AUTOMATIC VIDEO GAME TUTORIAL 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

Michael Cerny Green

September 2022

AUTOMATIC VIDEO GAME TUTORIAL 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

Michael Cerny Green

September 2022

Approved:

Department Chair Signature

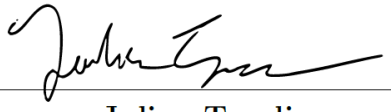
July 6, 2022

Date

University ID: N18400783
Net ID: mcg520

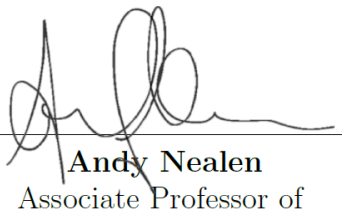
Approved by the Guidance Committee:

Major: Computer Science

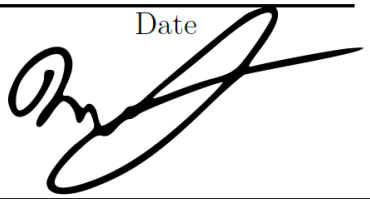


Julian Togelius
Associate Professor of
Computer Science and Engineering
6/21/2022
Date

Gillian Smith
Associate Professor of
Computer Science
6/22/2022
Date



Andy Nealen
Associate Professor of
Cinematic Arts and Computer Science
6/21/2022
Date



Michael Mateas
Professor of
Computational Media
6.27.2022
Date

Vita

Michael Cerny Green was born in Poughkeepsie, New York, USA on May 30th, 1994. He earned a Bachelor of Science in Computer Science and Business and a Bachelor of Arts in Classical Civilizations from Lehigh University in 2016. Throughout college, he experimented with programming video games and helped organize mobiLehigh, an annual mobile game jam.

In 2016, Michael matriculated into the New York University Tandon Engineering School's Ph.D. program within the Game Innovation Lab with lab director Julian Togelius and former director Andy Nealen. Since then, he has explored the intersection of video games, education, and AI with video game tutorial generation. He has published papers in international conferences and workshops like IEEE Conference on Games (COG), the International Conference on the Foundations of Digital Games (FDG), the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), the International Conference on Computational Creativity (ICCC), the World Congress on Computational Intelligence (WCCI), the Genetic and Evolutionary Computational Conference (GECCO), and AAAI conference on artificial intelligence (AAAI), as well as journals including IEEE Transactions on Games (TOG), KI-Künstliche Intelligenz, and Scientific Reports. Funding for tuition, travel, and persona expenses has been provided by various sources, including the GAANN Fellowship, the Tandon School Fellowship, Prof. Julian Togelius, and the OriGenAI education program.

From 2018-2020, Michael worked as a founding A.I. researcher and engineer at the energy startup OriGenAI. From 2020 until 2022, he was the company's AI software and technical project manager, responsible for building and deploying a real-time, scalable AI training and inferencing pipeline. During his time at NYU outside of OriGen, he has been a teaching assistant for courses on Artificial Intelligence, AI for Games, and Game Design.

Acknowledgements

I would like to thank my advisor Julian Togelius for being not only an amazing supervisor and boss, but also for being a great friend and mentor to me. I thank Andy Nealen, who helped guide the early days of my research in games. I also thank one of my greatest friends and colleagues Ahmed Khalifa, who is author on most if not all of the papers this thesis is founded upon, and whose support got me through the most difficult parts of this long journey. I thank my dissertation committee: Julian Togelius, Andy Nealen, Gillian Smith, and Michael Mateas. I thank all my co-authors for building great projects with me: Gabriella Barros, Philip Bontrager, Tiago Machado, Rodrigo Canaan, M Charity, Debosmita Bhaumik, Christoffer Holmgård, Ruben Rodriguez Torrado, Sam Earle, Lisa Soros, Luvneesh Mugrai, Antonios Liapis, Christoph Salge, Niels Justesen, Diego Perez-Liebana, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, Sebastian Risi, Athoug Alsoughayer, Divyesh Surana, Tyler Friesen, Sebastien Matringe, Pablo Ruiz, Luis Cueto-Felgueroso, Victoria Yen, Dipika Rajesh, Maria Edwards, Benjamin Sergent, Vibhor Kumar, Pushyami Shandilya, Claus Aranha, Adrian Brightmoore, Sean Butler, Michael Cook, Hagen Fischer, Christian Guckelsberger, Jupiter Hadley, Jean-Baptiste Hervé, Mark R Johnson, Quinn Kybartas, David Mason, Mike Preuss, Tristan Smith, and Ruck Thawonmas.

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: Ming Jin, Tae Jong Choi, Catalina Jaramillo, Graham Todd, Matthew Siper, Aaron Dharna, Chang Ye, Maxwell Sirotin, Chris DiMauro, Erica Ribeiro, Bruna Oliveira, Ramsey Nasser, Kaho Abe, Christopher Michaels, and Valeria Bontrager. I thank Annette Vera for her amazing support and love, without whom I would have gone insane a *long* time ago. I thank all my family for being my support network: my parents Mary and Robert, my sisters Rachel and Theresa, my brother Andrew, and of course Harper and Snowbelle. Finally, I would like to thank all the friends, new and old, for their support and for making life fun during this period of my life. I learned a lot from from all of you, and you helped make this possible.

Michael Cerny Green

September 2022

To my future self, so that I never forget what I'm capable of

ABSTRACT

AUTOMATIC VIDEO GAME TUTORIAL GENERATION

by

Michael Cerny Green

Advisor: Prof. Julian Togelius, Ph.D.

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy (Computer Science)

September 2022

This thesis explores techniques for automated tutorial generation for video games using artificial intelligence. We begin with an overview of tutorials (both roles and types) as well as a review of PCG algorithms and AI frameworks used in this thesis. We introduce several tree search methods for critical mechanic discovery, i.e. finding mechanics that can be taught with tutorials. We compare the discovery methods using four games in the General Video Game Artificial Intelligence (GVG-AI) corpus (Zelda, Solarfox, Plants, and RealPortals). We then present a theory of Game Mechanic Alignment (GMA) which permits mechanic analysis in regards to systemic and agential influences. We demonstrate the utility of GMA using three well-known games (Super Mario, Minecraft, and Bioshock) and present a method of GMA estimation which we apply to a dataset of human playtraces for the research game Minidungeons 2 to show off its ability to extract personalized mechanic insight. We present a method of rapid, accurate play persona classification in Minidungeons 2 using mechanic frequencies in order to identify players using their behavior. We demonstrate how mechanics can be used to build instruction-based and demonstration-based tutorials using the AtDelfi System for GVG-AI games. We follow up with the capability to generate tutorial scenes with single-objective optimization and quality diversity using mechanics or procedural personas modeled after playstyles using the Mario AI, GVG-AI, and Minidungeons

2 frameworks. We then show how these scenes can be stitched together into larger levels/sequential scenes in Mario AI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We often use the term "Artificial Intelligence" to describe an automated system that can solve problems, which it often does through learning. Within the domain of video games, we see this reflected in the many roles for which AI has been used: generating game content, playing games, and modeling players [176]. We can see AI occasionally being used in somewhat more obscure roles like spectacle, trainee or co-creator [167].

I think that AI can do more than just learn: I think that AI can *teach*. In this dissertation, I will show you how AI can be used to generate content from which we can learn, instead of just having AI learn from us. More specifically, I will introduce the research area of automated video game tutorial generation using artificial intelligence. My three main objectives are to explore methods with which to find mechanics to teach within tutorials, to present methods of play analysis, and to define different tutorial generation methods for instructions, demonstrations, and levels.

1.1 Thesis Statement

The computational creation of tutorials is a novel research area. Past expeditions into it have been limited to specific programs, tools, or games. Methods able to generalize across games and domains have not yet been discovered. I propose the following thesis statement which drives the core discussion of this dissertation:

We can automatically generate tutorials and tutorial levels for video games using mechanics and play personas.

1.2 Contributions

The published material in this thesis is extensive, and as such I find it prudent to summarize the areas where it adds value. My thesis both draws from and expands upon several applications of AI research, including procedural content generation, optimization using search-based methods, AI as gameplaying agents, and procedural personas. I claim that the most important contributions of this thesis are the following:

- **Mechanic graphs and critical mechanic discovery:** Tutorials teach mechanics to players, so organizing mechanic relationships using mechanic graphs can be an invaluable tool for tutorial generation. This work also presents two simple methods to search mechanic graphs for mechanics to teach within tutorials.

- **A theory of Game Mechanic Alignment:** Players play differently depending on their gameplay goals and what they consider to be enjoyable. Game Mechanic Alignment theory estimates these player behavioral differences on a mechanical level.

- **Play Persona classification:** Knowing who players are is a requirement to building content personalized for them. We demonstrate how a mechanic frequency representation can be used to quickly and accurately classify players by how they play.

- **A System for Tutorial Card Generation:** The AtDelfi System is a tutorial generation engine, capable of building tutorial cards with instructions and demonstrations of mechanics for any of the 100+ games in the GVG-AI framework. Although GVG-AI grants certain affordances which make mechanic extraction relatively easy, theoretically AtDelfi can also be used for any game in which critical mechanics can be extracted.

- **Techniques for Tutorial Level Generation using Mechanics and Procedural Personas:** Procedural tutorial level generation is the automated production of levels that teach the players game mechanics and how to play in different ways. We present experiments within multiple frameworks for generating tutorial levels, even providing a way to stitch small tutorial levels together into larger sequences of levels.

1.3 Publications

The work presented in this dissertation originates from the theory and experimental research of 10 papers, all of which have been peer-reviewed and accepted/published in the proceedings of various international conferences and workshops. The papers are listed below in chronological order, along with the chapter(s) in which they are presented:

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, and Julian Togellius. "Press Space to Fire": Automatic Video Game Tutorial Generation [64]. In the Proceedings of the the Artificial Intelligence and Interactive Digital Entertainment Conference through the Experimental AI for Games Workshop, 2017. Briefly discussed in Chapter 6.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Andy Nealen, and Julian Togelius. Generating levels that teach mechanics [63]. In the Proceedings of Foundations of Digital Games through the Procedural Content Generation Workshop, 2018. Briefly discussed in Chapter 10.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, Andy Nealen, and Julian Togelius. AtDELFI: automatically designing legible, full instructions for games [60]. In the Proceedings of Foundations of Digital Games, 2018. Extensively discussed in Chapter 6 and in Chapter 9.

- Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design [96]. In the Proceedings of The Genetic and Evolutionary Computation Conference, 2019. Extensively discussed in Chapters 10 and 11.

- M Charity, Michael Cerny Green, Ahmed Khalifa, and Julian Togelius. Mech-Elites: Illuminating the Mechanic Space of GVG-AI [20]. In the Proceedings of Foundation of Digital Games, 2020. Extensively discussed in Chapter 11.

- Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, and Julian Togelius. Automatic Critical Mechanic Discovery Using Playtraces in Video Games [62]. In the Proceedings of Foundation of Digital Games, 2020. Extensively discussed in Chapter 6.

- Michael Cerny Green, Luvneesh Mugrai, Ahmed Khalifa, and Julian Togelius. Mario Level Generation From Mechanics Using Scene Stitching [68]. In the Proceedings of Conference on Games, 2020. Extensively discussed in Chapter 13.

- Michael Cerny Green, Ahmed Khalifa, Philip Bontrager, Rodrigo Canaan, and Julian Togelius. Game Mechanic Alignment Theory and Discovery [65]. In the Proceedings of the Foundation of Digital Games, 2021. Extensively discussed in Chapter 7.

- Michael Cerny Green, Ahmed Khalifa, M Charity, Debosmita Bhaumik, and Julian Togelius. Predicting Personas Using Mechanic Frequencies and Game State Traces [66]. Accepted to the World Congress on Computational Intelligence, 2022. Extensively discussed in Chapter 8.

- Michael Cerny Green, Ahmed Khalifa, M Charity, and Julian Togelius. Persona-driven Dominant-Submissive Map (PDSM) Generation for Tutorials [67]. Accepted to the Foundations of Digital Games, 2022. Extensively discussed in Chapter 12.

Additionally, 14 papers were published during my PhD which in many cases inform the papers above and relate to the topic of the thesis, all of which are also all peer-reviewed and published. However, they are not included in my thesis for length or unrelated subject matter and are listed here in chronological order:

- Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. General video game rule generation [97]. In the Proceedings of Computational Intelligence and Games, 2017.

- Michael C. Green, Gabriella A. B. Barros, Antonios Liapis, and Julian Togelius. DATA Agent [58]. In the Proceedings of the Foundations of Digital Games, 2018.

- Gabriella A. B. Barros, Michael C. Green, Antonios Liapis, and Julian Togelius. Who killed Albert Einstein? From Open Data to murder mystery games [10]. In IEEE Transactions on Computational Intelligence and AI in Games, 2018.

- Christoffer Holmgard, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics [77]. In IEEE Transactions on Computational Intelligence and AI in Games, 2018.

- Christoph Salge, Michael Cerny Green, Rodrigo Canaan, and Julian Togelius. Generative Design in Minecraft (GDMC), Settlement Generation Competition [138]. In the Proceedings of the Foundations of Digital Games through the Procedural Content Generation Workshop, 2018.

- Gabriella A. B. Barros, Michael C. Green, Antonios Liapis, and Julian Togelius. Data-driven Design: A Case for Maximalist Game Design [9]. In the Proceedings of the International Conference on Computational Creativity, 2018.

- Michael Cerny Green, Benjamin Sergent, Pushyami Shandilya, Vibhor Kumar. Evolutionarily-Curated Curriculum Learning for Deep Reinforcement Learning Agents [70]. In the Proceedings of the Association for the Advancement of Artifical Intelligence through the Deep Reinforcement Learning in Games Workshop, 2019

- Debosmita Bhaumik, Ahmed Khalifa, Michael Cerny Green, and Julian Togelius. Tree Search vs Optimization Approaches for Map Generation [11]. In the Proceedings of Artificial Intelligence and Interactive Digital Entertainment, 2020.

- Christoph Salge, Christian Guckelsberger, Michael Cerny Green, Rodrigo Canaan, and Julian Togelius. Generative design in Minecraft: Chronicle Challenge [139]. In the Proceedings of International Conference on Computational Creativity, 2019

- Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. Two-step constructive approaches for dungeon generation [59]. In the Proceedings of Foundations of Digital Games through the Procedural Content Generation Workshop, 2019.

- Michael Cerny Green, Christoph Salge, and Julian Togelius. Organic Building Generation in Minecraft [69]. In the Proceedings of Foundations of Digital Games through the Procedural Content Generation Workshop, 2019.

- Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. The AI Settlement Generation Challenge in Minecraft [137]. In KI-Künstliche Intelligenz, 2020.

- Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. Bootstrapping Conditional GANs for Video Game Level Generation [164]. In the Proceedings of Conference on Games, 2020.

- Ruben Rodriguez-Torrado, Pablo Ruiz, Luis Cueto-Felgueroso, Michael Cerny Green, Tyler Friesen, Sebastien Matringe, and Julian Togelius.

Physics-informed Attention-based Neural Network for Solving Non-linear Partial Differential Equations [135]. Scientific Reports, 2022.

1.4 Outline

This work can be divided into three parts, each detailing a critical component of tutorial generation. The first part provides context for tutorial generation by presenting taxonomies on tutorial types and roles, related background work, the frameworks used in this thesis, and the algorithms utilized for tutorial generation. The second part defines methods for determining the relevancy of different mechanics as tutorial content and techniques for analyzing player behavior. The last part describes various ways to automatically generate different tutorial types. Figure 1.1 displays the project flow of this thesis and how each of the projects provide the ability to advance from game mechanics and playtraces to generated tutorials. A more detailed breakdown follows:

- **Part 1: Context** introduces the necessary techniques, domains, and representations used in this work
  - **Chapter 2: A Tutorial on Tutorials** covers the evolution of game tutorials across time and the role tutorials are meant to play. It also defines the tutorial types that this thesis focuses on generating.
  - **Chapter 3: Procedural Content Generation** presents an overview of PCG methods in games.
  - **Chapter 4: Frameworks** introduces the different frameworks that are used as test beds for tutorial generation projects.
  - **Chapter 5: Algorithms** explains the different algorithms that this thesis uses or builds upon.
- **Part 2: Mechanic Discovery and Analysis** introduces methods for game mechanic discovery and analysis.
  - **Chapter 6: Mechanic Discovery** defines the concept of mechanic graphs and two methods for searching mechanic graph spaces for relevant tutorial mechanics.
  - **Chapter 7: Mechanic Analysis** details a theory of Game Mechanic Alignment, a framework within which to analyze mechanics according to systemic and agential influences.

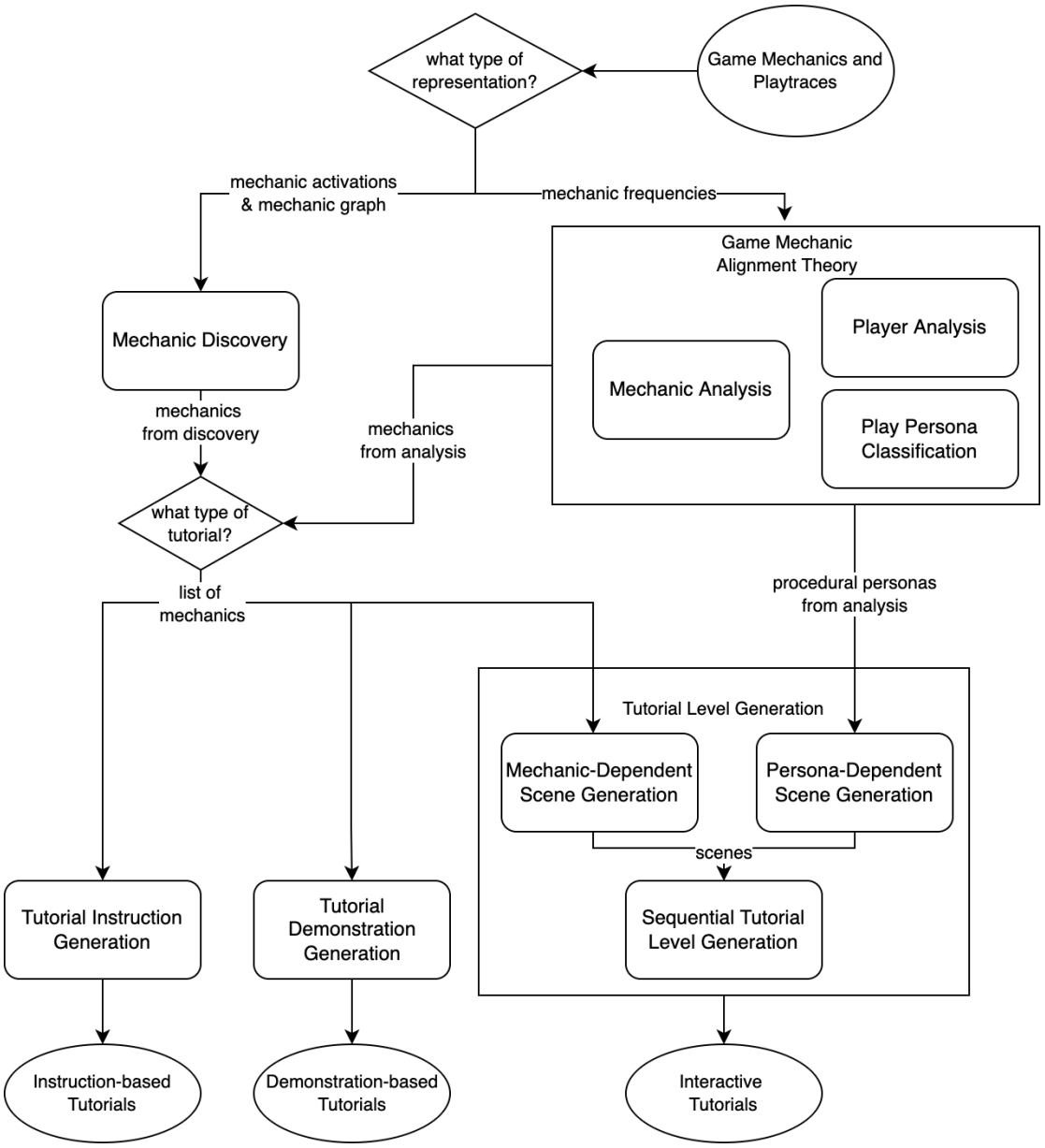


Figure 1.1: A map of how every project in this thesis fits together

  - **Chapter 8: Play Persona Classification** describes a set of experiments which use mechanical frequencies to classify players according to their most similar, designer-defined play personas.

- **Part 3: Tutorial Generation** introduces methods for generating the three kinds of tutorial types we identify in our research: instructions, demonstrations, and experiences.

  - **Chapter 9: Tutorial Instruction and Demonstration Generation** presents the AtDelfi system, which generates tutorial cards for GVG-AI games.
  - **Chapter 10: Mechanic-Dependent Tutorial Scene Generation Using Objective-driven Search** defines single-objective optimization methods for generating small tutorial levels called "scenes," which emphasizes usage of a mechanic or a subset of mechanics.
  - **Chapter 11: Mechanic-Dependent Tutorial Scene Generation Using Quality Diversity** defines methods for generating small tutorial scenes using quality diversity evolution.
  - **Chapter 12: Persona-dependent Tutorial Scene Generation** details a way to generate scenes using personas, drawing on concepts from Game Mechanic Alignment, play persona analysis, and mechanic-dependent tutorial scene generation.
  - **Chapter 13: Tutorial Sequential Level Generation** describes a way to link "scenes" together in a process we call "scene stitching" to produce longer, fuller levels.

- **Chapter 14: Conclusion** summarizes this thesis and discussing its contributions. We also talk about future work, impact on the field, and a vision for the future.

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 most the research discussed here, throughout the rest of this work the pronoun "we" will be used in favor of "I" when describing projects 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 gender inclusive speech. When I refer to **you**, the reader, I will address "you" appropriately.

Part I

Context

Chapter 2

A Tutorial on Tutorials

Tutorials are often the first interactions players have with a game. They help players understand game rules and, ultimately, learn how to play with them. At their core, tutorials *teach*, and we as players are meant to *learn* from them. A good tutorial is a helpful one; a bad tutorial is useless at best and misleading at worst. First I lead you through how tutorials have evolved throughout time, the role that I believe tutorials are meant to play in games, and tutorial patterns that we can easily identify. Then, I will present past and present research tangential to automated tutorial generation.

2.1 The Evolution of Tutorials

The first electronic games often lacked formal tutorials. Arcade games had simple mechanics that players could pick up easily: "Tilt the joystick to move," "Press button to shoot," are sometimes written onto the arcade machine itself. *Pong* (Atari 1972) is an example of a simple game without need of a tutorial. As arcade games became more popular, game designers began to include easy-to-read mechanic tables and videos on arcade machines. *StreetFighter* (Capcom 1987) and *Pac-Man* (Namco 1980) used both images and text written on the machine surface, as well as in-game videos of certain mechanics and moves players can use.

The introduction of consoles to the gaming community allowed designers to create games with more complex mechanics. To help players achieve their goals in the game, designers began to include formal tutorials teaching players the gameplay basics. Games like *Space Invaders* (ATARI 1978) and *Super Mario Bros* (Nintendo 1985) included booklets with text and photos explaining certain moves and mechanics. Designers experimented with other methods of teaching game rules, such as interactive in-game tutorials. Figure 2.1 shows an early example of interactive in-game tutorial in *Gauntlet* (Atari 1985), a complicated RPG dungeon

Figure 2.1: Gauntlet interactive tutorial about colliding with enemies.

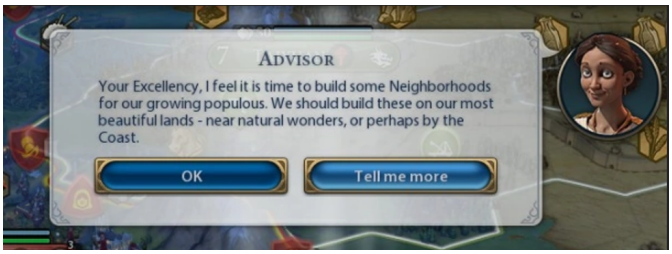


Figure 2.2: An advisor popup in Civilization VI, explaining ideal locations for neighborhood districts

crawler. To ease the player's learning curve, the tutorial was split among several levels. Whenever players first encountered a new element, the game paused and explanatory text appeared.

From AAA-titles to indie games, developers today have a plethora of ways to use tutorials. Civilization VI (Fraxis Games, 2016) uses an advisor, an in-game entity which communicates how different mechanics work to the player and advises good research options and building strategies (Figure 2.2). Players can tailor the advisor's messages to their familiarity with Civilization VI and its expansions. Dragon Age: Origins (Bioware, 2009) provides a tutorial questline for each of the 5 origin stories. Throughout these questlines, players learn how to converse with NPCs, combat (ranged, melee, and magic), inventory management, companion management, and map encounters like traps and ambushes. Elden Ring (FromSoftware, 2022) similarly presents a tutorial stage for the player to engage with; however, it is entirely optional (see Figure 2.3). Elden Ring also allows players leave messages in-game which will appear in other games. These messages may contain helpful tips ("Hidden treasure ahead!") or deceptive traps ("Jump off cliff, secret room") as a form of crowd-sourced (possibly deceptive) tutorials.

Some games have the ability to dynamically alter their difficulty, enemy behavior,

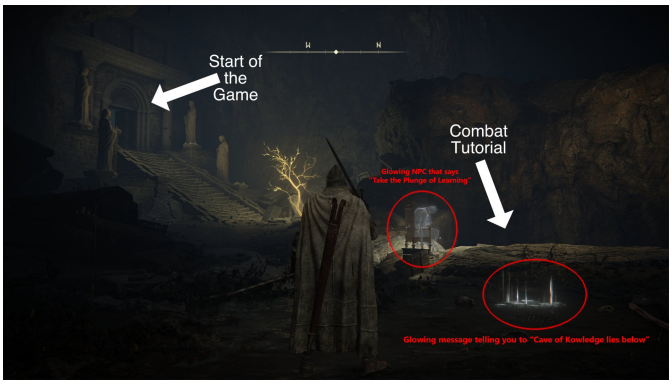


Figure 2.3: The *very optional* tutorial in Elden Ring

or map/item layout in response to the player's actions in real-time. Unreal Tournament (Epic Games & Digital Extremes, 1999) was praised for its bot behavior, in which players could select a skill level ranging from "novice" to "godlike" or allow the bot to dynamically adjust its skill level in response to the player's performance. In Batman: Arkham City (Rocksteady Studios, 2011), the boss fight with Mr. Freeze is well-known for its adaptive style. The slowly lumbering Mr. Freeze stomps around the level searching for the player, who hides in waiting and constantly looking for a new clever way to attack [54]. Every time an enemy manages to slay the player in Middle Earth: Shadow of Mordor (Monolith Productions, 2014), that enemy can be promoted to a higher rank in Sauron's army, gaining special benefits (and weaknesses) to accompany their new role. In fact, killing the player is not a hard requirement, sometimes merely encountering the player and surviving the ordeal is enough. A lowly orc set aflame by the player and managing to live may be promoted to a lieutenant with a crippling fear of fire [158]. Metal Gear Solid V: The Phantom Pain (Kojima Productions, 2015) contains a "revenge system," which dynamically adjusts the challenges of outpost infiltration depending on past player strategies.

2.2 The Role of Tutorials

Tutorials have evolved from being non-existent to being dynamic, interactive, in-game experiences. Ultimately, tutorials are supposed teach the player how to play. No matter how they look or are presented, one unifying theme of tutorial design is their content: *game mechanics*. A "mechanic" can be defined as an event within the game that is fired by a game element and impacts the game's state [147]. Mechanics define the boundaries of player affordance. The relationships between these mechanics and how they collectively interact to change the game at run-time is known as a game's dynamics [84]. To engage with the game, a player uses game

mechanics. Therefore, knowing *how to play* is equivalent to *knowing how to use the game's mechanics and dynamics*. This is especially true for mechanics that the player can directly trigger and mechanics that directly affect the player.

How much tutorials teach and how content is being taught varies from tutorial to tutorial and game to game. Tutorials tend to fall somewhere along the axis of Sheri Graner Ray's knowledge acquisition styles for players [133]. These styles range from Explorative Acquisition to Modeling Acquisition. *Explorative acquisition* emphasizes learning about something *by* doing it, whereas *modeling acquisition* focuses on studying how to do something *before* doing it. While one cannot say with absolute certainty if one technique is superior to another, different techniques suit different audiences and/or games [3, 133, 174]. Williams suggests that active learning tutorials, which stress player engagement and participation with the skills being learned, may be ineffective when the player never has an isolated place to practice a particularly complex skill [174]. In fact, Williams argues that some active learning tutorials actually ruin the entire game experience for the player because of this reason. According to Andersen et al., the effectiveness of tutorials on gameplay depends on how complex a game is to begin with [3], and sometimes are not useful at all. Game mechanics that are simple enough to be discovered using experimental methods may not require a tutorial to explain them.

To gain more understanding about tutorials and mechanics, researchers have created various languages to model games. Dan Cook's concept of *skill atoms* [28], which refers to the feedback loop through which a player learns a new skill during gameplay, is another example. Figure 2.4 shows the skill atom for learning how to jump. A skill atom can be divided into four separate elements:

- The *Action* the player performs to learn a new skill. This could involve anything from pressing a button or doing a complex series of actions to accomplish an end goal.
- The *Simulation* of that action in game. The player's action somehow affects the world.
- The *Feedback* from the simulation informs the player of the new state of the game.
- The *Modeling* the player now performs within their head, mapping the action they just took to the feedback from the simulation. "If I press this button, my character jumps up."

