etc). The least dimensional representative elite still activates five mechanics (with others still possible, just not activated during playthrough), whereas the most dimensional elite can be beaten by taking two steps to the right. All these are due to having a nonperfect agent that takes a long time to solve these easy levels and can't solve more complex levels that require using portals to pass.

7.6 General Video Game Framework Discussion

Constrained MAP-Elites was able to populate around 10% of the total cells with elites for Plants and Zelda. Compared to Solarfox (approx. 1%) and RealPortals (>1%), these two games' dimensions were relatively well-explored. At first glance, it would make sense that RealPortals was not as explored, due to its 34-dimensional complexity compared to Zelda's meager 8-dimensions. However, Solarfox (10) is also many less dimensional than RealPortals, but has a similarly low relative elite population. Keep in mind that though the dimensionality of Solarfox relative to Zelda is only 2 higher, the dimensional space increases from 256 possible cells to 1024. We also hypothesize that the elite population is impacted not only by the total number of game mechanics but also by the intricacies of how the mechanics interact with each other. In Solarfox, the player wins by collecting all the "blibs"; however, not all blibs are equally scored. If the player is patient, they can let certain "powerblibs" spawn more blibs, and therefore gain more points. We theorize that it is difficult to create levels that force OLETS agent to wait to collect these blibs, and therefore trigger these spawning mechanics in conjunction with others.

Across all games, when compared to the original levels, the generated levels provide a sense of uniformity. Solarfox levels tend to be sparsely populated with blibs, with a few exceptions (Figure 7.11b). There tend to be no water tiles present in RealPortals, or marsh tiles in Plants, relative to each games' original levels. The Zelda elites consist of wide-open spaces, instead of the usual maze-like patterns. We hypothesize that all of this is due to the entropy pressure from the fitness calculation specified in Equation 7.2, which drives

evolution towards creating simplistic levels. Also, having a nonperfect agent might not cause complex levels to be passed quickly causing it to score badly on constraints and might not pass. Using a better agent that can play these games will help us to find more complex levels (like using portals in RealPortals).

It is interesting that none of these levels incorporate the patterns that the original GVGAI levels contained. For example, most original Solarfox levels presented geometric arrangements of blibs and powerblibs, which were aesthetically pleasing levels to play. The original levels also contained enemies firing missiles from the top and bottom part of the level, with the player and blibs sandwiched between, unlike the system-generated levels in which enemies and blibs can be anywhere on the map. The entropy pressure exerted by the fitness function pushed for uniformity during generation, which resulted in either highly populated levels filled with certain tiles (Figures 7.11a and 7.11b), or lowly populated levels with large amounts of empty space (Figures 7.11c and 7.11d).

7.7 Summary

In this chapter, we introduced a new generic behavior characteristic that can be used across different games. The behavior characteristic is based on the actual game mechanics/events triggered during a playthrough of the level. We also introduced a new fitness function that is based on entropy that helps making the level looks simpler and easier to read. We tested these new ideas on two domains: Super Mario Bros and General Video Game Framework. The results from Mario looked very human and covered almost all the different mechanics, while the generated scenes from General Video Game Framework was different across different games with some success and failure.

We assume in this chapter that we can have access to the main game events/mechanics as this is not the core of this work. It usually can be automatically detected and extracted using Machine learning techniques [61]. We also focused on generating small scenes so we won't have long empty sections similar to the ones appearing in the generated levels in Chapter 6. These small scenes introduced a new problem which is stitching them together to generate the full level which we will tackle in the next Chapter 8.

Chapter 8

Stitching For Mechanical Experience

Generating complete levels is a hard problem. The fitness function and behavior characteristics tend to be an average of the experience over the whole level. This problem appears in the generated bullet hell levels in chapter 6 where high-density levels might have big empty sections. In chapter 7, we decided to generate scenes (a small section of a level) instead of levels. This method allowed us to generate a set of scenes with the required experience without having to worry about the distribution of these experiences since the scenes are small enough.

In this chapter, we are facing the problem of generating a full level again but this time we are armed with the generated scenes as starting blocks. We present a FI2Pop algorithm that stitches these scenes together to generate a full level that has the same mechanic sequence as the target mechanic sequence. The target mechanic sequence can be acquired from multiple different sources:

- it can be extracted from a player/agent playthrough to produce different structural levels with similar mechanics.
- it can be generated based on the player playing style, for example: if the player likes jumping produces more levels with more jumps. similar to work by Smith et al [123] and Shaker et al [118].
- It can be generated using a grammar approach similar to Smith et al work [124].
- It can be hand designed by a designer to generate levels with a certain experience.

We can use any of these previous methods to acquire the target mechanic sequence but we decided to go with the first technique (extracted from an agent playthrough) as we were curious if we are able to generate levels with the same experience as iconic levels like

Name	Description	Frequency
Low Jump	Mario performs a small hop	25.9%
High Jump	Mario jumps very high	39.44%
Short Jump	Mario jumps and hardly moves forward	28.33%
Long Jump	Mario jumps and moves forward a large amount	19.75%
Stomp Kill	Mario kills an enemy by jumping on it	78.77%
Shell Kill	An enemy is killed by a koopa shell	37.85%
Fall Kill	An enemy falls off the game screen	50%
Mode	Mario changes his mode (small, big, and fire)	22.77%
Coin	Mario collects a coin	50.5%
Brick Block	Mario bumps into a brick block	41.1%
? Block	Mario bumps into a ? mark block	59.79%

Table 8.1: A list of the Mario game mechanics and the percentage of evolved scenes that contain them.

Super Mario Bros' Level 1-1. We used the generated scenes[1] from the previous chapter on Super Mario Bros domain with three additional mechanics (explained later in the chapter) to capture more of the main game mechanics such as destroying bricks and bumping coins.

8.1 Methods

Using a target mechanic playtrace, our system is able to generate levels that are mechanically analogous to the input. We use a set of pre-evolved "scenes" produced using the same method explained in the previous chapter, which will be stitched together to create a full level.

Every scene is labeled with the mechanics that an agent triggered while playing it. Table 8.1 shows all the game mechanics in the Mario AI Framework that scenes may be labeled with. The scene library used in this system is evolved using the Constrained MAP-Elites algorithm introduced in the previous chapter using the mechanics listed in Table 8.1. The table also shows the percentage of scenes from the corpus that contains these specific mechanics. Jump is the most common mechanic which is not surprising since Mario is a jumping platform game (jumping over gaps, jumping on enemies, jumping to headbutt blocks, etc). In the following subsections, we will explain each part of the FI2Pop algorithm [83] that our system uses to stitch pre-generated scenes into playable Mario levels that an agent plays by triggering a specific game mechanic sequence.

[1] https://github.com/LuvneeshM/MarioExperiments/tree/master/scenelibrary

8.1.1 Chromosome Representation

A level consists of a number of scenes stitched together. A chromosome is synonymous with its level representation, and each scene within this chromosome is not limited to only having one mechanic label. Using scenes that contain multiple mechanics enables the generator to generate levels that are more condensed.

8.1.2 Genetic Operators

The system uses mutation and crossover as operators. For the crossover, we use a two-point crossover to allow the system to increase and decrease level length as well as swap-out any number of scenes, from a single scene to the entire level. For the mutation, the generator uses five different mutation types:

- **Delete:** delete a scene.
- **Add:** add a random scene adjacent to a scene. The random scene is selected with probability inversely proportional to the number of mechanics using rank selection.
- **Split:** replace the current scene with two other scenes such that half the mechanics go in a new left scene and the rest to go in the right scene.
- **Merge:** add the mechanics of a scene to the left scene or to the right scene, then replace both scenes with one from the corpus that has the combined list of mechanics.
- **Change:** changes a scene with another random scene. The random scene is selected with probability inversely proportional to the number of mechanics using rank selection.

The system selects a scene to apply one of the operators, with a higher likelihood to select scenes that have a high number of mechanics.

8.1.3 Constraints and Fitness Calculation

FI-2Pop uses both a feasible and an infeasible population. The infeasible population tries to make the chromosomes satisfy constraints (like making sure the level is playable), while the feasible population tries to make sure that the mechanics in the new levels are similar to the input mechanic sequence.

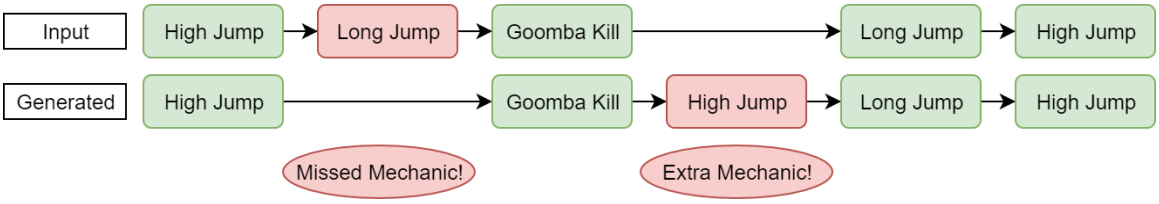


Figure 8.1: An example of missing and extra mechanic faults. The Long jump in the input playthrough is missing in the newly generated map's playthrough. The generated playthrough also contains an extra high jump not included in the input.

To calculate the constraints, an agent plays each chromosome N times. The system calculates the constraint value using the following equation:

$$C = \begin{cases} \frac{1}{N}\sum_{i=1}^{N}\frac{d_i}{d_{level}} & \text{if } \frac{1}{N}\sum_{i=1}^{N} w_i < p \\ 1 & \text{if } \frac{1}{N}\sum_{i=1}^{N} w_i \geq p \end{cases} \tag{8.1}$$

where d_i is the distance traveled by the A* agent on the level on the i^{th} iteration, d_{level} is the maximum length of the level, w_i is equal to 1 if the agent reached the end of the level on the i^{th} run, and p is the threshold percentage.

To calculate the fitness, the system uses the playtrace in which the agent not only won the level but also triggered the fewest mechanics. The fitness function compares this playtrace with the input playtrace and optimizes toward increasing the similarity between them. The level is assigned an initial score S, which is decremented based on the number of "faults". A fault is a mechanic sequence mismatch between the input sequence and the newly generated agent playtrace, defined either as an extra mechanic placed between a correct subsequence of mechanics, or else as a missing mechanic that would create an otherwise correct subsequence. Figure 8.1 displays an example of both. The system uses a sequence matching algorithm to calculate fault counts. The algorithm loops over the target sequence and searches for the first occurrence of each target mechanic in a sub-array of the generated mechanics list, from the beginning to the end of the list. If the mechanic is not found, it increments a counter tracking the number of missed mechanics. When the mechanic is found, the index of the mechanic is the number of extra mechanics between it and the previous matching mechanic. The pointer of the generated mechanic sequence is moved to point at this new position to continue the loop.

Based on how faults are calculated, it is possible for a level to have a negative fitness

score. The fitness function is calculated based on the following equation:

$$\begin{aligned} P_{missed} &= W_{missed} \cdot M_{missed} \\ P_{extra} &= W_{extra} \cdot \tanh(b \cdot M_{extra}) \\ F &= S - (P_{missed} + P_{extra} \cdot (S - P_{missed})) \end{aligned} \tag{8.2}$$

where P_{missed} is the penalty of missed mechanics, P_{extra} is the penalty of extra mechanics, M_{missed} is the number of missed mechanics, M_{extra} is the number of extra mechanics, S is the initial starting value, and b, W_{missed}, W_{extra} are predefined weights.

8.2 Experiments

To test the algorithms, we generate levels using three original Super Mario levels playtraces as targets (1-1, 4-2, 6-1 in Figures 8.2a, 8.2d, and 8.2g). The Robin Baumgarten A* algorithm, which was developed for the first Mario AI competition [135], is run once on each of the three levels. The resulting mechanic sequences are collected and used as targets.

Our system uses a population of 250 chromosomes each generation, with 70% crossover and 20% mutation rates, and 1 elite. Chromosomes are initialized using a random scene picker, which selects anywhere between 5 to 25 scenes with which to populate the level. Each scene is randomly selected from the corpus-based on the assigned number of mechanics to it. The number of mechanics for each scene is sampled from a Gaussian distribution with a mean as the average number of mechanics in the target playtrace and a standard deviation of 1. For the constraints calculation, we used p equal to 1, while for the fitness calculation, we used W_{missed} equal to 5.0, W_{extra} equals to 1.0, b equals to 0.065, and S equals to 100. These values were picked based on some preliminary experiments that prove that they lead to better performance.

The scene corpus is taken from the results of *Intentional Computational Level Design* [77]. The corpus [2] contains a total of 1691 Mario scenes, with an average of 5.45 mechanics in a scene and a standard deviation of 1.74. The library is generated using the Constrained MAP-Elites generator created in the previous chapter, with the dimensions corresponding to the mechanics shown in Table 8.1. The gameplaying agent in the MAP-Elites generator is the same agent we use: the winning A* agent from the 2013 Competition [136].

We compare the results from our system against two baselines. The random baseline generates levels with 5 to 25 scenes which are picked randomly from the corpus. The greedy baseline generates levels with 5 to 25 scenes, selected such that the resulting level

[2] https://github.com/LuvneeshM/MarioExperiments/tree/master/scenelibrary

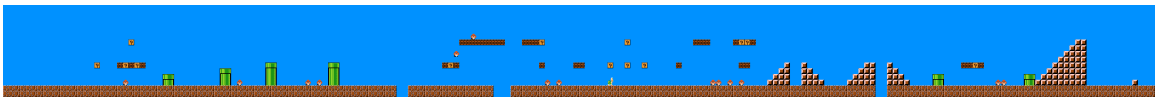
Mechanic	Level 1-1	Level 4-2	Level 6-1
Low Jump	14	20	18
High Jump	4	9	4
Short Jump	6	16	14
Long Jump	11	12	7
Stomp Kill	1	2	0
Shell Kill	0	0	0
Fall Kill	0	0	0
Mode	0	0	0
Coin	1	6	1
Brick Block	0	0	0
? Block	2	2	0
Total	39	67	44

Table 8.2: The frequency of each mechanic in the input playtrace.

maximizes the number of matched mechanics based on each scene labeled mechanics in the corpus. Each generator generates 20 levels for each input game world. Table 8.2 displays information about the mechanic makeup of the input playtraces. We can notice that all the playtraces have a low frequency of killing enemies, hitting blocks, or collecting coins. This was not surprising as the used A* agent is designed to reach the furthest to the right in the least amount of time without caring about its score.

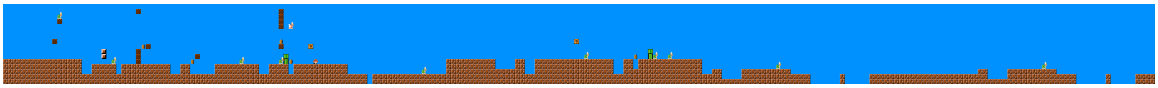
8.3 Results

Figure 8.2 shows the original levels from Super Mario Bros (Nintendo, 1985) and its greedy and evolved counterparts. Both greedy and evolved levels have less graphical variance, a result of the lack of diversity in the input scenes. This is due to the fitness function used to evolve these scenes [77] that most likely negatively impacts level diversity, as it aims to simply levels and create uniform spaces. For example: a scene containing a 3-tile-high-pipe requires the same jump that 3 breakable brick tiles stacked on top of each other does. The fitness function would prefer 3 breakable brick tiles stacked on top of each other as they are more uniform compared to a pipe. Also, most of the generated levels (greedy or evolution) seem flatter than their original counterparts for the same reason as the lack of graphical variance.