Skill atoms can be associated with other skill atoms to form *skill chains*. Using skill chains, one could presumably model any game. A related concept is strategy

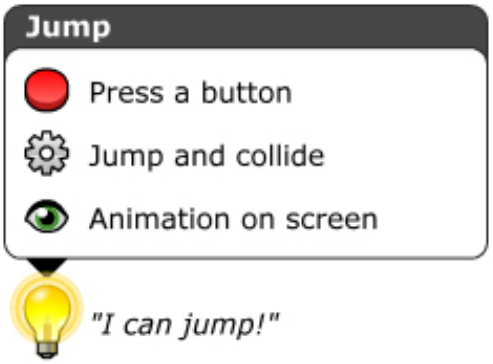


Figure 2.4: A skill atom for learning how to jump in any generic game, in the order of *action* (button), *simulation* (jump and collide), *feedback* (animation on screen), and *modeling* ("I can jump!")[1]

ladders, where each step is an addition to the previous step's strategy that makes a significant difference in playing. It has been proposed that a game's depth can be defined as the length of its longest strategy ladder [105].

With this, we can define two boundaries for tutorial generation: there are mechanics that are too simple to be taught in a tutorial, and there are mechanics complex enough that they may need to be practiced in a well-designed environment to hone. We can summarize that tutorials are meant to teach mechanics, particularly complex ones. The methods with which tutorials present mechanics to the player are covered next.

2.3 A Taxonomy of Tutorial Types

In general, a game developer would want to use the most suitable tutorial style for their game. For that purpose, they must understand different dimensions and factors that affect the tutorial design process and outcome. Andersen et al. [3] measures how game complexity affects the perceived outcome of tutorials. In their study, they define 4 dimensions of tutorial classification:

- **Tutorial Presence:** whether the game has a tutorial or not.
- **Context Sensitivity:** whether the tutorial is a part of story and game or separate and independent from them.
- **Freedom:** whether the player is free to experiment and explore or is forced to follow a set of commands.
- **Availability of Help:** whether the player can request for help or not.

The classification proposed by Andersen et al. is binary. However, it is useful to see tutorials situated on a continuum between these extremes, as this allows us to gain a more nuanced understanding of game tutorials. For example: Figure 2.5 displays the tutorial in *Braid* (Number None, Inc, 2008) for a time rewinding mechanic. The tutorial only appears based on a certain event, i.e. the player's death. Players will not know about the mechanic until their first death. Instead of having the tutorial available at anytime or showing how to use the mechanic at the beginning, the developer reveals it when it is first necessary.

Figure 2.5: Braid teaching time rewinding mechanic when the player dies.

By sampling tutorial classification in literature and comparing it with published game tutorials in the wild, we can find repeating tutorial patterns in multiple games. We highlight the following tutorial types, which are not the only tutorials present in the space, but appear to be the most common ones:

- **Teaching using instructions:** These tutorials explain how to play the game by providing the player with a group of instructions to follow, similar to what is seen in boardgames. For example: Strategy games, such as *Starcraft* (Blizzard, 1998), teach the player by taking them step by step towards understanding different aspects of the game through on-screen text.

- **Teaching using examples:** These tutorials explain how to play by showing the player an example of what will happen if they do a specific action. For example: Megaman X uses a Non Playable Character (NPC) to teach the player about the charging skill [47].

- **Teaching using an interactive experience:** These tutorials explain how to play the game by giving the player freedom to explore and experiment. Levels are a "container for gameplay" [18], acting as the grounds to explore

and experiment with game mechanic interactions. For example: in *Super Mario Bros* (Nintendo, 1985), the world 1-1 is designed to introduce players to various game elements, such as goombas and mushrooms, in a forgiving way [33]. Early obstacles in games are instances of *patterns*, which reoccur later in the game in more complex instances or combinations [34]. Through repetition and variation, players can master different skills.

Figure 2.6: Street Fighters arcade cabinet. The cabinet shows different combos that can be done.

A game can have more than a single tutorial type from the previous list. Arcade games used both demos and instructions to catch the attention of the player and help them learn. The demos help to attract more players, while simultaneously teaching them how to play. Showing an information screen before the game start, such as in *Pacman* (BANDAI NAMCO, 1980), or displaying instructions on the arcade cabin, frequently seen in fighting games, helps the player understand the game and be engaged. Figure 2.6 shows a *Street Fighters* arcade cabinet with different characters' combos and moves written on it. Megaman X uses a carefully designed level to teach the player how to perform a powershot attack, while also giving a demonstration of the powershot attack using a non-player character.

2.4 Tutorial Generation-like Research

Before this thesis, no project has yet attempted to develop a generalized framework for automated tutorial generation in games, though many projects exist that are tangentially related to tutorial generation. For example, generated heuristics for blackjack and poker can be used to create effective strategies for beginners[38, 39, 40]. *TutorialPlan* [106] generates text and image instructions for new users of AutoCAD. Mikami et al. generated tutorials for API coding libraries [115], claiming that the resulting generated tutorials helped users learn libraries more effectively than hand-made tutorials. Alexander et al. formalized the game logic of *Minecraft* (Mojang 2009) into mechanics to create action graphs, representing the player experience, and created quests and achievements based off those actions [1].

Game-O-Matic [165] generates arcade style games and instructions using a story-based concept-map inputted by a user. After the game is created, Game-O-Matic generates a tutorial page, explaining who the player will control, how to control them, and winning/losing conditions, by using the concept-map and relationships between objects within it. Mechanic Miner [29] evolves mechanics for 2D puzzle-platform games, using *Reflection* to find a new game mechanic then generate levels that utilize it. Mappy [118] and the updated Mappyland [119] can transform a series of button presses into a graph of room associations, transforming movement mechanics into level maps for Nintendo Entertainment System games. This is similar to what Summerville et al. created as a part of the *Gemini* system, a logic program that performs static reasoning over game specifications in order to find meaning [154]. Within the Gemini's Cygnus system, the player can derive higher-level meanings about the game in question from implicit rules (which they call "dynamics"). However, this derived knowledge is not structured in the form of a tutorial. Lastly, the Talin plugin [8] for Unity uses player skill to guide the output of the generator, giving dynamic feedback to players as they play.

[1]image from `https://www.gamasutra.com/view/feature/129948/the_chemistry_of_game_design.php?page=3`

Chapter 3

Procedural Content Generation

Procedural Content Generation (PCG) [76, 144] is the process of using computer algorithms to produce content. PCG techniques have been utilized in games going back decades, such as the level generation systems in Beneath the Apple Manor (Don Worth, 1978) and Rogue (Glenn Wichman, 1980). It helped developers produce enormous amounts of content with small memory and computational costs, like universe generation in Elite (David Braben and Ian Bell, 1984) [91]. Today, advances in technology no longer restrict developers in these areas, but PCG is widely used to generate content in games, such as maps in Spelunky (Derek Yu, 2008), terrain in Minecraft (Mojang, 2011), and worlds in Starbound (Chucklefish, 2016) and No Man's Sky (Hello Games, 2016). Research in PCG is expansive; within this thesis, we review several areas including constructive, search-based, machine learning, and reinforcement learning methods. We also give an overview of experienced-driven PCG (a subset of PCG methods meant to build targeted content for a specific person or persona) and mixed-initiative PCG (algorithms and humans working together to produce content).

3.1 Constructive Methods

Constructive content generation techniques are popular in commercial games for their speed and ease of implementation. The previously mentioned Rogue (Glenn Wichman, 1980) generates a new dungeon every playthrough by creating non-intersecting rectangles as rooms and connecting them using hallways. The defining feature of constructive methods is that they do not "generate and test." In other words, they do not continually optimize their generated content before they output it, like some of the methods in the following sections. Because of this, constructive methods must guarantee that the output levels contain all required features during the construction step. Often times there is no way to guarantee

level playability. Repair methods are often used to "fix" broken content after generation.

Different construction methods can be used depending on the domain and the desired output. For example, template-based generators fuse together hand-authored pieces to generate content. Games like Spelunky (Derek Yu, 2008), Spore (Maxis, 2008), Borderlands (Gearbox Software, 2009), and No Mans Sky (Hello Games 2016) all use template-based techniques. Grammar-based generation uses hand-designed rules to describe content space and manipulation within that space. Grammars have been used for the generation of trees [132], stories[14], and levels [42, 168]. Tracery [24, 25], an open-source grammar library, has been used to generate dialogue for social-media bots [169], art [26, 120], levels [99], level events [50], and level generators [101].

3.2 Search-Based Methods

Search-based PCG is a generative technique that uses search methods to find good content [163]. In practice, evolutionary algorithms are often used, as they can be applied to many domains, including content and level generation in video games. Search-based PCG has been applied in many different game frameworks and games, such as the General Video Game AI framework [125], PuzzleScript [95], Cut the Rope (ZeptoLab, 2010) [142], and Mazes [7]. Search-based techniques have been used to generate levels [6, 7, 11, 95], board games [15] and level generators [92, 101].

MAP-Elites [117], a quality-diversity algorithm, has been used to illuminate spaces by keeping saving solutions to a multidimensional array and organizing them by behavioral characteristics. MAP-Elites has been used in a variety of research applications such as generating Hearthstone Decks [51, 179] and platformer levels [172]. Constrained MAP-Elites (CME) is a hybrid algorithm that combines MAP-Elites with the FI2-Pop algorithm mentioned in Section 5.2.1. The algorithm for Constrained MAP-Elites is detailed in Section 5.2.2. CME has been used to generate levels for genres like bullet hells [99] and dungeon crawlers [2], as well as for several projects in this thesis (Chapters 11 and 12).

3.3 Machine Learning Methods

Procedural content generation via machine learning [155] (PCGML) is a family of PCG techniques that use machine learning algorithms to generate content. These methods are rarely seen outside of the research community because of their reliance on large datasets, long training times, and little control of the generated output.

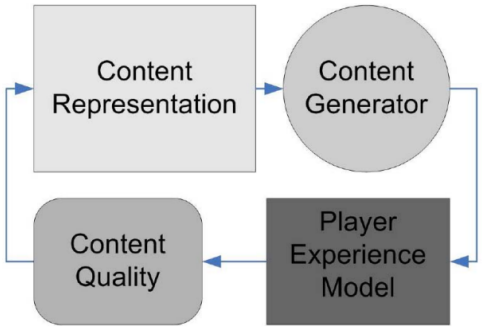


Figure 3.1: The main components of an experience-driven procedural content generator.

For example, Caves of Qud [71] uses PCGML to generate books and other aesthetic elements. Research applications with PCGML have used many different techniques including Markov Chains [150], N-Grams [36], GANs [164, 171], Autoencoders [87], and LSTMs [153].

Procedural content generation via reinforcement learning [93] (PCGRL) is a family of PCG techniques powered by reinforcement learning algorithms. PCGRL techniques have primarily been used in level/experience generation [23, 45, 114, 178], although there are few examples other than these. This is probably because of the difficulty of transforming level generation into a reinforcement environment. One way to do this in practice is to turn PCG into an iterative, co-creative process [73, 74].

3.4 Experience-Driven Generation

Most PCG methods generate content for the general audience of the game in question. Experience-driven Procedural Content Generation [175] (EDPCG) can create content designed for specific user experiences. EDPCG may use any of the methods specified in previous sections for a wide variety of applications such as generating levels [146], music [129], and therapeutic experiences [114]. Designing for a specific user experience can come in many forms, such as level difficulty (as measured by the player's skill) [83], aesthetics [109], or letting users pick for themselves [107].

3.5 Mixed-Initiative Generation

Mixed-initiative generation is a class of co-creation techniques where both humans and software work together to make content [108]. AI-assisted design tools can have wide-ranging requirements and can vary greatly in human- and

AI-affordances [121]. Sentient Sketchbook [110] gives level suggestion changes to the user for strategy game level generation. Tanagra [149] gives the user the ability to observe and manipulate geometric patterns in platformer levels, as well as the ability to auto-create level space without designer input. Ropossum [143] can let the user know if the level in its current form is playable in Cut The Rope (ZeptoLab, 2010). Similarly, the Baba is Y'all system contains a generator module which can generate levels that players can build off of while also checking for level playability [21]. When combined with machine learning, co-creative systems can take turns manipulating levels with the user [74] or can constrain user-manipulation [12]. Co-creativity can also involve the use of reinforcement learning, as done so in RL-Brush [41].

Chapter 4

Frameworks

This chapter introduces all of the frameworks we use across the thesis. Overall, we utilize three domains: General Video Game AI, Mario AI, and Minidungeons 2. Levels can be represented as a 2D tilemap in all three domains.

4.1 GVG-AI

The General Video Game Artificial Intelligence framework [127] (GVG-AI) is a framework built to run games written in the Video Game Description Language (VGDL) [46]. GVG-AI was originally developed in the context of the eponymous competition, but has since been used in various research projects. The framework allows the use of automated agents interchangeably between games. 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. Thus, success at the set of GVG-AI games involves adapting to changing mechanics, goals and strategy requirements in a similar fashion to how high-skilled players adapt to their opponent strategies to win.

The Video Game Description Language [46] (VGDL) is a game description language used to represent 2D arcade games (Pac-man), action (Space Invaders), and/or puzzle game (Sokoban) genres. VGDL games consist of 2 parts: a game description and level descriptions. The game description is responsible for holding information about game objects, game mechanics, and termination conditions. Figure 4.1 shows a simple Sokoban game and its game description. The game description consists of 4 subparts:

Sprite Set: a hierarchical list of game objects, called game sprites. Similar game sprites can be grouped under a common name in the hierarchy, which is considered a parent of these similar game sprites. For example, in Pac-man, two sprites are grouped into the "Pac-man" parent type: *hungry* and *powered*.

```
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
```

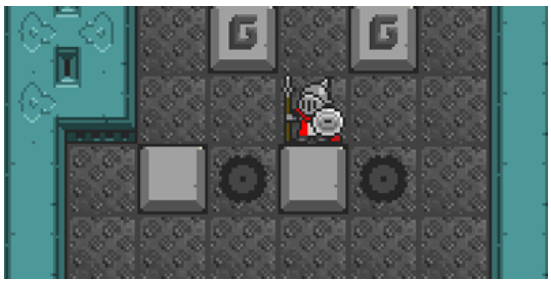


Figure 4.1: A definition of a simple game (a version of Sokoban) in VGDL, and a screenshot of part of the game in-engine.

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.

We do not utilize every game in the GVG-AI framework, of which there are currently over 100. The following is the subset of games we use across the projects presented within this work. These games are selected for their type and ruleset, based on an analysis by Bontrager et al [13]:

- **Aliens** is a clone of *Space Invaders*. The player uses arrow keys to move left and right on the bottom of the screen, shooting down aliens and avoiding missiles they shoot back. The player score for each alien destroyed. If the player collides with an alien or an alien's missile, they lose the game.

- **Butterflies** is a game where players must collect all butterflies and keep them away from the flowers (as they eat the flowers). Each butterfly collected increases the score, and after collecting all butterflies, the player wins. If a butterfly eats a flower, it multiplies, and if all flowers are eaten the player loses. The game's challenge is balancing flower management to score high without losing.

- **Camel Race** is a racing game. To win, the player needs to be the first camel to touch the opposite side of the screen. There are various obstacles the player needs to avoid in order to accomplish this goal. The only way to gain any points is by winning the game.

- **Jaws** is a survival game where the player needs to evade/kill all sharks within 1000 ticks. The sharks are divided into two types, passive and aggressive. Passive sharks swim in a straight line and can be killed by the player's gun, while aggressive sharks swim towards the player and cannot be killed by bullets. When sharks die, they turn into jewels, which can be collected by the player for points.

- **Pacman** is game set in a 2-dimensional maze, which the player traverses while collecting all the pellets and fruit in the level. Four deadly ghosts chase the player in the maze and must be avoided. Some of the pellets are "power pellets" which grant the player a temporary invulnerability to the ghosts, allowing the player to eat them and force them to respawn in the center of the maze.

- **Plants** is a GVG-AI clone of *Plants vs. Zombies* (PopCap Games 2009). The goal is to survive for 1000 ticks. Zombies spawn on the right side of the screen moving towards the left, and the player loses if a zombie reaches the left side of the screen. The player needs to build plants, which fire zombie-killing pea sprites.

- **RealPortals** is a GVG-AI clone of *Portal* (Valve 2007). The player must reach the door, which sometimes is behind another locked door that needs a key. The player is restricted by water, which often lies between them and the door. To succeed, players need to pick up wands, which allow them to create *portals* through which they can spontaneously travel across the map. There are two different types of wands, and each corresponds to a different portal gateway.

- **Solarfox** is a GVG-AI adaptation of *Solar Fox* (Bally/Midway Mfg. Co 1981). The player must dodge both enemies and their flaming projectiles

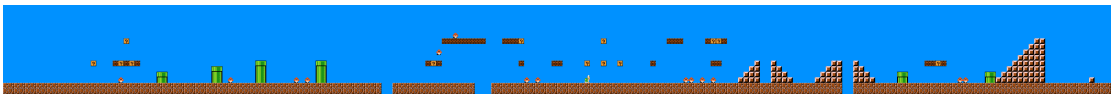


Figure 4.2: Super Mario Bros World 1-1

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 in 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. Good gameplay invokes a balance of short- and long-term strategy, balancing the greed of winning the level against getting more points and risking loss.

GVG-AI is the framework for projects described in Chapters 6, 9, and 11.

4.2 Mario AI Framework

Infinite Mario Bros. (IMB) was developed by Markus Persson (Notch) [128] 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 amount 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, when in the fire state, shoot fireballs. 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 [90], having been used in AI competitions in the past [90, 162]. It improved on limitations of IMB's level generator, and several techniques have been applied to automatically play [90] or create levels [145, 151], some of which are search-based methods.

The Mario AI Framework is used for projects presented in Chapters 10 and 13.

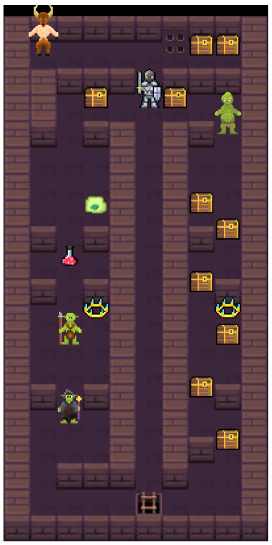


Figure 4.3: A Minidungeons 2 map

4.3 Minidungeons 2

MiniDungeons 2 [1] is a 2-dimensional deterministic, turn-based, rogue-like game, first described in [82], in which the player takes on the role of a hero traversing a dungeon level, with the end goal of reaching the exit. Typically set on a 10 by 20 tile grid, the game map is made up of a mix of impassible *walls* and passable *floors*. Interactive items and characters are scattered throughout the level. To win, the player must reach the exit, represented as a staircase. All game characters have Hit Points (HP) and deal damage when the player collides with them. The player begins the game with 10 HP, and if the player runs out of HP, they die and lose the level. Figure 4.3 displays a map from Minidungeons 2.

Each turn, the player selects an action to perform, and all game characters will then move after the player completes their action. Any game character may move in one of the four cardinal directions (North, South, East, West) on their turn so long as the tile in that direction is not a wall. The player is given one re-usable javelin at the start of every level. The player may choose to throw this javelin and do 1 damage to any monster within their unbroken line of sight. After using the javelin, the hero must traverse to the tile to which it was thrown in order to pick it up and use it again.

While exploring a map, the player can find a various different interactive objects that provides the player with various effects:

- **Potions** increase the HP of the hero by 1, up to the maximum of 10.
- **Treasures** are used to increase the treasure score of the hero.

[1] http://minidungeons.com/

- **Portals** come in pairs. If the hero collides with a portal, they are transported to the location of the paired portal on the same turn.
- **Traps** deal 1 damage to any game character moving through them, including monsters and enemies.

In addition to the above objects, the player may encounter monsters, all of which desire to attack a player within line of sight and some of which have additional secondary goals:

- **Goblins** move 1 tile every turn towards the player if within line of sight. They have 1 HP and deal 1 damage upon collision. Goblins try to avoid colliding with other goblins and goblin wizards.
- **Goblin Wizards** cast a 1 damage spell at the hero if they have line of sight within 5 tiles of the player. If they are over 5 tiles from the player but have line of sight, they will move 1 tile towards the player each turn. Wizards have 1 HP and deal no damage on collision.
- **Blobs** move toward either a potion or the hero within line of sight. They will move 1 tile towards the closest one per turn, preferring potions over the hero in case of a tie. A blob colliding with a potion consumes it and enables it to level-up into a more powerful blob, which also happens if a blob collides with another blobs and merge with each other. The lowest level blob has 1 HP and does 1 damage upon collision. The 2^{nd} level blob has 2 HP and does 2 damage. The most powerful blob has 3 HP and does 3 damage.
- **Ogres** will move 1 tile towards either the player or a treasure per turn within line of sight, preferring treasures over the hero in case of a tie. When an ogre collides with a treasure, they consume it, and their sprite becomes fancier to look at. Ogres have 2 HP and deal 2 damage to anything they collide with, including other ogres.
- **Minitaurs** always move 1 step along the shortest path to the hero as determined by A* search, regardless of line of sight. Collision with the minitaur will deal 1 damage to the player. A minitaur has no HP and is immortal. When damaged, the minitaur is stunned for 5 moves (and can be passed through).

In a Minidungeons 2 gameplay session, players usually move around with their character to interact with different game objects and characters. At every frame, the current game state is recorded as a 2D map of tiles as well as the player's current health, score, location, and last performed action. Every interaction (game

mechanics) that happens in the game is also recorded. There are a total of 17 mechanics tracked during play:

- **Enemy Kill:** the player slays any enemy in the game.
- **Monster Hit:** a specific monster is hit by either the player or the javelin. This mechanic is recorded separately for each different monster so we have **Goblin Hit**, **Minitaur Hit**, **Goblin Wizard Hit**, **Blob Hit**, or **Ogre Hit**.
- **Ogre Treasure:** the ogre collects a treasure
- **Blob Potion:** the blob consumes a potion
- **Blob Combine:** the blob combines with another blob
- **Javelin Throw:** the player throws the javelin
- **Collect Treasure:** the player collects a treasure
- **Consume Potion:** the player consumes a potion
- **Trigger Trap:** a trap is triggered
- **Use Portal:** the player uses a portal
- **Take Turn:** the player makes a move and ends their turn
- **Die:** the player dies
- **Reach Stairs:** the player reaches the exit stairs

Minidungeons 2 is the framework of choice for projects presented in Chapters 7, 8, and 12. Below, we explain the Minidungeons 2 playtrace datasets that are utilized by multiple projects in this thesis. Each datapoint contains of a list of sequential gamestates including map information, player health information, and mechanic information at each state. Every datapoint is also labeled with persona labels, which is performed using action agreement ratios for the human playtrace dataset as explained in Section 4.3.4.

4.3.1 Minidungeons 2 Maps

Five hand-designed maps were developed to build the Minidungeons 2 playtrace dataset. All maps contained a mix of game elements and offered multiple routes for the player to take to complete the level. Figure 4.4 displays all the five levels. As we can see all the maps except Map 202 provide a clear path for each of the

different personas. For example, Map 100 has a center path for runner, right path for treasure collector, and left path for monster killer. Meanwhile, Map 202 is an open space with monsters and treasures distributed all over the map. Every map contains $5-6$ monsters, $6-9$ treasures, and at least one straightforward path to the exit.

(a) Map 100 (b) Map 101 (c) Map 102 (d) Map 201 (e) Map 202

Figure 4.4: User study maps used to generate the human playtrace dataset

4.3.2 Synthetic Agents

For the purposes of this thesis, we curated a group of artificial agent personas to generate the synthetic playtraces. These agents are Best First Search (BestFS) agents which use utility functions created by Holmgaard et al [77, 80] in order to resemble a player archetype (persona). Previous research [77] uses MCTS agents, but in a fully deterministic game, BestFS generates more consistent results, both for action agreement and for gameplay ability. There are three agent personas in MiniDungeons that are used to capture agent playtraces with the following described goals:

1. **Runner:** complete the level as fast as possible.
2. **Monster Killer:** slay as many monsters as possible before completing the level.
3. **Treasure Collector:** open as many chests as possible before completing the level.

Agents perform online planning to move, meaning that each turn, they build a search tree to find the optimal next move to make. After doing preliminary experiments, we realized that the synthetic agents *always* make the same movements, and therefore multiple playtraces on the same map will not expand the dataset. To

overcome this, we added random movement to the synthetic agents when generating the synthetic dataset: every move there is a 25% chance that the agent will take a random action rather than their searched action. This randomness is *not* used when performing action agreement.

At each turn, the BestFS agent uses one of the following heuristic for the respective persona. In the following equations, h_{agent} denotes the heuristic function for that specific persona. The agents could be runner (r), monster killer (mk), or treasure collector (tc). The runner persona is rewarded to move to the exit in the fewest amount of steps as possible. Equation 4.1 shows the heuristic (h_r) function of the runner persona.

$$h_r = dist_{exit} - steps \tag{4.1}$$

where $dist_{exit}$ is the distance from the current player location to the exit, and *steps* is the amount of steps taken since the start of the game.

The MK persona will prioritize killing all monsters while attempting not to die and getting to the exit when all monsters have been slain. Equation 4.2 shows the heuristic (h_{mk}) function of the monster killer persona.

$$\begin{aligned} h_{mk} &= c * N_{monster} + k * Dead + p_{monster} \\ p_{monster} &= \begin{cases} min(dist_{monster}) & N_{monster} > 0 \\ dist_{exit} & N_{monster} == 0 \end{cases} \end{aligned} \tag{4.2}$$

where c and k are constants, $min(dist_{monster})$ is the distance to the closest monster from the player location, $dist_{exit}$ is the distance between the player and the exit, $N_{monster}$ is the number of alive monsters in the level, and $Dead$ is a binary value that is equal to 1 if the player is dead and 0 otherwise.

The TC persona will prioritize collecting treasure while attempting not to die and getting to the exit when all treasures have been collected. It uses a similar equation to 4.2 but replacing $min(dist_{monster})$ with $min(dist_{treasure})$ (the distance to the closest treasure from the player location) and $N_{monster}$ with $N_{treasure}$ (number of unopened treasures in the level).

4.3.3 Synthetic Dataset

When an agent plays a map, it will continue to play until they win or lose. Agents are given 1.0 second to plan before making their next move, i.e. build their search tree. For the monster killer agent and treasure collector, c is equal to 45 and k equal to *double.max* to encourage the agent to stay alive and favor more monsters being slain in case of the monster killer or treasures collected in case of the treasure collector. We run each persona 100 times on each of the five map

described in Section 4.3.1, for a total of 1500 playtraces.

4.3.4 Human Dataset

We performed a user study to gather player data for Minidungeons 2. A link to the user study was spread by a combination of tweets and academic email lists. In this study, participants are given 3 tutorial levels to acclimate them to the game. These levels are very simple, beginning with a small linear map and increasing in size and mechanical complexity until introducing most of the basic mechanics by the 3rd level. The user is presented with a series of levels randomly selected from the maps described in Section 4.3.1. A total of 853 human playtraces were collected using this method.

These playtraces are labeled by comparing the user's actions with the procedural agents' actions - the action agreement label. Given a playtrace, a replay agent steps through every move that the human made. At every step, the system runs each of the 3 generative agents for 1 second, searching for the moves they would have made at that step. This means that each move takes 3 seconds to calculate agreement. User playtraces can be as long as 140 moves or more, meaning that a single playtrace could take several minutes to compute without parallelization. If the user's action agrees with an agent's action, that persona gets a point. The persona's label is added to the playtrace if they have more than 50% action agreement, calculated by taking that persona's point score divided by the total number of steps taken. Using this method, user data can have multiple persona labels or possibly none at all.

4.3.5 AAR Label Analysis

Table 4.1 displays the breakdown of the training player distributions using every AAR multilabel combination. One interesting finding is that there is a sizable group of playtraces (82) with no label at all, suggesting the presence of a fourth persona not measured here. Another curious group of players are the 117 traces that are classified as all three persona types. The mechanics showcased here are selected due to their traditional correlation with a specific persona, and values in **bold** correspond to that combination's correlated mechanics.

All combinations which include the "R" label contain a lower amount of steps taken than other label combinations, the lowest coming from "Pure R" at 18.49 and "R&MK" at 15.25. "Pure TC" and "Pure MK" distributions contain similar amounts of treasures collected and monsters killed, but there are only 2 "Pure MK" playtraces in the entire dataset. The presence of many "R&MK" (56), "TC&MK" (24) and "R&TC&MK" playtraces but few "Pure MK" traces promotes the idea

that players that are driven to kill monsters are also driven toward other persona subgoals during gameplay. The highest combinations of treasures collected and enemies killed comes from the "TC&MK" labeled traces, 6.83 and 5.0 respectively, suggesting TC and MK subgoals have great synergy. The "R&TC" traces have a greater amount of enemies killed than "R&MK," but the treasure collection amounts differ greatly (3.36 and 0.18 respectively). Due to the nature of enemy movement in Minidungeons 2 and the placement of enemies and treasure on some of the user study maps (Section 4.3.1, it is difficult to collect treasure and not kill monsters. However, it is quite simple to kill monsters and *not* collect the nearby treasure, which is immobile. We think that "R&TC" players will kill monsters on the way to collect treasure, whereas "R&MK" players kill monsters and ignore the treasure. Also of note is the different in steps taken between the two distributions: "R&TC" players (36.82) take over twice as many steps as "R&MK" players (15.25) in order to collect more treasure. This result suggests that "R&TC" players may have competing subgoals: "go to the exit as fast as possible, but also go out of your way to collect treasure."

	Count	*Steps Taken*	*Treasures Collected*	*Enemies Slain*
No Label	82	21.46±30.77	0.6±1.55	1.21±2.29
Pure R	144	**18.49±8.07**	0.48±1.19	1.47±1.69
Pure TC	64	73.84±29.98	**5.75±1.79**	3.48±2.16
Pure MK	2	92.5±0.71	5.0±1.41	**2.5±0.71**
R&TC	76	**36.82±23.67**	**3.36±3.1**	2.91±1.96
R&MK	56	**15.25±9.61**	0.18±0.95	**1.45±1.32**
TC&MK	24	113.17±19.03	**6.83±0.56**	**5.0±2.3**
R&TC&MK	117	**43.95±23.59**	**3.63±2.86**	**3.85±1.96**

Table 4.1: Average scores and standard deviations in several game metrics for every AAR multilabel combination on the 565 human playtrace training dataset. The notable mechanics for a specific persona label combination are **bold**.

Chapter 5

Algorithms

This chapter summarizes the different algorithms used throughout this thesis. This chapter is divided into two sections. Section 5.1 reviews the literature concerning tree search algorithms, while Section 5.2 reviews literature about evolutionary algorithms.

5.1 Tree Search Algorithms

Tree Search Algorithms are a family of algorithms used to search spaces efficiently. Tree generation/traversal can be performed using a variety of different algorithms depending on the domain and the need for search time against globally or locally optimal solutions. We explain four different tree search algorithms below.

5.1.1 Breadth-First Search

Breadth-First Search [31] is a form of uninformed graph traversal in which each level of a tree is fully explored before the next level is begun. To do so, the algorithm takes the form of a first-in first-out queue, in which nodes are added one child at a time starting from the root at the top of the tree. The uninformed critical mechanic discovery method in Chapter 6 utilizes breadth-first search on a mechanic graph.

5.1.2 Best-First Search

Best-First Search [31] is a form of informed graph traversal which uses a priority queue. Nodes are expanded in order of their score using a heuristic evaluation. The mechanic informed critical discovery method in Chapter 6 utilizes a modified best-first search algorithm.

5.1.3 A-Star Search

A-Star Search [31] is a subtype of Best-First Search, leveraging both a heuristic estimate and a cost evaluator to rank its priority queue of nodes. A-Star is most useful in deterministic environments, and generally fails to perform well in stochastic ones, as it does not take randomness into account in its forward model. In the projects which use the deterministic Minidungeons 2 and Mario AI frameworks (Chapters 10, 11, 12, and 13) we use A-Star to power game-playing agents.

5.1.4 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search [16, 32] is a heuristic search algorithm, like A-Star, which uses random sampling for deterministic problems. Unlike other tree search algorithms like Minmax, breadth-first, or depth-first, MCTS focuses on *exploiting* the most promising moves to expand next, while balancing that by *exploring* more neglected branches of the tree. The balance of exploitation versus exploration is traditionally handled through the evaluation of the Upper Confidence Bound for Trees equation, which applies UCB1 to the tree. The tree is built incrementally, with each iteration following a simple formula:

1. **Selection**: MCTS chooses the next node to expand via the *tree policy*, starting at the root node and recursively picking the highest scoring child "until the most urgent expandable node is reached" [16] or a terminal state (i.e. the game is won or lost). The score for traversing the tree in MCTS is termed *tree policy*, and in traditional MCTS approaches it is given by the Upper Confidence Bound (UCB1) equation:

$$UCB = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(t)}{n_i}} \tag{5.1}$$

where w_i is the number of wins which originate from taking move i, n_i is the times move i was visited, c is the exploration constant. It is typically chosen that $c = \sqrt{2}$, since this value has been shown to guarantee convergence to a value function within finite time for single-player games terminal states and rewards bounded to the range $[0, 1]$[16]. t is the total number of simulations for the node considered and is equivalent to the sum of all n_i for all possible moves. The UCB1 equation attempts to balance exploration (looking into paths not yet simulated) and exploitation (looking into paths previously simulated that show good results).

2. **Expansion**: unless the selected node is a terminal state (i.e. the game is over), a child node (W) is created for the next action. Typically, this next action

is randomly selected from all possible future actions.

3. **Simulation**: the *default policy* is used to simulate a random rollout from W. The rollout consists of performing actions at random until the game reaches a terminal state, or up to a fixed number of moves.

4. **Backpropagation**: the result (i.e. utility score) of the simulation is backpropagated to every node, from the expanded W to the root node. This affects future policy decisions, i.e. future selection steps.

MCTS is used for informed critical mechanic discovery in Chapter 6.

5.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a family of population-based optimization algorithms inspired by mechanisms of biological evolution such as natural selection, reproduction, mutation, and fitness. EAs search though generations of candidate solutions (known as individuals or "chromosomes"), beginning with an initial population (usually randomized) and then subjected to an iterative cycle of evaluation of fitness, selection based on this fitness, and re-population via genetic operators to create a new population and begin the cycle again. Some of the more popular algorithms include genetic algorithms and evolutionary strategy [140]. For example, $(\mu + \lambda)$-EA is a technique in which a population of solutions (μ) are iteratively mutated and evolved into a new population (λ). Within this thesis, we leverage two kinds of evolution: Feasible-Infeasible 2-Population (FI2-Pop) and Constrained Map Elites (CME), which are explained in the subsequent sections.

5.2.1 FI2-Pop

FI2-Pop [102] (Feasible/Infeasible 2-Population) is a genetic algorithm which uses 2 populations, one containing feasible and the other infeasible chromosomes. The infeasible population pressures chromosomes to satisfy constraints, while the feasible population pressures chromosomes to increase their fitness. Though each population evolves on its own, chromosomes can freely transfer between them across generations if they satisfy (or fail to satisfy) defined constraints. We use FI2-Pop for projects in Chapters 10 and 13.

5.2.2 MAP-Elites & Constrained MAP-Elites

Constrained MAP-Elites [99] (CME) is a fusion of FI2Pop [103] and MAP-Elites [117], a quality-diversity algorithm which can illuminate spaces. In MAP-Elites, a map of n-dimensions tracks a diverse set of elite chromosomes by storing

each chromosome of differing behavioral characteristics within a cell. Constrained MAP-Elites similarly uses a map, but each cell contains both a feasible and infeasible population rather than a single individual. Just like FI-2Pop, the feasible population maximizes fitness, and the infeasible population aims to overcome constraints. A chromosome can move freely both between populations within a cell if its constraint satisfaction changes and between different cells if its behavioral characteristics change along one of the map's dimensions. CME is used for projects in Chapters 11 and 12.

Part II

Mechanic Discovery and Analysis

Our journey into automated tutorial generation begins with mechanics. Tutorials teach mechanics, and to build a tutorial, you must know what mechanics should be taught. We've broken down this part into three chapters. The first introduces mechanic graphs as a conceptual framework to organize mechanic relationships and two methods of critical mechanic discovery, i.e. algorithms which can search mechanic space for important mechanics to teach. The second chapter explains a theory of Game Mechanic Alignment to analyze mechanics in regards to systemic and agential rewards. The third chapter describes a method which uses mechanics to quickly and accurately classify players according to their in-game behaviors, i.e. their play personas.

Chapter 6

Mechanic Discovery

In this chapter, we define "mechanic discovery" as the process of finding a set of mechanics according to a specific goal, such as teaching them within a tutorial. "Critical mechanic discovery" focuses on finding "critical mechanics": the minimum set of unique mechanics that the player **must** trigger in order to win the game or level in question [61]. The methods we present in this chapter for critical mechanic discovery require the use of a directed "mechanic graph" which can encapsulate mechanic relationships with various game entities. In the following sections, we explain mechanic graph generation and how mechanic graphs can be used within the GVG-AI framework. We then describe and provide examples in GVG-AI for two different search-based methods for critical mechanic discovery: an *uninformed* method and an *informed* method. To gauge the performance of these methods, we run a two-step evaluation procedure: a user study which captures and compares system output against human-identified tutorial content, and a critical-mechanic-informed synthetic agent comparison.

This chapter is based on the work of several papers. The concept of atomic and mechanic graphs (Section 6.1) comes from "'Press Space to Fire': Automatic Video Game Tutorial Generation" published at the EXAG workshop at AIIDE 2017 [64] and "AtDelfi: Automatically Designing Legible, Full Instructions For Games" published at FDG 2018 [60], both with myself as the first author. They also are the source for uninformed critical mechanic discovery (Section 6.2.1) and sibling mechanics 6.2.3. Informed critical mechanic discovery (Section 6.2.2) and experiments comparing the two methods of discovery (Section 6.3) come from "Automatic Critical Mechanic Discovery Using Playtraces in Video Games" published at FDG 2020 [62], with myself as first author. The work within this chapter uses games from GVG-AI framework described in Section 4.1.

Figure 6.1: The atomic interaction graph for GVG-AI's aliens game

6.1 Atomic Interaction and Mechanic Graphs

An "atomic iteration graph" is a directed graph which tracks all actions, game objects, winning/losing conditions, and user-input information in the game. It encapsulates objects, conditions, and actions as nodes. It contains edges, which represent relationships between nodes, inspired by Dormans' mission graph concept [42, 43, 44]. Within GVG-AI, all internal types of objects, conditions, and action nodes in the atomic interaction graph are derived directly from a game's VGDL description. Figure 6.1 displays an atomic interaction graph for the GVG-AI's "aliens" game.

Atomic interaction graphs can be abstracted into "mechanic graphs," where each mechanic is represented as a single node. By representing the game space as a graph, one can easily trace mechanics and find skill chains described by Cook [28] (see Section 2.2).

6.1.1 Nodes

There are three unique types of nodes: *objects*, *conditions*, and *actions*. Objects are anything the player can interact with or see in a game, e.g. characters, enemies and the player avatar. In GVG-AI, "time" is also included as an in-game entity, as some games have termination conditions based on a certain amount of time passing. Actions are verbs the game uses to change in-game objects, such as destroying

a specified game object, transforming a object into another object, or winning the game. Conditions mark events that must take place to enact an action, for example pressing a button, checking if the amount of passed game time is over a certain value, or if the player's avatar has been destroyed.

To integrate with GVG-AI games, the system reads the Sprite Set, Interaction Set, and Termination Set from a game description file written in Video Game Description Language (VGDL). Every sprite found in the Sprite Set is given an *object node* within the graph. Every interaction and termination in the Interaction and Termination Set is given a *condition node* and an *action node*.

In GVG-AI, it is safe to assume that any given interaction within the Interaction Set in a game will have a "collide" condition. However, the Termination Set might contain mechanics with unique conditions, such as counting how many sprites exist in the game or ending the game after a certain amount of time. Sprite nodes involved in an interaction or termination rule are linked in the directed graph to their associated condition nodes, while condition nodes are linked to their associated action nodes. This process' end result is a fully functional directed graph that adequately maps every mechanic in the game.

6.1.2 Controls and Points

Depending on the type of game being played, information about control and input may be contained within a node representing the player character, or stored separately from the graph in case the player is not explicitly controlling a single character. This type of information includes anything that explains the player's ability to affect the game through direct input, such as pressing buttons or using a game-pad.

GVG-AI information about user controls is implicitly stored when the avatar sprite (the sprite whom the player always has direct control of in any GVG-AI game) is registered into a sprite node, as all GVG-AI games require the player to explicitly control a single sprite. The avatar type determines controls and movement afforded to the player. Table 6.1 shows the different types of avatars and how movement information is parsed from them.

Information about points is contained in the specific action that would cause an increase or decrease of points. For example, stomping on a Goomba in *Super Mario Bros* (Nintendo 1985) gives the player 100 points. Within GVG-AI, each interaction in the Interaction Set that results in a change in points registers its mechanic as a point-changing mechanic during graph building, storing how it affects point totals.

Table 6.1: Avatar Type and Movement Parsing

Avatar Type	**Movement**
Moving	"...use the four arrow keys to move."
Horizontal	"...use the left and right arrow keys to move."
Flak	"...use the left and right arrow keys to move. Press space to shoot/use/release [object]."
Vertical	"...use the up and down arrow keys to move."
Ongoing	"...use the arrow keys to change direction. You will not stop traveling in that direction until you change direction again."
Ongoing Shoot	"...use the arrow keys to change direction. Press space to shoot/use/release [object]."
Ongoing Turning	"...use the arrow keys to change direction. You cannot do 180 degree turns!"
Oriented	"...use the arrow keys to turn and move."
Shoot	"...use the arrow keys to turn and move. Press space to shoot/use/release [object]."

6.1.3 Mechanic Abstraction

The system then abstracts these node elements into a "mechanic graph," where each mechanic is represented as a single node. This abstraction is done to better organize the search space into concretely defined mechanic nodes, in contrast to the atomic interaction representation.

In a mechanic graph, any object, condition, or action can be a part of a mechanic node, but they do not exclusively belong to a mechanic. For example, a player object can be a member of a "pickup key" mechanic node, as well as an "open door" mechanic node. To complete this transformation, the algorithm loops over all nodes in the atomic interaction graph; object nodes that are linked directly to a unique condition-action node pair are considered a single mechanic. Mechanics which share input and/or output game objects are linked using an edge, as in Figure 6.3.

6.2 Critical Mechanic Discovery

After building a mechanic graph, we can then begin the process of critical mechanic discovery, a category of tutorial content discovery. "Critical Mechanics" are the set of mechanics necessary to trigger in order to win a level/game. In other words, every winning playthrough will contain this set of mechanics[1].

"Critical paths" are sequences of critical mechanics starting from an explicit

[1]Though, one could imagine a scenario where the player could have multiple choices in a level, resulting in a disjointed set of critical mechanics, depending on the gameplay path selected.

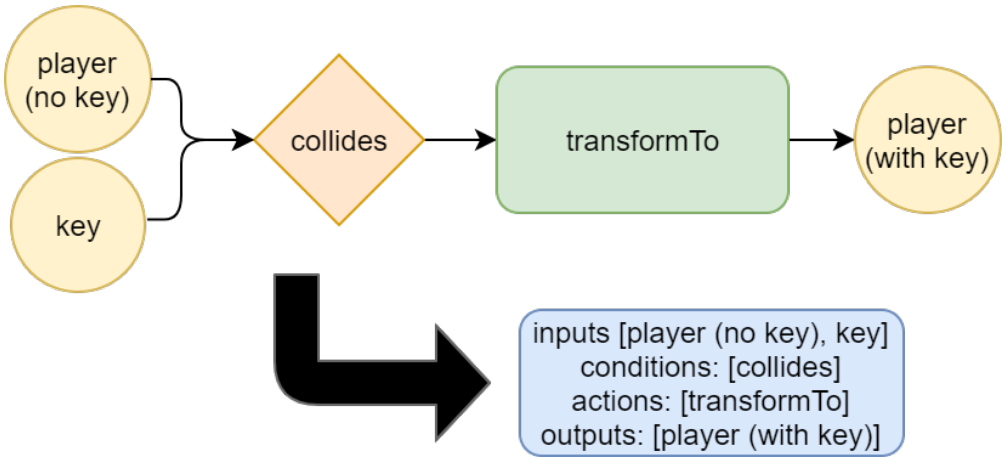


Figure 6.2: A collection of nodes representing a pickup-key mechanic. A player colliding with a key results in the player picking up the key. This can be transformed into a single *mechanic node*.

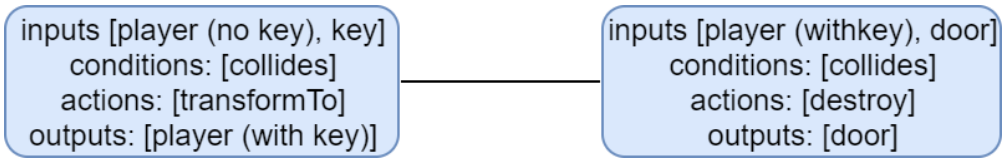


Figure 6.3: An example of how mechanic nodes that share inputs/outputs are linked using an edge. The shared I/O is from the atomic node "player (withkey)."

player input and ending with a winning or losing terminal state within a generated mechanic graph. For example, a player wins *Space Invaders* (Nishikado 1978) when all aliens in a level are destroyed. A critical path for *Space Invaders* would be something like: "Use the joystick to move, press A to shoot. If a missile and an alien collide, destroy the alien (and get points). If all aliens are destroyed, the player wins." Below, we present two different pathfinding algorithms that can be used to find critical paths in mechanic graphs.

6.2.1 Uninformed

Uninformed discovery is named because of its total reliance on mechanic graph information, without augmentation from any agents or players playing the game.

Within GVG-AI, the system finds every avatar sprite node in the graph (in some GVG-AI games, there are multiple avatar sprites). From each avatar sprite node, it then finds every possible path from that node to every winning terminal mechanic. Afterwards, it sorts these paths by length and picks the shortest path for every terminal mechanic. Finally, it iterates over every avatar sprite and picks the longest "shortest path" for each terminal mechanic. Preliminary testing indicates that this method results in a concise critical path which does not cut out important mechanics. In other words, the process searches for the longest path in a list of

breadth-first search results going from avatar-input mechanics to winning terminal mechanics.

For critical paths leading to losses, the system observes every losing terminal mechanic. From each of these mechanics, it works backwards attempting to decipher what might cause the mechanic to activate. In GVG-AI, there are two terminal conditions: either a timeout or the current number of clones of a certain sprite is at some threshold. The first is easy for the system to decipher, as no specific mechanic causes a timeout. The second requires the system to look at the sprite involved in the terminal mechanic, and see what other mechanics would impact its value. Any mechanic that achieves this goal is included in the losing critical path, i.e. any mechanic causing the number of clones of this sprite to rise or fall.

6.2.2 Informed

Informed discovery is named for its use of playtraces from artificial agents or human players to augment the mechanic graph. After a mechanic graph is created and all possible game mechanics are represented, the system informs the graph with playtrace information, which can be collected from human players or automated agents. Given a collection of playtraces for a single game level, the system looks for the playtrace that (1) contains the lowest amount of unique mechanics represented on the graph, *and* (2) in which the player won the level. In doing so, it infers that the playtrace must contain knowledge of which mechanics must be triggered in order to beat the level. By singling out the playtrace with the lowest amount of unique mechanics, it can minimize gameplay "noise", such as accidentally walking into walls (which triggers an interaction with the wall), or triggering other events that have nothing to do with winning the game.

Each mechanic in the playtrace is linked to the particular game-frame in which it occurred. For each unique mechanic triggered during that playtrace, the system looks for the earliest frame during gameplay when that mechanic occurred and enters it into the corresponding node in the graph. Once this has been done for all mechanics, the system performs a modified best first search algorithm over the graph, starting from player-centric mechanics (i.e. those that the player either initiates or is otherwise involved in, like colliding with coins or swinging a sword) and ending with a positive terminating one (i.e. winning the game). The algorithm thus behaves like a greedy best first algorithm that records all mechanics visited until reaching a terminal one. The cost of a node is that node's frame value. The algorithms ends if the current picked node is a terminal node. The pseudocode for this process can be found in Algorithm 1. Therefore, the search generates a path of the earliest occurring mechanics, which then becomes the list of the game's

Algorithm 1 Finding the Critical Path of a Game

```
searchlist = getAllPlayerCentricMechanics()
criticalPath = []
while searchList ≠ [] do
    targetMechanic = targetSeq[pTarget]
    genSubList = generatedSeq.subList(pGenerated, len(generatedSeq))
    mechanicIndex = genSubList.indexOf(targetMechanic)
    if mechanicIndex ≡ -1 then
        missedMechs = missedMechs + 1
    else
        pGenerated = pGenerated + mechanicIndex + 1
        extraMechs = extraMechs + mechanicIndex
    end
    if pGenerated >= len(generatedSeq) then
        pTarget = pTarget + 1
        break
    end
end
return extraMechs, missedMechs
```

critical mechanics.

6.2.3 Sibling Mechanics

Mechanics that are nearly identical in nature, otherwise known as "sibling-mechanics," have the potential to be combined to simplify how content is presented to the user. The exact implementation behind this depends almost entirely on the game, but the motivation is the same: make the tutorial more legible by grouping similar mechanics. For example, instead of saying "the Player can kill Goombas using fireballs and the Player can kill Koopas using fireballs," in Super Mario Bros, the system can simplify it to "the player can kill enemies using fireballs."

Within GVG-AI, sibling mechanics contain identical condition-action pairs and involve sprites that are classified in VGDL as having the same parent [2]. In GVG-AI's Zelda, hitting either a bat or a spider with the sword results in that entity's destruction. In the game's description file, bat and spiders are both identified under a single parent (enemy). If either the bat-sword mechanic or the spider-sword mechanic is contained within the critical mechanic list, the other one can also be included.

[2]https://github.com/GAIGResearch/GVGAI/wiki/Sprites

Age			Game Playing Frequency			
<25	25-34	35+	None	Casually	Often	Everyday
24.7%	68.8%	6.5%	4.3%	36.6%	16.1%	43.0%

Table 6.2: User study participant demographics

6.3 Evaluating Critical Mechanic Discovery

We perform a two-step evaluation method to evaluate if the critical mechanics found using a critical mechanic discovery method are good content for a tutorial. First, a user study compares human-identified mechanics against method-identified ones. The user study experiment evaluates how closely a method matches what humans identify as important mechanics. Second, method-identified critical mechanics can be inserted into MCTS reward functions. An agent-comparison experiment verifies that (at least from the perspective of a game-playing artificial agent) being rewarded for triggering method-discovered critical mechanics results in better agent performance. The following subsections explain the human-identified mechanic comparison study and present the results of MCTS agent comparison study in detail.

6.3.1 User Study Evaluation

Tutorials are primarily meant for human consumption, and therefore, we felt it prudent to compare the output of critical mechanic discovery methods to what human's believe to be the point of the game. Study participants were chosen by sending out a university-wide email to students asking for participation, as well as forwarding to friends and colleagues at other universities. Demographic information about the 93 participants is shown in Table 6.2.

Our user study application displayed a prompt describing the study's purpose. Participants played a minimum of 3 different levels each for 4 GVG-AI games. After completing the levels of a game, participants would be given the following prompt: "In short sentences, describe what the player needs to do in order to perform well in the game." The participants responded using a free-text answer space. We deliberately chose the prompt wording and the answer space to gain insight into what humans consider to be good tutorial content, not just what are critical mechanics. To reiterate, the goal of content discovery is to find good content for tutorials, which may go beyond just winning.

Table 6.3 displays the results of both evaluations. In each game, for every critical mechanic that each discovery method identified, we record the percentage of users who believed the mechanic is important. We also include all other mechanics that participants thought are important but the discovery method does not. The

Game	Mechanic	Percentage	Informed Method	Uninformed Method
Solarfox	Avoid Flames	68%		
	Collide with gems to pick them up	64%	X	X
	Avoid Walls	18%		
	Match Rate	-	**45.45%**	**45.45%**
Zelda	Collide with the key to pick it up	80%	X	X
	Unlock the door with the key	80%	X	X
	Kill Enemies with Sword	76%		
	Avoid dying by colliding with Enemies	60%		
	Navigate the level walls using arrow keys	20%		
	Move quickly	12%		
	Match Rate	-	**48.8%**	**48.8%**
Plants	Press Space to use the shovel	100%	X	
	Use the shovel on grass to plant plants	100%	X	
	Plants kill zombies by shooting pellets	76%	X	
	When plants get hit with axes, both are destroyed	53%	X	
	Protect the villagers from zombies for some time	35%	X	X
	Add plants to different areas to get good coverage	29%		
	Axes don't affect player	6%		
	Match Rate	-	**81.8%**	**11.9%**
RealPortals	Press space to shoot a missile	72%	X	
	If the missile collides with a wall, it turns into a portal	72%	X	
	If a potion collides with water, the water is turned into ground	72%	X	
	Unlock the door with the key	68%	X	
	Collide with the goal to capture it	52%	X	X
	Collide with the key to pick it up	48%	X	
	Pick up different wands to toggle between portal types	44%	X	
	Teleport from the portal entrance to the portal exit	44%	X	
	Collide with a potion to push it	40%	X	
	Avoid dying by colliding with water or portal entrance with no exit	32%		
	If a potion collides with the portal entrance, it is teleported to the portal exit	16%	X	
	You can't go through the portal exit	0%	X	
	Match Rate	-	**94.3%**	**9.3%**

Table 6.3: The *Percentage* column designates the percentage of each mechanic being mentioned by humans in the user study. The X's in the *Method* columns designate that the mechanic was included in the critical mechanic list for that method. The *Match Rate* defines how closely this method agreed with human-identified critical mechanics. For all games, player movement (up-down-left-right) is an implied critical mechanic.

"Mechanic" column contains the aggregated and summarized responses of the user study participants. Because the prompt was free-text, the exact wording of different game mechanics varied, but we attempted to approximate these into the mechanics of the game as they are written in a game's VGDL file. The "Percentage" column shows what percentage of the participants wrote down some form of this mechanic. For each of the games, we calculated each technique's match rate by summing the human-identified percentage value of the critical mechanics discovered by a method. That sum is then normalized over the summation of all percentages. The match rate therefore gives higher weight to the mechanics that more humans identified to be important. These values can be seen at the bottom of each game's section on Table 6.3.

The new informed method either is equivalent to or vastly improves over the baseline for every game when it comes to matching human opinion. Mechanics identified by the informed critical discovery method have the highest percentages of being mentioned by participants in all games except Solarfox. In Solarfox, a slightly higher number of people think that avoiding flames is more important than collecting the gems. We postulate that the constant movement of the player (the player can only change directions, not speed) and the large collision areas of the flames caused some users to focus more on flame avoidance than collecting gems.

Humans not only identify important mechanics for winning but also ones to avoid losing. For example, in Zelda, "Avoid dying by colliding with enemies" is identified by 60% of participants. Other participants note subgoals that usually reflect a better playing strategy, such as "Add plants to different areas to get good coverage." The last mechanic type identified by participants pertains to scoring higher. In Zelda, the "kill enemies with sword" mechanic appears 76% of time, and in Plants, the "Plants kill zombies by shooting pellets" mechanic also appears 76% of time. Interestingly, the informed method does not classify this as a critical mechanic, instead opting to include plants getting hit with axes instead. We believe this is because plants shoot pellets independently of player actions, so by default planting more plants would result in more pellets and thus a higher chance of winning. Thus, this can be condensed down into just "plant more plants." However, axes have a direct negative affect on plants and therefore impact a player's chance of winning. The algorithm found this shorter interaction path ("create plants" - "axes destroy them") to be a simpler choice than including pellet interactions ("create plants" - "plants create pellets" - "pellets destroy zombies").

One system-identified mechanic in Portals, "You can't go through the portal exit," was never mentioned by any of the participants. We hypothesize there may be several reasons for this, one being that the mechanic seems very trivial to humans. It occurs in the playtraces because of the way the game is implemented in VGDL: after teleporting from entrance to exit, the game forces the player to step away from the exit. Participants who beat the game may not have thought it important enough to mention, and players who were unable to beat the game might have never realized that the portals were different types and colors.

6.3.2 Agent Evaluation

Logic would suggest that, when playing a game, knowing how to win and how to lose *should* give you an advantage and allow you to play *better* than if you do not have that knowledge. Therefore, we incorporated critical mechanic knowledge into an MCTS agent and compared its performance against a baseline agent, to see if it played better.

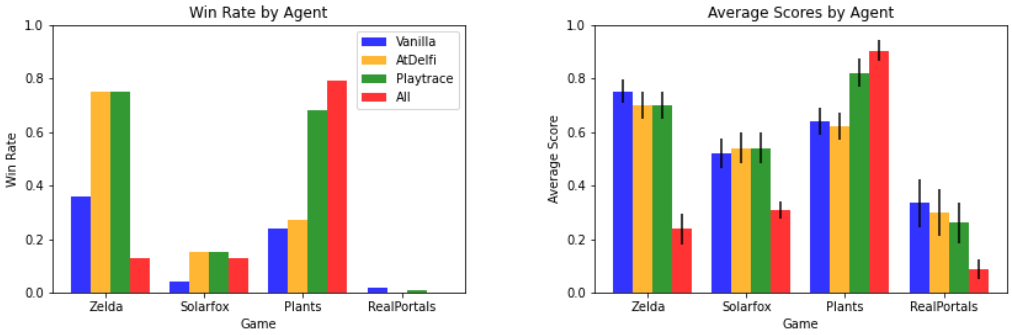
Traditionally, the evaluation function of an MCTS agent takes into account the game state at the end of the *simulation* phase of the algorithm, and then the reward is backpropagated up the tree. We can modify this evaluation function to take into account all simulated event data as well, adding additional rewards for any simulated events that match conditions of critical path mechanics. This is a similar approach to the use of subgoals in hierarchical planning [170], one difference being that the agent is a simple MCTS agent rather than a more complex

hierarchical MCTS or reinforcement learning agent. Another notable difference is that the subgoals defined here are represented as game mechanics, rather than game states. This partial state abstraction affords a greater degree of generality across domains.

The value of these additional mechanic rewards decreases with frequency. Each time the agent triggers a specific mechanic in its past during play, the subsequent reward decreases by $1/frequency$, in order to both encourage the agent to trigger multiple mechanics, and to discourage the agent to keep triggering the same mechanic repeatedly. This reward is also decreased the further out in planning the agent finds the mechanic, similar to discount factors in reinforcement learning. Therefore, mechanics triggered earlier on in planning backpropagate greater rewards than those that happened later. This allows an agent to better focus its search to areas where mechanics trigger early and frequently. The reward equation for a single instance of a critical mechanic during planning is given in Equation 6.1, where F is the number of occurrences this mechanic has been triggered until now, $T_{current}$ is the in game frame where the mechanic was triggered, and T_{root} is game frame at the root node.

$$R = \frac{1}{F * 1.1^{T_{current} - T_{root}}} \tag{6.1}$$

In this evaluation, we compare the performance of an MCTS agent with no mechanic information (vanilla) against MCTS agents augmented with the critical mechanics for Solarfox, Zelda, Plants, and RealPortals discovered using the uninformed the informed methods. The vanilla MCTS agent is a clone of the MCTS agent that comes with the GVG-AI framework and used for benchmarking in other GVG-AI projects. Agents given critical mechanic information have an identical configuration to the vanilla agent, with the sole exception being the Reward Calculation, which is replaced with the process explained above instead of game score. Finally, a second benchmarking agent is given all the mechanics for each game, also being rewarded each time any of these mechanics are triggered. The C value for all agents in the UCT equation was fixed to 0.125. Regardless of mechanic information given, each agent is given 5 unique levels to play for each game, 3 of these levels being identical to the user study levels and 2 being unique to this evaluation. Each level is played 20 times, for a total of 100 playthroughs per game. An agent is permitted to build a search tree of up to 5000 nodes before deciding its next action every turn. An agent is permitted a maximum rollout of 50 moves for each node expansion. All experiments took place on Intel Xeon E5-2690v4 2.6GHz CPU processor within a Java Virtual Machine limited to 8GB of memory. An experiment was allowed to be a maximum of 48 hours long; however,



(a) The win rates of agents on all four games.

(b) The mean normalized scores of agents on all four games.

Figure 6.4: Comparing the performance between the different agents.

none reached this limit.