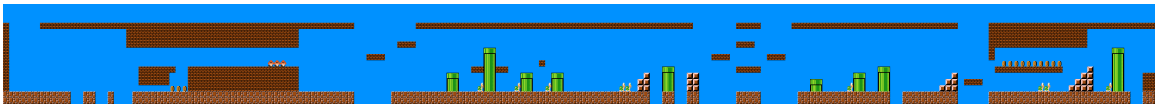


(a) Level 1-1 original

(b) Level 1-1 greedy



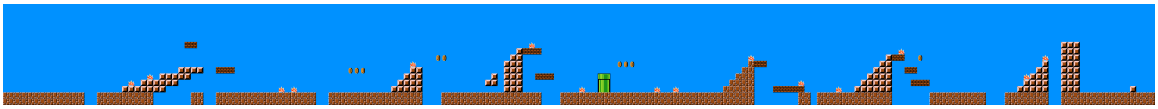
(c) Level 1-1 evolved



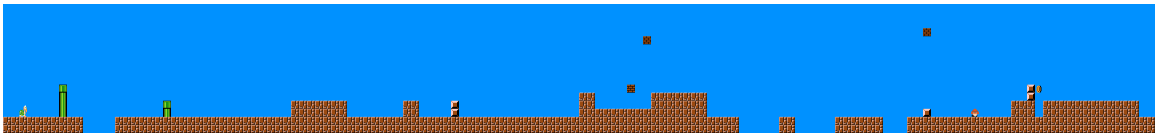
(d) Level 4-2 original

(e) Level 4-2 greedy

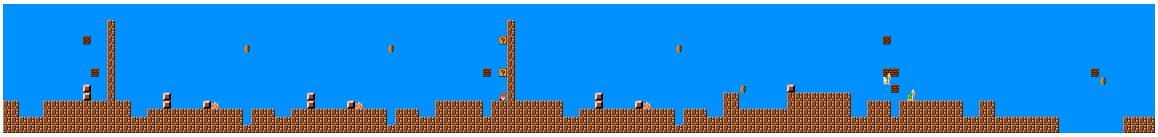
(f) Level 4-2 evolved



(g) Level 6-1 original



(h) Level 6-1 greedy



(i) Level 6-1 evolved

Figure 8.2: A random sampling of greedy-generated and system-evolved levels, compared to their original equivalents

Experiment	Playability	Inter-TPKLDiv	Intra-TPKLDiv
Original Levels	52%	0.715 ± 0.410	-
Random Levels 1-1	10.75%	0.697 ± 0.265	2.941 ± 1.005
Greedy World 1-1	28.5%	0.675 ± 0.228	2.636 ± 0.795
Evolution World 1-1	100%	0.269 ± 0.127	1.601 ± 0.573
Random Levels 4-2	10.75%	0.697 ± 0.265	2.941 ± 1.005
Greedy World 4-2	26.25%	0.648 ± 0.181	3.329 ± 0.647
Evolution World 4-2	99.5%	0.264 ± 0.094	1.997 ± 0.466
Random Levels 6-1	10.75%	0.697 ± 0.265	2.941 ± 1.005
Greedy World 6-1	25%	0.648 ± 0.172	2.601 ± 0.577
Evolution World 6-1	87.25%	0.348 ± 0.117	1.505 ± 0.404

Table 8.3: Different level statistics calculated over 20 generated levels using different techniques and compared to the original levels as a reference point.

8.3.1 Level Playability

To calculate playability for each group of levels, we run Robin Baumgarten's A* agent [135] 20 times per level and average the results over the whole group of levels. Table 8.3 shows the playability percentages of each of these groups. We find it notable that the A* can only win 52% of the original levels, which includes all the levels from the original Super Mario Bros except for the underwater levels and castle levels. This demonstrates that the A* is not a perfect algorithm and not able to beat every level every time. For random, greedy, and evolved levels, we calculated the results over 20 generated levels. The evolved 1-1, 4-2, and 6-1 levels all have close to 100% playability. In contrast, greedy stitching seems to make poor quality levels in terms of playability ($25 - 28.5\%$). Random stitching predictably creates barely playable levels (10%).

8.3.2 Mechanic Similarity

Figure 8.3 displays the mechanic evaluation across all three target levels for all generators. "Matches" is the number of matched mechanics from our matching algorithm divided by the total number of target mechanics (can be found in Table 8.1) for each level, while "Extras" is the total number of mechanics in the current level that didn't get matched using our matching algorithm divided by the total number of target mechanics (can be found in Table 8.1) for each level. The graph shows the average (the solid line) and standard deviation (as light shade) over 20 different runs for all the three techniques. As fitness is impacted by matching mechanics it makes sense that the number of matches will increases in the evolutionary generated levels, just as fitness does. Across all three levels, the evolution

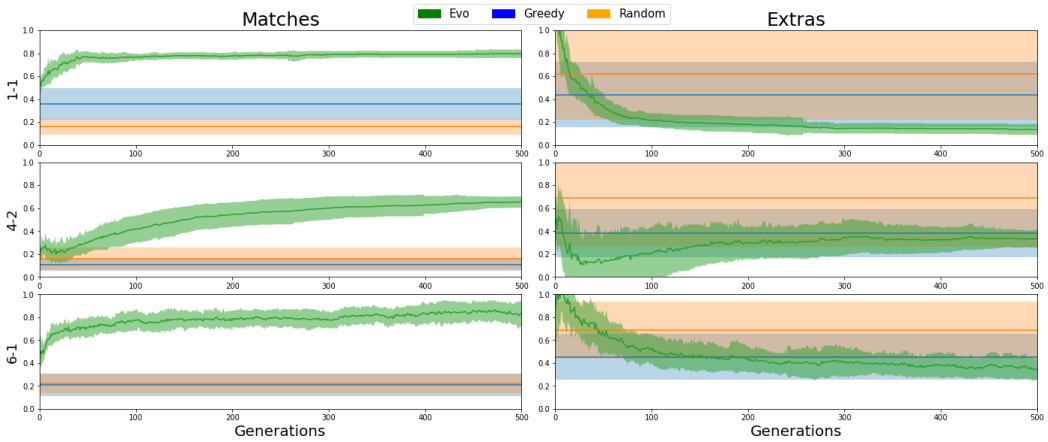


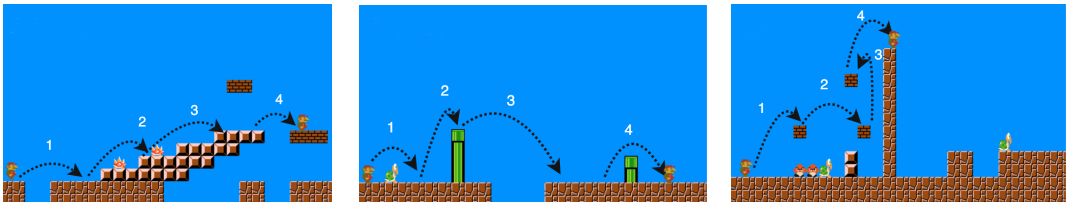
Figure 8.3: Tracking mechanic statistics throughout generations across all three target levels for all generators.

agent far outperforms both the greedy and random generators. This is also true for the extra mechanics (which are minimized) on 1-1 and 6-1. However, on 4-2 the greedy generator seems to add more extra mechanics after a brief drop ending around generation 50 which might be inevitable to increase the matched mechanics. This stabilizes around generation 200, which is also when the match mechanic count stabilizes.

8.3.3 Structural Diversity

We use Tile Pattern KL-Divergence (TPKLDiv) calculations [91] to measure the structural similarity between the 20 generated levels that we call Inter-TPKLDiv and between the generated levels and the corresponding original level which we call Intra-TPKLDiv. We use a 3x3 tile pattern window to calculate the TPKLDiv. For the Inter-TPKLDiv, this value reflects how different these 20 levels from each other. In order to calculate it, we calculate the TPKLDiv between each level and the remaining levels and take the average of the minimum 20 values such that all the levels are present. For the Intra-TPKlDiv, this value reflects how different the generated levels from the original level. We compute this value by calculating the TPKLDiv between each generated level and the original level, then we talk the average over all these values. The value of TPKLDiv reaches 0 when both levels are identical but increases the more the levels are different.

Table 8.3 shows both Inter-TPKLDiv and Intra-TPKLDiv for the 20 generated levels. From observing the values of the Inter-TPKLDiv, it is obvious that every technique has a lower diversity score than the original levels of Super Mario Bros. We noticed that the randomly generated levels have a lower TPKLDiv value, showing that the evolved levels have less diversity overall. Also, the levels generated for World 1-1 have low diversity



(a) Original Sequence (b) Greedy Sequence (c) Evolved Sequence

Figure 8.4: An example of a 4 jump sequence in the original 6-1 level, and how the two generators try to copy it.

relative to the others. In the context of it being the very first level of the game with the most basic and simple mechanics for new players, this diversity score makes sense. This makes it harder to find more diverse structural levels. It is surprising that World 4-2 has less diversity than World 1-1. The World 4-2 playtrace has so many fired mechanics, it might be more difficult to find a playable level with such a high amount even with 25 scenes, especially when scenes with small numbers of mechanics are selected more often. One last note, the greedy algorithm having higher diversity than all the evolution levels but at the same time, they have less playable levels compared to the evolved levels.

By observing the Intra-TPKLDiv values, the random generator's levels have the largest TPKLDiv except for in World 4-2 where we were surprised to find that greedy generator has a higher value. This might be due to the greedy generator creating longer levels in reaction to the longer length of the input mechanic sequence for that level. The evolved levels have nearly half the TPKLDiv value of the random generator levels except for World 4-2, also probably in response to that playtrace's mechanic count. To reference the findings presented in Table 8.3, 4-2 levels seem to contain only small Inter-TPKLDiv (all the 20 generated levels are structurally similar to each other) but show an increase in the Intra-TPKLDiv (all the 20 generated levels are structurally different from the original level).

8.4 Discussion

Looking into results shown by Figure 8.3, it is clear the FI2Pop generator outperformed the baselines in the defined terms of matching the mechanic sequence of the input. As we allow for the length of the levels to vary within a defined range, it is important to observe the convergence of said level lengths. A typical level from Super Mario is 14 scenes in length and the levels generated for level 1-1 and level 6-1 converge toward and hover around that length. However, the generator for level 4-2 converges to a length of roughly 23 scenes, nearly 1.64 times that of the other 2 observed levels. We hypothesis the reason behind this

influx in scene length is due to the sheer number of mechanics present in the original level 4-2. Level 4-2 has the most number of mechanics triggered, close to 1.7 times the amount of level 1-1 and to 1.5 times the amount for level 6-1. We believe the generator could not guarantee levels where the input mechanic sequence to occur in the given length and thus favored levels with more scenes. By spreading out the mechanics, it allows the generator to better guarantee generating levels with a higher likelihood of being aligned to the input from a mechanics sequence standpoint. This is evident as the number of matches increases as the overall length of the level increases for level 4-2. Since the overall length of level 4-2 increases, the likelihood of additional mechanics occurring between the wanted mechanics also increases, explaining the rise in extra mechanics for the generated levels for 4-2.

The evolutionary generator is influenced to ensure mechanics from the input sequence are forced to happen in their particular order in the generated levels. Evolution is driven by the matching pressure in the fitness function to guarantee the agent had no other choice but to perform certain mechanics before progressing forward. Figure 8.4 shows an example of this with a zoom into a subsection of 6-1 from the original level, greedy generated level, and evolved level. The original level requires the agent to perform a 4-jump sequence in order to progress forward in the level. A similar manner of triggered mechanics can be seen in both the greedy and evolved levels. The greedy generator simply places scenes next to each other in which jumps occur. However, it cannot guarantee or force the agent to perform the jumps outside of having a strong likelihood of the jumps occurring. If the agent was delayed to react it might stomp on the Koopa instead of jumping over it, leading to a different mechanic sequence. Looking to the evolved example, we see there is a long wall that acts as a hard gate, blocking the agent from progressing forward, without first performing the 4 jumps.

The corpus from Khalifa et al. [77] was built using a fitness function that encouraged simplicity, this is reflected on the levels built using the corpus as most of them look simple and flat. In Figure 8.4, the "original" two-scene section that requires 4 jumps to overcome translates to another two scene section with a Koopa turtle, a pipe, a gap, and a little pipe in the "greedy" level section. The evolutionary generator, driven by matching pressure, shrinks this down into a single scene to force the agent into having no other choice but this mechanic sequence. In a way, the evolution method is performing a type of minimalist level generation, creating the simplest levels which are mechanically analogous to the original.

Based on the results of Section 8.3.3 and Table 8.3, we hypothesize that a mechanical sequence populated with small amounts of relatively simple mechanics (Evolution World 1-1) has only a small range of mechanically similar cousins. It seems that having a large number of target mechanics also makes it difficult to curate diversity (World 4-2). It's possible that World 6-2 represents a sweet spot in terms of diversity. In future work, we'd

like to test this theory using a more diverse array of scenes and levels.

We think that it can be solved by using scenes with more diversity as the current evolved scenes don't care about visual diversity and treat jumping over a gap the same as jumping over a wall. Another solution can be done by using more detailed mechanics For example, instead of having Jump as a mechanic, we can Jump over a Goomba, Jump over a Pipe, Jump over a gap as mechanics instead. The problem of that technique is that having more detailed mechanics might cause the generator to collapse between different runs and end up generating the same level every time you run.

Looking back on the whole technique, we can see the similarity between it and texture synthesis techniques [47] and wave function collapse [60]. In these techniques, we try to create a bigger texture from small patterns by minimizing the error between the final texture and the local patterns. Similarly, we could have used any of these techniques to batch these small scenes in a bigger one. We believe that their performance won't differ from the greedy technique proposed in this paper. These techniques only care about local relations and won't be taking any precautions about emergent behaviors that could arise from combing two scenes beside each other. For example: combine an empty scene with a jump near the end, with a scene with a Goomba near the beginning that might cause a Fall Kill mechanic to be triggered.

8.5 Summary

By the end of this chapter, we have finished our pipeline of generating general game levels using a search-based approach. The pipeline starts by creating small level sections (scenes) which can be used to generate a big corpus of different scenes that are mechanically different from each other. In this chapter, we introduced the last piece of the framework where it stitches the generated scenes together to generate a full level. The stitching process tries to generate a playable level that follows a target mechanic sequence. The target sequence can be extracted from playthrough, playing style, generated via grammar, etc. We successfully generated levels that mimic an agent playthrough but at the same time, the levels felt very flat and not aesthetically pleasing. This can be fixed by either adding more detailed mechanics during scene generation or by using a fitness function that rewards visual diversity. This work can be further expanded to generate personalized levels for different player personas. This can be done by generating a playthrough from an agent that follows a certain persona similar to work by Holmgard et al [66].

Part 3

Searching for Generators

Chapter 9

Learning Level Generators through Reinforcements

This chapter and the next one are tackling the problem of general level generation from a different point of view. Instead of trying to search the space of game levels, but rather the space of policies that generate game levels. At each step, the policy is asked to take the action that leads to the highest expected final level quality. To the best of our knowledge, there are only a few papers [74, 62, 27, 44] that address some form of this problem. These works either didn't focus on the problem itself [62, 27, 44] or used a very specific generator representation which can't be transformed easily from one game to another [74]. One might wonder if PCGML [130] can be used for the general level generation, the problem of PCGML so far is it needs a big amount of input data to work well. Recently, new research [138] showed the ability to generate levels from a small amount of data but the generated model so far can only be used to sample new levels which is not easy to be used directly for human collaboration, content repairing, etc.

In this chapter, we investigate how to generate game levels using reinforcement learning. To the best of our knowledge, this is the first time reinforcement learning is brought to bear on this problem. This is probably because it is not immediately obvious how to cast a level generation problem as a reinforcement learning problem. The core question that this chapter attempts to answer is therefore how level generation can be formulated as a tractable reinforcement learning problem. We formulate observation spaces, action spaces, and reward schemes so as to make existing RL algorithms learn policies that result in high-quality game levels.

Reinforcement learning approaches to PCG have several potential advantages over existing methods. Compared to search-based methods [137], the inference stage – generating a new level once a model has been trained – is much faster as no search needs to be performed.

This comes at the cost of having a long training phase which search-based PCG methods do not have, so in other words, we move time consumption from inference to training. Compared to supervised learning methods [130], the big advantage is that no training data is necessary. Another distinct advantage is that the incremental nature of the trained policies makes them potentially more suitable to interactive and mixed-initiative approaches to PCG, where content is created together with human users [150].

In our experiments, we focus on two-dimensional levels for three different game environments. Two of these are actual games – the classic puzzle game Sokoban (Thinking Rabbit, 1982) and a simplified version of The Legend of Zelda (Nintendo, 1986) – and the third is a simple maze environment where the objective is to generate mazes containing long paths. We formulate three different representations of game levels as reinforcement learning problems: the narrow representation, where the agent is asked to change a single tile at a random location in the observation; the turtle representation, which similarly focuses on a single tile, but where the agent can control its own location by moving to adjacent tiles; and the wide representation, where the agent can change any tile in the level at any point. We find that all three representations can be successful in all three game scenarios, given that the right choices are made regarding reward schemes and episode lengths, but that there are interesting differences in the generated artifacts.

9.1 PCGRL Framework

The PCGRL Framework casts the PCG process as an iterative task (like some of the constructive generators described in chapter 2) instead of generating the whole content at once (like Volz et al. [143] who generate Super Mario Bros levels using GANs and Latent Variable Evolution). We thus see content generation as a MDP, where at each step the agent gets an observation and reward then responds with an action.

To realize the idea of iterative content generation, we start with a level populated by random tiles. At each step, the agent is allowed to make a small change in the level (such as one tile). This change will be judged by the system with respect to a target goal for the level and assigned a reward. The reward should reflect how much closer that small change has brought the agent to its goal state. For example: if we are generating a PacMan (NAMCO, 1980) level, one of the goals is to have only one player; so a change that adds a player when there is none is a positive change, and negative otherwise. The system should also determine the halting point of the generation process (limiting the number of iterations so that it doesn't take forever).

To make the framework easy to implement for any game, we break it down into three

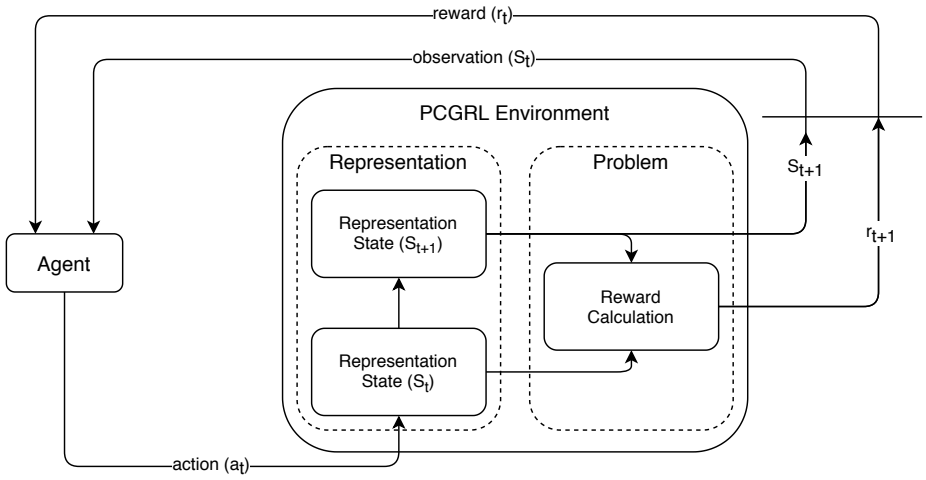


Figure 9.1: The system architecture for the PCGRL environment for content generation.

parts and isolate game-related information from the generation process. These parts are: the Problem module, the Representation module, and the Change Percentage. The problem module stores information about the generated level such as goal, reward function, etc. The representation module is responsible for transforming the current level into a viable observation state for the agent. The change percentage limits the number of changes the agent can affect the content over the course of an episode, preventing it from changing the content too much. We will discuss these parts in detail in the following subsections.

Figure 9.1 shows the PCGRL agent-environment loop. The agent observes the current state (S_t), and based on it, sends an action (a_t) to the representation module, which in turn transforms the state (S_t) into the next state (S_{t+1}). These two states (S_t and S_{t+1}) are both sent to the problem module, which assesses the change's effect on map quality and returns the new reward (r_{t+1}). The environment returns the new reward (r_{t+1}) and the new state (S_{t+1}) back to the agent, and the loop continues.

After training, these agents can be used as generators: they can iterate over a randomly-initialized map for a fixed number of steps, either improving it slightly or transforming it completely.

9.1.1 Problem

The problem module is responsible for providing all the information about the current generation task. For example: if we were trying to generate a Super Mario Bros (Nintendo, 1985) level, the problem module would support us with the level size, the types of objects that can appear in the level, etc. This module provides two functions. The first function assesses the change in quality of generated content after a certain agent action. For example:

1	1	1	1	1	1	1
1	4	0	0	0	0	1
1	0	3	1	3	0	1
1	1	0	1	0	1	1
1	0	2	0	3	0	1
1	4	0	0	0	4	1
1	1	1	1	1	1	1

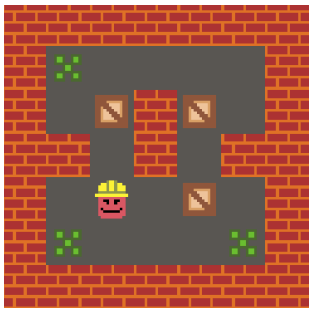


Figure 9.2: Sokoban level as 2D integer array.

if the agent removes an object from a game level, the problem module will assess the resultant change in level quality and return a reward value that the agent can use to learn. The second function determines when the goal is reached, which terminates the generation process. For example: if we are generating a house layout, we can terminate generation after we have a certain number of rooms created. It is important to define a goal for our problem that leads to many possible levels, as we are trying to learn a generator and not find a single level.

9.1.2 Representation

To model content generation as a MDP, we need to define the state space, action space, and transition function. The representation module is responsible for this transformation. Its role is to initialize the problem, maintain the current state, and modify the state based on the agent's action.

For the sake of simplicity, we represent a generated level as a 2D array of integers where the location in the array corresponds to the location in the level, and the value defines the type of object in that location. For example: Figure 9.2 shows a Sokoban (Thinking Rabbit, 1982) level as 2D array, and the corresponding level. This constraint makes it easy for us to adopt the same representations that were used in the work by Bhaumik et al. on generating levels using tree search algorithms [14]. In that work, they defined the following representations:

- **Narrow:** the simplest way of representing the problem. It is inspired by cellular automata [148] where at each step the agent observes the current state and a given location, and needs to decide what change to make, if any, at that location. The observation space is defined as the current state (as the 2D integer array) and the modification location (as an x and y index in the 2D array). The action space is defined as: a no-action (which skips the current location) or a change tile action (value

between 0 to $n-1$ where n is the number of different objects provided by the problem module) where that value will replace the tile at the current location.

- **Turtle:** inspired from the turtle graphics languages such as Logo [15] where at each step, the agent can move around and modify certain tiles along the way. The observation space is represented as the current state (as the 2D integer array) and the current agent location (as an x and y index in the 2D array). The action space is defined as: a movement action (which changes the agent's current location by moving it either up, $down$, $left$, or $right$) or a change tile action (value between 0 to $n-1$ where n is the number of different objects provided by the problem module).

- **Wide:** is similar to Earle's work on playing SimCity [44]. At each step, the agent has full control over the location and value of the changed tile. The observation space is the current state (as the 2D integer array). The action space is defined as: the affected location (x and y position on the level) and the change tile action (value between 0 to $n-1$ where n is the number of different objects provided by the problem module).

Each representation corresponds to a distinct class of agents. Those with **Narrow** representations are beholden to a predetermined sequence of build-locations; those with **Turtle** representations have local control over the current location, but only relative to the last; and those with **Wide** representations have full control. We might also develop hybrid representations (e.g. a mix of narrow and wide, where the agent can modify a small area around a given location) or modify their action schemes (e.g. changing multiple tiles instead of a single tile).

9.1.3 Change Percentage

The change percentage is an important parameter that defines how many tiles the agent is allowed to change as a percentage of the full size of the level. The change percentage limits the episode length during training, it terminated the current episode if the agent made changes that is more than this percentage. We can think about it in a similar manner as the discount factor, as it defines how greedy the agent should be. For example: if the percentage is small, it means the agent is only allowed to make a very small amount of changes to the map, so the agent will end up learning more greedy actions to get higher short-term rewards. On the other hand, if the percentage is high, it means the agent is allowed to change as much of the map as possible, so the agent will end up learning a less greedy and more optimal solution to the problem.

One might wonder why we would want a more greedy agent, rather than learning an optimal agent. The problem with allowing the agent to make as many changes as it wants is that the reward scheme is based on the global performance of the level and not relative to the initial random level layout. This means, that given complete information, the agent will converge to a single optimal solution. This is not the goal, as we desire an agent that acts as a designer that can transform an input level into a new level that is inspired by the input. Two solutions that we saw to this is to either limit the agents actions to force it to make minimal changes to the environment, or to define a general reward transformation that rewards based on performance relative to the changes in the starting level. In this work we opted for limiting the agents actions as it is the most straight forward approach.

Limiting the number of changes an agent can make will directly force the agent to be conservative with its changes and adapt to the initial design it is given. It does pose a learning challenge earlier presented by Bontrager et. al. [16], referred to as Subverted Generalization. With Subverted Generalization, the agent learns that certain actions, such as improving the level design, results in a positive reward, but then generalizing on this knowledge isn't safe as too many level improvements will cause the game to prematurely end. Limiting changes allows the agent to be reactive but does pose this learning challenge which might make it a little more difficult to train the agent.

9.2 Experiments

Our PCGRL framework[1] is implemented as an OpenAI Gym [20] interface, making it easy to use existing agents with our framework. We tested the framework using the three main representations (Narrow, Turtle, and Wide) on three different problems (Binary, Zelda, and Sokoban). For all the problems, the reward function rewards actions that help reach the goal of the problem while punishing actions that move away from it (check Chapter 3 for more details). The problem terminates when the goal is reached. For binary, we decided the longest shortest path length is more than or equal to 20; while for Zelda, the solution length is more than or equal to 16, and enemies are not too close more than 3 tiles away; finally for Sokoban, the solution length is more than or equal to 18.

After some preliminary tests, we decided to fix the change percentage to 20%, to encourage the agent to react, rather than to overwrite, the starting state. The starting state is randomly sampled from a random distribution which was tuned to fit each problem. For example: the player tile in Zelda and Sokoban have very low probability to appear in the starting state. Figures 9.5a,9.6a, and 9.7a show examples of different starting states for each

[1] `https://github.com/amidos2006/gym-pcgrl`

of the three problems.

We randomize the starting position of the turtle and narrow representations, to encourage generalization [70]. In narrow, each build location is chosen at random, so that the network will learn to react to any location, rather than overfitting on any given sequence of locations. This ensures that all locations are visited with equal frequency during level-design. It need not be true during inference. We provide the agent with a one-hot encoding of the level map to help training.

For training, we use *Stable Baselines* which is an improved implementation of OpenAI baselines [40]. We use Proximal Policy Optimization (PPO) [114] to train our agents. We use two different architectures for the agent networks. The first is used for the binary and turtle representation. Its body consists of 3 convolution layers followed by a fully connected layer, and an additional fully connected layer for both the action and value heads. This architecture is similar to Google's ALE DeepQ architecture [95]. The wide representation needed a different architecture due to its large action space (equal to the input space, e.g. 14x14 in binary). To solve that problem, we decided to use the similar architecture from Earle's [44] work in playing SimCity. The body of the network consists of 8 convolutions, which is used directly for the action head, while the value head has 3 additional convolutions.

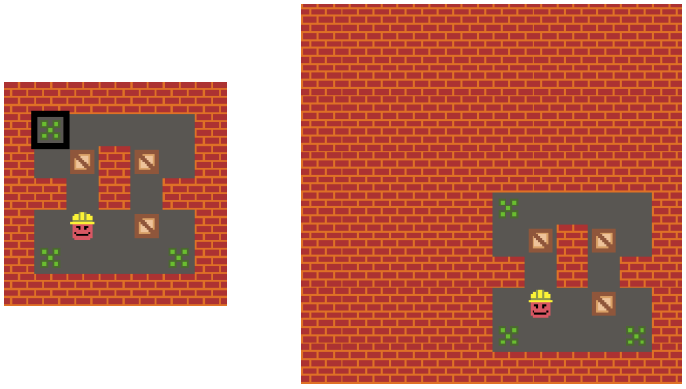


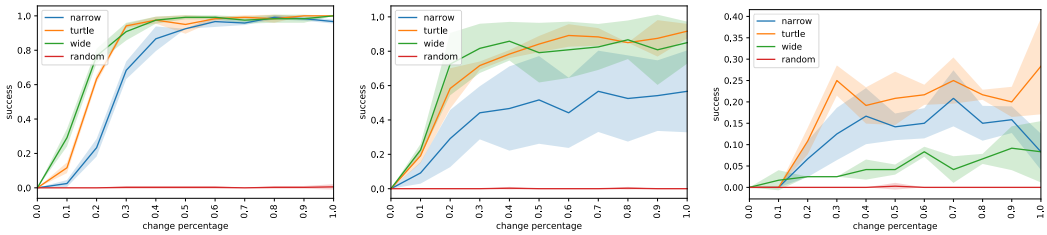
Figure 9.3: Location data is being transformed as an image translation in a double sized map.

For each problem/representation experiment, we trained 3 different models to show the stability of the training. Each model was trained for 100 million frames. Since both Narrow and Turtle have location information, we encode this information as a translation of the map around that location which means the new map is twice as big as the old map and the center of that new map is the location needed. Figure 9.3 shows the location information as a black rectangle on the left image. This information has been transformed as a translation information in the right image where the position of that tile is the center of the new map. The new map is twice the size of the original map and is originally filled with a default tile

(solid in this case).