Figure 6.4a displays a comparison of win rates between the agents, and Figure 6.4b displays average normalized scores with a 95% confidence interval. Scores are normalized by level using the maximum and minimum obtainable scores for that level and then averaged together. Zelda and Solarfox both have fixed maximum and minimum scores for all levels. Because the maximum score value in Plants is based on randomness, we instead score agent performance by their survival time. RealPortals does not have an upper bound on score due to the nature of its game mechanics, so we clamp scoring to the minimum optimal score needed to solve each level.

From Figure 6.4, it can be seen that the informed method was able to achieve better performance than the vanilla MCTS on all games, and better performance than the uninformed method on Plants. The uninformed method appears to have a higher average score on RealPortals. However, due to the confidence interval for both of the augmented agents, we can assume that the score difference between the two is most likely the result of random noise. The low win rates in RealPortals may be a response to the complexity of the game. Because of this complexity, achieving a higher score is a mixed signal: it could mean the agent is closer to winning, but it could also mean the agent is simply abusing the game rule of repetitively going through a portal to get more points. The *All* mechanics agent seems to perform better on Plants, with comparable performance in Solarfox and the worst performance in Zelda.

6.4 Discussion

Based on the results from the user study and the agent experiments, we conclude that the informed method is more successful than the uninformed method

at correctly identifying critical mechanics. This suggests that the method could be a crucial component in a tutorial generation system.

For Zelda, Solarfox, and Plants, the informed method agent results demonstrate significant win-rate improvement over the vanilla MCTS agent when critical mechanics are incorporated into the search algorithm. The informed method outperforms the uninformed method in Plants, due to the inclusion of some highly important mechanics about planting defenses. None of the methods help agents win RealPortals, suggesting there is room for improvement in how this information is incorporated into the agent. This is further supported by the inconsistant gameplay of the All agent, which is rewarded for any game mechanic being triggered. In a game like Plants, where there are 15 mechanics in total, incorporating every mechanic seems to have a strong positive affect. But in Zelda, which contains nearly three times as many, this causes the opposite. We speculate this has something to do with how the MCTS agents are rewarded for mechanic triggers. For Plants, most mechanics are directly or indirectly caused by the player planting more plants. Any action, therefore that involves planting a plant is highly rewarded. In Zelda, however, a good portion of the mechanics involve enemies bumping into walls and each other. When every branch in the tree is rewarded for these stochastic occurrences, MCTS agents behave more like breadth first search agents. The uninformed and informed methods overcome this by allowing the agent to focus on the game-winning mechanics only, and therefore can take advantage of MCTS' "exploitation" factor.

The informed method demonstrates matched or significant increased performance over the uninformed method based on the match rates shown in Table 6.3. Interestingly enough, although the informed method has its highest match rate with RealPortals, an agent augmented with those mechanics only manages to win the game 1% of the time. This situation proves that a two-step evaluation procedure provides a deeper understanding than either being a stand-alone process. In the context of tutorial generation for humans, *a method which helps an AI achieve a stable win rate yet fails to address many of the human-identified critical mechanics cannot be considered very successful.*

Humans identify important mechanics not present in the informed method's critical mechanic set, like the fact that the player can kill enemies in Zelda, that one should avoid flames in Solarfox, or that peas kill zombies in Plants. We can attribute this to the way the informed method searches for the critical path. The goal of the system is to find a *least cost path* using mechanics that result in a winning state, thus it *does not* search for a result that *avoids* a losing state. As a result, it will not actively include mechanics that may be important to players in order to avoid dying or losing the game (such as avoiding flames in Solarfox). We

had expected the informed method to include mechanics like slaying monsters in Zelda or that peas slay zombies in Plants. Due to the way playtrace information is inserted (only one playtrace with the minimum amount of noise is used), the critical mechanics may change depending on what happened this particular playtrace and when mechanics were triggered in relation to each other.

Our method is focused on mechanics being triggered during play, but what it admittedly fails to capture are any mechanics one would *not* want to trigger to win. Solarfox best exemplifies this, where running into walls or flames would cause a loss, i.e. make it impossible to win. Players believed this to be important to mention in Table 6.3. We limited the scope of this paper to include only these "positive" mechanics, as we believe that discovering "negative" mechanics is a non-trivial problem by itself. Hence, our definition of critical mechanic limits itself to mechanics that must be triggered to win. We believe this problem is a research question by itself, and plan to improve our approach to include it in future work.

There is an interesting discussion point to be had in regards to a game like Realportals. Even though agent performance is higher on a scoring basis, neither augmented agent can reliably win levels, and the way that it is gaining points (going back and forth between portals repetitively) can hardly be considered a successful strategy for a human being. Despite this, users strongly concurred with the informed method's mechanics, suggesting that for this game (and perhaps others similar to it in complexity) the agent will have to be more intelligently augmented with mechanics.

Would critical mechanic discovery work for a game not written in VGDL? GVG-AI provides a lot of "shortcuts" for mechanic discovery, enabling us to easily find mechanic relationships with which to build mechanic graphs. However, critical mechanic discovery methods can theoretically work on any directed mechanic graph. Methods for automatically identifying maps [118, 119], mechanics [29] and other characteristics given only an executable version of the game or even video footage [48, 75] show promise here.

6.5 Summary

This chapter introduces atomic interaction graphs and mechanic graphs, as well as an example for how to build them for GVG-AI games by leveraging VGDL. We also defines two search-based methods for "critical mechanic discovery," the process of finding the subset of mechanics which the player must trigger to win a game or level, using mechanic graphs. It stands to reason that critical mechanics could be the set of foundational mechanics to be taught in tutorials, which would make them inputs for a tutorial generative engine.

These experiments are not just in the earlier sections of my thesis by design: they were some of the first theories I wrote during my PhD. But I often second-guessed myself about the concept of critical mechanics being the sole input to tutorials. You see, critical mechanics are the same for everyone: all players receive the same opportunities and winning conditions in a fair game. The next chapter explains the course of my research after I (as well as two of my committee members Dr. Smith and Dr. Togelius) wondered "Who is the tutorial for, and what do they care about?"

Chapter 7

Game Mechanic Alignment Theory

Up to this point, we have defined critical mechanics as those required to trigger to win. We've also demonstrated that we can discover critical mechanics using mechanic graphs and tree search methods in order to be used as tutorial content. But player curiosity and enjoyment should not be limited to a binary choice of "winning" and "losing". We should also strive for tutorials to be more than explanations of how to win. Players engage with games with their own biases, motivations, and goals, which impact the enjoyment they receive from play. For example, in the game Minecraft (Mojang 2007), a voxel-based open-world sandbox game, the designers present the option for players to travel to the "End," a dangerous zone of monsters and treacherous terrain, to defeat the Ender Dragon, which could act as the game's final boss. However, many players may choose to never travel to the End, instead selecting to build large castles and design elegant structures that are aesthetically pleasing. Some players set TNT alight in an explosive attempt to blow up as much of the rendered world as possible without crashing their server [1]. Admittedly, defeating the Ender Dragon does not explicitly end the game either, as the player can keep on playing in their world with the ability to slay the Ender Dragon repeatedly.

How can you measure the impact of a player's goals on their in-game behavior? Elias, Gutschera and Garfield [49] distinguish between "systemic" and "agential" properties of game characteristics of a game. Systemic properties derive from the game's rules, and agential properties depend on the players. In this work, we consider that the choice of a player to trigger a mechanic can be made through a combination of systemic and agential factors: a mechanic provides "systemic rewards" if it results in the player obtaining an external (to the player) reward signal

[1] Video of TNT: https://www.youtube.com/watch?v=o9V2FOyD-GM

created by the designer to encourage certain behavior (e.g. points) or if it directly contributes to progression or winning. However, mechanics can also be pursued due to "agential motivations" that come from the player. These motivations may be aligned with or run against systemic rewards.

This chapter presents a *Game Mechanic Alignment* theory, a framework in which mechanics can be analyzed in terms of agential motivations and systemic rewards in an attempt to extract the impact of player goals on in-game actions. It also presents a way to use Game Mechanic Alignment as a tool to extract important mechanics to teach within tutorials. Our review includes examples of Game Mechanic Alignment applied to some well-known games, a methodology that can estimate mechanic alignment given playtrace data, and an application of this methodology in the research framework Minidungeons 2 (Section 4.3). We conclude how the theory can be applied as input to automated tutorial generation systems as an alternative method to critical mechanic discovery. The theory and estimation method part of this chapter comes from "Game Mechanic Alignment Theory" with myself as first author published at the International Conference on the Foundations of Digital Games, 2021 [65]. The findings presented in Section 7.4 is new research that has not yet been published.

7.1 The Theory

This "Game Mechanic Alignment" theory offers a framework in which game mechanics can be categorized in terms of the *system* (the game) and the *agent* engaging with it. Usually, games contain designer-defined reward systems which are consistent regardless of who is playing. The impact these reward systems have on play and the form they take highly depends on the conscious decisions of the designers. In a general sense, these reward systems can be interpreted as systemic penalties versus systemic rewards, which usually guide the player toward winning and away from losing. This is a separate concept from the environmental reward explicitly defined in reinforcement learning environments [157]. In games that do not have explicit winning conditions, such as Minecraft (Mojang, 2008) or The Sims (Maxis, 2000), this condition can be substituted with another win-like condition, such as defeating the Ender Dragon in Minecraft, getting a job in The Sims, having X amount of happy customers in RollerCoaster Tycoon (Chris Sawyer, 1999), etc.

In addition to built-in systemic rewards, player-specific incentives also influence how a player engages with the game. For example, speed runners and casual players have very different goals and motivations. A speed runner may bypass, skip, or glitch their way to the end of the game. A casual player may take their time and explore the environment. They may even test out losing to better understand

how certain failure mechanics work. Thus, a player's goals may be independent of the game's systemic rewards. A game may reward a player when they move toward the right side of the screen, but a player may want to first collect every powerup before moving on to the next checkpoint. In an extreme case, a player interested in exploring the losing mechanics of a game may find themselves moving counter to the systemic rewards. For example, players in Minecraft may wish to build traps for their bases, which they themselves may test the effectiveness of. If agential incentives and systemic rewards are in agreement for a specific mechanic, we consider this mechanic to be *in alignment*.

We can think of mechanics as existing in a 2D space according to how they correlate to systemic reward systems and agential incentives. The extremes of the "Systemic Rewards" axis would coincide with critical and fatal mechanic correlations, referring back to critical mechanics in Chapter 6. "Fatal mechanics" are the opposite of critical mechanics: they are the set of game mechanics which, if triggered, result in losing the game. At the extremes of the "Agential Incentives" axis lie mechanics that the player feels interanally incentivized to pursue or to avoid:

- *Incentive Mechanics*: The set of mechanics that a specific player is incentivized to trigger over the course of the level to accomplish that player's subgoals.
- *Avoidance Mechanics*: the opposite of incentive mechanics. The set of mechanics that a specific player avoids triggering over the course of the level to accomplish that player's subgoals.

The intersection of agential incentives and systemic reward axes creates an origin point, aka the "neutral" zone, where the player's behavior is neither influenced by systemic reward/penalty or agential motivations/aversions or only marginally influenced. The quadrant that a mechanic inhabits will embody the relationship that a mechanic has in terms of the environment and the player. Figure 7.1 shows the player incentives and systemic rewards space plotted as 2D axes where x-axis represents systemic rewards and the y-axis represents agential incentives. These two axes divide the space into four quadrants. They are, in counter-clockwise order:

- Quadrant 1 (Green): Both the environmental rewards and the player's motivations encourage the player to trigger this mechanic.
- Quadrant 2 (Yellow): The environment punishes the player but the player wants to trigger this mechanic anyways.

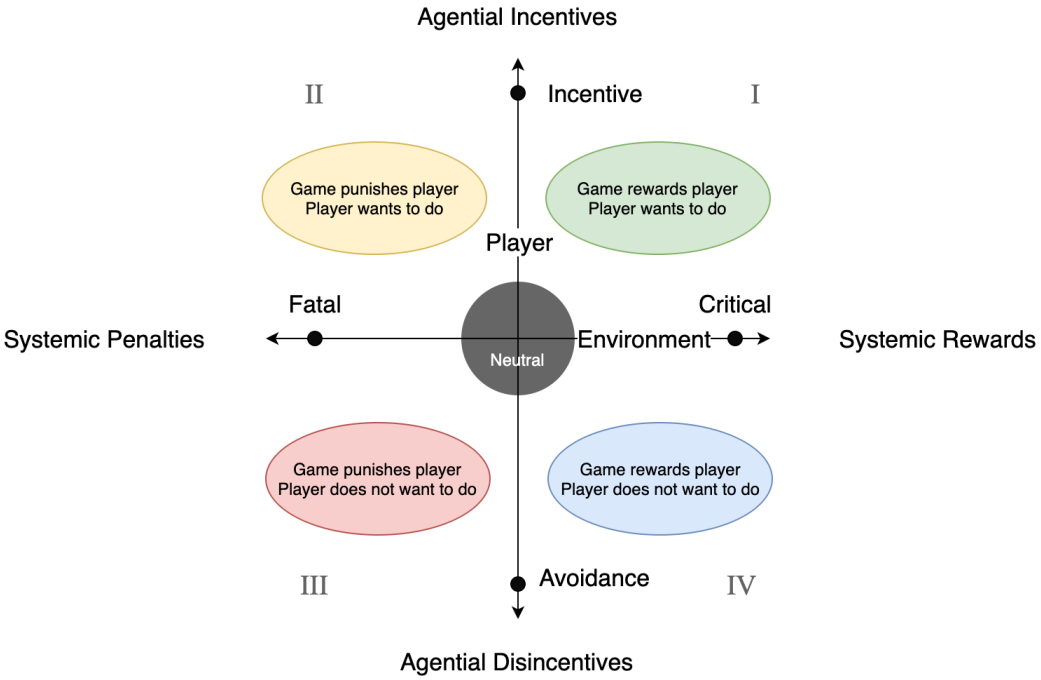


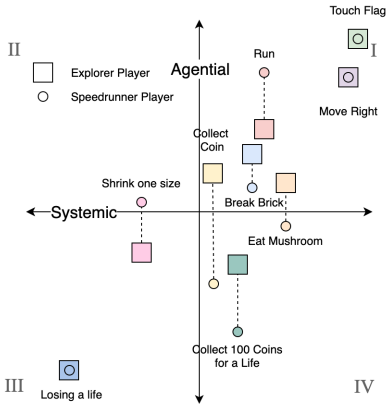
Figure 7.1: Game Mechanic Alignment Theory

- Quadrant 3 (Red): Both the systemic rewards and the agential motivations discourage the player from triggering this mechanic.
- Quadrant 4 (Blue): The environment rewards the player but the player wants to avoid triggering this mechanic regardless.

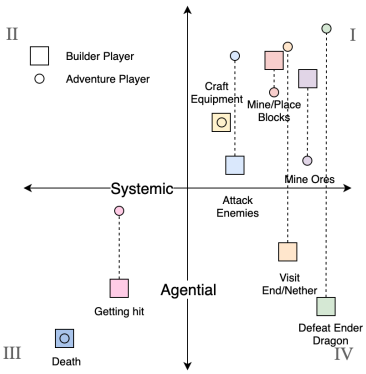
When game mechanics are within Quadrant 1 or Quadrant 3 (green and red zones), they can be considered to be "in alignment". In other words, both the environment and the player are in agreement in regards to rewards and incentives. Perfect alignment would be at the $y = x$ line. However, if mechanics lie within Quadrant 2 and/or Quadrant 4 (yellow and blue zones), they are player-environment misaligned. The player may be motivated to take actions that cause environmental penalties, or else refuse to take actions that would otherwise give them rewards. It is important to note that being "in alignment" is not a judgement of player ability, nor is it a statement on what is "correct" behavior in the game. It is simply a way to categorize mechanics relative to the systemic rewards of the game and how often the player is motivated to trigger them during play.

7.2 Examples of GMA

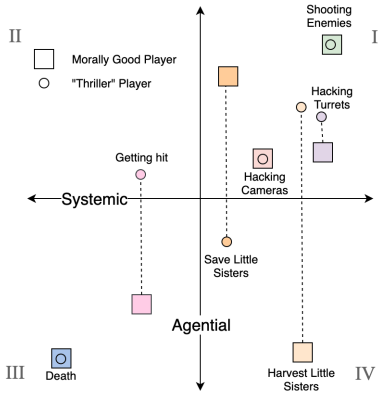
To demonstrate the usefulness of this framework, in this section we present examples using several well-known games: *Super Mario Bros* (Nintendo 1985),



(a) Explorer and Speedrunner in Super Mario Bros



(b) Builder and Adventurer in Minecraft



(c) Morally Good and Thriller players in Bioshock

Figure 7.2: Examples of alignment axes for different players in Super Mario Bros, Minecraft, and Bioshock

Minecraft (Mojang 2009), and *Bioshock* (Bioware 2007). For each game, we highlight aspects relevant to the discussion of player incentives and systemic rewards and provide mechanic alignment charts as examples of how player incentives associated with certain actions could differ for two hypothetical player profiles. Since these examples are simply for illustration purposes, the values on the x and y axis are meant to be taken qualitatively. In each example, we simplify the number and description of the game mechanics: each has many more mechanics and they may be described subjectively different depending on the individual doing analysis.

7.2.1 Super Mario Bros

While much has been said about how the Mario series teaches the player through careful level design [53], a lot of the discussion around the series focuses on critical and fatal mechanics such as moving to the right, collecting power ups and jumping over hazards. These mechanics directly help the player progress towards

winning states or avoid losing states. However, the game also acknowledges and incorporates in its design various agential motivations that players are likely to exhibit.

For example, the scoring system featured in the early games of the franchise rewards the players both for actions that directly help the player progress and avoid losing, such as killing enemies and picking up power-ups, and also for actions that display mastery over the game which have high appeal to advanced players, such as grabbing the flagpole at a higher spot and finishing the level faster.

The collectible coins fulfill multiple roles. At first, they provide no immediate benefit other than serving as a token of player-motivated achievements such as mastering a tricky jump or uncovering secrets. But once a certain number of coins are collected, the player is given an extra life, which helps avoid a game-over state. Finally, coins and other collectibles in Mario and similar games can be used to lead the player guide the player toward secrets or suggest alternate paths that may require a power-up, which Khalifa et al. [94] call *foreshadowing*. Following these cues can lead to both systemic rewards and fulfilling player goals.

Designers can take steps to align systemic rewards with the actions that satisfy the player's motivations. Collecting the coin in Mario is not only systematically rewarding, its also aesthetically pleasing to the common player. The coin collection sound is a positive, happy sound that players may enjoy hearing. Collecting a coin satisfies a curious player's urge to explore. Figure 7.2a suggests a possible Agential-Systemic Mechanical Alignment chart for the mechanics of the game and the hypothetical player profiles of "explorer" and "speedrunner". Both players are ultimately interested in beating the level, but the explorer player does this in a slower-paced and safer way. The explorer enjoys eating Mushrooms for extra safety, breaking Bricks out of curiosity for what's inside, and collecting coins that are easily accessible. The speedrunner, as the name suggests, wants to beat the level with the lowest possible in-game time and will avoid collectibles and bricks unless this results in a faster clear. The speedrunner will even occasionally take damage on purpose to shrink size in order to go through narrow paths, making its y-axis value higher than the explorer player. While neither player wants to purposefully waste time, the explorer will only choose to run when it is safe to do so, while the speedrunner will run most of the time, even at the risk of losing a life (and thus restarting the level and the timer). That is why the speed runner has a higher value for "Run" mechanic over explorer.

For the x-axis values, we assigned relative order for these mechanics for what we think the game designer wants the player to do to finish the game. That's why "Touch Flag" and "Move Right" have the highest x-axis value, as they provide progress toward finishing the game, while "Losing Life" and "Shrink one size" have

the lowest values on the x-axis as they prevent progress.

7.2.2 Minecraft

Minecraft presents an interesting case of study for our framework since much of the appeal of the game comes from fulfilling player-incentivized goals. While triggering the credits by beating the Ender Dragon could be considered "winning" the game in a traditional sense, players are free to ignore this goal (potentially indefinitely) and focus on exploring, mining, crafting and building structures.

Figure 7.2b illustrates the mechanical alignment for two hypothetical player profiles: the "builder", who takes enjoyment from building structures, and the "adventurer", who seeks challenging mob encounters, including the Ender Dragon and other opponents found in the Nether and the End dimensions. Both players place equal value on crafting equipment as it enables both in reaching their goals. Neither player desires to be hit, but the adventurer is more comfortable with the possibility, and gets hit more often as consequence of the more frequent combat encounters.

For the x-axis values, we assigned them from the perspective of the designer who wants the player to reach the credit scene and finishing the game. As you can see, all mechanics which provide progress toward reaching this goal are on the extreme right side such as "Visit End/Nether," "Mine Ores," and of course Defeating the Ender Dragon. While all the mechanics that hinders this progress lies on the extreme left such as "Getting hit" and "Death." Minecraft contains hundreds of block types, and several different types of ores. To simplify this exercise, we have elected to merge all blocks and ores into generic mechanics. However, in theory a designer could make each of these their own mechanic for more granular analysis.

7.2.3 Bioshock

In *Bioshock*, the player is faced with an important choice regarding the fate of characters known as *Little Sisters* which incurs in both moral (player incentivizes) and systemic implications. When meeting one of these genetically-modified young girls, the player can choose to either harvest or save them. Harvesting them yields more ADAM, a systemic reward that serves as in-game currency for upgrades, but results in the child's death. Saving them yields less ADAM but can create other environmental benefits and can lead to a "happier" ending. Thus the game pits a player's motivations for taking a moral path and reaching the "happy" ending against environmental considerations.

Figure 7.2c illustrates a possible alignment chart for two hypothetical player profiles: the "morally good" player and the "thriller" player. Both these players

have the same final goal which is beating the game, but the morally good player is trying to reach it with the least amount of killing, while the thriller only cares about power and destruction. That is why they differ on the "Little Sisters": the thriller desires more power, so they is highly motivated to harvest the sisters. On the other hand, the morally good player feels obligated to save the sisters, and they might want to reach the good ending of the game.

Similarly to all the previous games, we sort the mechanics on the x-axis such as mechanics near the right are pushing the player to progress in the game such as "Shooting Enemies" and "Hacking Turrets". While, mechanics near the left blocks that progress and cause the player to replay certain areas such as "Death" and "Getting hit".

7.3 GMA Estimation

In this section, we propose a method for automatically estimating agential incentives and systemic rewards for the game mechanics for a given game/level. This method uses a sizable distribution of playtraces from a diverse set of artificial agents/human players where each agent/player plays the same level multiple times. A large group of player data allows for the influence of player incentives to be separated out from the environment specific motivations and thus mechanic alignments to be calculated for many types of players. Similar to any statistical technique, the bigger and more diverse the input data, the more accurate are the estimations.

Without loss of generality, the method to calculate how critical/fatal a mechanic is is fairly straightforward and intuitive. We simply want to know how often a mechanic occurs in a winning/losing playtrace as opposed to how often it occurs in any playtrace in general. Rather than calculate a binary value for *if* a mechanic is critical or not, we can calculate an approximate value for **critical-ness**. The frequency of a mechanic across playtraces can be viewed as a probability density function (PDF) that represents how likely a mechanic is to occur in a playtrace. It then follows that the difference between the PDF of the mechanic and the PDF of the mechanic given winning/losing would give a quantitative measure of how critical a mechanic is. This can be applied directly to player motivations by simply conditioning the mechanic on playtraces of the given player. We use the Wasserstein distance [173] to calculate the distance between the two distributions. The following equation shows the general form of the distance calculation.

$$D_{m,c} = W_1(PDF(m|c), PDF(m)) \tag{7.1}$$

where m is the current mechanic, c is the current condition (win in case of systemic rewards and $agent$ in case of agential motivations), and W_1 is the first Wasserstein distance, $PDF(m|c)$ is the distribution of the mechanic m given the condition c happening, and $PDF(m)$ is the distribution of the mechanic m in all the playtraces.

The PDF can be calculated directly from the discrete data. The frequencies for each playtrace can be normalized to form a rough, discrete, approximation of the true PDF, i.e. a histogram. The Wasserstein distance can then be computed directly on this discrete data to compute the estimated distance. This approach has the benefit of not needing to fit any distribution to the data and therefore does not require any assumptions about the data distribution. Calculating the result over a PDF also would allow for this approach to be used when the PDF is known through other means.

The distance proposed in equation 7.1 is a scalar value between 0 and 1, and it does not express direction in the space. To calculate the direction, we compare the mean of $PDF(m)$. If the conditional distribution has a high average compared to the overall distribution, this means it is encouraged, while if it has lower average, it is discouraged.

$$S_{m,c} = Sign(\mu_{PDF(m|c)} - \mu_{PDF(m)}) \tag{7.2}$$

To calculate the final rewards, both systemic (E_m) and agential (I_m), we combine both equations 7.2 and 7.1 to get the final values. Equation 7.3 shows the final equation for both rewards where the difference is the agential incentive estimation is conditioned on a certain agent(s) are playing, while the systemic reward is condition on having a winning playtraces.

$$\begin{aligned} I_m &= S_{m,agent} \cdot D_{m,agent} \\ E_m &= S_{m,win} \cdot D_{m,win} \end{aligned} \tag{7.3}$$

7.4 Application to Minidungeons 2

In this section, we apply GMA on the 858 persona-labeled human playtraces of Minidungeons 2 (training and testing sets) in order to understand the level of incentive/avoidance that players have about different mechanics. The specifics of how this dataset is made is covered in Section 4.3. Unfortunately for critical/fatal analysis, not many humans lose ($< 2\%$ of the playtraces), so our methods to calculate critical/fatal are not very accurate due to the lack of diversity. However, there is a diversity of playstyles as shown by the "Count" in Table 7.1. In this dataset, there are 3 pure playstyle labels:

1. R: Runners who race through the level as fast as possible.

	Count	EnemyKill	TakeTurn	UsePortal	CollectTreasure	CollectPotion	JavelinThrow
No Label	136	-0.147	-0.105	0.026	-0.202	-0.177	-0.013
R	243	-0.11	-0.122	-0.022	-0.213	-0.225	-0.035
TC	92	0.113	0.242	0.05	0.397	0.206	0.104
MK	3	0.157	0.309	-0.044	0.212	0.324	0.174
R/TC	108	0.052	-0.046	0.016	0.113	-0.07	-0.014
R/MK	69	-0.119	-0.132	-0.035	-0.22	0.16	-0.029
TC/MK	34	0.232	0.494	-0.044	0.517	0.584	0.058
R/TC/MK	173	0.161	0.083	0.009	0.161	0.209	0.015

Table 7.1: GMA scores on the entire dataset for each of the 8 Minidungeon 2 personas. Also included is the number of playtraces in each persona category. Gray highlighting signifies a negative score.

	1	2	3	4	5	6
No Label	CollectTreasure	CollectPotion	EnemyKill	TakeTurn	UsePortal	JavelinThrow
R	CollectPotion	CollectTreasure	TakeTurn	EnemyKill	JavelinThrow	UsePortal
TC	CollectTreasure	TakeTurn	CollectPotion	EnemyKill	JavelinThrow	UsePortal
MK	CollectPotion	TakeTurn	CollectTreasure	JavelinThrow	EnemyKill	UsePortal
R/TC	CollectTreasure	CollectPotion	EnemyKill	TakeTurn	UsePortal	JavelinThrow
R/MK	CollectTreasure	CollectPotion	TakeTurn	EnemyKill	UsePortal	JavelinThrow
TC/MK	CollectPotion	CollectTreasure	TakeTurn	EnemyKill	JavelinThrow	UsePortal
R/TC/MK	CollectPotion	CollectTreasure	EnemyKill	TakeTurn	JavelinThrow	UsePortal

Table 7.2: Mechanic importance as ranked by GMA values for each persona. Mechanics in gray are discouraged.

2. TC: Treasure Collectors, who want to collect treasure.

3. MK: Monster Killers, who want to slay enemies.

There are also personas which are made up of mixtures of the above playstyles, like R/MK, or play like none of the playstyles which are labeled "No Label", as explained in Section 4.3.4.

7.4.1 GMA Calculation

For the purposes of this experiment, we have cut down the 17 mechanics of Minidungeons into 6. The EnemyKill mechanic references anytime the player slays an enemy, regardless of enemy type, so we use that rather than individual MonsterHit mechanics. We remove the Reach Stairs and Die mechanics since we do not want to analyze winning and losing. We also remove the BlobPotion, BlobCombine, and OgreTreasure, and TriggerTrap mechanics for simplicity. Table 7.1 displays the GMA vectors for all 8 playstyles when calculated using the agential estimation equation (Equation 7.3). In this table, scores that are negative have been highlighted in gray.

To better understand these results, we rank the mechanics based on the magnitude of their GMA scores within each persona in Table 7.2, where an "incentive" mechanic is in white and an "avoidance" mechanic is in gray. "No Label" and "Runner" personas avoid collecting any items, taking many turns, or killing any

monsters. “Treasure Collector” and related TC-hybrid personas are incentivized to take many turns and collect treasures. There are only 3 total instances of the pure “Monster Killer” persona, so we should take with a grain of salt the fact that slaying enemies is not a heavily prioritized mechanic, even less than collecting treasure. The TC/MK and R/TC/MK personas slay enemies, while R/MK personas seem to want to avoid slaying enemies as much as pure Runner personas do. We believe this to be an artifact of Action Agreement Ratios, where a playtrace is labeled as a persona if it’s action sequence agrees with that of that persona at least 50% of the time. With this in mind, its possible for a playtrace to run toward monsters placed near the exit and avoid killing them, thus agreeing with both a Runner and a Monster Killer persona. Map 101 and Map 201 from Figure 4.4 are perfect examples of where this occurs. Some playtraces on these maps will move down toward the exit, while also on the path to the nearest monster for more than half of their moves.