9.3 Results

We trained 3 models for each representation and problem configuration. To analyze these models, we collected 40 generated levels for each model. To generate the levels, 40 different random level layouts were generated, the models were then tasked with modifying these random layouts into good levels. We analyzed the final modified levels using different change percentages, ranging from 0% to 100%, where the percentage represents the fraction of tiles the agent is allowed to change during inference.



(a) Binary (b) Zelda (c) Sokoban

Figure 9.4: The Success percentage of generating levels from random a initial state with respect to the change percentage.

Figure 9.4 shows the percentage of generated levels that satisfy the goal criterion for each game. The horizontal axis represents the change percentage during inference while the vertical axis represents the percentage of successful levels. The solid line is the average of the three trained models while the shaded area is their standard deviation. We added a random agent to the mix. The random agent results were averaged between all three representation which turns out to be a 0% success rate. We can see that some representations perform better than the others in some tasks while in others they perform the same. Predictably, when only allowed to make very few changes, the agent isn't able to design very good levels but possibly surprising is that the agent needs only to change at most 40% of the tiles to design a successful level. This is especially interesting since the agent was only allowed to change 20% of the tiles during training.

Figure 9.4b shows that the narrow representation fails to get a high success rate compared to the other representations. We found that the narrow representation could design mostly good levels but fails to make their solution paths longer than 16 steps. We believe that narrow might be struggling to improve the solution path due to the random nature of selecting the next tile during training and the need for long waits to make sure the system visits every

	Narrow	Turtle	Wide
Binary	$29.7\% \pm 1.4\%$	$19.7\% \pm 0.8\%$	$15.6\% \pm 1\%$
Zelda	$39.25\% \pm 14.9\%$	$21.6\% \pm 2.6\%$	$21.8\% \pm 11.2\%$
Sokoban	$26.9\% \pm 1.6\%$	$25.1\% \pm 2.2\%$	$26.5\% \pm 2.3\%$

Table 9.1: Percentage of map changed by the agent during inference when allowed unlimited changes.

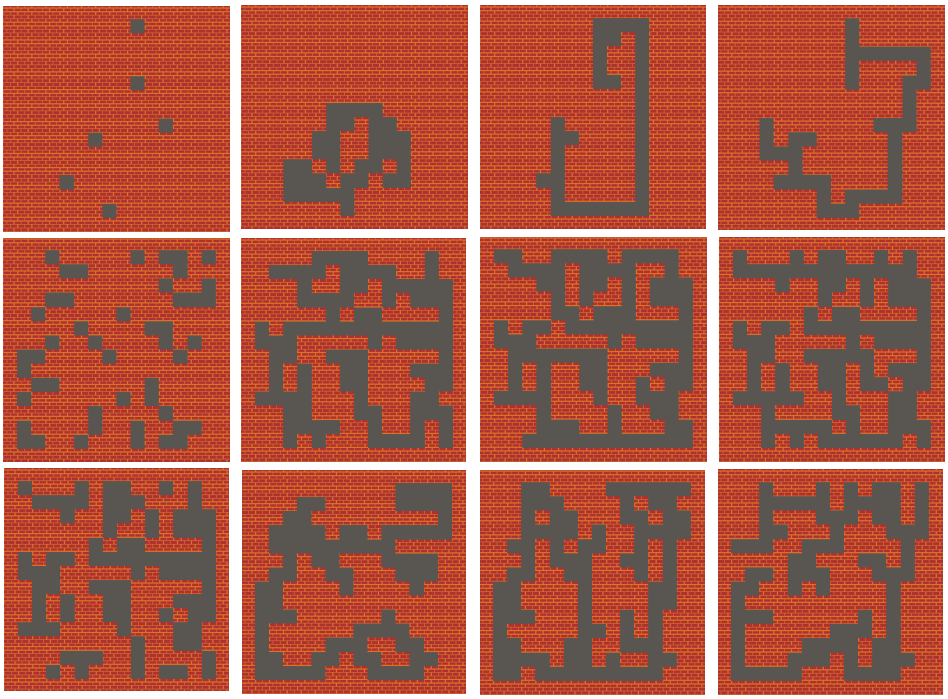
single tile. We believe that it could be solved if the model were trained for a longer time (as it was still learning after 100 million frames).

Figure 9.4c shows the result from generating Sokoban levels. We can see that the percentage of success is relatively low, especially in the wide representation. We looked into the generated levels and found that most of them are easy levels that can be solved in few steps (less than 18). Narrow models generate 86.7% solvable (average solution length of 9.4), Turtle models generate 88.3% solvable (average solution length of 12), while Wide models generate 67.5% solvable levels (average solution length of 6.2) at 100% change percentage. Based on that, we think that this problem could be solved if the model were trained for a longer time using a more powerful Sokoban solver, as the current solver only solves simple levels to reduce training time.

We also looked at how many changes the trained models are making on average. Table 9.1 shows the percentage of changed tiles in the map if the agent has no upper bound (change percentage is equal to 100% during inference). We can see that most of the agents still don't make a lot of changes, which proves that the models are reacting to the input map instead of overwriting it with an optimal solution. These results suggest that these trained agents could be used in mixed initiative approaches [150] as they are able to react to the map and the user inputs.

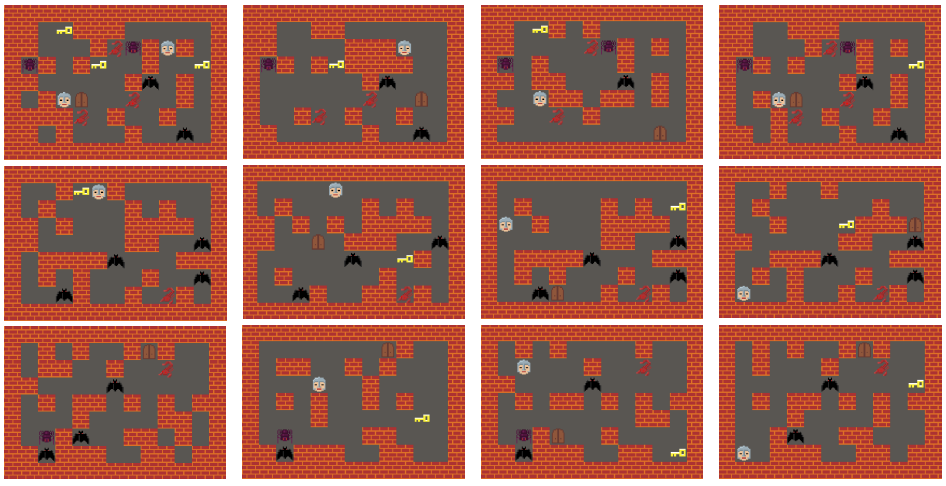
Figures 9.5,9.6, and 9.7 show the results from running a trained model on the Binary, Zelda, and Sokoban problems respectively. In these experiments, we fix the starting state and run the trained model for each representation. We run the model with 100% change percentage instead of the 20% change percentage (used during training). The reason for the increase in the change percentage is to allow the algorithm to make more changes in case it started from very bad starting state.

As shown in figure 9.4, different representations do not seem to have a large effect on success across every problem. The real difference can be seen in design styles. This is especially clear in the Binary problem (figure 9.5). In the Binary problem, all representations achieve similar success percentages, but each results in a stylistically distinct set of generated levels. In the narrow representation, the agent has no control over when it is allowed to



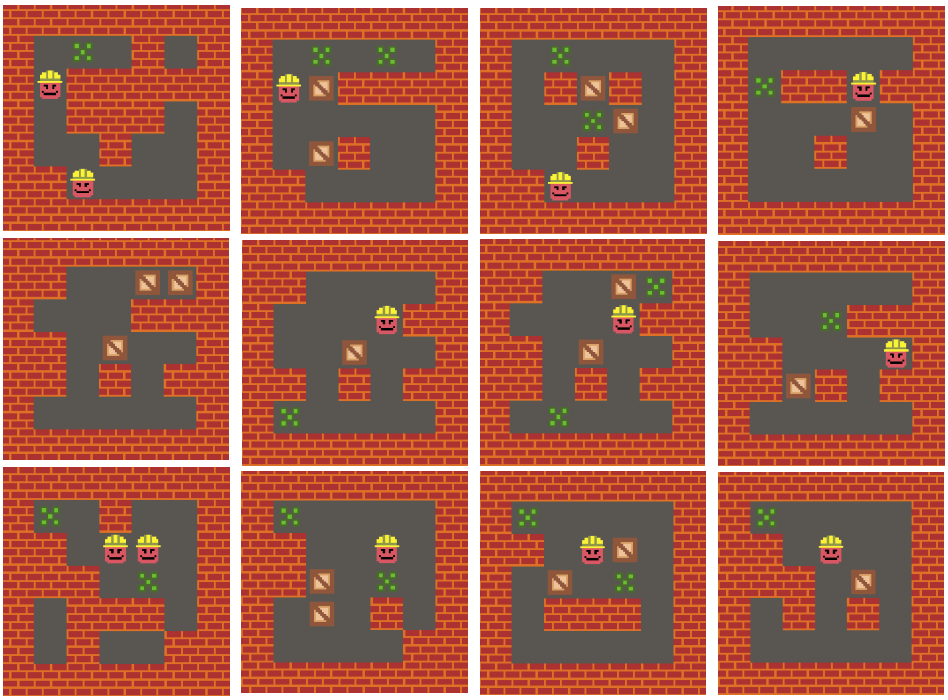
(a) Initial (b) Narrow (c) Turtle (d) Wide

Figure 9.5: Binary generated examples using different representation for the same starting state.



(a) Initial (b) Narrow (c) Turtle (d) Wide

Figure 9.6: Zelda generated examples using different representation for the same starting state.



(a) Initial (b) Narrow (c) Turtle (d) Wide

Figure 9.7: Sokoban generated examples using different representation for the same starting state.

make a change in a specific area. From this it seems to have learned to make more frequent sub-optimal changes rather than waiting for the ideal change. This makes this representation the least "controlled" by the initial conditions and to have the most varied results. The turtle representation has control over where it goes, but only loosely, as navigation is difficult and it's possible for the game to end if it does not move efficiently. It seems to favor updating the board in a spiral manner. Lastly, the wide representation is the most efficient with its changes, with its designs being closest to the original pattern. This agent could always control a change exactly, so it never needed to risk ending the game with too many changes. The downside of using the wide representation is that it must learn a much larger action space than the other agents.

9.4 Unhelpful Representations

To help the agents better learn and track the changes they are making, we experimented with providing the agents with general, non game-specific, information that could be incorporated into any PCGRL environment. We found these approaches to almost universally be unhelpful at best. We show them in table 9.2 to assist others who want to experiment further with the

Representation	Expected Benefit	Outcome
Heatmap of changes	Agent can track changes and relative number of changes	No change in the performance of the agents
Heatmap of visited Locations	Agent can track where it's been and avoid returning	It slightly helped the turtle agent to navigate but it ended up with similar results
Percentage of changes	Agent can track the percentage of changes (0 means no changes have happen, while 1 means the agent can't do any change any more)	The agents waited till certain percentages to act while not changing any results

Table 9.2: Additional Representations

representation we are proposing here.

The information that we believed would be so useful is a heatmap of the changes an agent has made. The idea of adding the heatmap is to give the agent an external memory so the agent is not changing the same tiles multiple times and so the agent can plan larger changes. After experimenting with that heatmap, the outcome didn't differ from our current trained algorithm.

The idea outlined on the second line of Table 9.2 is a heatmap of visits. It is similar to the map of changes but tracks all the tiles that have been visited, regardless if they are changed or not. We added this information especially for the turtle representation to help it to easily learn navigation and avoid revisiting the same location which we noticed in earlier experiments. Adding the heatmap of visits was slightly successful, it managed to make the turtle not revisit a lot of old places but overall it didn't change the success percentage for any of the problems. We theorize that both heatmaps didn't work because our problems are too simple and don't require external memory to plan. They might be more successful with more complex problems which we would like to investigate in future work.

Lastly, we decided to provided the percentage of changes as a single value so the agent can know how many changes are remaining. The idea is to allow the agent to know when it can make big changes or when it is almost done and should be greedy. The results were not good, the models usually wasted time by changing tiles back and forth until the percentage got closer to zero and then it started to work towards the goal.

9.5 Discussion

The two major conceptual differences between this work and almost all published research on PCG is to search in content generator space rather than content space, and to see content generation as an iterative improvement problem. Searching in content generator space was prefigured by the Procedural Procedural Level Generator Generator (PPLGG), which evolves Super Mario Bros level generators [74]. However, the search in PPLGG proceeds through interactive evolution rather than reinforcement learning, and the generators are represented as simple multi-agent systems. The paradigm presented here is more scalable, not least for its automatic learning. Other work seeing game content generation as a sequential task has been discussed above in chapter 2.

It is particularly exciting to see the multiple uses PCGRL could have in mixed-initiative editing tools. As the interface of a trained model is to make a single small change intended to improve the level, this fits in very well with a turn-taking paradigm. One could imagine multiple trained PCGRL agents working as suggestion engines, pointing out where to improve the current design or as "brushes," applied by the human user to e.g. increase difficulty in a region.

One of the main reasons that approaches to PCG based on search, optimization and solving have not been applied widely in the game industry is that they take too much time. And it is indeed true that if any kind of simulation is incorporated into an evaluation function, the search for good game content can be costly. Here, PCGRL offers the possibility of moving most of the time consumption from inference to training, or, in game development terms, from runtime to development. This might make PCGRL a viable choice for runtime content generation in games even on devices with limited computational power. The problem of designing an appropriate reward scheme remains, however, and is similar to that of designing an evaluation function in search-based PCG.

9.6 Summary

We introduce the PCGRL framework [2] to tackle the problem of general level generation from a different angle. We decided to search for a generator for every new game and then later we can use it to get new levels for our game. We decided to represent the generator as a neural network which we train using reinforcement learning. We introduced three different representations to transform the problem of level generation to a Markov Decision Process. These representations are different families from the agent point of view as they allow of

[2] https://github.com/amidos2006/gym-pcgrl

different degrees of agent controlling the next affected location on the generated map. We trained a PPO algorithm using all the 3 representations on our 3 problems. The trained agent was able to successfully generate levels for all the experiments except for Sokoban where it was only able to generate easy levels. We think that more training time or better models will be able to handle bigger levels and more difficult levels. The different representations didn't affect the ability of the agent to learn but affected the style of the generated content.

Chapter 10

Constructive Level Generator through Multi-Objective Evolution

One of the problems with the previous chapter approach is that it is not easy to interpret what is happening inside the neural network, which doesn't help humans to understand more about level generation. It is also hard for any human to hand tune the network directly, the only way is to do it indirectly using active learning.

In this chapter, we tackle the previous problem by using a more understandable representation. We use Marahel [81] as our generator representation. Marahel provides us with a more understandable search space where we can understand every step of the generation process and modify it easily. We search this space using NSGA-II to find generators that can perform as well as the trained network from the previous Chapter on the same three problems (Binary, Zelda, and Sokoban).

10.1 Marahel Language

Marahel[1] is a language and framework for constructive 2D tile-based level generation. The language is an attempt to formalize the principles behind a number of popular algorithms that can be used for constructive (i.e. not based on generate-and-test) level generation for tile-based games so that they can easily be recombined.

Any valid *Marahel* string constitutes a specification for a level generator, which when interpreted by the *Marahel* parser can produce 2D tile-based levels. Not all valid *Marahel* scripts will produce usable levels for all games because game mechanics play an important role in defining the space of plausible levels. For example: if the player is able to dig through

[1] *Marahel* means *levels* in Arabic.

walls, it is okay to have isolated areas. The user of *Marahel* (a human and/or an algorithm) must make sure that the script is not only a valid script but also a suitable one, i.e. it fits the requirements of the current game.

Marahel approaches level generation as a description language that describes the steps of the generation process instead of the required level. By comparing level generation techniques to programming paradigms, we can see that techniques such as constraint-based generation follow a declarative programming paradigm, while other techniques such as constructive generation follow an imperative programming paradigm. Answer Set Programming [122] can be described as a language that follows a declarative programming paradigm where the user defines the features required in the output and the system finds a solution for it. Following this logic, *Marahel* can be described as a language that follows an imperative programming paradigm (like C++) where the user defines the steps required by the generator to change the current map.

Each script consists of 5 sections: Metadata, Entities, Neighborhoods, Regions, and Explorers. The first 3 sections (Metadata, Entities, and Neighborhoods) defines different data required during the generation process, while the rest (Regions and Explorers) defines the steps of the generation. Comparing these sections to an imperative programming language, the first 3 sections will be similar to the input data and constant values required/used by the program, while the last 2 sections are the actual program itself.

The following five steps are taken by the *Marahel* when implementing a generator description:

1. Parse the first 3 sections (Metadata, Entities, and Neighborhoods) and save them for later usage.

2. Define a 2D array of the dimension specified in the previous step and initialize it with "unknown".

3. Use the algorithm defined in the Regions section to divide the map into several areas.

4. Apply all the defined explorers sequentially to modify the 2D array based on their rules.

5. Return the 2D array to the user.

The *Marahel* language can be described as a context-free grammar. Figure 10.1a shows the full definition of the current version of *Marahel*. Terminals in *Marahel* are a list of the currently supported features in the system. Adding a new terminal to the list extends

⟨*script*⟩ ::= ⟨*metadata*⟩ ⟨*entities*⟩ ⟨*neighborhoods*⟩ ⟨*regions*⟩ ⟨*explorers*⟩

⟨*metadata*⟩ ::= ⟨*generalInfo*⟩ ⟨*metadata*⟩ | ⟨*generalInfo*⟩

⟨*generalInfo*⟩ ::= minDimension | maxDimension | dimension

⟨*entities*⟩ ::= entityName ⟨*entities*⟩ | entityName

⟨*neighborhoods*⟩ ::= ⟨*neighbor*⟩ ⟨*neighborhoods*⟩ | ϵ

⟨*neighbor*⟩ ::= neighborName relativePoints

⟨*regions*⟩ ::= numOfRegions ⟨*divider*⟩

⟨*divider*⟩ ::= 'equal' | 'bsp' | 'sampling'

⟨*explorers*⟩ ::= ⟨*explorer*⟩ ⟨*explorers*⟩ | ϵ

⟨*explorer*⟩ ::= ⟨*appliedRegion*⟩ ⟨*generalParam*⟩ ⟨*expType*⟩ ⟨*rules*⟩

⟨*appliedRegion*⟩ ::= 'map' | 'all' | 'some' | 'specific'

⟨*generalParam*⟩ ::= ⟨*param*⟩ ⟨*generalParam*⟩ | ϵ

⟨*param*⟩ ::= borderSize | borderHandling | replacingTech | iterations

⟨*expType*⟩ ::= 'narrow' | 'turtle' | 'wide'

⟨*rules*⟩ ::= ⟨*rule*⟩ ⟨*rules*⟩ | ⟨*rule*⟩

⟨*rule*⟩ ::= ⟨*conditions*⟩ ⟨*executers*⟩

⟨*conditions*⟩ ::= ⟨*cond*⟩ ⟨*conditions*⟩ | ϵ

⟨*executors*⟩ ::= ⟨*execut*⟩ ⟨*executors*⟩ | ⟨*execut*⟩

⟨*cond*⟩ ::= ⟨*bioperator*⟩ ⟨*estimator*⟩ ⟨*estimator*⟩ | ⟨*unioperator*⟩ ⟨*estimator*⟩

⟨*estimator*⟩ ::= constant | random | noise | entityEstimator | neighborhoodEstimator | distanceEstimator

⟨*bioperator*⟩ ::= 'equal' | 'notEqual' | 'greater' | 'less'

⟨*unioperator*⟩ ::= 'isEven' | 'isOdd' | 'numberOf'

⟨*execut*⟩ ::= neighborhoodExecuter

(a)

```
{
  metadata:{
    minDimension:"40x30",
    maxDimension:"60x45"
  },
  entities:["empty", "solid"],
  neighborhoods:{
    all:"111,131,111",
    plus:"010,121,010"
  },
  regions:{
    type:"bsp",
    numberOfRegions:7,
    parameters:{
      min:"8x8",
      max:"15x15"
    }
  },
  explorers:[
    {
      type:"narrow",
      region:"map",
      rules:["self(any)->self(solid)"]
    },
    {
      type:"narrow",
      region:"all",
      rules:["dist(out)>1,self(any)->
          self(empty)"]
    },
    {
      type:"turtle_connect",
      region:"map",
      parameters:{
        directions:"plus",
        entities:"empty"
      },
      rules:["self(any)->self(empty)"]
    }
  ]
}
```

(b)

Figure 10.1: The left figure shows *Marahel* language as a context free grammar. Angular brackets values such as <entities> are non terminal, quoted values such as 'equal' are string terminals, while other values such as entityName are user defined terminals. The right figure shows an example of a full Marahel script.

Marahel's capabilities. For example: If a new divider algorithm is required, we only need to add a new terminal to "<divider>".

Figure 10.1b shows an example of a full *Marahel* script compatible with the current Javascript implementation[2]. This script generates dungeons that consist of 7 connected rooms of different sizes. Below we describe the different sections of a Marahel script.

[2]https://github.com/amidos2006/Marahel

10.1.1 Metadata

The Metadata section contains all the information that is related to the whole generation process such as the minimum and the maximum dimensions of the generated map. In figure 10.1b, the generator is able to generate levels with minimum size of *40x30* tiles and maximum size of *60x45*.

10.1.2 Entities

The Entities section contains a list of all the names of the entities that can appear in the final generated map and is the "ontology" of the levels. Entities are the base unit of any generated level. A level is a 2D array of entities. For example, in figure 10.1b has two types of entities which are "solid" and "empty".

10.1.3 Neighborhoods

The neighborhoods section is a section that contains a list of different neighborhoods. A neighborhood is an entity that defines relations between multiple locations and a center one. Neighborhoods can be represented using various methods such as a list of points, a 2D array of numbers, etc. For example: "[(1,1), (0,-3)]" shows a list version of a neighborhood where (1,1) and (0,-3) are relative points. To calculate the relative locations from a certain point such as (2,2) using this neighborhood, you need to add each point separately to get (3,3) and (2,-1) as a result.

In the current implementation, *Marahel* uses a 2D array of numbers to specify these relative points. The 2D array is flattened into a string to make it easy to write. Each neighborhood contains a name and a 2D array written as string. The numbers indicate the relation between their locations in the matrix with respect to a certain location.

Figure 10.1b shows two neighborhoods: *all* and *plus*. The comma signify a new row in the matrix, while each number represent a column. Each *1* in the string tells the generator to use that location relative to the location of the value *2* or *3*. The value *3* is same as *2* but it tells the generator that this location is a relative location too. For example: "111,121,111" checks all the 3x3 tiles except the center tile, while "111,131,111" checks all the 3x3 tiles including the center tile.

10.1.4 Regions

The Regions section defines the algorithm that is used to divide the map into several regions. Regions are portions of the generated map that are generated using the selected algorithm. In

each *Marahel* script, The user selects the "divider algorithm" and the "number of regions". *Marahel* currently supports three different algorithms to generate rectangular regions:

- **Equal:** divides the map into equal-sized portions using a grid then selects randomly some/all regions based on the required "number of regions".
- **Binary Space Partitioning:** divides the map into different size region by splitting each region either vertically or horizontally. The algorithm keeps splitting each region till the termination conditions are met. After that, it selects randomly some/all regions based on the required "number of regions".
- **Sampling:** adds regions to the generated map that do not intersect with the previous ones. The algorithm continues until the required "number of regions" is met.

The generator in figure 10.1b uses binary space partitioning to divide the generated map to *7* different regions such that no region should be smaller than *8x8* and no bigger than *15x15*.

10.1.5 Explorers

Explorers are the core of the generation process. Explorers use an algorithm to visit different tiles on a defined region of the map. At each step of the algorithm, the explorer is at a certain location(s) where it will apply the defined rules. Explorers and rules together define how the system modifies the generated map. A *Marahel* script can have more than one explorer where they are applied sequentially. Explorers consist of 3 main parts:

- **Type and Parameters:** specifies the type of the explorer and its parameters. Different supported types will be discussed later in this section. Also, It specifies some general parameters such as "number of repetition" which allows the system to repeat this specific explorer any number of times.
- **Applied Region:** selects the area of the map that is affected by the explorer. The user can select either the whole map, all/some regions generated by the region divider, or manually defined regions. Any tile outside the applied region(s) won't be affected by the explorer.
- **Rules:** is a list of conditional rules that change the generated map. *Marahel* goes over the list in order until the first rule is satisfied; that rule will then be applied.

$$rule\colon condition, ..., condition \rightarrow executer, ..., executer \tag{10.1}$$

Equation 10.1 shows the structure of rules in *Marahel*. Rules in *Marahel* consists of two sides: Left hand side and Right hand side. The left hand side is a group of conditions that need to be satisfied before applying the right hand side. If any condition fails, the rule fails.

$$condition\colon estimator <op> estimator \tag{10.2}$$

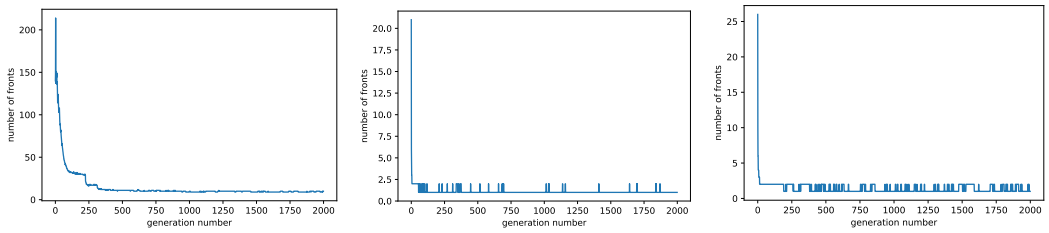
Conditions can be anything that returns either true or false. In the current implementation of *Marahel*, only comparative conditions exists (bi-operator conditions). Equation 10.2 shows the structure of the comparitive conditions. <op> is either greater than ($>$), less than ($<$), equal ($==$), or not equal ($!=$). Estimators are functions that return a numerical value, it can be anything from a constant number to a complex equation. Estimators, in the current implementation of *Marahel*, are either a neighborhood estimator, a distance estimator, a number estimator, or an entity estimator.

The neighborhood estimator calculates the number of a certain entity/entities around the current location using the relative points defined by a specified neighborhood. The distance estimator calculates the minimum Manhattan distance between the current location and a specified entity/entities. The number estimator is either a fixed number, a random number between 0 and 1, or a Perlin noise value between -1 and 1 for the current location either in the applied region or in the whole map. The entity estimator gets the total number of a specified entity either in the applied region or in the whole map.

Executors are simpler than conditions. Executors modify locations on the map relative to the current location. In the current implementation of *Marahel*, it supports one type of executor where it changes the current location and/or the surrounding locations (using the relative points of a specified neighborhood) to a certain entity. If the executor has a list of entities, it will pick one of them at random.

Marahel currently supports three main types of explorers. These types are inspired from the PCGRL representation:

- **Narrow**: The system control the location of the next visited tile based on outside factor. The outside factors can be horizontal movement (*narrow_horz*) or vertical movement (*narrow_vert*) or just pick random locations (*narrow_rnd*).
- **Turtle**: The agent control its next location with respect to its current location. It can choose the next location randomly (*turtle_drunk*), using an estimator function (*turtle_heur*), or get closer to disjoint area (*turtle_connect*). The relative locations,



(a) Binary (b) Zelda (c) Sokoban

Figure 10.2: Number of Pareto fronts at each generation for all the three problems.

the agent selects from, can be defined using one of the neighborhoods. The default neighborhood used is similar to the plus neighborhood defined in figure 10.1b.

- **Wide**: The agent controls its next location by selecting any location in the map. This selection can be based on an estimator function (*wide_heur*) or random (*wide_rand*). The wide explorer guarantees that no tiles will be visited more than once per iteration of the explorer.

In figure 10.1b, we are having 3 different explorers that are being executed after each other. The first explorer is a narrow explorer that initialize the whole map to solid. Followed by another narrow explorer that initialize every region to empty except the borders of the region. Lastly, a turtle explorer is used over the whole map that connects all the isolated empty areas (connects all the regions).

10.2 Marahel Evolution

We use grammatical evolution [113, 101] to evolve our Marahel programs. We restricted our evolution to only evolve five different explorers. The reason is to force evolution to find interesting small programs than allowing for big ones. We also introduce an explorer before these five to initialize the map with random tiles to allow evolution only to focus on achieving the target results. There are no regions, all the explorers are applied on the full map. For neighborhoods, we fixed them to a predefined set of 18 different ones. These neighborhoods cover different configurations that can be used such as Moore neighborhood, Von Neumann neighborhood, diagonal neighborhood, etc. These neighborhoods have 3 different sizes 1x1, 3x3, and 5x5.

We are going to search for generators for the same three problems introduced in the PCGRL Framework [75]. The goal of the generation is to find a playable level. The initialization explorer that we added is adjusted similarly to the one used in the PCGRL

framework where it is biased to have a good starting state. In Binary, the empty is equal to solid equal to 50%. In Zelda, Empty is 50%, Solid is 25%, Enemies is 10%, Player is 5%, Key is 5%, and Door is 5%. Lastly in Sokoban, Solid is 40%, Empty is 45%, Player is 5%, Crate is 5%, and Target is 5%. The reason is to have the same starting point similar to the PCGRL framework which allows us to compare it. Also, these values generate levels that require a small number of changes to make it playable therefore helping the evolution.

We decided on using multi-objective evolution instead of normal objective-based evolution as we found from an earlier experiment that normal objective-based evolution (GA) doesn't improve much in all the different requirements. We think the reason that our constrained search space makes it not possible to optimize all these values at the same time. An increase in one value will cause a decrease in another one which can be seen later in section 10.3.

10.2.1 Representation

The chromosome consists of 102 integer numbers where each number is a value between 0 and 49. The first number identifies the number of explorers used (min 1 and maximum 5), the second number is the seed for a random number generator, every 20 integers after that correspond to an explorer. Each explorer number directs the grammar expansions for non-terminal symbols.

10.2.2 Genetic Operators

We are using two genetic operators: Crossover and Mutation. The crossover operator allows for bigger meaningful changes. it can swap either the seed number, number of explorers, or one of the explorers (all the 20 numbers). On the other hand, the mutation operator does a very small change. It picks a random location from the array and replaces it with another random value.