7.4.2 Extracting Mechanics

A player may eventually know about *every* game mechanic at some point to fully experience a game. Discovering mechanics and how they interact with one another is most often a core part of the gameplay and player enjoyment. Using GMA, we can estimate the priority of certain mechanics over others in regards to “who” is playing. For example, if I wanted to make an instruction-based tutorial for Runner personas, my instructions in order of importance may look like this:

1. Do not collect potions.
2. Do not collect treasures.
3. Do not take many turns/be fast.
4. Do not kill enemies.

In this example, we removed the last two mechanics (JavelinThrow and Use Portal) as they have low GMA magnitudes (-0.035 and -0.022 respectively). We can also use different wording depending on priority to better emphasize mechanic importance to the player:

1. NEVER collect potions.
2. NEVER collect treasures.
3. Do not take many turns.
4. Try not to kill many enemies.

(a) A level giving the player the option to collect items

(b) A level that forces speedrunning past enemies

Figure 7.3: A sequence of Minidungeons 2 levels which teach a player to play like a Runner

Another way to teach these mechanics is through a level or series of levels in which the player is faced with choices to either collect or not to collect potions and treasures, like the level in Figure 7.3a. In this level, the player can rush straight to the exit as a Runner does, or they can collect some of the additional on screen objects. We can follow that level with the one in Figure 7.3b. If the player initially runs to the exit in the first level, then this behavior is reinforced in the second level, which forces the player to rush through without collecting treasure or else they will lose. If the player initially collects the treasure in the first level, then the second level will teach them to not always collect treasure. Persona identification is covered more thoroughly in Chapter 8 while persona-dependent level generation is covered in Chapter 12. By using GMA to estimate priority, we can utilize it interchangeably with critical mechanic discovery methods depending on input representation (mechanic graphs or playtrace data) and whether personalization matters or not, as Figure 7.4 displays. In either case, we can extract mechanics as content for tutorials.

7.5 Discussion

Game Mechanic Alignment allows us to visualise how different players engage with various game mechanics according to their internal incentives and systemic rewards. This has many potential applications, such as categorizing players based on the y-value of different events (e.g. through clustering), validating a designer's assumptions of what features of the game will be most enjoyed by players, building reward systems that are aligned (to the desired extent) with player's preferences

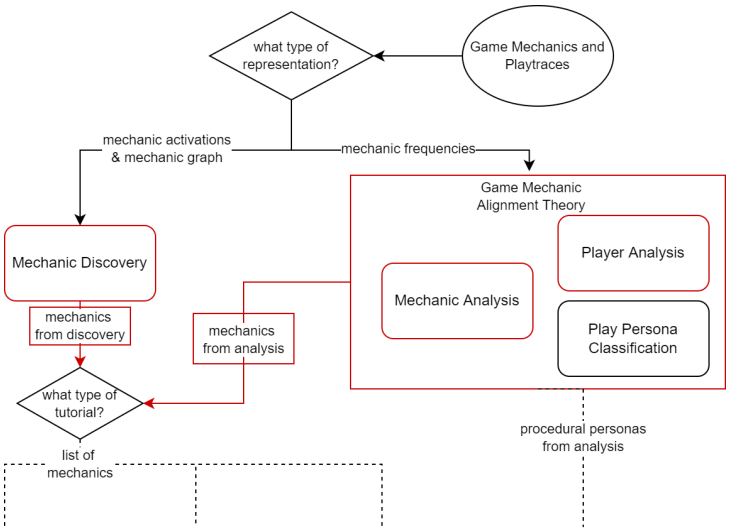


Figure 7.4: Outlined in red, mechanics for tutorial content could come from either mechanic discovery or from Game Mechanic Alignment using mechanic/player analysis.

and building tutorials that support actions preferred by different types of players, even if they are not all equally rewarded by the reward systems.

Within the context of this thesis, GMA provides an alternative way to extract mechanics for tutorial content. These mechanics can be a set of mechanics prioritized by their critical/fatal importance (the x-axis of GMA), or it can be a set of mechanics prioritized by as player's preferences (the y-axis of GMA). Figure 7.4 displays the top half of Figure 1.1 from Chapter 1 to explain how mechanic content can come from either method to be used in any of the three tutorial types. To determine which method to use, you need to figure out your input data representation (mechanic graph/mechanic activations from playtraces or just mechanic frequencies from playtraces) and if you want a personalized tutorial or not.

7.5.1 Skill and Difficulty

One limitation of GMA estimation is that it relies on the correlation between triggering a mechanic and winning the game (for the x axis) or between triggering a mechanic and belonging to a certain player profile (for the y axis). These correlations are not necessarily causal/intentional and could be affected by spurious factors. One such factor is skill. Consider a hypothetical level where a cosmetic item with no functional value is hidden near the exit. The event of picking up this item would have a higher correlation with winning than losing, and therefore would be to the right in the x-axis, even though it does not improve a player's chance to beat the level. If two players A and B are equally incentivized to pick up this

item, but player A reaches the end of the level more often, the event would show up higher on the y-axis for player A, because player A had more opportunities to trigger it due to their higher player skill.

The *Talin* [8] system arguably touches upon this more directly. In Talin, mechanics are dynamically taught to the player based on their skill level. Mechanic mastery is represented using scalar values initialized by the game designer. As the player plays and either triggers or fails to trigger that mechanic, its value rises or falls. A similar system could be used to measure player skill/mastery of particular mechanics to explain how much of a y-value difference between different players is a result of their skill versus incentives. If we can be sure of the method's accuracy, it may be more beneficial to designate another axis to account for player skill. However, we want to make it clear that being unable to differentiate between incentive or skill when it comes to player behavior does not negate the utility of this technique. Two experiences could be offered to the player to counter this: one being a tutorial giving an easier level to help "practice", the other giving the player a more challenging level with more complex uses of the mechanic to provide entertainment. Regardless of it being due to skill or incentives, the game system/designer has a deeper understanding of what the player was doing during play than without the technique.

Level difficulty also may play a role here. Many games present the player with procedurally generated sets of levels, such as Spelunky (Derek Yu, 2008). In games like these, a designer may not be able to make the assumption that all levels are equally difficult. If we assume there exists a method for measuring level difficulty, then we could test our agents only against the levels with similar difficulty and the same set of game mechanics.

7.5.2 Game and Level Design

Idiosyncrasies of a particular game or level could also contribute to spurious correlations. For example, if another hypothetical level contains a bifurcation where the player can choose between a red path heading North and a blue path heading South, and these paths feature opportunities to trigger different sets of events, then a naive analysis of the prevalence of these events might be confounded by an aesthetic preference for the red or blue color.

It is possible, in principle, to attempt to correct for spurious correlations. For example, if we know an event can only be triggered at a certain point of the map, we could consider only play traces that got within a certain distance of that point. Or we could, at each time-step of the play trace, simulate the game for a number of steps with the goal of triggering the event, and consider only play traces where

it was possible to do so. We could also perform A/B tests where some features of the level (e.g. the events on each path) are kept constant where others (e.g. the colors) are manually swapped. But these corrections would come at the cost of domain knowledge and/or computational power, which is why they were not considered in our experiments.

7.5.3 Time-dependent Mechanics

All of our above examples refer to game mechanics with rather obvious immediate rewards/penalties. Let us consider the game Fallout 2 (Bethesda 1998), a first person shooter set in a post-apocalyptic Oregon, United States. In the game, the player may encounter a consumable substance called "Jet," a highly addictive meta-amphetamine which grants the player temporary short-term bonuses to their combat abilities. After the initial bonus period, however, the player will suffer heavy penalties to these same skills. Additionally, the player may become addicted to the substance, requiring them to keep dosing themselves every day or suffer additional penalties. In this example, consuming Jet has opposing systemic reward/penalties, depending on the time horizon: rewarding in the short-term, penalizing in the long-term. To visualize this using Game Mechanic Alignment, we may need to introduce a third axis to represent "time."

7.6 Summary

This chapter presents a theory of Game Mechanic Alignment as a framework with which to articulate the systemic-agential forces in play on a mechanical level. It builds upon the previous chapter about mechanic discovery, and provides space for future work on other subsets of discovery such as "incentive mechanic discovery, "avoidance mechanic discovery," and "fatal mechanic discovery." GMA does not require the use of a mechanic graph like critical mechanic discovery does, but it does require a set of playtrace data. However, GMA does not define any single way to interpret the results. The output of GMA analysis is up to the analyzer: if a player tends to utilize a certain mechanic more than most, is that on purpose or on accident? How deeply does player skill play a role? Context is key, and automatically verifying player intent is a non-trivial endeavor. However, Game Mechanic Alignment provides the groundwork necessary to analyze mechanics and sets the stage for detailed player analysis and tutorial generation in the later parts of this thesis.

Chapter 8

Play Persona Classification

In this chapter, we present a way to rapidly classify players by their gameplay behavior. "Play personas" are defined as archetypal models of player behavior inferred from experience or observed data. Knowing a player's play persona can be critical to correctly serving personalized tutorial content. If a player's persona is known, play personas can be computed by calculating the action agreement ratio (AAR) between a player and a generative model of playing behavior, a so-called procedural persona [79, 80, 81]. However, AAR calculations are computationally expensive and assume that appropriate procedural personas are readily available. Often times in industry, game studios have hundreds to thousands of trace telemetry datapoints streaming in by the hour. For large companies this can be even greater. For example, King, the developer of Candy Crush Saga[1], reports a daily average of 13 million active players[2]. For large or streaming datasets, calculating AAR can be an incredibly expensive computational challenge, even with parallelization.

We propose that mechanic frequencies and game trace states can be used in lieu of AAR to rapidly classify players with acceptable levels of accuracy. To prove this, we train classifiers on human playtrace mechanic frequencies using AAR labels in MiniDungeons 2, and infer on a testing set of more human playtraces using AAR labels. For comparison, we also train a classifier on the actual playtrace (sequence of game states) and compare its results to the ones trained on the mechanic frequency vector. Additionally, we train both classifiers on synthetic generated data using the same personas used to generate the AAR labels. While the classifiers trained on synthetic data fail to generalize to the test set, the classifiers trained on AAR labeled human data is able to accurately classify human playtraces. All of the work in this chapter comes from "Predicting Personas Using Mechanic Frequencies and Game State Traces" published at the World Congress for Computational Intelligence 2022 [66] and uses the Minidungeons 2 framework described in Section

[1]https://www.king.com/game/candycrush
[2]https://activeplayer.io/candy-crush-saga/

4.3.

8.1 Classifying Players

In a Minidungeons 2 gameplay session, we record every state and game element interaction. Please revisit 4.3 for details about game state and mechanics. We use the Minidungeons 2 human-playtrace dataset described in Section 4.3.4. Out of the total 853 playtraces, 565 playtraces are used as the training set data, and 293 playtraces are used as the test dataset. The training set data id split 70-30 for training and validation. All human playtraces are labeled using Action Agreement Ratios. We also use the Minidungeons 2 synthetic dataset of 1500 playtraces described in section 4.3.3. Players can be labeled as Runners (R), Treasure Collectors (TC), or Monster Killers (MK). We train two models using two data representations and test them on the test set. Our models use the following data representations:

- **Cropped playtraces:** We convert the states of the playtraces into a smaller state by cropping the states to focus on the 3x3 area around the player. This particular size was used because player interactions and the mechanics examined for the MiniDungeons game only affect the immediate surrounding area of the player (i.e. collecting nearby treasure or potions or attacking.) The smaller size also prevents the machine learning model from overfitting and ideally helps it generalize better [177].

- **Mechanic frequencies:** We use the frequency of the 17 triggered mechanics specified in Section 4.3 as the input for our model. By "frequency" we mean the exact number of times each mechanic is triggered over the course of one level. These mechanics are represented in vector format, and normalized on a mechanic-basis across the entire dataset before training.

These two input representations are used for both the agent and human datasets specified in Sections 4.3.3 and 4.3.4.

Using these different input data representations, we train two different types of models:

- **Long Short Term Memory (LSTM) Model:** this model is trained on the cropped playtrace representation. LSTM can train and generalized on sequential data. The model consists of a LSTM layer with 100 hidden nodes followed by an output layer of 3 nodes for each persona probability. The LSTM model is trained using stochastic gradient descent for 200 epochs and

learning rate of 0.001. For each experiment, we train 3 networks to make sure the results are stable.

- **Support Vector Machine (SVM) Model:** this model is trained on the mechanic frequency representation. We picked SVMs because they are a well established machine learning method that is known for its capability to generalize on small datasets without overfitting.

8.2 Results and Discussion

Table 8.1 displays the results of training and validation of the classifiers on the synthetic and human datasets. The SVM trained on synthetic data clearly fail to capture the distribution of the human players. Although there are $1,500$ playtraces to train on, the synthetic playtraces distribution is far away from human data distribution. We think due to the purity of the synthetic agents (only runner, only monster killer, or only treasure collector), the trained classifier fails to recognize any multi-label persona. As a result, they do not perform well on the human testing set with an accuracy of 4.8%. The LSTM model is also not able to generalize well on the synthetic data, having a low accuracy during training (58.1%) and testing (18.6%). During training, the LSTM is not able to differentiate between treasure collector and monster killer personas (due to proximity of treasures and monsters) which causes it to classify all the treasure collector as monster killers during training. We believe that the lack of variety between playtraces explains this poor performance for both the LSTM and the SVM.

The SVM and LSTM models trained on the human dataset labeled using Action-Agreement is able to generalize to perform well on the test set, with an accuracy of 70% and 72.6% respectively. The testing accuracy is also similar to its training/validation scores, meaning that both distributions of human players, which total 858 traces, contain similar mechanical patterns that the classifier picked up on. The similarity in scores between LSTM model and SVM model supports our hypothesis that mechanic frequency is a good data augmentation method, being able to capture the essence of the player persona similar to using playtrace data.

8.3 Summary

In this chapter, we demonstrate a method of classifying play personas using normalized mechanic frequency vectors. If a player's persona is known, then it is possible to generate personalized levels and/or tutorials for them. Personalized tutorials could help a player better master their persona style or even introduce

Model	*Training Set*	*Training*	*Validation*	*Testing*
LSTM	*Synthetic*	0.581 ± 0.047	0.483 ± 0.08	0.186 ± 0.029
	Human	0.837 ± 0.03	0.784 ± 0.067	**0.726 ± 0.029**
SVM	*Synthetic*	0.596	0.567	0.048
	Human	0.777	0.694	**0.700**

Table 8.1: The training, validation, testing results for the LSTM and SVM trained on different datasets/labels

them to a new persona they have not explored. Outside of PCG, play persona statistics can be a valuable source of feedback for designers to improve and iterate levels. For example, designers could learn how players react to certain level patterns and what kind of behaviors are most commonly used to overcome them. Procedural personas which mimic play personas can be integrated in the development engines as tool to direct designers about map and level experience. Knowing the persona makeup of the player base can guide developers toward building content that their player base enjoys.

This concludes the second part of this thesis. Building on mechanics as the conceptual foundation of tutorials, we have investigated several ways to extract mechanics for tutorial content using different input representations and techniques. We have introduced a theory of Game Mechanic Alignment to uncover correlations between mechanics, players, and winning/losing. We have also proven a way to classify players by their play style by analyzing their mechanic use. In the final part of this thesis, we will use the concepts reviewed here to generate different types of tutorials.

Part III

Tutorial Generation

Figure 8.1: Capturing a Ralts tutorial in Pokemon: Sapphire

Tutorials can come in many different shapes and sizes. They can be in the form of written instructions in a video game booklet, such as *Space Invaders* (ATARI 1978) and *Super Mario Bros* (Nintendo 1985). They can be demonstrations that the players observe, such as in *Pokemon: Sapphire* (Nintendo, 2002) when the player watches Wally navigate menus to catch a Ralts[3] (Figure 8.1). They can also be as set of interactive stages in which the player engages in learning-by-playing, such as the beginning levels of the *Super Meat Boy* campaign (Team Meat, 2010).

The following chapters describe our exploration into generating different tutorial types. Chapter 9 introduces the AtDelfi System which can build "tutorial cards" for GVGAI games to present instructions and demonstrations of mechanics to the player. Chapters 10 and 11 present methods of generating tutorial levels which emphasize the use of specific mechanics during play using objectives and quality diversity, respectively. Within the same vein, Chapter 12 describes a method of level generation which emphasizes (or discourages) different play persona behaviors. Chapter 13 introduces a way to stitch the smaller levels generated using the previous three chapters into longer fuller levels.

[3] `https://www.youtube.com/watch?v=xERTYBk3Txk`

Chapter 9

Tutorial Instruction and Demonstration Generation

Instruction-based tutorials explain how to play the game by providing the player with a group of instructions to follow, similar to what is seen in boardgames. Demonstration-based tutorials show the player an example of what will happen if they do a specific action. In this chapter, we present work where we generate tutorial cards containing critical mechanic information in the form of instruction and demonstration tutorials for several games in the GVG-AI framework (described in Section 4.1): Aliens, Butterflies, CamelRace, Jaws, Plants, RealPortals, Survive-Zombies, and Zelda. All of the work described in this chapter comes from "AtDelfi: Automatically Designing Legible, Full Instructions For Games" with myself as first author published at the International Conference on the Foundations of Digital Games, 2018 [60] and won the best paper award.

9.1 Tutorial Card Content

The mechanic content for the tutorial cards is found using the uninformed critical mechanic discovery method explained in Chapter 6. The tutorial cards display this critical mechanic information to the player to teach them how to win the game, how they could lose, and how to get points.

9.1.1 Instructions

To create instructions, the system performs text-replacement using a decision tree. A subsection of this tree is displayed in Figure 9.1. Using critical mechanic information for winning/losing paths and points information from the mechanic graph, the system can generate human-readable text, which is displayed to the user. Table 9.1 shows an example of generated tutorial instructions for GVGAI's

Aliens. It is important to note that this text replacement is generalized across every game in the GVGAI framework. As a result, the names of sprites (i.e. "avatar (FlakAvatar)" in Aliens) are displayed as they are written in Video Game Description Language (VGDL).

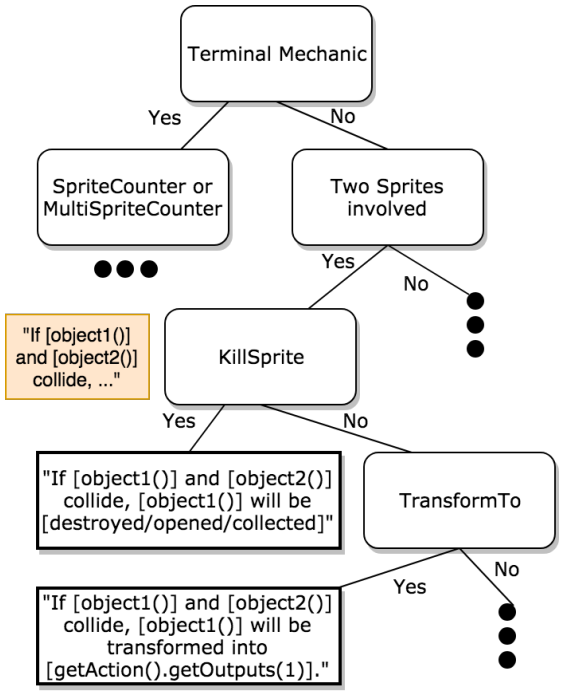


Figure 9.1: A subtree of the overall text-replacement tree for all GVGAI games in the AtDELFI System

9.1.2 Demonstrations

Tutorial videos show animated examples of mechanics to the player. Using an adapted version of *Seekwhence* [113], the system uses frames captured by artificial agents playing the game and transform them into animations for the player to watch on a *tutorial card*, which displays tutorial videos in an easy-to-read format. Figure 9.2 shows a section of such a card for GVGAI's Aliens.

There is yet to exist an agent that can beat every game in the GVGAI framework. Therefore, in an attempt to make the generator as robust as possible, the system uses a cocktail of winning competitors from past GVGAI competitions, all of which can be found on the GVGAI website[1]: adrienctx [125], NovelTS, NovTea [57], Number27 [123], and YOLOBOT [89]. To capture the full extent of winning and losing conditions, our system also uses a one-step look-ahead agent and a doNothing agent. The former looks only one step ahead into the forward model, while the latter literally does nothing. Each agent plays every level of a given game.

[1] www.gvgai.net

Table 9.1: Tutorial instructions for GVGAI's Aliens

Controls:	As the avatar, use the arrow keys to turn and move.
	Press space to shoot the sam (missile).
Winning:	If you press space, then avatar (FlakAvatar) will shoot a sam (missile).
	If alien and sam (missile) collide, then the alien sprite will be destroyed.
	If there are no more portalSlow (portal) sprites or portalFast (portal) sprites or alienGreen (alien) sprites or alienBlue (alien) sprites then you will win.
Losing:	If avatar (FlakAvatar) and bomb (misisle) collide, then the avatar (FlakAvatar) sprite will be destroyed.
	If avatar (FlakAvatar) and alien collide, then the avatar (FlakAvatar) sprite will be destroyed.
	If there are no more avatar (FlakAvatar sprites then you will lose.
Points:	If the alien and the sam (missile) collide, then you will gain 2 points.
	If the base and the sam (missile) collide, then you will gain 1 point.

Meanwhile, the system captures every frame and stores it in a database. After all agents have played, the system then queries the database for the exact frame where each mechanic in the critical paths and the points sections of the tutorial occurred. In the event that an agent was unable to beat a level, it might be missing frames for a winning critical path mechanic query, in which case the system will look at another playthrough. If an agent triggered this mechanic, the system requests 5 frames around the triggered mechanic to be placed in the tutorial video. Unused frames are deleted to save space. Mechanic frames found are bundled together with the mechanic they show and displayed on a *tutorial card* in an easy to read format. Frames are looped through to create an animation demonstrating the mechanic. If no agent triggered a tutorial-referenced mechanic or if the mechanic references a spacebar mechanic ("Press space to fire a missile"), the system does not display any frames. For mechanics that are merged together, the system displays a video for every mechanic that was merged.

9.2 Tutorial Card Results

We generate turtorial cards for 8 GVGAI games: *Aliens*, *Butterflies*, *Camelrace*, *Jaws*, *Plants*, *RealPortals*, *SurviveZombies*, and *Zelda*. These games were selected from the four difficulty groups identified by Bontrager et al. [13]:

1. games that require only a few steps of look ahead and offer rewards often

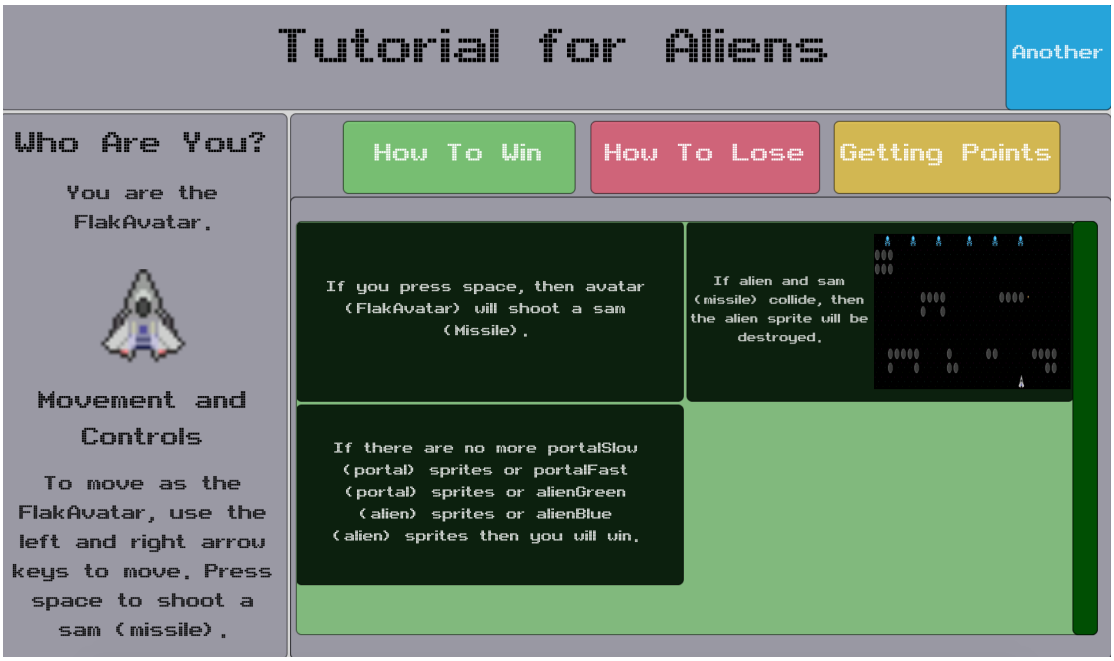


Figure 9.2: A tutorial demonstrating mechanics for GVGAI's Aliens, the buttons allow the user to switch from winning, losing, and point-gaining information

(*Aliens* and *Butterflies*)

2. games which require searching a large space or looking far into the future for any reward (*Plants* and *RealPortals*)

3. games which fall in between the above extremes and on which MCTS techniques perform well (*Jaws* and *SurviveZombies*)

4. games which fall in between the above extremes but on which MCTS techniques perform poorly (*Camelrace* and *Zelda*)

The readability of generated tutorial cards is evaluated subjectively. For each tutorial, we read the generated instructions, observed the demonstrated mechanics, and compared it to the VGDL game description written by the GVGAI game developers. Our evaluations are described below:

Aliens Aliens is a clone of *Space Invaders*. The player uses arrow keys to move left and right on the bottom of the screen, shooting down aliens and avoiding missiles they shoot back. The player score for each alien destroyed. If the player collides with an alien or an alien's missile, they lose the game. The Aliens generated tutorial was one of the most understandable among the 8 games. Information given in all four sections of the instructions adequately explained main characteristics of the game: destroy all aliens and do not get blown up. Agents collected frames that highlighted every mechanic, making it user-friendly.

Butterflies In Butterflies, players must collect all butterflies and keep them away from the flowers (as they eat the flowers). Each butterfly collected increases the score, and after collecting all butterflies, the player wins. If a butterfly eats a flower, it multiplies, and if all flowers are eaten the player loses. The game's challenge is balancing flower management to score high without losing. The Butterflies tutorial did an adequate job at explaining the objectives of the game. Instructions do not touch upon the strategy of letting butterflies multiply in order to maximize point gain, but it does explain how to win, how to lose, and that collecting butterflies is a way to increase score. In addition to winning, losing, and collecting butterflies for points, the videos show a sequence where a butterfly collides with a flower and multiplies, but it does not explicitly comment on this mechanic. It displays this information only to show that, by allowing all flowers to be destroyed, the player will lose the game.

Camel Race Camel Race is a racing game. To win, the player needs to be the first camel to touch the opposite side of the screen. There are various obstacles the player needs to avoid in order to accomplish this goal. The only way to gain any points is by winning the game. The generated tutorial explains the winning and losing mechanics. Agent videos show win and lose situations when the goal-line is on either side of the screen.

Jaws Jaws is a survival game where the player needs to evade/kill all sharks within 1000 ticks. The sharks are divided into two types, passive and aggressive. Passive sharks swim in a straight line and can be killed by the player's gun, while aggressive sharks swim towards the player and cannot be killed by bullets. When sharks die, they turn into jewels, which can be collected by the player for points. The instructions describe the winning and losing paths correctly. However, although they explain that by collecting jewels the score will increase, they fail to include that jewels are spawned after destroying a shark. This lack of detail could confuse a player, who does not immediately see where jewels come from. Only through watching a video can players see how jewels are spawned.

Plants Plants is a GVGAI clone of *Plants vs. Zombies* (PopCap Games 2009). The goal is to survive for 1000 ticks. Zombies spawn on the right side of the screen moving towards the left, and the player loses if a zombie reaches the left side of the screen. The player needs to build plants, which fire zombie-killing pea sprites. Generated instructions explain how to win and to lose, but come short when it comes to mechanics involving points: they only explain that when peas collide with zombies the score will increase. It does not explain that plants spawn peas or that the player can create plants to defend themselves. Additionally, the

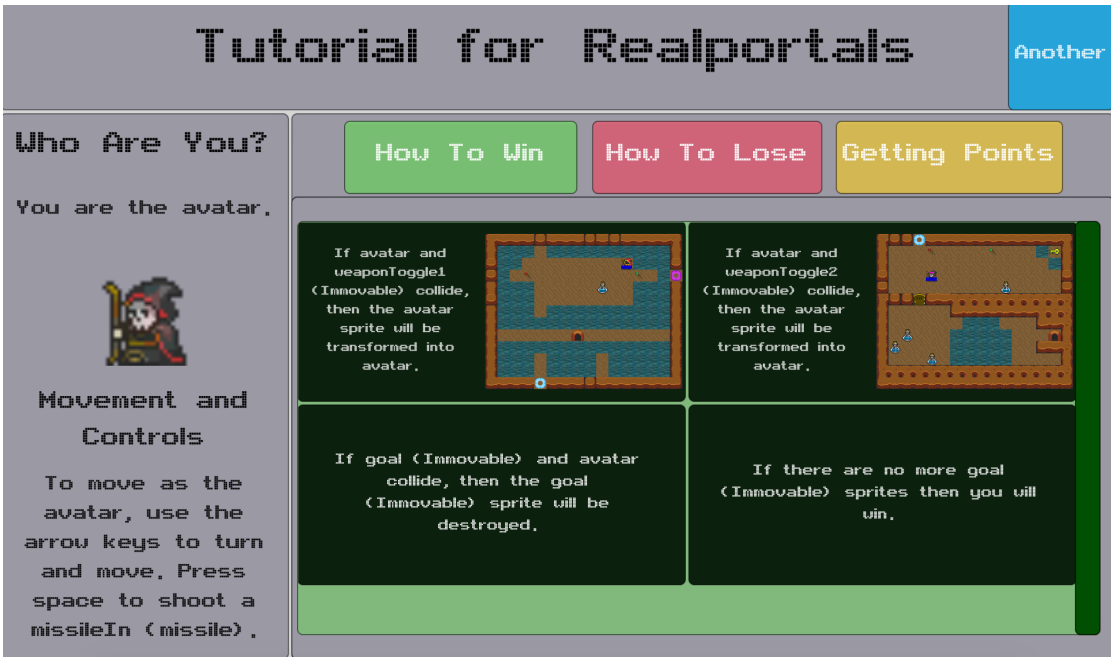


Figure 9.3: A tutorial demonstrating mechanics for GVGAI's RealPortals, although there are agent videos for points and losing, the win section is incomplete due to the fact that none of the agents could beat the game.

videos do include a sequence showing a pea spawning from a plant and hitting a zombie along side text "If the zombie and the pea collide, then you will gain 1 point," but does not explicitly comment on the pea spawning mechanic.

RealPortals RealPortals is a GVGAI clone of *Portal* (Valve 2007). The player must reach the door, which sometimes is behind another locked door that needs a key. The player is restricted by water, which often lies between them and the door. To succeed, players need to pick up wands, which allow them to create *portals* through which they can spontaneously travel across the map. There are two different types of wands, and each corresponds to a different portal gateway. The generated tutorial explain how to change portal types (by colliding with different wands), but does not explain how each wand is used to create portals, nor how the player can travel using the portals. The instructions fail to explain that the portals are necessary in order to win the game, however it does include that the player needs to collide with the goal to win. They also explain that using a portal gives the player points. None of the agents were able to beat RealPortals. They show an agent using portals to gain points, but always in compromising positions (e.g. the agent uses the portal to teleport into the water, proceeding to die immediately after). Because none of the agents won, the winning critical path information is incomplete, specifically the part about colliding with the goal to win the game, as seen in Figure 9.3.

SurviveZombies SurviveZombies is a zombie-survival game, where the player needs to survive for 1000 ticks to win. The player can increase their score and get more hit points if they pick up honey, which is created when bee sprites collide with zombies. The player will lose hit points if they collide with any zombies, and if their hit points are depleted, they lose the game. The instructions describe surviving to win, but do not mention the hit point mechanic. This is due to this version of tutorial generation not being created with the understanding of hit point mechanics. It compensates for this by describing that collision with zombies is deadly: "If avatar and hell collide, then the avatar sprite will be destroyed." The instructions also do not mention how honey regenerates hit points, only explaining that collecting honey awards points. However, the videos do show the hit point mechanic by displaying the moment hit points change when zombies are collided with or honey is collected.

Zelda Zelda is a simplified GVGAI clone of the dungeon system in *The Legend of Zelda* (Nintendo 1986). The goal is to pick up a key and unlock the door in the level. The player will encounter monsters, which can kill the player, causing them to lose. The player can swing a sword; if they hit a monster, the monster is destroyed, and the player gains points. Instructions explain the key and door mechanic used to win the game, as well as how the player can kill and be killed by monsters. The agent videos include sequences of every type of monster, killing and being killed by, the player, making this a user-friendly tutorial.

9.3 Summary

In this chapter, we present a way to generate tutorial text and demonstrations from game mechanics. This chapter combines work performed in Chapter 6 as part of the AtDelfi project, an end-to-end tutorial generative system. As such, the usefulness of the tutorials are determined in part by the methods used to curate the mechanics to teach. These mechanics can come from anywhere, be it from critical mechanic discovery, GMA curation, or manually selected.

We presented instructions and demonstrations through the use of minimalist tutorial cards. However, many games will present text and examples in-game. Civilization VI (Fraxis Games, 2016), for example, uses an advisor system which will pop-up with advice on any given game turn. Dynamically generated advice which can adapt to the environment and player preferences could be highly beneficial for a player who is struggling or wants to master a certain strategy. Though AtDelfi does not build the third tutorial type (levels), the next several chapters cover ways in which tutorial levels can be built using mechanics and procedural personas.

Chapter 10

Mechanic-Dependent Tutorial Scene Generation Using Objective-driven Search

Interactive tutorials give the player freedom to explore and experiment. For example: in *Super Mario Bros* (Nintendo 1985), the world 1-1 is designed to introduce players to different game elements, such as goombas and mushrooms, in a way that the player can not miss [33]. In *Super Meat Boy's* (Team Meat 2010) "Hello World" level, the only way to die is to purposefully jump off the screen, serving to introduce basic controls to the player. Often in tutorial levels, early obstacles are instances of *patterns* which reoccur later in the game in more complex instances or combinations [34].

This chapter details the use of single-objective optimization (specifically FI2Pop, Section 5.2.1) with different optimization functions to generate *scenes* for the Mario AI framework (Section 4.2). Scenes are defined in game design [5] as small levels which contain a desired experience (a single jump, killing an enemy, etc). Figure 10.1 displays an example of a scene from the first *Super Mario Bros* (Nintendo 1985) where the player must jump from the first pyramid to the second pyramid without falling in the gap. The research in this chapter comes from two papers, "Generating Levels that Teach Mechanics" which is published at the PCG Workshop at FDG 2018 with myself as first author [63], and "Intentional Computational Level Design" expanding upon it published at GECCO 2019 with Ahmed Khalifa as first author [96].

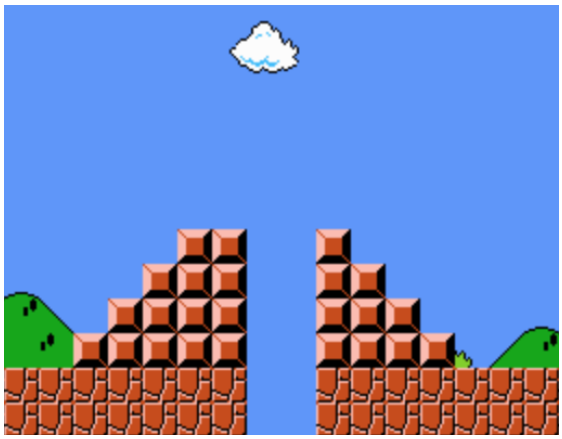


Figure 10.1: Super Mario Bros scene where the player needs to jump over a gap from the first pyramid to the second pyramid.

10.1 FI2Pop Generation Methods

We use two different objective approaches to generate scenes which we call "Limited Agents" and "Punishing Model". The "Limited Agents" technique uses two separate agents to generate levels. One of these agents is, ideally, a *perfect* agent, meaning that it is always able to find a way to beat the level if any such way exists. The second agent is a variant of a perfect agent but limited in some mechanic-based way (for example, being completely unable to jump in the game). The technique strives to make sure that, given a generated scene, a perfect agent (i.e. an agent that knows all mechanics) can beat it, but an agent that cannot perform a required mechanic cannot. The "Punishing Model" uses one agent (the perfect agent) but with two different forward models. One of these forward models is perfect while the other punishes the agent for causing certain mechanics during play. This technique attempts to make sure within the generated scene, the perfect model can beat the level while the model that has limited foresight cannot.

10.2 Scene Representation

We assume 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 10.2. This padding is necessary to ensure that there is a safe area where Mario starts and finishes the level. Additionally, these slices were collected from the levels presented in the Video Game Level Corpus (VGLC) [156]. There are 3721 vertical slices in 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

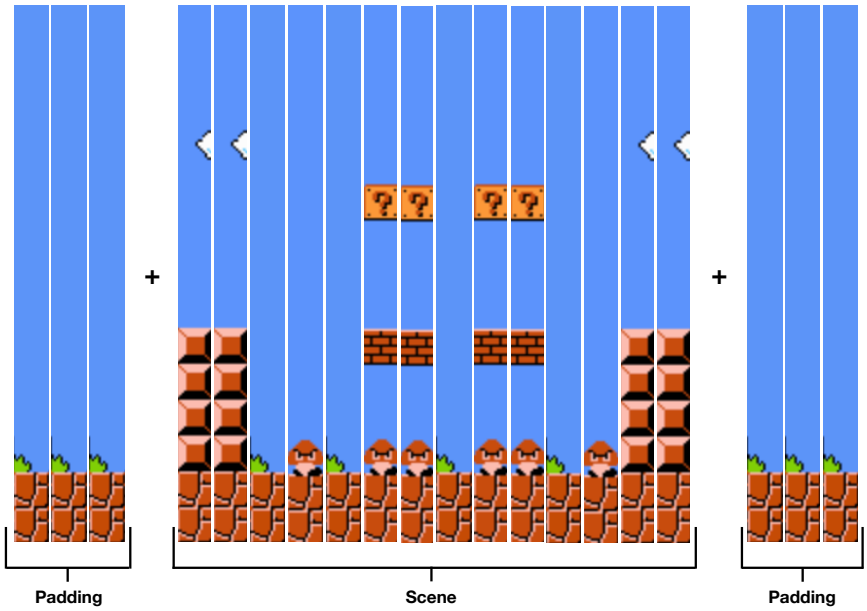


Figure 10.2: The chromosome consists of 14 vertical slices padded with 3 floor slices on both ends.

chance to be picked compared to other slices, since it appears more often in the corpus.

10.3 Scene Evaluation

Scenes are evaluated in relation to specific constraints, which differ between the different techniques and will be discussed in later sections. If the scene satisfies these constraints, it is also evaluated in terms of simplicity. The simplicity fitness is the same between the techniques and tries to reduce the entropy in the scene. The entropy calculation pressures generated scenes to place tiles efficiently and to have high horizontal consistency. For example, a scene in which tiles placed next to each other are constantly changing (empty space - wall - empty space - wall) will have low horizontal consistency. A scene in which tiles placed next to each other are uniform (wall - wall - wall) will be more horizontally consistent. Equation 10.1 shows the fitness calculation equation.

$$fitness = (0.2 \cdot (1 - H(X))) + \left(0.8 \cdot \left(1 - H\left(\bar{X}\right)\right)\right) \tag{10.1}$$

where X is the set of different used tile types in the scene and $\bar{X}$ is the set of horizontal tile changes in the scene. We 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.

Regarding the aforementioned constraints, both approaches make sure that the scene is beatable by using Robin Baumgarten's A* agent, the winner of the first Mario AI competition [160]. It is important to note here that the A* agent is not perfect in reality. A* is deterministic algorithm. However, Robin Baumgarten's A* agent contains a forward model that contains stochastic elements and is therefore not entirely consistent with the framework's game state. At the time of publication, this agent was about as close to perfect an agent for the Mario AI Framework could be.

Besides the playability constraint, another constraint is included to make sure that the agents playing the scenes utilize the targeted mechanic, whatever that may be. This additional constraint will be discussed in detail for each approach in the following subsections.

10.3.1 Limited Agents

This technique uses two agents: Baumgarten's A* agent (the "perfect agent") and a modified version of the agent that we call the "limited agent". The constraint is satisfied when the perfect agent can win the scene while the limited agent cannot. In case the constraint is not satisfied, two scenarios may occur. If both agents win the level, the fitness of the chromosome is 0. If neither agent wins the level, then the chromosome's fitness is proportional to the difference of the distance traveled by the two agents. Maximizing the relative distance between the perfect agent and the limited agent is directly proportional to satisfying the constraint. A good scene can be defined as the perfect agent traversing the entire scene while the limited agent travels less. Equation 10.2 shows the constraints equation:

$$constraints = \begin{cases} 1 & \text{if } A_{perf} = 1 \ \& \ A_{limit} = 0 \\ \frac{d_{perf} - d_{limit}}{d_{scene}} & \text{otherwise} \end{cases} \tag{10.2}$$

where A_{perf} and A_{limit} are the result of the perfect agent and the limited agent playing the scene, respectively being 1 when the agent wins and 0 otherwise; d_{perf} and d_{limit} are the distances traveled by the perfect- and the limited agent. d_{scene} is the length of the full scene, used to normalize the result to the -1 and 1 range. Three different mechanics are observed for the limited agents method: running, jumping, and enemy collision.

Agent	Limitation
No Run	can't press the run button.
Limited Jump	can't hold the jump button for too long.
Enemy Blind	doesn't see enemies.

Table 10.1: Agents' Limitation in Mario

Mechanic Model	Punishment
High Jump	kills the player if they hold the jump button for too long.
Speed	kills the player if they exceed a certain x velocity.
Stomp	kills the player if they stomp an enemy.
Shell Kill	kills the player if they kill an enemy using a koopa shell.
Mushroom	kills the player if they collect a mushroom.
Coin	kills the player if they collect a coin.

Table 10.2: Mechanic models in Mario and their perceived punishments

10.3.2 Punishing Model

This technique uses only one agent (the perfect agent), but relies on two different forward models. The first forward model, called "normal model", behaves as expected in the game. However, the second forward model, the "punishing model", is written in such a way that the agent believes it will die when a certain mechanic is fired. For example, the agent believes it will die if it stomps on an enemy and consequently tries to avoid doing so. The constraints' value is calculated using equation 10.2, using the punishing forward model's results as substitute for the limited agent's.

Table 10.2 shows six different punishing models used in this project. These are six different basic mechanics that appear in the original *Super Mario Bros.* We exclude mechanics that only appeared in following titles, such as wall jumping and ground pound.

10.4 Genetic Operators

We use 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.* Elitism is used to preserve the best chromosome across generations. As a result of A* being only a close approximation to a perfect agent, chromosomes do not always receive the same constrained value over multiple playthroughs (the agent may perform different actions in the exact same scenario). To counter this, we keep the lowest

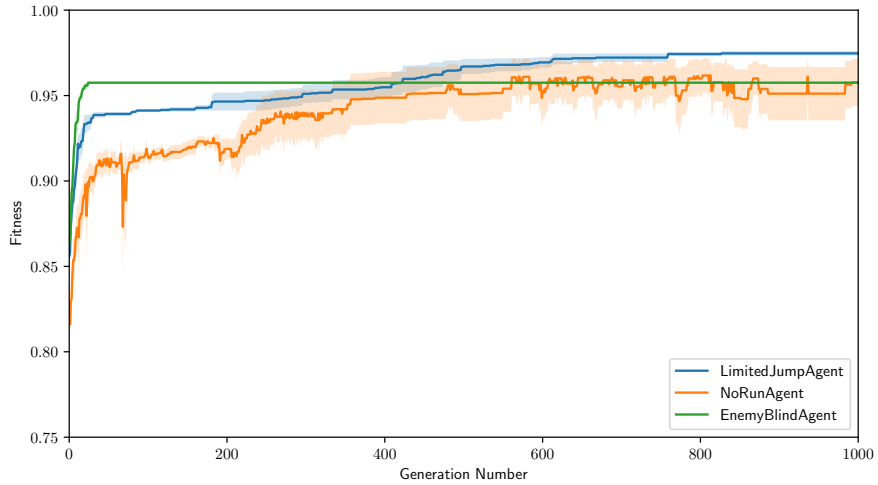


Figure 10.3: The average maximum fitness value and the standard error of the limited agents approaches over the 5 runs.

constrained value after evaluation of the chromosomes between generations.

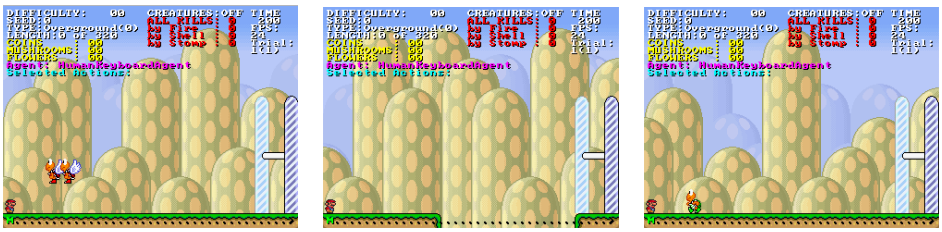
10.5 Results

Each technique is run 5 times total, each time for 1000 generations, where each generation consists of 100 new chromosomes. These values were selected based on our current available resources and also to make sure that generation algorithm will finish within 12 hours using 25 CPU cores. The following subsections present the results of each approach. All results are evaluated by members of our team by playing the generated levels themselves and capturing their observations.

10.5.1 Limited Agents

The "Limited Agents" approach reaches convergence very rapidly. Figure 10.3 shows the average maximum fitness for each of the targeted mechanics. The *Enemy Blind* agent is the fastest, finding the best chromosome by generation eight. Its constraints are simpler than that of other agents, as it causes the limited agent to run through enemies instead of more specific mechanics like stomping on them. On the other hand, the *No Run* agent quickly finds a good solution, albeit unstable and thus resulting in a noisy fitness line. Unstable solutions are solutions where constraints are satisfied due to the agent being imperfect, thus their constraints' values change between multiple runs, causing them to sometimes move to the infeasible population instead.

Figure 10.4 shows the best chromosome at the end of the 1000 generations for each targeted mechanic from one of the 5 runs. Looking at the *Enemy Blind*



(a) No Run Agent (b) Limited Jump Agent (c) Enemy Blind Agent

Figure 10.4: The results for the limited agent approach after 1000 generations.

and *Limited Jump* results, the generated scenes do require the use of the targeted mechanics to beat. The *Enemy Blind* scene requires the agent to avoid a koopa troopa in order to proceed, either by stomping on it or jumping over it. The *Limited Jump* scene requires the player to both run and hold the jump button down for a few game ticks to long jump over a gap. However, the *No Run* scene can be beaten without running if the player waits (until the koopa paratroopa fly overhead) and then moves to finish the level, or by jumping on top of the koopa paratroopas. This scene is generated because the agent moves forward immediately on scene initialization, dying as soon as the koopa paratroopa lands on them and rendering that scene unstable. Also of note, the *Limited Jump* scene can act as a *No Run* agent's scene as well, as the gap cannot be passed without running while jumping. While l analyzing the best chromosomes from all the 5 runs, we found that all the generated levels are different from each other except for the *Enemy Blind* scenes. *Enemy Blind* levels look mostly similar to Figure 10.4c but with different enemy x positions or types.

10.5.2 Punishing Model

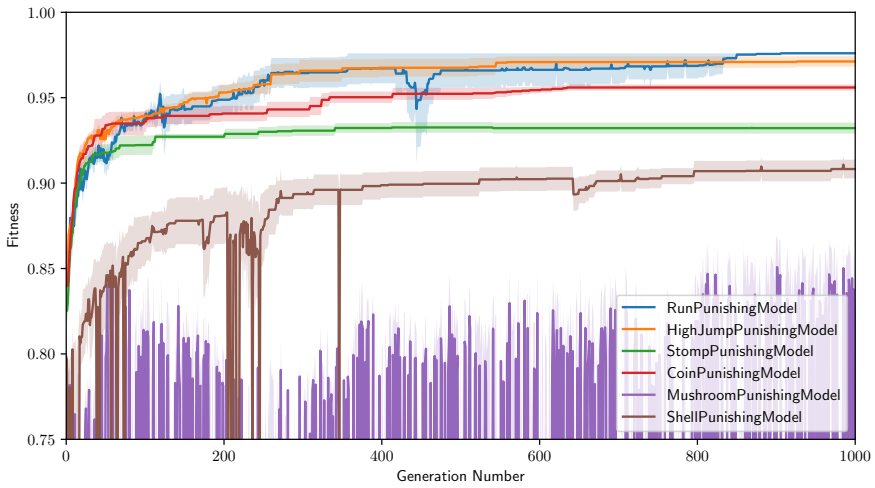
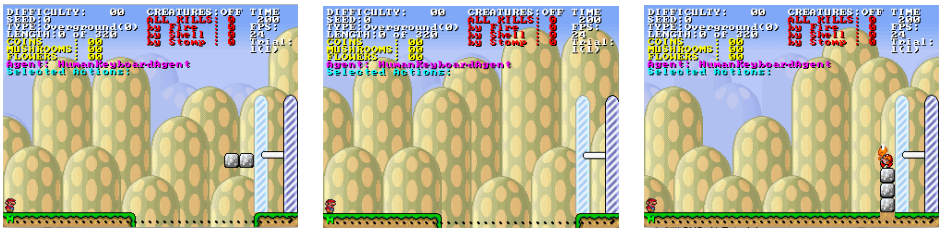


Figure 10.5: The average maximum fitness value and the standard error of the punishing model approaches over 5 runs.

The punishing model is slower to find targeted mechanic levels since the punishing model spends most of its search time computing alternative paths to avoid punishment. Figure 10.5 shows the average maximum fitness for each of these mechanics. Results show several unstable scenes for most iterations before stabilizing. The *Mushroom Punishing Model* never stabilizes and always fluctuates between a 0 fitness (no scenes satisfied the constraints) and above a 0.8. This instability is caused from forcing the AI to try find a possible path at every tick when a mushroom is spawned (which happens after hitting a "?" block). Occasionally the agent misses a viable solution without triggering the mechanic in one generation but finds it in the next generation, causing whiplash in the chromosome's constraints and fitness.



(a) High Jump Punishing Model (b) Speed Punishing Model (c) Stomp Punishing Model

(d) Shell Kill Punishing Model (e) Mushroom Punishing Model (f) Coin Punishing Model

Figure 10.6: The results for the punishing model approach after 1000 generations.

Figure 10.6 shows results for each targeted mechanic from one of the 5 runs. The scenes generated for the *High Jump-*, *Speed-*, *Stomp-*, and *Coin Punishing Model* are impossible to win unless the targeted mechanic is used. The remaining two generated scenes rely heavily on the use of the targeted mechanic, but a creative agent (or human) may notice alternate solutions that do not require that mechanic. These scenes exist because the agent is not exhaustively searching for all possible solutions to ensure they require use of the mechanic. By looking at all the best chromosomes from all the 5 runs, we find that all the generated scenes, while not explicitly requiring the mechanic, do provide all the *ingredients* needed for the mechanic to occur. For example, the shell kill punishing scene requires having at least two koopas, the speed punishing scene requires having a large gap, etc.

10.6 Discussion

From the results, we can see that each approach has its own advantages and disadvantages. The "Limited Agent" is the simpler and developmentally faster than the "Punishing Model", as the former doesn't involve modifying the game engine itself (or at least a forward model of the game engine) but only the agents. Its main disadvantage is that in order to use this technique, it is necessary to modify the action space in a way that revolves around a certain game mechanic. Modifying an agent like this can be challenging because of how difficult it can be to identify all game mechanics just from action space. For example, enemy blindness is easy because Mario dies when he directly collides with an enemy and therefore can be "blind" to the collision, but identifying a shell kill is harder (how does an agent *see* or *not see* a shell killing an enemy from agent action space?). Another shortcoming is that generated scenes from the Limited Agent approach do not require that the player uses that mechanic in order to beat the scene. Figure 10.4a shows a *No Run* agent scene that the player can beat if they wait for the koopa paratroopa to pass by.

The "Punishing Model" method is able to generate scenes that revolved around the targeted mechanic and are quite difficult to beat without triggering it. Theoretically, all the generated scenes should guarantee the required mechanic is triggered to beat it, but the usage of an agent that does not exhaustively search for all solutions prevents this. Because the Baumgarten's A* agent makes mistakes, some generated scenes were winnable even if you ignore the mechanic (e.g. Figure 10.6d and 10.6e). In most of the runs for complicated mechanics such as mushroom-related ones, the algorithm spent most generations attempting to stabilize and therefore did not have enough time to simplify its creations, resulting in complex scenes.

Both FI2Pop techniques are vulnerable to agent error due to the stochasticity in the Baumgarten A* forward model. For example, if an agent dies because of a mistake the algorithm does not account for, the scene may pass as a good scene, even if the scene might not have the mechanic anywhere in it due to how the fitness is calculated (one dies while the other lives). Possible solutions for this might be to use an agent with a perfect forward model, or to run the Baumgarten agent multiple times and pick the best run. To guarantee that these techniques generate 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 within an acceptable timeframe. Then the target mechanic is recorded as fired only if it is triggered every possible solution. However, this would be impractical even for Super Mario Bros' complexity.

10.7 Summary

In this chapter, we present two objective-optimization techniques to generate mechanic-dependent scenes, i.e. levels that offer the potential to use a mechanic or set of mechanics. The FI2Pop methods use different agent/foward model combinations.

Procedurally generating levels which enable players to experience mechanics individually or in different combinations is a great step for automated tutorial generation. The minimizing entropy pressure used in this chapter to produce simplistic levels is a reoccurring theme throughout the rest of the thesis. One could imagine simplistic levels as being the "beginner levels," the levels in which a player first learns about a mechanic.

While the results shown are promising, there are still a few notable limitations. The "Limited Agents" approach relies on manually identifying mechanics and implementing agents that have a limited action space. The second approach, "Punishing Models" requires the creation of multiple alternate forward models for each targeted mechanic. Both of these models therefore take additional development time to successfully implement. Neither approach guarantees that the target mechanics are absolutely necessary to beat the scene. Quality diversity methods could be used explore mechanic-dependent generation further to evolve levels that are beatable and simple, while using the illuminating power of MAP-Elites to categorize levels by mechanics triggered. Using a QD method, one could generate a sampling of maps with possible mechanic combinations. We will explore using QD methods to generate mechanic-dependent scenes in the next chapter.

Chapter 11

Mechanic-Dependent Tutorial Scene Generation Using Quality Diversity

In the previous chapter, we explored using the single-objective optimization algorithm FI2Pop (Section 10.1) to generate mechanic-dependent tutorial scenes. This chapter details mechanic-dependent tutorial scene generation by leveraging the illuminating power of Constrained MAP-Elites (CME, Section 5.2.2). We propose using quality diversity over objective optimization strategies for two reasons. The first is that QD algorithms do not require manual agent/model manipulation like Limited Agents or Punishing Model strategies do. The second is that QD algorithms can provide a diverse sampling of level space more so than FI2Pop can. The domains we utilize are the Mario AI framework described in Section 4.2 and several games from the GVG-AI framework (Zelda, Solarfox, Plants, and Realportals) described in Section 4.1. The research in this chapter comes from two papers, "Intentional Computational Level Design" published at GECCO 2019 with Ahmed Khalifa as first author [96] (also mentioned in Chapter 10) and "Mech-Elites: Illuminating the Mechanic Space of GVG-AI" published at FDG 2020 with M Charity as first author [20].

11.1 Scene Representation

11.1.1 Mario Scene Representation

Mario scenes are represented the same as they are in the previous chapter (Section 10.2). Chromosomes are defined as a sequence of vertical slices sampled from the first *Super Mario Bros.* As before, these slices were collected from the levels presented in the Video Game Level Corpus (VGLC) [156].

11.1.2 GVG-AI Scene Representation

The vertical slice representation is specific to Mario as levels are traversed from left to right allowing for fixed height levels. In GVG-AI, however, we represent the levels as a 2D array of tiles where each tile corresponds to a game sprite. The different game tiles are extracted automatically from the VGDL description of the game. This representation is generic and can work between different games with no needed modifications.

11.2 Scene Evaluation

11.2.1 Mario Scene Evaluation

Similar to the previous chapter (see Section 10.3), generated scenes have playability constraints and entropy fitness. The CME technique within the Mario domain uses only one agent (the perfect agent) and one forward model (the normal forward model), in contrast to the Limited Agent and Punishing Model techniques mentioned before. Also as in Chapter 10, the agent used in this experiment is the Robin Baumgarten A* agent.

A scene's constraints value is equal to 1 if the agent can beat the scene, and directly proportional to the traversed distance otherwise, as shown in Equation 11.1.

$$constraints = \begin{cases} 1 & \text{if } A_p = 1 \\ \frac{d_p}{d_s} & \text{otherwise} \end{cases} \tag{11.1}$$

where A_p is the result of a perfect agent playing the scene, d_p is the distance traversed by the agent, and d_s is the scene length, used to normalize the result between $0 - 1$.

After calculating the constraints, the playthrough is analyzed to extract the different types of mechanics that were fired during play. This extracted information is used to create binary dimensions for the CME algorithm. Table 11.1 shows the different mechanics extracted from the playthrough. These mechanics represent the six mechanics used in the punishing model approach in Section 10.3.2 while also including additional mechanics like "Jump" and "Fall Kill," which were more difficult or impossible to represent/isolate in the punishing model or limited agent approach. For example, fall kill was added to help differentiate between scenes that requires overcoming enemies through action and scenes where enemies will fall out of the scene if the player has waited or played imperfectly (killing through inaction). Furthermore, the different jump types were added to differentiate between scenes that required (1) running and long jumping, or (2) holding the jump button for a

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 11.1: Constrained Map-Elites' dimensions for Mario.

bit longer to high jump, or (3) just a regular jump.

11.2.2 GVG-AI Scene Evalutation

Similar to the experiments in Mario, the GVG-AI constraint value focuses on providing a beatable level, while the fitness focuses on simplifying the levels so they can be easily parsed by humans. However, the constraint calculation looks a little different than Mario, which can be measured in terms of horizontal distance traveled. In GVG-AI games, the goal is not always to *get somewhere*, so constraint calculations are a function of time rather than distance.

11.2.2.1 Constraints

There are two types of constraint evaluation for the GVG-AI experiments: playability constraints and accessibility constraints. Playability constraints exist to ensure the level is playable and can be won in an appropriate amount of time, which we call the "ideal time". The accessibility constraints exist to ensure that the agent does not lose within the first few frames of the game. This allows the generated levels to be playable by humans, as players have enough time to react and (hopefully) will not be defeated immediately at the level's start. We use the AdrianCTX algorithm [125], winner of the 2016 planning competition, as the constraint evaluation agent.

For the playability constraints, if the agent successfully completed the level, then only the completion time is evaluated. A preset value which we call the "ideal time" is compared to the agent's completion time. The closer the completion

time is to the ideal time, the better the constraint value. This is to pressure the generator to build levels that the agent does complete too quickly - so the player cannot see the demonstration of the game's mechanics - or too slowly - so the tutorial level doesn't drag out longer than needed. Otherwise if the agent does not win the level, the constraint value is inversely propotional to difference between the survival time and the ideal time. We multiply this value by 0.25 to penalize it for not winning the level. Equation 11.2 shows the part of the constraint calculation that applies to the time to complete a level,

$$P = \frac{win}{|T_w - T_i| + 1} + \frac{(1 - win) * 0.25}{|T_s - T_i| + 1} \tag{11.2}$$

where win represents a 1 if the agent finished the level successfully and a 0 otherwise. T_i represents the ideal time pre-defined for the system, and T_w and T_s represent the finishing time of the agent before successful completion and unsuccessful completion respectively.

For the accessibility constraint, a "Do Nothing" agent is run on the level for a certain number of trials. This agent does not perform any user actions and remains idle in the level. If this agent dies before reaching T_i, 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. This is to remove the chance that the evaluation agent happened to "get lucky" on its performance in the level and keep the level reasonably user-friendly. Equation 11.3 demonstrates how the idle agent's trials were applied,

$$A = \begin{cases} 1 & \text{if } \frac{N_p}{N_t} \geq 0.5 \\ \frac{N_p}{N_t} & \text{otherwise} \end{cases} \tag{11.3}$$

where N_p representing the number of times the idle agent survived to the ideal time, and N_t representing the number of idle agent tests.