10.2.3 Fitness Functions

In this work, we want to find generators that can produce playable levels for all three problems. The problem of using playability only as our fitness is its rough fitness landscape. All the random initialized generators for most of the problems will produce 100% unplayable levels. Having other fitness functions allow the space to be smoother or optimized toward these ones to reach the goal.

All our fitness function are designed to reflect how close the current value to a desired value. For example: if we want to have one player, so our fitness function will be 1 if the number of player is 1 and less than one otherwise. The value is calculate using the following equation.

$$f(x) = \begin{cases} \frac{range_{min} - x}{range_{min}} & \text{if } x < range_{min} \\ 1 & \text{if } range_{min} \leq x \geq range_{max} \\ \frac{x - range_{max}}{max - range_{max}} & \text{if } x > range_{max} \end{cases} \tag{10.3}$$

where x is the current input value to be scaled, $range_{min}$ is the minimal acceptable value, $range_{max}$ is the maximum acceptable value, and max is the maximum possible value. The $f(x)$ is clamped to be always between 0 and 1.

Binary has two fitness functions:

- **Number of Regions:** the number of regions in the generated map. The goal is to have one region so $range_{min}$ equals to $range_{max}$ equals to 1 and max is 10.
- **Path Length Improvement:** the number of increase in the shortest longest path after the random initialization explorer. The goal is to have an increase of at least 20. To achieve that, $range_{min}$ is equal to 20 and $range_{max}$ is infinity.

Zelda has five fitness functions:

- **Number of Players:** the number of players in the generated map. The goal is to have one player. To achieve that, $range_{min}$ equal to $range_{max}$ equal to 1, and max is equal to 10.
- **Number of Keys:** the number of keys in the generated map. Similar to the number of players, the goal is to have one key.
- **Number of Doors:** the number of doors in the generated map. Similar to the number of players, the goal is to have one door.
- **Number of Enemies:** the number of keys in the generated map. The goal is to have not many enemies and not too few enemies. To achieve that, the $range_{min}$ is 2, $range_{max}$ is 4, and max is equal to 10.
- **Solution Length:** the number of steps the player needs to reach the key then the door. The goal is to have at least 20 steps to finish the level. Similar to path length improvement, we set $range_{min}$ to 20 and $range_{max}$ to infinity.

Sokoban has four fitness functions:

- **Number of Players:** the number of players in the generated map. The goal is to have one player. To achieve that, $range_{min}$ equal to $range_{max}$ equal to 1, and max is equal to 10.
- **Number of Crates:** the number of crates in the generated map. The goal is to have not too many crates and not too few so we set $range_{min}$ to 2, $range_{max}$ to 4, and max to 10.
- **Absolute Difference:** the absolute difference between the number of crates and targets in the generated map. The goal is to have the number of crates equal to the number of targets so the level can be won. To achieve that, we set $range_{min}$ and $range_{max}$ to 0, and max to 10.
- **Solution Length:** the number of steps the player needs to win a Sokoban level (all crates are on targets). The goal is to have at least 20 steps to finish the level. Similar to path length improvement, we set $range_{min}$ to 20 and $range_{max}$ to infinity.

10.3 Results

For evolution, we used NSGA-II algorithm. We used tournament selection of size 2, the population size of 500, 2000 generations, crossover rate equal to 70%, and mutation rate equal to 30%. For each problem, we have different size maps similar to the same sizes from the PCGRL framework. For Binary, the map size is 14x14, while Zelda is 11x7, and finally, Sokoban is 5x5. Since the evolved generator will always generate different levels, so the fitness values is calculated by averaging the values of 50 different generated maps.

Because of the too many fitness functions for Zelda and Sokoban, the Pareto Front might be missing some interesting generators. Figure 10.2 shows the number of Pareto fronts at each generation. At generation 2000, we can see that for Zelda and Sokoban all there is only one front (all 500 chromosomes are in it). This shows that there might be more interesting generators that are not appearing. This doesn't happen in the Binary problem as it has 10 fronts at generation 2000.

10.3.1 Binary

As discussed before in section 10.2, the current representation, and restrictions don't allow us to find a generator that satisfies both fitness functions. Having a high path length leads

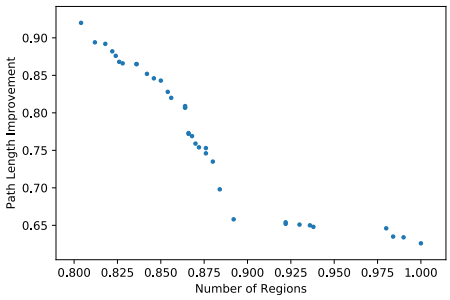


Figure 10.3: The Pareto front for the Binary Problem for both fitness functions.

to having more than one region while having one region leads to having less than 20 path length improvement. Figure 10.3 shows the Pareto front of the Binary problem after 2000 generation. The Pareto front contains 36 chromosomes out of the 500 while the rest are distributed on the other 9 fronts.

Figure 10.4 shows one of the Pareto front generators that has the highest path length improvement 0.92 (an average of 18 increase) and connectivity of 0.8 (an average of 2 regions). Looking at the generator, the generator has 3 explorers. The first explorer connects the random initialized level using vertical lines of empty which leads to a fully connected level with big open space as the connection using vertical lines instead of one tile. The second explorer is another connecting one but since the first one connected everything then it won't happen. Finally, the last explorer goes on every tile in the map and check if it is empty using 5x5 Moore neighborhood, it converts the center to solid. This last explorer is the reason for having more than 1 region but at the same time, it is what guarantees the long path as it causes a lot of diagonal solid in big open areas. Looking at the examples in figure 10.4, we can see the long vertical connections led to having a circular dungeon generator and we can see a small few disconnected areas.

10.3.2 Zelda

For the Zelda problem, at the last generation, all the chromosomes (500) exist in one front which shows that there are more chromosomes that can exist in the front. The Pareto Front consists of five dimensions so we decided to plot it into several 2D plots of different fitness function combinations (figure 10.5). Figure 10.5a shows the Pareto front between the number of players and the number of enemies fitness functions. It is interesting to see that it is inversely proportional as it means that having more enemies means less chance of having a single player. This makes sense as having more enemies means fewer tiles for the player avatar.

```
{"explorers": [
{
  "type": "connect",
  "parameters": {
      "repeats": "1",
    "replace": "buffer",
    "directions": "plus",
    "entities": "empty"
  },
  "rules": [
    "self(any) -> vert(empty)"
  ]
},
{
  "type": "connect",
  "parameters": {
      "repeats": "1",
      "replace": "same",
    "directions": "plus",
    "entities": "empty"
  },
  "rules": [
    "self(out) -> plusnc(solid|empty)"
  ]
},
{
  "type": "horz",
  "parameters": {
      "repeats": "1",
    "replace": "buffer"
  },
  "rules": [
    "plusfive(empty) -> down(solid)"
  ]
}]}
```

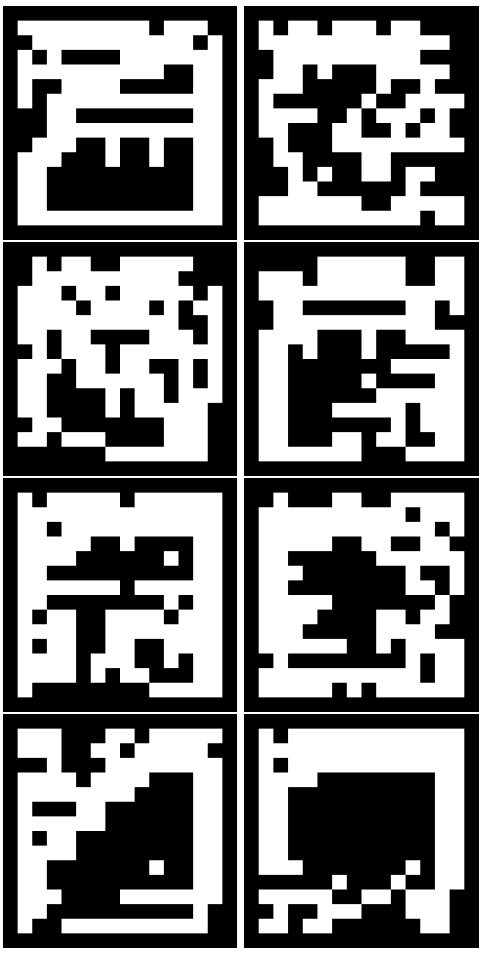
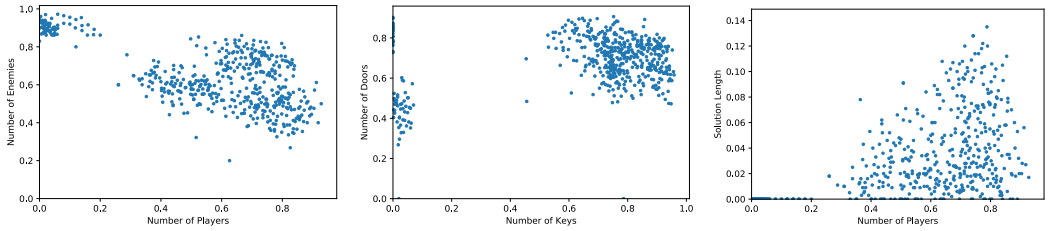


Figure 10.4: The evolved binary generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. The black tiles are solid, while the white tiles are empty. This generator has number of regions fitness value equal to 0.8 and path length improvement fitness value equal to 0.92



(a) (b) (c)

Figure 10.5: The Pareto front for the Zelda Problem for different combination of the fitness functions.

Figure 10.5b shows the Pareto front between the number of doors and the number of keys fitness functions. Interesting enough, the graph is directly proportional everywhere except near the end where we almost have a fitness of 1 for the number of keys. Looking at the script, we found that the script erases a lot of entities while trying to connect the isolated area from the map. This erase behavior might break the generated levels if there are few numbers of keys or doors which are the case when key fitness function approach to 1.

Figure 10.5c shows the Pareto front between the number of players and solution length. The solution length fitness is pretty low overall with a maximum of 0.14. This doesn't mean it is unplayable, it also could mean very short length. It is obvious when we low the number of player fitness we have low solution length as you can't play a level if you don't have one player. On the other hand, with high value for the number of players, the solution length is bouncing between almost 0 and 0.14. We think this noise is due to the other two fitness functions: the number of keys and the number of doors. The only way to have a solution length is to have one player, one key, and one door. This makes it a very hard fitness function to satisfy causing these low fitness values.

Figure 10.6 shows a Marahel Zelda generator evolved after 2000 generation. We picked this generator as it has the highest solution length fitness. Looking on the 10 generated examples, we can notice that none of them are playable but some can be fixed to be playable easily. The interesting thing about this generator example is it have a small number of players and keys and doors which makes it have a high chance to generate playable levels. Looking into the generator itself, we can see it is more of an eraser. It depends on the starting noise and it tries to substitute some of the tiles by solid using a noise function while moving on the path to connect entities.

10.3.3 Sokoban

Similar to Zelda, all the 500 chromosomes appear in the one front. The Pareto front consists of four dimensions to plot it so we decided to split it into two 2D plots for 2 combinations of the fitness functions (figure 10.7). Figure 10.7a shows the Pareto front between the number of players' fitness value and the solution length fitness value. It is obvious that having a higher number of players will cause the solution length value to increase (you can't play a level if you don't have one player). The solution length value is very noisy and with a peak of 4.5% around 0.6 number of player value. Similar to Zelda, we think that the low solution length fitness value due to the cascaded fitness as a level is only playable if it has one player, the number of crates more than 0 and equal to the number of targets. Even when all this happens, the level has a higher chance to be unplayable compared to Zelda levels as

```
{"explorers": [
{
  "type": "greedy",
  "parameters": {
    "repeats": "3",
    "replace": "buffer",
    "directions": "all",
    "heuristics": "dist(empty)"
  },
  "rules": [
    "diagfive(key) -> down(door)"
  ]
},
{
  "type": "connect",
  "parameters": {
    "repeats": "1",
    "replace": "buffer",
    "directions": "all",
    "entities": "empty"
  },
  "rules": [
    "noise>=-0.5
-> left(solid)"
  ]
}]}
```

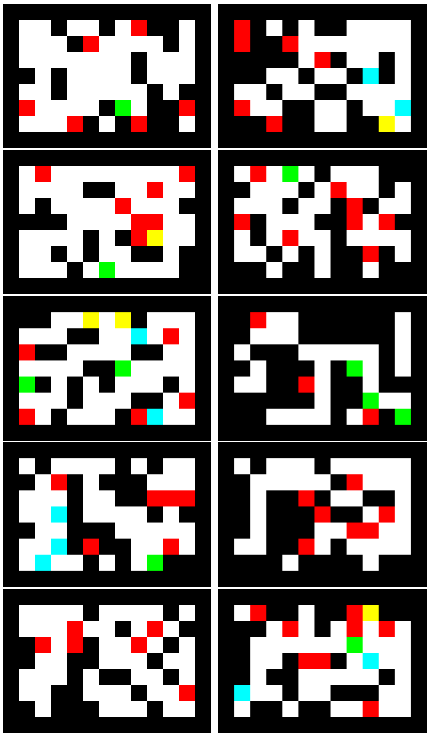
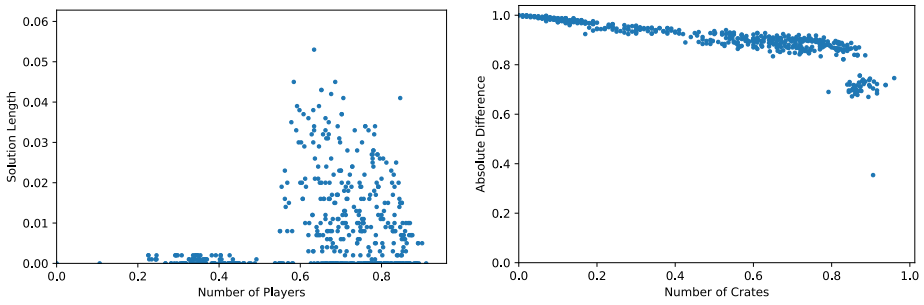


Figure 10.6: The evolved zelda generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. Black tiles are solid, white tiles are empty, green tiles are player, red tiles are enemies, yellow are keys, and cyan are doors. This generator has number of players fitness value equals to 0.79, number of keys fitness value equals to 0.67, number of doors fitness value equals to 0.7, number of enemies fitness value equals to 0.59, and solution length fitness value equal to 0.14



(a) (b)

Figure 10.7: The Pareto front for the Sokoban problem for different combination of the fitness functions.

```
{"explorers":[
{
    "type": "connect",
    "parameters": {
        "repeats": "1",
        "replace": "same",
        "directions": "plus",
        "entities": "empty",
        "out": "crate",
        "changes": "4"
    },
    "rules": [
        "noise>0.3,up(player) -> up(target)",
        "noise<=0.6, random>=0.5,vertl(crate) ->
            diag(empty)"
    ]
},
{
    "type": "connect",
    "parameters": {
        "repeats": "1",
        "replace": "same",
        "directions": "plus",
        "entities": "empty"
    },
    "rules": [
        "allfive(empty) -> down(solid)"
    ]
}]}
```

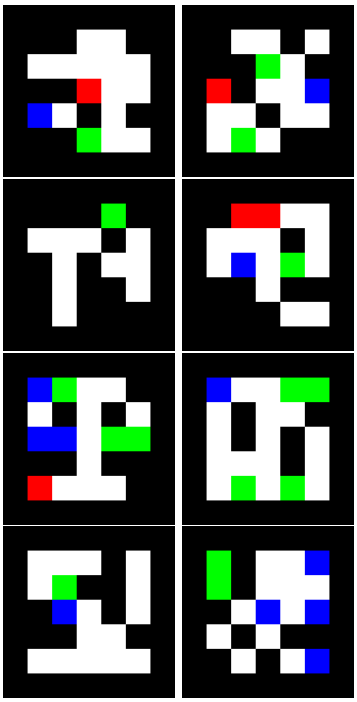


Figure 10.8: The evolved Sokoban generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. Black tiles are solid, white tiles are empty, green tiles are player, red tiles are crates, and blue tiles are targets. This generator has number of players fitness value equals to 0.69, number of crates fitness value equals to 0.66, absolute difference fitness value equal to 0.87, and solution length fitness value equal to 0.045.

crates can start in a locked position not allowing them to move even when all the constraints are satisfied. For example: the top left level on in figure 10.8 satisfies all the playability constraints but you can't win it as the crate (red tile) can't be moved.