The total constraint score of a level is therefore equivalent to C in Equation 11.4,

$$C = P + A \tag{11.4}$$

where P is from Equation 11.2 and A is from Equation 11.3. In order to be considered a "feasible" level for a MAP-Elites cell, the level must reach a certain threshold of constraints. If the level evaluation does not reach this threshold, the level is placed in the MAP-Elites cell's "infeasible" population instead.

The constraint threshold is chosen with the intention that a level will both be beatable and pass the idle agent tests, but may be within a certain range for T_i. For example, if the constraint threshold is set to 0.1, based on the function

defined for the constraint value, a level can be considered elite in two cases: if it is beatable and passes the DoNothing tests, so long as T_w is within 10 timesteps of T_i; or if it is unbeatable, but passes the DoNothing tests and T_s is within 2 timesteps of T_i. For our experiment, we set this threshold value to 0.1.

It is possible for an unbeatable level to be evaluated as a "feasible" level or beatable level to evaluated as an "infeasible" level. In the case of beatable levels, this could happen if the level has a poor finishing time (i.e. T_w was not close enough to T_i) or the level is failing the idle agent tests. In the case of unbeatable levels, a level can still have a good constraint value (i.e. pass the constraint threshold) if T_s is extremely close to T_i and also passes the idle tests. However, this end value will also be penalized to 0.25 of the original value since the level was still not finished successfully.

11.2.2.2 Fitness

The fitness of a level is found by calculating the tile entropy and the derivative-tile entropy of the level. We use the same fitness function as we did in GVG-AI as we do in Mario, in Section 10.3. The system generates open, mostly empty-tiled levels or levels with similar tiles placed adjacent to each other that still demonstrate the game mechanics needed to play the game. Weights were given to the importance of the tile entropy versus the tile derivative entropy to create less noisy levels. Equation 11.5 was used for the level fitnesses:

$$fitness = H(lvl) * w + H(\Delta lvl) * (1 - w) \tag{11.5}$$

where $H(lvl)$ represents the raw tile entropy of a level, $H(\Delta lvl)$ represents the entropy of the derivative of the same level, and w represents the pre-set weight value. This equation was based on the entropy tile fitness equation used by [96], however, the level derivative is calculated differently. Since the level was not seperated into vertical slices and mutated on the slices like the Mario levels, the derivative was calculated based on the vertical and horizontal changes instead of only horizontal changes. For each tile, we calculate the number of different neighboring tiles on the north, south, east, and west of it use the value as the dervative value for the map.

11.3 Genetic Operators

11.3.1 Mario Operators

We utilize the same genetic operators detailed in Section 10.4. We use a two point crossover that swaps any number of slices, ranging from a single slice to the full scene. The mutation operator replaces a single slice with a random slice sampled from slices in the original *Super Mario Bros.*

11.3.2 GVG-AI Operators

In GVG-AI, we do not use crossover as it is difficult to define a meaningful crossover operation using GVG-AI tile-based representation compared to the Mario vertical slice representation. The GVG-AI levels are represented as 2D ASCII maps - generated with randomized dimensions. Selecting a portion of a large generated level for crossover could entirely erase the contents of another smaller level. Determining the crossover point and amount would also lead to level mutation inconsistencies. Our mutation operator selects a random tile of the level and turned 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. The end result is a mutated version of the input level, ideally a better level. This mutated level is evaluated in the next iteration's population of levels before being added back to the MAP.

11.4 Results

We ran the CME Mario approach 5 times for 1000 generations/iterations, where each generation/iteration consists of 100 new chromosomes. The infeasible population size was fixed to twenty chromosomes, and the feasible population cell size was fixed to one chromosome. These values were selected based on our current available resources and also to make sure that generation algorithm will finish in under than 12 hours using 25 CPU cores.

Four games from the GVG-AI framework were used to test the effectiveness of the CME method. Zelda, Solarfox, Plants, and RealPortals are all described in Section 4.1. The dimensions used in by the MAP-Elites system are shown in Tables 11.2, 11.3, 11.4, and 11.5. These mechanics were extracted from each game automatically by using the AtDelfi system [60], which is able to parse game rules directly from a GVG-AI game's VDGL description file. For all experiments, we generate 50 chromosomes for each iteration. In each iteration, 20% of the levels are randomly initialized, and the remainder are filled with mutated versions of

the selected chromosomes. For the Solarfox experiment, levels are much more dependent on having fewer empty tiles for functional gameplay and thus a less constrained initialized population. To assist evolution, only 10% of Solarfox levels were randomly initialized each iteration, and 90% filled with mutated versions.

In GVG-AI, each experiment is run for 500 iterations. The behavioral characteristcs for a level are calculated using the AdrianCTX agent's playthrough. The *idle agent* is run a total of 5 times on the level, of which it must survive 3 in order to pass the constraint test. In RealPortals, it is impossible for a non-moving agent to die (the only way to lose is to fall into water), and therefore this constraint test is not necessary. Both agent's "ideal times" (T_i) were set to 70 time-steps for all experiments. If the constraint test is passed, the level's fitness is evaluated according to Equation 11.5, where w is 0.25 and $(1 - w)$ is 0.75 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 while $(1 - w)$ is 0.8 for these experiments. After evaluation, chromosomes are compared to their respective dimensional family and will be placed within an Elite cell if applicable. For all four games, a single MAP-Elite cell is allowed to store a maximum of 20 infeasible levels and 1 feasible "elite" level. A newly initialized level has a 50% chance of being mutated from the elite level of a MAP-Elite cell. Otherwise, the level is mutated from the cell's best level from the infeasible population.

In the following subsections, we present a representative subset of each game's generated levels for Mario and GVG-AI. The *mean* and *mode* GVG-AI 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 the elites shown in this Section for both Mario and the GVG-AI games are only a small subset of the possible elites, and that dimensional similarity does not necessarily equate to structural similarity.

Figure 11.1 shows the average number of elites found during 1000 CME iterations in Mario. These numbers always increase, meaning that the algorithm is finding more elite scenes but begins to plateau around generation 200. The normalized elite counts across generations for the GVG-AI experiments is displayed in Figure 11.2. Each experiment was normalized against its total possible elite cell count, calculated using the game's mechanic dimensionality specified in Tables 11.2, 11.3, 11.4, and 11.5.

All results are evaluated by members of our team through playing the generated levels themselves and capturing their observations, as described below.

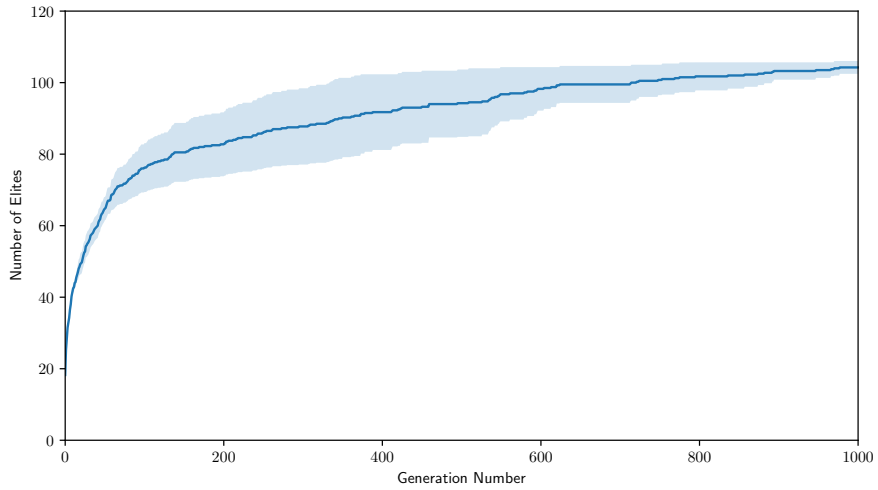


Figure 11.1: The average number of elites and the standard error in the CME approach over 5 runs. After 1000 Generations, this translates to roughly 39% of the map filled

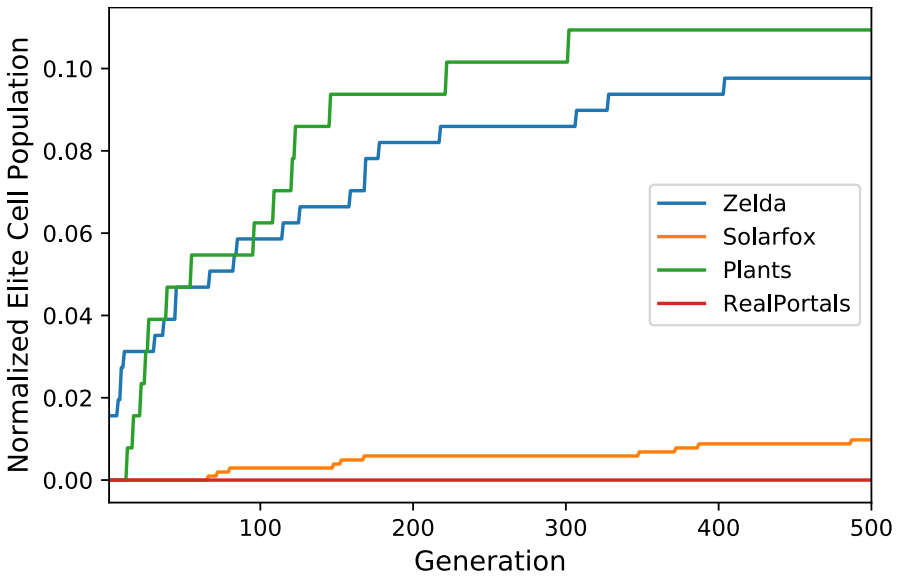
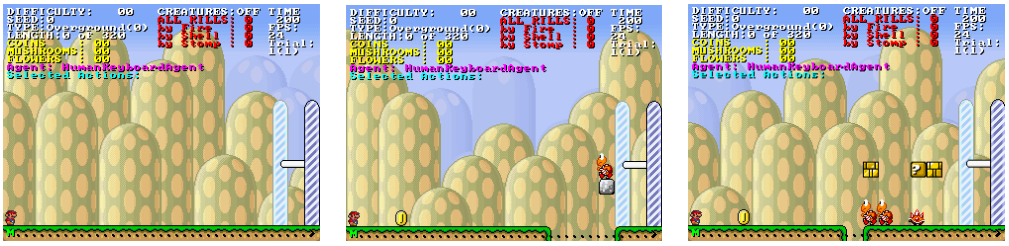


Figure 11.2: Number of filled Elite MAP Cells (Normalized) across generations. Although RealPortals is considerably lower than the other games, it's map contained 231 elites, nearly ten times more than any other game in GVG-AI.

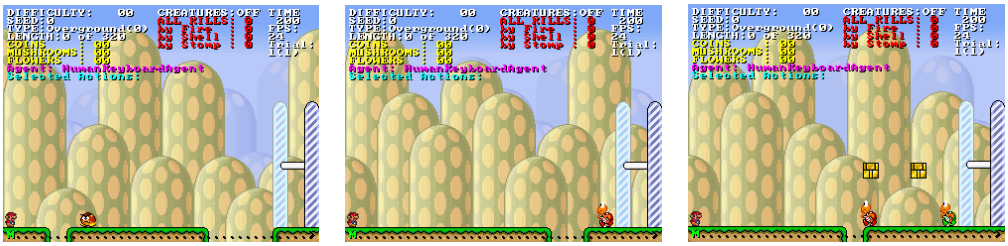
11.4.0.1 Mario



(a) No fired mechanics (b) 50% of fired mechanics (c) All fired mechanics

Figure 11.3: Three generated scenes with various degrees of number of fired mechanics.

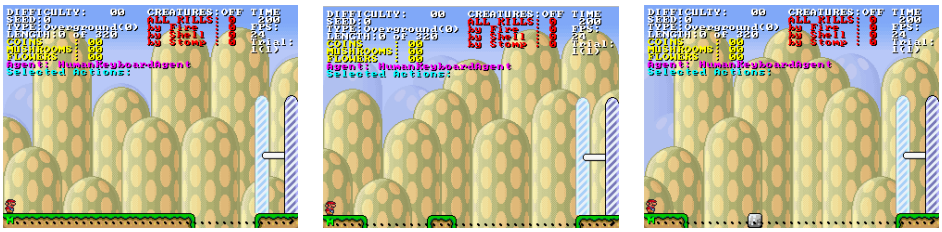
The CME algorithm is able to find a huge set of different combinations of triggered mechanics that range from no mechanics to all of them. Figure 11.3 shows several generated scenes with various fired mechanics. The level in Figure 11.3a does not require the use of any mechanic, as the floor is a flat surface without any game objects. Figure 11.3b shows a scene within which a player might trigger four different mechanics (jump, high jump, stomp, and coin) and Figure 11.3c displays a scene in which all eight mechanics can be triggered. It has two enemies to enable a shell kill, a question mark block for the mushroom, a coin to collect, etc. However, one may notice that these scenes do not guarantee a mechanic will be fired similar to the Limited Agents and Punishing Model techniques presented in Chapter 10). For example, Figure 11.3c can be beaten from the top of the blocks without triggering any stomps or shell kills, depending on what the player decides to do.



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

Figure 11.4: Three generated scenes with different ways to kill the enemies.

The CME method can capture multiple mechanics concerning how enemies can be slain. Figure 11.4 displays three scenes with different types of enemy kill mechanics. In a fall kill scene (Figure 11.4a), the generated scene contains goombas, as they can fall of the edge. In a stomp and fall kill scene (Figure 11.4b), the scene contains a red koopa troopa since they cannot fall of the edge of the screen unless they are first stomped. Scenes which contain every type of kill (Figure 11.4c) contain at least two enemies with one of them being a koopa troopa.



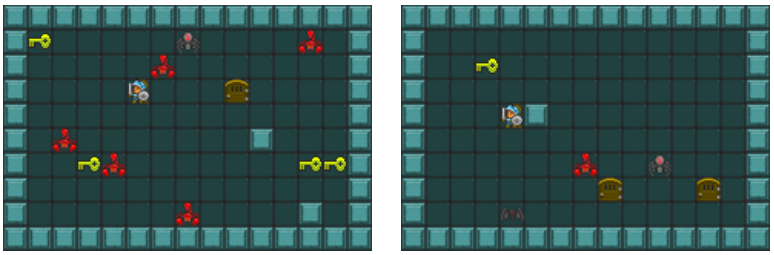
(a) Normal Jump (b) Normal Jump and Long Jump (c) Normal Jump, Long Jump, and High Jump

Figure 11.5: Three generated scenes with different ways to jump.

*	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

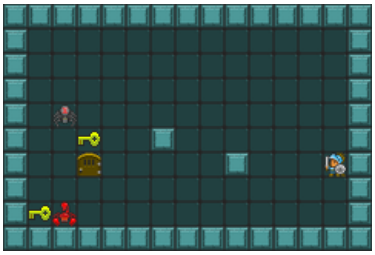
Table 11.2: Constrained MAP-Elites dimensions for the GVG-AI game Zelda

Similar to the kill mechanics, the CME method can also capture a variety of jump mechanics in a scene. Figure 11.5 displays 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 11.5a). 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) by performing a jump while running. The scene in Figure 11.5b requires that the second jump is long enough to reach the other side of the gap. Lastly, the long high jump requires a even longer distance, i.e. holding the jump button longer (Figure 11.5c) with a larger second gap while also performing a long jump. Similar to the punishing model experiments, all the generated level from the 5 runs have different layout while still maintaining the necessary set-up for the mechanic to occur.

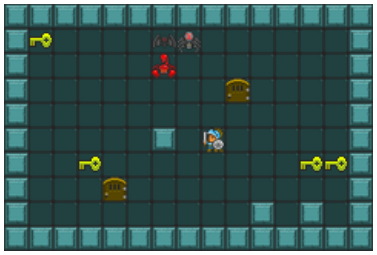


(a) mean: z1, z2, z3, z5, z7

(b) mode: z1, z2, z3, z7, z8



(c) least: z7, z8



(d) most: z1, z2, z3, z5, z6, z7, z8

Figure 11.6: A subset of generated elite levels for Zelda. Their string representation corresponds to their showcased mechanics detailed in Table 11.2.

11.4.0.2 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 11.6 displays four 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 11.6c) showcased 2 mechanics, out of 8 possible. The map with the most mechanic-dimensionality 11.6d contained 7. The least dimensional map matches up with the AtDelfi system's critical mechanics [60]. We can see however, that other mechanics from Table 11.2 can be triggered in this space, such as killing monsters and bumping into walls. We think that, due to the openness of the space, the agent failed to bump into any walls or monsters on its bee-line route to the key and the door. 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 and the monster movement patterns.

11.4.0.3 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

*	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 11.3: 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

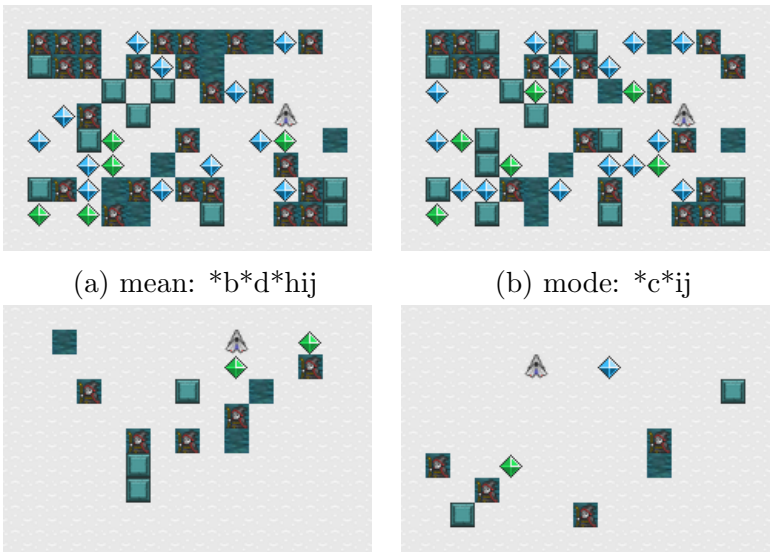
Table 11.4: Constrained MAP-Elites dimensions for the GVG-AI game Plants

map. The average fitness across all cells was 0.4311. Figure 11.7 displays four 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 11.7c), whereas the most (Figure 11.7d) contained 8 out of the possible 10 mechanics.

The representative least dimensional elite (Figure 11.7c) 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 start for an easy win. The most dimensional elite contains nearly every mechanic in the game except for 2: the agent hitting a wall and touching enemy ground tile where both kills the agent upon touching them.

11.4.0.4 Plants

After 500 iterations, 31 out of 128 possible cells are 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 11.8 displays four elites of varying



(a) mean: *b*d*hij (b) mode: *c*ij

(c) least: *hij (d) most: *cdefghij

Figure 11.7: A subset of generated elite levels for Solarfox. The string representation corresponds to their showcased mechanics detailed in Table 11.3.

dimensions. The mean and mode are calculated to be 3.93 and 5 respectively. The map with the least dimensionality showcase just 1 mechanic. The map with the most mechanic-dimensionality contains 6 out of the 7 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 algorithm optimized the zombie spawner placement such that it takes almost T_s before losing which will also allow the idle agent to pass almost all its trials. 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.

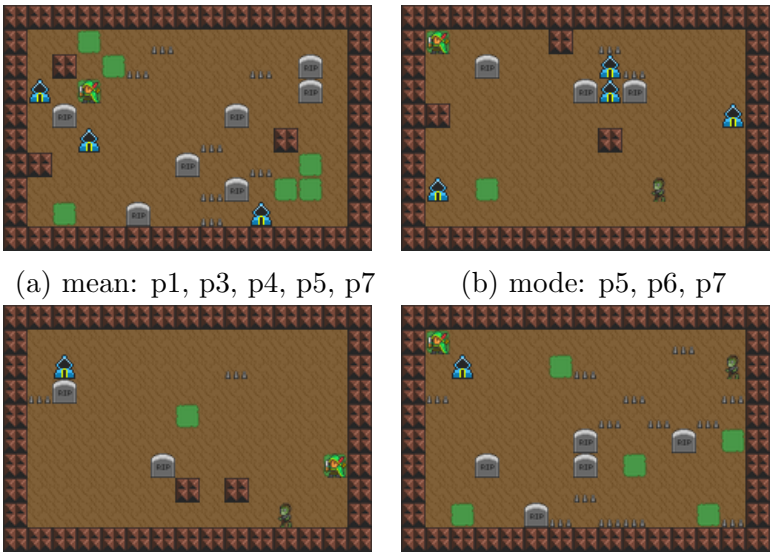
11.4.0.5 RealPortals

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 11.9 displays four 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 35 possible.

In contrast to the other games described above, RealPortals is extremely complex, as seen in the sheer amount of elites that can and do exist. Ironically,

*	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 11.5: Constrained MAP-Elites dimensions for the GVG-AI game RealPortals



(a) mean: p1, p3, p4, p5, p7 (b) mode: p5, p6, p7

(c) least: p4 (d) most: p1, p2, p3, p4, p5, p7

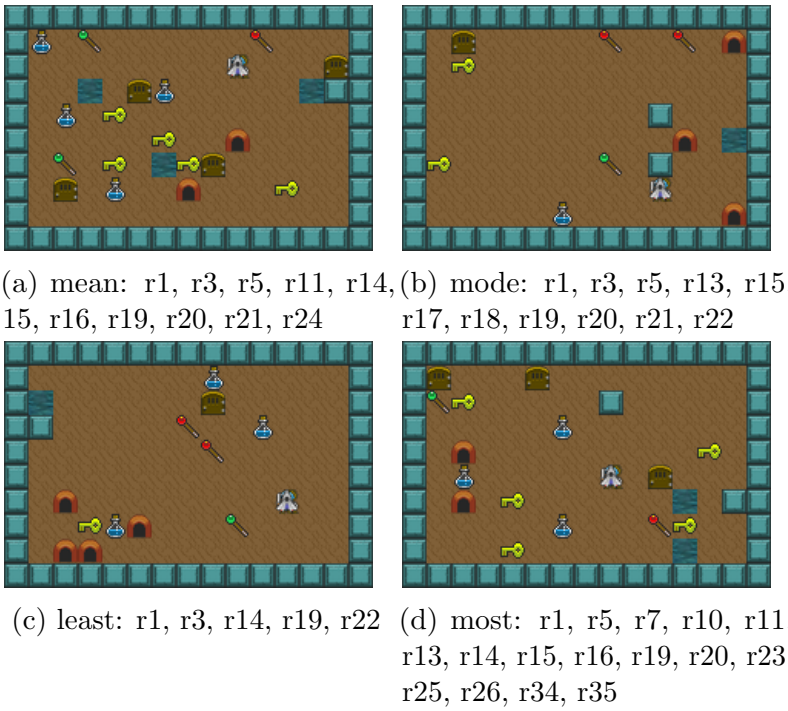
Figure 11.8: A subset of generated elite levels for Plants. Their string representation corresponds to their showcased mechanics detailed in Table 11.4.

the generated levels are all extremely simple to solve, unlike any of the RealPortals levels that come with the GVG-AI framework. Without water dividing the map and requiring the player to use portaling, the levels are transformed into a find-the-goal game that is 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). This is due to the inability of AdrianCTX agent to solve complex RealPortals levels. 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 moving two steps to the right from player start.

11.5 Discussion

Similar to shortcomings discussed in the Limited Agents and Punishing Model techniques discussed in Chapter 10, the Baumgarten's A* agent and AdrienCTX agent make mistakes. These mistakes result in some generated scenes being beatable even if the target mechanics are ignored.

However, the CME technique does guarantee that the mechanic *could* be used if the player desires it. It is able to generate scenes with multiple targeted mechanics, and it uses a single agent in the generation process, rather than two. Furthermore, the CME method does not search for one scene like the other two approaches, instead searching for diverse sets of scenes that contain any combination



(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

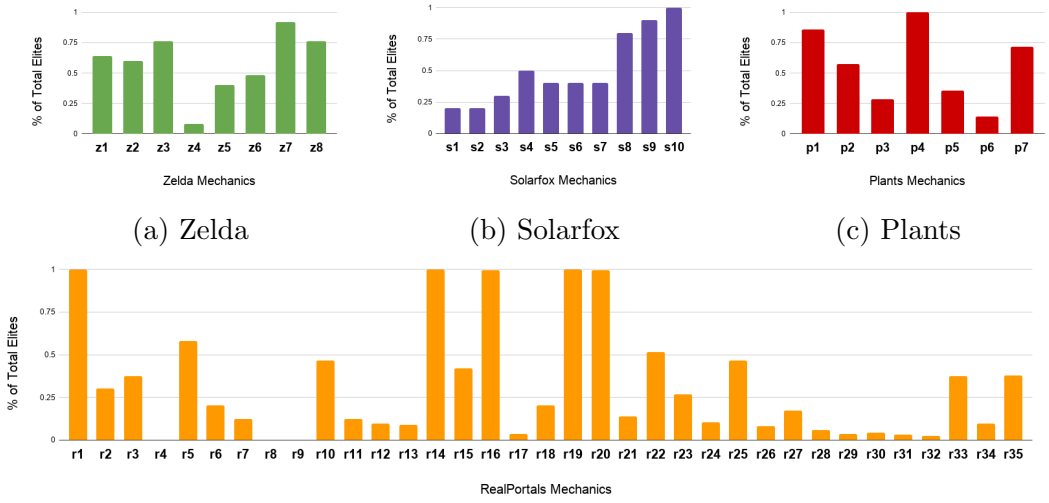
(d) most: r1, r5, r7, r10, r11, r13, r14, r15, r16, r19, r20, r23, r25, r26, r34, r35

Figure 11.9: A subset of generated elite levels for RealPortals. Their string representation corresponds to their showcased mechanics detailed in Table 11.5.

of mechanics. This way of generation might not be optimal in some cases, e.g. if the target mechanics are not that easy to find, the algorithm might spend too much time finding other scenes that contain unwanted mechanics. But it does illuminate mechanic space more efficiently than a single-objective method.

Within GVG-AI, CME is able to populate more than 10% and slightly less than 10% of the total cells with elites for Plants and Zelda respectively. 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 is not as explored, due to its 34-dimensional complexity compared to Zelda's meager 8-dimensions. However, while Solarfox (10) has less dimensions than RealPortals, it has a similarly low elite population.Though Solarfox is only 2 dimensions greater than Zelda, the dimensional space from Zelda to Solarfox increases from 256 possible cells to 1024. We also hypothesize that elite population is impacted not only by the total number of game mechanics, but also by the ability of the agent to solve the level, since AdrianCTX is unable to beat complex Solarfox levels [13]. Agents struggle on Solarfox because of the continuous movement of the player's avatar. This mechanic forces the agent to be responsive at each frame otherwise the agent can die quickly via inaction compared to the other games.

Similar to the Limited Agents and Punishing Model techniques, the quality diversity technique can theoretically generate scenes which require targeted mechanics by using an exhaustive search agent. Then the target mechanic is recorded



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

(d) RealPortals

Figure 11.10: 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 11.2; Solarfox: Table 11.3; Plants: Table 11.4; RealPortals: Table 11.5.

as fired only if it is triggered in all possible solutions. It would also be interesting to have an agent that can adapt to a game using some game specific information. One way to do this might be to take advantage of hyperstate information [30]. Another way would be use the information extracted from the VGDL file as an evaluation function for the agent [61]. Yet another direction would be to aggregate the unique mechanics triggered across a multitude of agents, to get a better sense of the mechanic space.

Across all games, when compared to the original levels, the generated levels provide a sense of uniformity. Mario levels are simple, even when containing the potential for lots of mechanics. The Zelda elites consist of wide open spaces, instead of the usual maze-like patterns. Solarfox levels tend to be sparsely populated with blibs, with a few exceptions (Figure 11.7b) instead of geometric arrangements of blibs and powerblibs. There tend to be no water tiles present in RealPortals, or marsh tiles in Plants, relative to each games' original levels. We hypothesize that all of this is due to the entropy pressure from the fitness calculation specified in Equation 10.1 for Mario and Equation 11.5 for GVG-AI, which drive evolution towards creating simplistic levels with large amounts of empty tiles or highly populated levels filled with a lot of similar tiles.

11.6 Summary

In this chapter, we present experiments in which we generate mechanic-dependent levels, i.e. levels that are generated to offer the potential to use a mechanic or set of mechanics, using quality diversity. We propose using QD over single-objective optimization methods like FI2Pop for several reasons. The first is unlike the FI2Pop methods, the QD approach does not require manual agent or forward model manipulation. The second is that QD algorithms can provide a diverse sampling of level space more so than FI2Pop can.

Having a more diverse sampling of level space is highly useful for tutorial level content. If a large variety of scenes are available, a tutorial generative system has greater agency in constructing level curricula for player consumption. We will revisit this idea in Chapter 13 when we build longer Mario levels out of scenes. In the next chapter, we explore an entirely different approach to generating tutorial scenes through the use of procedural personas.

Chapter 12

Persona-Dependent Tutorial Scene Generation

In the previous chapters, we presented evolutionary methods to generate levels which showcase the use of *game mechanics*. Remember how in Chapters 7 and 8 we analyzed and classified players by their playstyle using their in-game mechanic behavior? In this chapter, we demonstrate a way to evolve small tutorial levels tailored to a specific playstyle. There are several manners in which one could do this. One way is to create a level that mandates the use of certain mechanics (or in general, overcomes a specific type of challenge) and that is winnable using the currently used playstyle. Another type of tutorial level is one that requires the player adopt a different playstyle to win. In the latter case, the playstyle may be something the game wishes to teach in order to enable the player to enjoy more of the game. The game can also teach when to avoid a certain playstyle to help the player understand the strategic depth of the game. If you can identify a player's playstyle, then using the method presented in this chapter you could generate levels specifically for that player.

Here we investigate methods for generating levels tailored to playstyles in MiniDungeons 2 [82], which is described in Section 4.3. For this, we use the concept of procedural personas. A procedural persona is a generative model (i.e., a game-playing agent) of an archetypical playstyle [77, 78]. The three procedural personas used in this study – the runner, the monster killer, and the treasure collector – exemplify sharply different strategies of playing a game. We then use a quality-diversity algorithm, Constrained MAP-Elites [99] (CME, Section 5.2.2), to find large sets of different levels. The levels systematically vary depending on which personas can perform well on them. Thus, the system can generate levels that can be played equally well by multiple playstyles, that encourage the use of a particular playstyle, or that discourage the use of a particular playstyle. The work

in this chapter comes from a paper with myself as first author published at the International Conference on the Foundations of Digital Games, 2022 [67].