Figure 10.7b shows the Pareto front between the number of crates fitness value and absolute difference fitness value. The relation is inversely proportional as having a higher number of crates fitness value causes the absolute difference value decrease. This is due to having more crates will increase the risk of having more errors (not having the number of crates equal to the number of targets) in the generated levels. Finding a generator that produces no crates and no targets is a very easy task (a generator that erases everything). This generator will always be in the front as it will always have an absolute difference fitness value equal to 1 which no other generator achieved.

Figure 10.8 shows the evolved Sokoban generator with 8 different generated levels using that generator. Similarly, we picked this generator as it has the highest solution length fitness value. The generator is a bit simple and it starts by trying to connect between isolated areas in the map and use noise function with some more constraints to either add empty or target

tiles. Then later it visits the new isolated tiles and adding solid tiles if it is too big of empty space. This generator also depends highly on the starting level as most of these conditions only valid in certain cases and not all the time.

10.4 Discussion

Comparing the results between both techniques, the PCGRL manages to surpass all the proposed technique as it finds successful levels in all the three problems while we only nearly solved the binary problem only. On the other hand, The evolution of Marahel resulted in a more understandable generator compared to the other technique introduced in the previous chapter. The interpretability of Marahel language is a big advantage as we can debug these generators or edit them easily. In the future, one could try to mix the PCGRL with Marahel by using reinforcement learning to train an agent to be able to generate Marahel scripts. This problem, as interesting as it sounds, is not easy as we will need a way to transform the Marahel generation problem into a Markov Decision Process. Besides that, we will also need a more complex network architecture (such as Transformers, LSTMS, etc) than the one used in chapter 9 to learn Marahel as the Marahel language is Turing Complete (it can represent cellular automata [148]).

Our representation and restrictions helped the evolution to find a small concise generator that can be understood easily but at the same time, it was harder to search the space. This can be noticed from our fitness functions, they created a Pareto front instead of working in tandem. It would be interesting to experiment with less restricted evolution and try to see if this will change the results.

We also noticed that having an initialization explorer that initializes the map before the generated explorers start, helped to find generators that react to the current initialization by erasing instead of adding (as most of the fitness functions need less number of entities than more of them). It would be interesting to remove that initialization generator and see if we can achieve different results that try to add more entities than erasing.

Another idea, we would like in the future to try to change the fitness function to be more about improvement (similar to path length improvement) instead of optimizing towards a certain value (like the number of players, the number of regions, etc). We think these types of fitness functions help the evolution to find more interesting layouts and levels as it doesn't depend on the starting state.

One last thing, the average operator (used to aggregate the fitness of the generated sample maps) sometimes biases the generation towards mediocre generators. We think that using a different type of operators like mixmin operator (mixing the average value and the minimum

value) might force the generator to move away from these mediocre generators.

10.5 Summary

This chapter introduced a multi-objective optimization method to evolve constructive level generators. The generators used Marahel [81] as their representation. We restricted the evolution to small size Marahel scripts to force the evolution to find understandable and efficient generators. The results show that our restrictions might have caused a Pareto front and not be able to find a generator that can achieve all the fitness functions 100%. We also see that only the binary problem was able to explore most of its Pareto front, while for Zelda and Sokoban the results were a subset of the full front. The final generator for the Binary problem was interesting as it resulted in these long loopy dungeons. On the other hand, the Zelda generator acted as an eraser erasing extra objects from the random initialization, while Sokoban generator just re-sampled the level from a different distribution that have a higher chance to be playable levels. One advantage of that technique compared to the neural networks represented in the previous chapter, the evolved generators were easy to read and understand which allowed us to analyze them and see their logic behind them which we couldn't do using neural networks.

Chapter 11

Conclusion

This is the last chapter of our journey but it is not the end. This work proposes new techniques, representations, and metrics that we used for different solutions to the general level generation problem. These contributions have their limitations and also they are not the only solutions. This chapter looks back on all the work and contributions in the thesis and puts down the stepping stones towards the upcoming steps to be explored in this problem.

11.1 Summary

In this thesis, we set towards a journey to find a solution for level generation for any video game. We introduced two different approaches to the problem. The first approach uses quality-diversity algorithms to find good levels for any game, while the second approach searches the level generator space using reinforcement learning or multi-objective evolution to find a level generator that can be used for our problem.

11.1.1 Searching For Content

We proposed using search-based procedural content generation [137] (SBPCG) as a framework for the general level generation problem. SBPCG is a flexible framework where it is possible to guarantee certain specifications in the generated content. In Chapter 5, we introduced a group of general constraints and playability metrics that can help us find playable levels. We used the FI2Pop genetic algorithm to satisfy these constraints. We applied this algorithm on multiple different games from the GVGAI framework to showcase the generality of these metrics. The algorithm was able to generate content that follows these rules but at the same time it was noisy, took a long time to find the solution, and there was no guarantee that different runs will generate different content.

We solved these problems in Chapter 6 by introducing a new quality -diversity algorithm called Constrained MAP-Elites. The algorithm was used to generate a big corpus of different levels for bullet hell games that are playable and fit different play styles. Having a big diverse corpus of levels was our solution for long generation time and low diversity between runs. Instead of generating a single level each time, we save time by generating a big set of diverse levels in a single run. At the same time, this technique introduced a new problem: what is a good general metric to measure diversity in games?

In Chapter 7, we presented a new diversity metric that can be used in any game. The diversity metric measures the difference between levels based on the different mechanics being triggered during a winning playthrough. We tested this new metric to generate small levels (scenes) for Super Mario Bros and four different GVGAI games. For games with a small amount of tracked mechanics, the generated levels are mechanically different and look pleasing and simple. While for games with more complex interactions and more tracked mechanics, the technique failed to generate a high number of diverse playable levels. In this approach, we decided to use simplicity as our fitness function as we believed it will make the generated levels easier to read and play. This fitness was able to make stuff look nice and clean but at the same time, made most of the levels structurally similar.

Finally, in chapter 8, we provided a method to generate bigger levels from the generated scenes. The method uses FI2Pop to stitch the generated scenes in a bigger level such that the output level follows a certain mechanical experience. The mechanical experience is automatically extracted by using an automated playing on a certain Super Mario Bros level. The generated levels have a highly similar play traces to the input play trace extracted from playing an original level. At the same time, the generated levels look structurally flat and uninteresting compared to their corresponding original level. The reason behind that is the used scenes are structurally flat and also the automated player used try to find the fastest shortest path to the goal avoiding firing interesting behavior.

Overall, part 2 succeeded in finding generic techniques to generate levels that can work between multiple games. These techniques can work out of the box directly with any game. The quality of the result heavily depends on the quality of the playtraces and the amount of tracked game mechanics. Having a small amount of different mechanics can work directly out of the box but increasing their numbers might need some adaptation. Having a human-like automated player that follows a certain persona, will lead towards more interesting levels than uniform/plain levels with minimum interactions.

11.1.2 Searching For Generators

Our second solution was searching for the level generator itself. This solution is harder because we need a good representation of the search space. In this work, we proposed two different representations for the level generator space. One is based on neural networks and the other is based on a description language. In chapter 9, we investigated the idea of using neural networks as a content generator and we trained them using reinforcement learning. We proposed three different ways (Narrow, Turtle, and Wide) to transform the level generation problem into a Markov Decision Process (MDP). We also used Proximal Policy Optimization to train the network on three different problems (Binary, Zelda, and Sokoban). The results show the ability of the network to learn to behave as a content generator by modifying the input from random noise to a playable level.

We provided a different representation of the problem in Chapter 10. We designed a new constructive level generation language called Marahel. We used this language as our level generator generation space. We used a multi-objective optimization algorithm (NSGA-II) to find generators for the same three problems used in PCGRL (Binary, Zelda, and Sokoban). The results are less impressive compared to the PCGRL. We were not able to find a generator that can solve any of these problems but we found different generators that can satisfy some of the objectives. Most of them worked as a filter by deleting some of the extra game objects produced in the random initialization.

Overall, neither of the two introduced approaches (searching for content or searching for generators) is a silver bullet to solve the problem of general level generation but each one has advantages and disadvantages. A core difference between both approaches is where the computation time is being used. For searching for content, all the computation time is used during generating the levels themselves. This forces the approach to be used in an offline generation which gives us more control in curating the content and making sure it fits with our target game. While searching for generators, all the computation time is used for finding the generator itself. This renders the approach more suitable for an online generation as we represent the found generators as a constructive generators. This makes the level generation part fast and could be done in real time but at the same time there is no guarantee on the quality of the generated levels from these generators.

11.2 Discussion

This thesis presents these techniques not to replace level designers but to assist them during the creation process. They suppose to serve as a tool for inspiration or as an assistant that

can help you during level creation. We wanted these techniques to be a framework and not a direct solution. Designers will have to tune these techniques toward their game to be able to work well and generate good results. For example: If designers want to generate levels for Lode Runner (an arcade game by Hudson Soft, 1983), the designer needs to provide the system with a way to evaluate levels in the form of an automated playing agent, reward function, fitness function, etc. Without these modifications, the system (in its current format) won't be able to work on any game. Trying to use generic techniques between different games usually end up not working on all games. For example: In chapter 7, our technique generated levels that don't use the main mechanic of the game (traveling through portals).

We believe that games are an art form where every game is unique and there are no two games that are similar. But because of that, we believe that there exists a subset of games in that space that is different from each other but can be captured by Artificial Intelligence techniques. Our techniques can go fully automated on this subset of games, that doesn't mean that these games are any better or worse than human-designed ones but it is very different. This is the same problem as any other art form. For example, GPT-3 [21] is a technical achievement and the generated text usually can't be differentiated from the human-written text. This doesn't mean that GPT-3 will replace writers or writing is not an art form. It is just a different subset of creative writing that people enjoy.

11.3 Future Work

This thesis explored different techniques, algorithms, and ways of measuring quality and diversity of levels. The proposed work is a stepping stone towards general level generation but there is more to be explored. This section presents some of these ideas and applications where this work can be expanded towards.

11.3.1 Searching for Content

In Part 2, we shown how quality diversity plays an important role in finding levels for any game, we used behavioral characteristics and fitness function that can easily be transferred between different games with a minimum amount of work. We think there is a lot more that needs to be analyzed and explored more:

- **Explore Different Fitness Functions:** We introduced entropy as a potential fitness function to reflect level simplicity but it usually leads to more uniform/plain levels with few structural changes between them. It would be interesting to try new fitness

functions that explores some concepts from other visual domains (such as art [10], photography [85], etc) about what make a content looks or feels good for humans.

- **Building Level Curriculum:** The work presented in chapter 7 was originally motivated by tutorial generation. This is why the fitness function revolves around finding the simplest level that contains specific game mechanics. It would be interesting to sequence the generated levels in an order that can work as a tutorial level for human players [56]. We could also use them to build a curriculum for reinforcement learning agents that will help them to learn and play games better [13, 70].

- **Enhancing Playability Checking:** The bottleneck for our proposed technique is the full dependency on the automated game playing agent. The game-playing agent identify the constraints for level playability and its playtrace is used to collect triggered game mechanics for diversity. If the agent is not optimal, it usually takes a long time to finish which stale the whole generation process and it would also produce a lot, unnecessary triggered game mechanics. An interesting direction would be using an automated playing agent that can adapt itself with every introduced new level. This could be done using reinforcement learning [70], evolving playing agents in tandem with the levels [145], imitation learning [46], etc.

- **Automatically Detecting Mechanics:** In this thesis, we assumed that the triggered mechanics can be easily extracted from a playtrace. This is a valid assumption as designers and developers can always add loggers that record all the triggered mechanics during game playing. Sometimes adding all the code for logging might be an overkill for the project. A future direction is to explore how to automatically extract the triggered game mechanics by analyzing the game screen/states. This could be done by borrowing from the image processing field and use techniques similar to the detection of soccer events [49], game highlights [34], etc.

11.3.2 Searching for Generators

In this thesis, we only scratched the surface of the searching for generators. There is a lot of different stuff that can be explored:

- **Exploring Different Representations:** We only explored two types of representations for generators: neural networks and Marahel scripts. These are not the only possible representations, we could represent the generator as a group of constraints that are satisfied using a constraint satisfaction algorithm such as Answer Set Programming [122]. We can also represent it as a group of Wang tiles relationships where

we try to find a set of tiles that can lead to interesting generation [139]. We think that each of these new representations would have some advantages and limitations which should be analyzed and compared with respect to each other.

- **Trying New Techniques:** The introduced methods are not yet fully explored, there is much more that can be done with them. For the PCGRL, it is interesting to explore all the different RL training algorithm (such as Self-Play [121], NeuroEvolution [28], Imitation Learning [43], etc) and their effect on the found generators. For the evolution of Marahel, we only explored using NSGA-II for searching a limited space, we can try to use Quality Diversity techniques (such as Constrained Novelty Search [89], MAP-Elites [96], etc) and analyze their effect on the found generators.

- **Testing on Harder Games:** Our algorithms were explored on 3 different types of problems. These problems vary in their required skill to solve but at the same time, they are simplified. Sokoban levels are only 5x5, Zelda levels are from the GVGAI framework which is a demake of the original Legend of Zelda, and Binary levels are simple with only two types of tiles. It would be interesting to test these techniques on games that have bigger maps such as Super Mario Bros, games that require dexterity such as Bullet Hells, etc. We think the currently proposed techniques will need some modifications before being able to use it on more complex problems. A proposed solution could be dividing the levels into smaller areas similar to the proposed work in Chapter 7 and later combine them. Another direction is to divide the generation problem into smaller sub-problems then find a generator to solve each of these problems. This direction is similar to the work by Green et al. [25] where one agent was responsible for generating the layout and the other responsible for adding the other sprites given the layout.

- **Finding the Reward/Fitness Function:** In all the proposed work, we were giving the fitness/reward function as an input and we are trying to find generators that maximize these functions. A potential direction is to find these functions automatically. We could try to learn the function from a small human data set such as an example level. Another direction is to abandon objectives and use diversity as the core element towards searching for new generators. We could also use mixed-initiative techniques where user selection acts as the fitness/reward function.

Bibliography

[1] Chad Adams and Sushil Louis. Procedural maze level generation with evolutionary cellular automata. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

[2] Zach Aikman. Generating procedural dungeons in galak z. `https://www.youtube.com/watch?v=ySTpjT6JYFU`, 2014.

[3] Alberto Alvarez, Steve Dahlskog, Jose Font, and Julian Togelius. Empowering quality diversity in dungeon design with interactive constrained map-elites. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[4] Anna Anthropy and Naomi Clark. *A game design vocabulary: Exploring the foundational principles behind good game design*. Pearson Education, 2014.

[5] Daniel Ashlock. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 289–296. IEEE, 2010.

[6] Daniel Ashlock. Evolvable fashion-based cellular automata for generating cavern systems. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 306–313. IEEE, 2015.

[7] Daniel Ashlock, Colin Lee, and Cameron McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, 2011.

[8] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[9] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. CRC Press, 1997.

[10] Molly Bang. *Picture this: How pictures work*. Chronicle Books, 2016.

[11] Spencer Beaupre, Thomas Wiles, Sean Briggs, and Gillian Smith. A design pattern approach for multi-game level generation. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[12] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[13] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[14] Debosmita Bhaumik, Ahmed Khalifa, Michael Cerny Green, and Julian Togelius. Tree search vs optimization approaches for map generation. *arXiv preprint arXiv:1903.11678*, 2019.

[15] Beranek Bolt and Robert Newman. Logo (programming language). `https://en.wikipedia.org/wiki/Logo_(programming_language)`, 1967.

[16] Philip Bontrager, Ahmed Khalifa, Damien Anderson, Matthew Stephenson, Christoph Salge, and Julian Togelius. "superstition" in the network: Deep reinforcement learning plays deceptive games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 10–16, 2019.

[17] Philip Bontrager, Ahmed Khalifa, Andre Mendes, and Julian Togelius. Matching games and algorithms for general video game playing. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[18] Philip Bontrager, Aditi Roy, Julian Togelius, Nasir Memon, and Arun Ross. Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–9. IEEE, 2018.

[19] William F Brewer and Edward H Lichtenstein. Event schemas, story schemas, and story grammars. *Center for the Study of Reading Technical Report; no. 197*, 1980.

[20] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv*, 2020.

[22] Cameron Browne. Uct for pcg. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[23] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[24] Joel Burgess. How we used iterative level design to ship skyrim and fallout 3. https://www.youtube.com/watch?v=PhW8CY8XkFg, 2014.

[25] Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. Two-step constructive approaches for dungeon generation. In *Foundation of Digital Games at PCGWorkshop*. ACM, 2019.

[26] Megan Charity, Michael Cerny Green, Ahmed Khalifa, and Julian Togelius. Mech-elites: Illuminating the mechanic space of gvgai. *arXiv preprint arXiv:2002.04733*, 2020.

[27] Zhengxing Chen, Christopher Amato, Truong-Huy D Nguyen, Seth Cooper, Yizhou Sun, and Magy Seif El-Nasr. Q-deckrec: A fast deck recommendation system for collectible card games. In *Computational Intelligence and Games*. IEEE, 2018.

[28] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to basics: Benchmarking canonical evolution strategies for playing atari. *arXiv preprint arXiv:1802.08842*, 2018.

[29] Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics (TOG)*, 22(3):287–294, 2003.

[30] Kate Compton, Benjamin Filstrup, et al. Tracery: Approachable story grammar authoring for casual users. In *Seventh Intelligent Narrative Technologies Workshop*, 2014.

[31] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: an author-focused generative text tool. In *International Conference on Interactive Digital Storytelling*, pages 154–161. Springer, 2015.

[32] Kate Compton, Johnathan Pagnutti, and Jim Whitehead. A shared language for creative communities of artbots. In *Proceedings of the 2017 Co-Creation Workshop*, 2017.

[33] Michael Cook, Mirjam Eladhari, Andy Nealen, Mike Treanor, Eddy Boxerman, Alex Jaffe, Paul Sottosanti, and Steve Swink. Pcg-based game design patterns. *arXiv preprint arXiv:1610.03138*, 2016.

[34] David Cottrell. System and method for automated creation of video game highlights, August 20 2013. US Patent 8,515,253.

[35] Steve Dahlskog and Julian Togelius. Patterns as objectives for level generation. In *International Conference on Foundations of Digital Games*. ACM, 2013.

[36] Steve Dahlskog, Julian Togelius, and Mark J Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, pages 200–206, 2014.

[37] Charles Darwin's. On the origin of species. *published on*, 24, 1859.

[38] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[39] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[40] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. *GitHub repository*, 2017.

[41] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, pages 1–8, 2010.

[42] Olve Drageset, Mark HM Winands, Raluca D Gaina, and Diego Perez-Liebana. Optimising level generators for general video game ai. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[43] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in neural information processing systems*, pages 1087–1098, 2017.

[44] Sam Earle. Using fractal neural networks to play simcity 1 and conway's game of life at variable scales. In *Experimental AI in Games Workshop*, 2019.

[45] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. In *Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, 2013.

[46] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[47] Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1033–1038. IEEE, 1999.

[48] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[49] Ahmet Ekin, A Murat Tekalp, and Rajiv Mehrotra. Automatic soccer video analysis and summarization. *IEEE Transactions on Image processing*, 12(7):796–807, 2003.

[50] Steve Engels, Tiffany Tong, and Fabian Chan. Automatic real-time music generation for games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[51] Lucas Ferreira and Claudio Toledo. A search-based approach for generating angry birds levels. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[52] Luis Flores and David Thue. Level of detail event generation. In *International Conference on Interactive Digital Storytelling*, pages 75–86. Springer, 2017.

[53] Matthew C Fontaine, Scott Lee, LB Soros, Fernando De Mesentier Silva, Julian Togelius, and Amy K Hoover. Mapping hearthstone deck spaces through map-elites with sliding boundaries. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 161–169, 2019.

[54] Daniele Gravina, Antonios Liapis, and Georgios Yannakakis. Surprise search: Beyond objectives and novelty. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 677–684, 2016.

[55] Daniele Gravina, Antonios Liapis, and Georgios N Yannakakis. Constrained surprise search for content generation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[56] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, Andy Nealen, and Julian Togelius. Atdelfi: automatically designing legible, full instructions for games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[57] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Andy Nealen, and Julian Togelius. Generating levels that teach mechanics. In *International Conference on the Foundations of Digital Games*, page 55. ACM, 2018.

[58] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, and Julian Togellius. " press space to fire": Automatic video game tutorial generation. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2017.

[59] Jason Grinblat. Markov by candelight. `https://www.youtube.com/watch?v=3AjlsTtrfVY`, 2016.

[60] Maxim Gumin. Wave function collapse. `https://github.com/mxgmn/WaveFunctionCollapse`, 2016.

[61] Matthew Guzdial, Boyang Li, and Mark O Riedl. Game engine learning from video. In *IJCAI*, pages 3707–3713, 2017.

[62] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark O Riedl. Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2019.

[63] Matthew Guzdial, Nicholas Liao, and Mark Riedl. Co-creative level design via machine learning. *arXiv preprint arXiv:1809.09420*, 2018.

[64] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[65] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Evolving content in the galactic arms race video game. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 241–248. IEEE, 2009.

[66] Christoffer Holmgard, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Automated playtesting with procedural personas with evolved heuristics. *IEEE Transactions on Games*, 2018.

[67] Aaron Isaksen, Drew Wallace, Adam Finkelstein, and Andy Nealen. Simulating strategy and dexterity for puzzle games. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 142–149. IEEE, 2017.

[68] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC Workshop on Computational Creativity and Games*, 2016.

[69] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

[70] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.

[71] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[72] Isaac Karth. Elite (1984). `https://procedural-generation.tumblr.com/post/112509130817/elite-1984-elite-created-by-ian-bell-and-david`, 2015.

[73] Paul Kent and Juergen Branke. Bop-elites, a bayesian optimisation algorithm for quality-diversity search. *arXiv preprint arXiv:2005.04320*, 2020.

[74] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N Yannakakis. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 335–341. IEEE, 2012.

[75] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. *arXiv preprint arXiv:2001.09212*, 2020.

[76] Ahmed Khalifa and Magda Fayek. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*, 2015.

[77] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 796–803, 2019.

[78] Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 170–177. IEEE, 2017.

[79] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. Talakat: Bullet hell generation through constrained map-elites. In *The Genetic and Evolutionary Computation Conference*. ACM, 2018.

[80] 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*, pages 253–259, 2016.

[81] Ahmed Khalifa and Julian Togelius. Marahel: A language for constructive level generation. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2017.

[82] Ahmed Khalifa and Julian Togelius. Multi-objective level generator generation with marahel. *arXiv preprint arXiv:2005.08368*, 2020.

[83] 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. *European Journal of Operational Research*, 190(2):310–327, 2008.

[84] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[85] Bert Krages. *Photography: the art of composition*. Simon and Schuster, 2012.

[86] Joel Lehman, Sebastian Risi, and Jeff Clune. Creative generation of 3d objects with deep learning and innovation engines. In *Proceedings of the 7th International Conference on Computational Creativity*, 2016.

[87] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[88] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Designer modeling for sentient sketchbook. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[89] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Constrained novelty search: A study on game content generation. *Evolutionary computation*, 23(1):101–129, 2015.

[90] Yang Liu. A fast and elitist multi-objective particle swarm algorithm: Nspso. In *2008 IEEE International Conference on Granular Computing*, pages 470–475. IEEE, 2008.

[91] Simon M Lucas and Vanessa Volz. Tile pattern kl-divergence for analysing and evolving game levels. In *GECCO*, 2019.

[92] Nicholas McDonald. Markov chains for procedural buildings. `https://weigert.vsos.ethz.ch/2019/10/30/markov-chains-for-procedural-buildings/`, 2019.

[93] Fawzi Mesmar. *Al-Khallab on the Art of Game design*, volume 1. CreateSpace Independent Publishing Platform, 2018.

[94] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[95] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

[96] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[97] Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, pages 207–212. IEEE, 2015.

[98] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 959–966, 2015.

[99] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. General video game evaluation using relative algorithm performance profiles. In *European Conference on the Applications of Evolutionary Computation*, pages 369–380. Springer, 2015.

[100] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. Towards generating arcade game rules with vgdl. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 185–192. IEEE, 2015.

[101] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[102] Michael O'Neil and Conor Ryan. Grammatical evolution. In *Grammatical evolution*, pages 33–47. Springer, 2003.

[103] Johnathan Pagnutti, Kate Compton, and Jim Whitehead. Do you like this art i made you: introducing techne, a creative artbot commune. In *Proceedings of 1st International Joint Conference of DiGRA and FDG*, 2016.

[104] Andrew Pech, Philip Hingston, Martin Masek, and Chiou Peng Lam. Evolving cellular automata for maze generation. In *Australasian conference on artificial life and computational intelligence*, pages 112–124. Springer, 2015.

[105] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

[106] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2015.

[107] Marcus Persson. Infinite mario bros. *Online Game). Last Accessed: December*, 11, 2008.

[108] David Plans and Davide Morelli. Experience-driven procedural music generation for games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):192–198, 2012.

[109] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[110] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

[111] Sebastian Risi, Joel Lehman, David B D'Ambrosio, Ryan Hall, and Kenneth O Stanley. Petalz: Search-based procedural content generation for the casual gamer. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):244–255, 2015.

[112] Graeme Ritchie. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines*, 17(1):67–99, 2007.

[113] Conor Ryan, John James Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer, 1998.

[114] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[115] Noor Shaker, Mohammad Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[116] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[117] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, et al. The 2010 mario ai championship: Level generation track. *Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.

[118] Noor Shaker, Georgios N Yannakakis, and Julian Togelius. Towards player-driven procedural content generation. In *Proceedings of the 9th conference on Computing Frontiers*, pages 237–240, 2012.

[119] Tanya Short and Tarn Adams. *Procedural generation in game design*. CRC Press, 2017.

[120] Miguel Sicart. Defining game mechanics. *Game Studies*, 8(2):n, 2008.

[121] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[122] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[123] Gillian Smith, Alexei Othenin-Girard, Jim Whitehead, and Noah Wardrip-Fruin. Pcg-based game design: creating endless web. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 188–195, 2012.

[124] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182, 2009.

[125] Sam Snodgrass and Santiago Ontañón. Experiments in map generation using markov chains. In *FDG*, 2014.

[126] Sam Snodgrass and Santiago Ontanón. Learning to generate video game maps using markov models. *IEEE transactions on computational intelligence and AI in games*, 9(4):410–422, 2016.

[127] Irwin Sobel and Gary Feldman. A 3x3 isotropic gradient operator for image processing. *a talk at the Stanford Artificial Project in*, pages 271–272, 1968.

[128] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In *European Conference on the Applications of Evolutionary Computation*, pages 131–140. Springer, 2010.

[129] Adam Summerville and Michael Mateas. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*, 2016.

[130] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[131] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487*, 2016.

[132] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[133] Joshua Taylor and Ian Parberry. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pages 5–12, 2011.

[134] Julian Togelius, Tróndur Justinussen, and Anders Hartzen. Compositional procedural content generation. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, pages 1–4, 2012.

[135] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[136] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The mario ai championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

[137] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[138] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. Bootstrapping conditional gans for video game level generation. *arXiv preprint arXiv:1910.01603*, 2019.

[139] Nick Turner. Seeded exploration of wang tiled environments. `https://www.gamasutra.com/blogs/NickTurner/20170411/295783/Seeded_Exploration_of_Wang_Tiled_Environments.php`, 2017.

[140] Neil Urquhart and Emma Hart. Optimisation and illumination of a real-world workforce scheduling and routing application (wsrp) via map-elites. In *International Conference on Parallel Problem Solving from Nature*, pages 488–499. Springer, 2018.

[141] Roland Van der Linden, Ricardo Lopes, and Rafael Bidarra. Designing procedurally generated levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[142] Tony Veale and Mike Cook. *Twitterbots: Making Machines that Make Meaning*. MIT Press, 2018.

[143] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228, 2018.

[144] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 52–67, 2018.

[145] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Poet: open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 142–151, 2019.

[146] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[147] Wikipedia. Yavalath (board game). `https://de.wikipedia.org/wiki/Yavalath`. Accessed: November 3, 2015.

[148] Stephen Wolfram. Cellular automata. *Los Alamos Science*, 9:42, 1983.

[149] Stephen Wolfram. Theory and applications of cellular automata. *World Scientific*, 1986.

[150] Georgios N Yannakakis, Antonios Liapis, and Constantine Alexopoulos. Mixed-initiative co-creativity. In *Foundations of Digital Games*. ACM, 2014.

[151] Georgios N Yannakakis and Héctor P Martínez. Ratings are overrated! *Frontiers in ICT*, 2:13, 2015.

[152] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Rotation, translation, and cropping for zero-shot generalization. In *Conference on Games*, 2020.

[153] Derek Yu. *Spelunky: Boss Fight Books# 11*, volume 11. Boss Fight Books, 2016.

[154] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.