12.1 Persona-Driven Level Generator

We use the evolutionary search-based algorithm known as Constrained MAP-Elites (CME) [99] to generate levels, which is described in Section 5.2.2. The evolutionary pipeline consists of 5 distinct steps: initialization, evaluation, replacement, selection, and mutation. The system uses these steps in the following fashion to generate levels:

1. Initialize a multi-dimensional grid of cells where each cell have 2 populations for the feasible and the infeasible chromosomes where each population of size "MAP_CELL_SIZE".

2. Generate an initial population of chromosomes of size "POPSIZE" using the **Initialization** step.

3. Evaluate the current population of chromosomes fitness and behavior characteristics using the **Evaluation** step.

4. Insert the current population of chromosomes into the multi-dimensional grid using the **Replacement** step.

5. Generate a new population of chromosomes of size "POPSIZE"

   (a) Select a chromosome from the grid using the **Selection** step.

   (b) make a new copy of the selected chromosome using the **Mutation** step.

6. Repeat steps 3 to 5 for number of iterations equal to "ITERATIONS".

Chromosomes (game levels) are represented as two 2D array of size "LEVEL_WIDTH" x "LEVEL_HEIGHT" where each tile can be any of the possible tile types in MD2 (empty, solid, hero, potion, treasure, trap, portal, goblin, goblin wizard, blob, ogre, and minitaur). Table 12.1 provides a description of all hyperparameters as they are referenced in the previous steps and the following sections as well as their experiment values.

12.1.1 Initialization

The system initially creates blank maps (empty map with only wall border) with a fixed width and height ("LEVEL_WIDTH" x "LEVEL_HEIGHT"). In our experiments, level width and height are fixed to a size of 10x10, in contrast to the

Phase	Parameter	Description	Value
Setup	B	Bucket size of the MAP-Elites matrix	5
	P	Number of dimensions that is used for the MAP-Elites matrix	3
	MAP_CELL_SIZE	Maximum number of members allowed to be stored in a cell for the feasible/infeasible population	5
	LEVEL_WIDTH	The width in tiles of evolved MD2 levels	10
	LEVEL_HEIGHT	The height in tiles of evolved MD2 levels	10
Evolve	ITERATIONS	How many times to evolve (evaluate, select, mutate) a population	500
	POPSIZE	Maximum size of the population. Members may be removed if they are invalid to be evaluated	60
Initialize	EMPTY_INIT_RATE	Probability to place an empty tile in the initialization phase of the map	0.5
	WALL_INIT_RATE	Probability to place a wall tile in the initialization phase of the map	0.3
Mutation	EMPTY_MUT_RATE	Probability to place an empty tile in the mutation phase of the map	0.5
	MUTATION_RATE	The probability to mutate the current tile (iterates over every tile in the map excluding borders)	0.1
Selection	ELITE_PROB	Probability to select a sample from the elite population	0.8
	FEAS_PROB	Probability to select a sample from the feasible population	0.6
Personas	C	Weight constant for agent's utility cost	45
	K	Weight constant for agent's death cost	1

Table 12.1: Hyperparameter descriptions and experimental values

traditional MD2 levels which are 10x20. We selected this smaller size as it allows for faster agent evaluation which is described in Section 12.2. This also makes for smaller and simpler levels, which is inline with the primary motivation of tutorial levels.

Every tile in the map (except the border tiles) is iteratively changed to a random tile based on predefined probabilities where empty and wall tiles have higher probabilities than other game elements (empty probability and wall probabilities are defined in Table 12.1 as "EMPTY_INIT_RATE" and "WALL_INIT_RATE" respectively). After this step, the randomly initialized levels are repaired such that these levels contain exactly one player, one exit, and either exactly 2 or 0 portals. The algorithm repairs the generated levels by either placing the missing tile at a random location or removes the excessive tiles until the levels are fixed.

12.1.2 Evaluation

The maps are placed into a Constrained MAP-Elites grid with each cell in the matrix containing two populations: feasible and infeasible. The feasibility of the map is determined by calculating a constraint value. The constraint value for the maps are based on the connectivity of the map. A breadth-first algorithm is conducted on every non-wall tile in the map starting from the player's initial position, and if every non-wall tile is reached the constraint value is 1 and the map is considered feasible. Infeasible maps contain disconnected non-wall tiles and the constraint value is based on the ratio between reachable tiles (t_r) and all non-wall tiles (t_t) as shown in the following equation:

$$S = \frac{t_r}{t_t} \tag{12.1}$$

The fitness function measures the simplicity of the levels. To calculate simplicity of the generated levels, we calculate the entropy of the different tiles in the generate level like in Chapters 10, 11, and 12. With this fitness, the generator will try

to evolve levels that have uniform tile types which makes levels look less noisy. Equation 12.2 displays the map fitness equation, where p_i is the percentage of a specific tile i existing on the level and n is the total number of distinct tiles in MD2.

$$fitness = 1 + \sum_{i=0}^{n} p_i \cdot \log(p_i) \tag{12.2}$$

In order to place a newly generated level in the CME grid, its behavior characteristics need to be calculated. In this work, the behavior characteristics are the remaining hit points after level completion by each persona, ranging from 0 (death) to 10 (full health). The health is divided into buckets of $10/B$ health where "B" is the hyperparameter that determines the size of each dimension. With multiple different personas, the size of the MAP-Elites matrix will be B^P cells, where "P" is a hyperparameter that determines the number of used personas. In our experiments, we use 3 different personas which are runner, monster killer, and treasure collector; and 5 buckets per dimension. More information about the different personas will be described in Section 12.2.

12.1.3 Replacement

The elites of this matrix are defined as the levels with the highest fitness values in the feasible population of the cells. Each cell has a limit to the size of the population ("MAP_CELL_SIZE"). When a new generated level has a better fitness or constrained value than levels in the current cell's feasible or infeasible population, the least fit level in that population is replaced by the new level. In the case of the feasible population, if the new level has better fitness than the least fit level in this population, the new level is inserted and the least fit level will be moved to the infeasible population instead of being completely deleted. On the other hand, if the new level is going to be inserted in the infeasible population, it must be better than the least performing level to replace it. After a couple of iterations, the infeasible population will contain substantially fewer maps with low constraint values as more fitted individuals are being pushed into the cell. This "survival of the fittest" replacement for the feasible and infeasible populations maintains the quality of the matrix and all of the maps contained in the cells while providing better selection possibilities for future populations.

12.1.4 Selection

In the selection step, it is important to note that behavioral characteristics are not considered at all. Every elite level from the matrix is added into an elite selection pool indiscriminately. Members of the feasible and infeasible populations

are added to their own respective pools. Elites are given the highest probability ("ELITE_PROB") to be selected, then feasible ("FEAS_PROB"), and lastly infeasible ($1 - ELITE\_PROB - FEAS\_PROB$). When a set is chosen, a chromosome is selected randomly from the set to be mutated (explained in Section 12.1.5) before being added to the next generation's population.

12.1.5 Mutation

During the mutation step, every tile of a selected map is given a chance of mutating as controlled by the rate of tile mutation ("MUTATION_RATE"). If a tile is set to mutate, it may change itself into any tile in the game with equal probability except for an empty tile, which has a higher chance to be selected as the end mutation ("EMPTY_MUT_RATE"). All the other tiles have equal probability to be selected. We have the empty tile with higher probability than the rest to encourage the evolution to erase more often than adding. The same repair function from the initialization phase is used to ensure there is only 1 player, 1 exit, and 2 or 0 portals (see Section 12.1.1). If a mutation makes it physically impossible for the player to ever reach the exit, the level will be removed and will be inserted back to the CME grid. Since they are unplayable, these levels cannot allow agents to determine their behavior characteristics. Consequentially, a generation may contain less chromosomes than the predefined size ("POPSIZE"), however all of the levels will be guaranteed to be playable.

12.2 Procedural Personas

Our experiments use an online-planning best-first search (BestFS) agent to evaluate levels. Each turn, the agent is limited to building a 500 node tree to decide their next action. 500 nodes is chosen after preliminary experiments suggest that a BestFS agent can play well but still be deceived with the resulting limited planning horizon. In theory, this could be varied to approximate player skill [86], however we do not perform that in this paper. The BestFS agent plays every level three times using one of the three different personas. The results of these three runs determine that level's behavioral characteristics. Best-first search does not need to play the level more than once per persona due to the deterministic nature of both the game and the algorithm.

As mentioned in Section 12.1.2, the behavior characteristics of a level correspond to the remaining health of the different personas after completing the tested level. In previous work by Holmgaard et al. [77, 80], three main personas are identified for MD2:

1. **Runner (R):** complete the level as fast as possible.
2. **Monster Killer (MK):** slay as many monsters as possible before completing the level.
3. **Treasure Collector (TC):** open as many chests as possible before completing the level.

We describe the utility functions for these agents in detail in Section 4.3.2.

12.3 Results

In this section, we review the results from CME over five evolutionary runs. When we refer to a level's behavioral characteristics, we refer to its bucketed values in order of runner, treasure collector, and monster killer HP results. Buckets are numbered 0 to 4, totaling 5 buckets. For example, a level with dimensions of 444 means that the agent finished the level in the 4th bucket (the highest amount of HP) with every persona. We divide the levels into 3 major types:

- **Balanced:** These are levels where all three personas complete the level with approximately the same HP. These are the diagonal cells in the CME matrix (000, 111, 222, 333, and 444).
- **Dominant:** These are levels where a certain persona (R, TC, or MK) completes the level with more HP than the other two personas. We call them Runner dominant, Monster Killer dominant, and Treasure Collector dominant levels.
- **Submissive:** These are levels where a certain persona (R, TC, or MK) completes the level with less HP than the other two persona. We call them Runner submissive, Monster Killer submissive, and Treasure Collector submissive levels.

By differentiating levels this way, we can better analyze a level's ability to encourage or discourage a player to behave as a specific persona. First, we analyze the resulting MAP-Elite matrix, focusing on the elite coverage. We then analyze the levels in terms of their persona HP outcomes, as defined in the paragraph above.

12.3.1 Elite Matrix

The elite map contains 5x5x5 cells, or 125, since each dimension is made of 5 buckets. If we aggregate the five runs, CME fills an average 34.2 cells (27.36%)

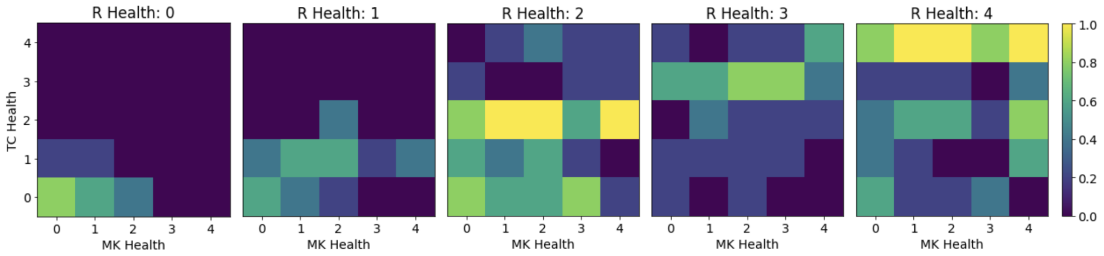


Figure 12.1: The probability of an elite being found for each cell over 5 different runs.

Persona	Dominant	Submissive	Max Levels
Runner	**9.0 ± 3.16**	2.0 ± 1.67	30
Treasure Collector	1.8 ± 0.75	**7.4 ± 1.74**	30
Monster Killer	6.0 ± 2.45	**11.8 ± 2.31**	30
Balanced	**4.2 ± 0.75**		5

Table 12.2: Mean and standard deviation of the number of different level types discovered across 5 runs.

with standard deviation of 4.82. Figure 12.1 displays the elite map of populated cells across 5 experimental runs. Higher runner health seems to correlate to higher cell coverage. This is likely to be because monster killer and treasure collector heuristics will default to runner heuristics (get to the exit) once their primary objectives are completed. Generating a level where a runner persona loses health while other algorithms loses less health in comparison is difficult to do, as it is not easy to deceive a runner without also deceiving the other two personas. The runner's final health tends to act as an upper bound for the final health of the other agents. Although there are maps where TC and MK personas complete the level with higher health, they are less likely to be found.

To understand more about the ability of CME to generate different types of levels, we analyze the number of *dominant* and *submissive* level types across 5 experimental runs as shown in table 12.2. The runner persona tends to dominate the other two personas more often than it is submissive. As mentioned before, both treasure collector and monster killer personas behave similar to the runner when there are no treasures or monsters respectively. In contrast, balanced levels seem relatively easy to discover, as on average 4 levels are found out of a maximum of five per experimental run. If both treasure and monsters are placed along or nearby the shortest path to the exit, all three personas tend to behave the same and therefore have identical HP outcomes. It is also relatively simple to create impossible levels by placing lots and lots of monsters near the exit (see Figure 12.2a).

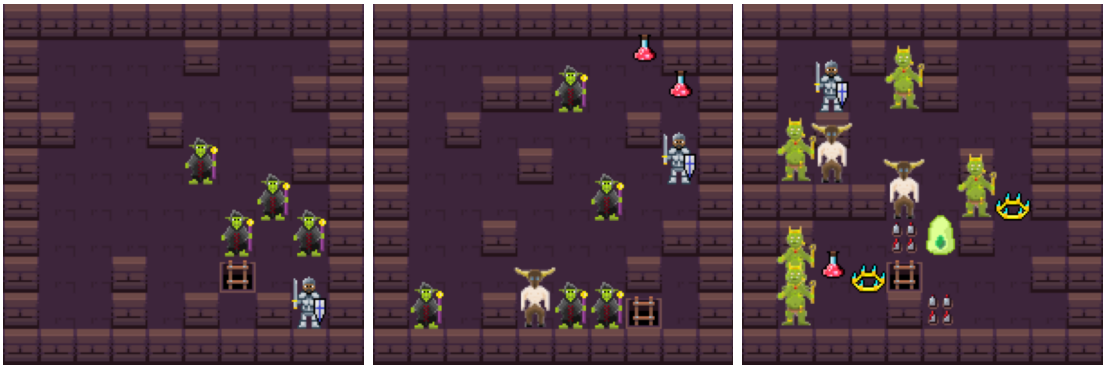
12.3.2 Balanced Levels

Balanced levels are the levels in which all three personas finish with a similar health percentage. These levels have behavior characteristics of either 000, 111, 222, 333, or 444. A level with a behavior characteristic of 444, means that the agent finished the level with at least 8 or more HP remaining with every persona.

Figure 12.2 displays 3 example levels for 3 different balanced levels values: 000, 222, and 444. The 444 levels (Figure 12.2c) are levels in which all 3 personas finish with nearly all of their HP. These are the simplest, easiest possible levels, consisting of just the hero and the exit. The only difference between these levels is the position of the player and exit location. Since the system has no constraint on the solution length, a level where the player starts next to an exit is scored similar to a level that requires the player to learn how to move and navigate a maze. However, the simplicity-based fitness function will pressure the generator to evolve empty levels since they will have higher fitness. We did not include path length as part of the constraints or fitness, since we wanted these levels to encourage a user to play as a certain persona and not how to move and play in general. Figure 12.2a displays levels where all three personas die or have very low health (000 levels). Placing lots of enemies along the shortest path towards the exit quickly obliterates HP regardless of the persona. Figure 12.2b displays levels where all 3 persona have medium amount of health (between 4 to 6 HP). Levels such as these tend to have a small amount of monsters that will deal some damage to the player. Similar to the hard levels in Figure 12.2a, the enemies are close to shortest path towards the exit.

12.3.3 Dominant Levels

Dominant levels are levels where one persona completes the level with higher HP than the other two personas. For example, a 321 level is a runner dominant level, while a 242 level is a treasure collector dominant level, and a 024 level is a monster killer dominant level. Since the dominant levels cover 25% (30 cells out 125 cells as shown in table 12.2) of the map, we will focus on the dominant levels that have the highest fitness in the elite map. Figure 12.3 displays expressive range analysis of all the discovered dominant levels across 5 runs. Runner dominant levels (Figure 12.3a) tend to have at least 1 treasure or more and consistently have short distances to the exit. Monster killer dominant levels (Figure 12.3c) have longer paths to exit with fewer monsters than the runner dominant levels. There are not too many treasure collector dominant levels as they appear to be difficult to discover (only 9 levels across 5 runs). We believe this is because of the difficulty of using treasures to guide the treasure collector down a different path where they do not take damage while the runner does.



R: 0| TC: 0| MK: 0
Fitness: 0.6

R: 0| TC: 0| MK: 0
Fitness: 0.53

R: 0| TC: 0| MK: 0
Fitness: 0.44

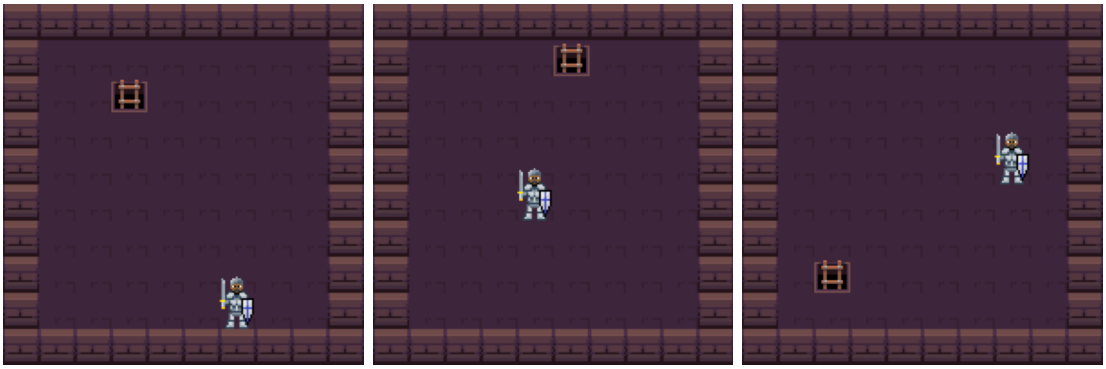
(a) All personas die/nearly die

R: 2| TC: 2| MK: 2
Fitness: 0.63

R: 2| TC: 2| MK: 2
Fitness: 0.6

R: 2| TC: 2| MK: 2
Fitness: 0.6

(b) All persona lose some health



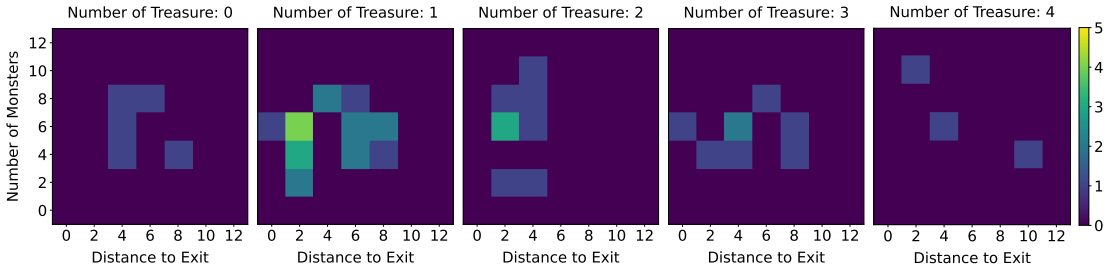
R: 4| TC: 4| MK: 4
Fitness: 0.67

R: 4| TC: 4| MK: 4
Fitness: 0.67

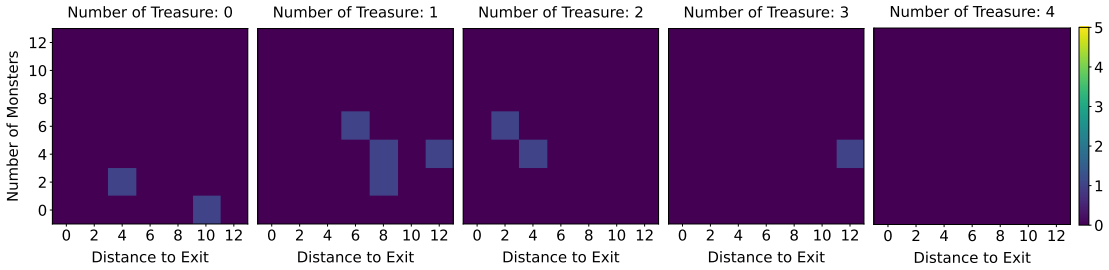
R: 4| TC: 4| MK: 4
Fitness: 0.67

(c) No personas lose health

Figure 12.2: Simplest generated balanced levels across 5 runs.



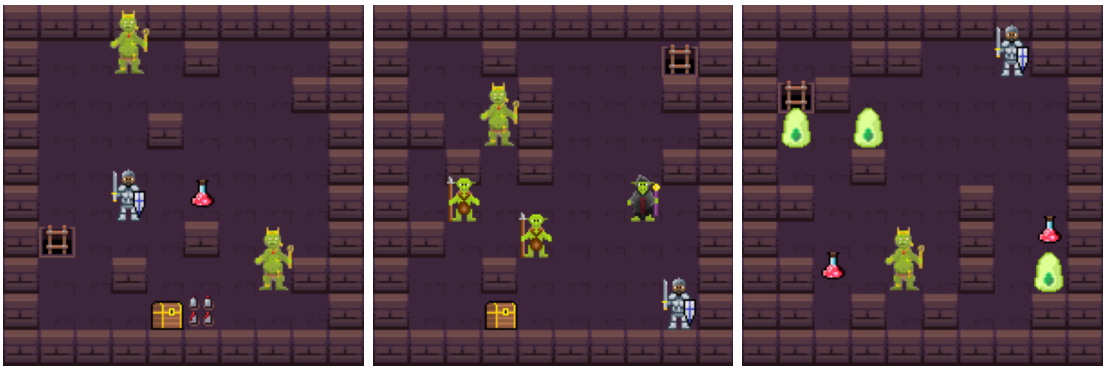
(a) Runner Dominant Levels



(b) Treasure Collector Dominant Levels

(c) Monster Killer Dominant Levels

Figure 12.3: Expressive range analysis for the different dominant levels across 5 runs.

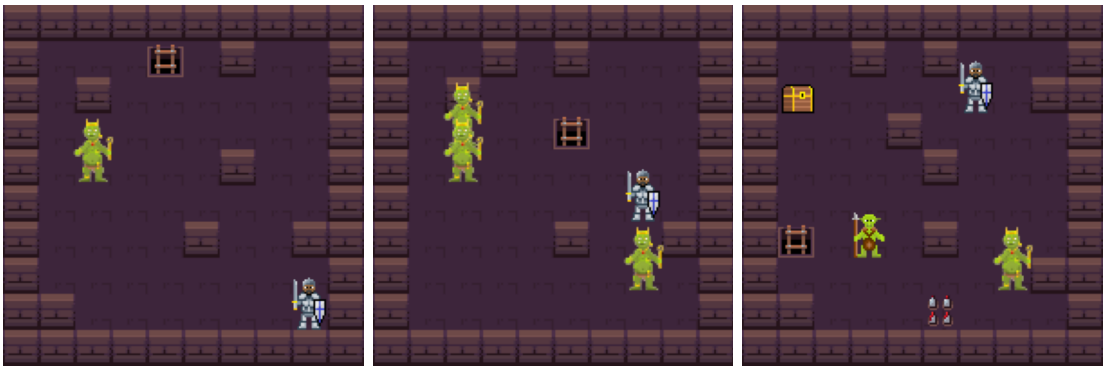


R: 4| TC: 2| MK: 2
Fitness: 0.56

R: 3| TC: 2| MK: 1
Fitness: 0.56

R: 4| TC: 2| MK: 2
Fitness: 0.55

(a) Runner Dominant Levels



R: 2| TC: 4| MK: 2
Fitness: 0.64

R: 2| TC: 4| MK: 1
Fitness: 0.61

R: 3| TC: 4| MK: 3
Fitness: 0.57

(b) Treasure Collector Dominant Levels

R: 2| TC: 2| MK: 4
Fitness: 0.64

R: 2| TC: 2| MK: 4
Fitness: 0.64

R: 2| TC: 2| MK: 4
Fitness: 0.64

(c) Monster Killer Dominant Levels

Figure 12.4: Simplest generated dominant levels where a certain persona ends with higher level than the other two.

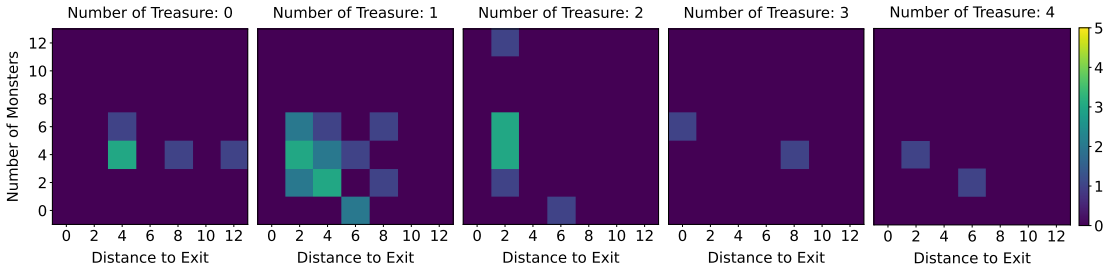
Figure 12.4 show the highest fittest levels where runner, treasure collector, and monster killer dominates respectively. The runner dominant levels (Figure 12.4a) contain short direct paths to the exit with a few enemies nearby. A treasure chest is sometimes placed close to the rest of the monsters so that the treasure collector takes damage from the surrounding monsters on its way back to exit after collecting the treasure. Occasionally there is no treasure but the treasure collector still loses health, due to the different cost functions between the personas influencing the agent to take different paths. The monster killer dominant levels always have one enemy beside the shortest path to the exit so the monster killer can just kill it using the javelin without taking any damage while the other two personas encounter them. The treasure collector dominant levels (Figure 12.4b) usually have treasures around the map to guide the treasure collector persona away from monsters (see Figure 12.3b). We show two of the simplest maps that do not have treasure yet the treasure collector manages to avoid taking much damage anyway, again due to the nuances of the collector's heuristic and cost functions. We did not expect this type of behavior from treasure collectors, and we want to review the heuristic and cost functions of collectors in future work in order to generate more treasure collector dominant levels with treasures in them.

12.3.4 Submissive Levels

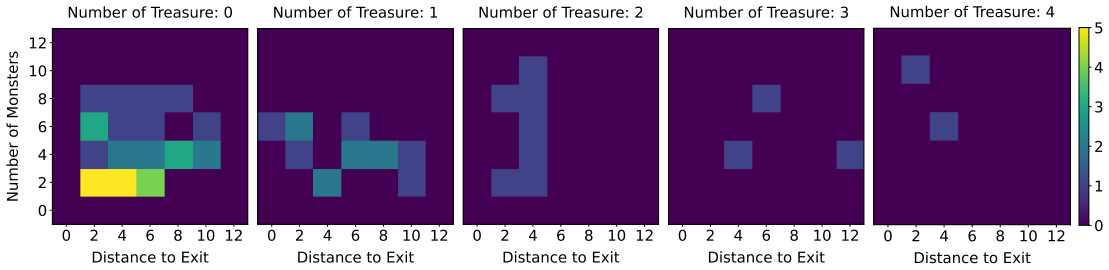
Submissive levels are levels where one of the personas finishes with lower HP than the other two. For example, a 143 level is a runner submissive level, a 301 level is a treasure collector submissive level, and a 241 is a monster killer submissive level. Figure 12.5 displays the expressive range analysis for the submissive levels. Runner submissive levels (Figure 12.5a) are difficult to find (10 levels across 5 runs), which is expected as it is difficult to make a runner persona lose health without the other personas also losing health (monster killers especially). Treasure collector submissive levels (Figure 12.5b) tend to have at least one treasure chest, usually in proximity of monsters who attacks the nearby collector. Also, these levels usually have a short path to the exit allowing runner to run to exit with less taken damage. Monster killer submissive levels (Figure 12.5c) tend to have 0 or 1 treasures and small amount of monsters with short to decent distance to exit. Not having a lot of treasures influences the collector and runner personas to move towards the exit without battling with monsters while monster killer will always go to battle them and lose health.

Similar to the dominant levels, we show the highest fitness level for each submissive persona in Figure 12.6. Looking at table 12.2 and figure 12.5, monster killer or treasure collector submissive levels seem to be easier to discover than

(a) Runner Submissive Levels



(b) Treasure Collector Submissive Levels



(c) Monster Killer Submissive Levels

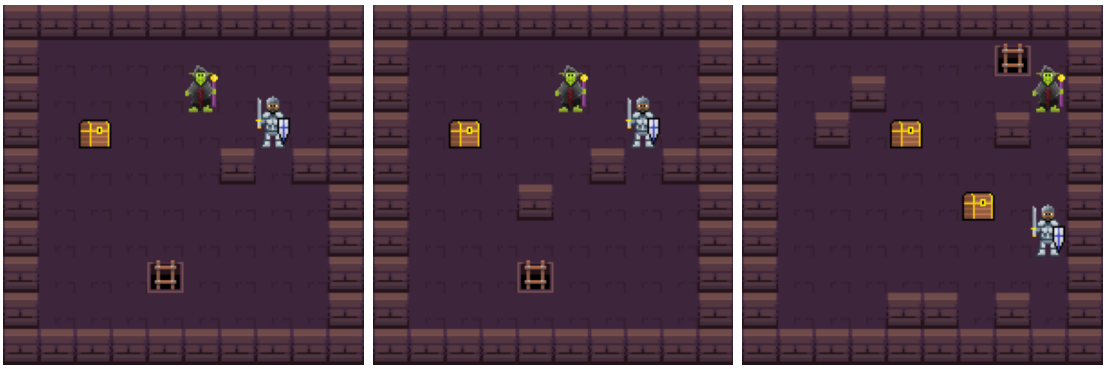
Figure 12.5: Expressive range analysis for the different submissive levels across 5 runs.

R: 2| TC: 4| MK: 4
Fitness: 0.64

R: 2| TC: 3| MK: 3
Fitness: 0.62

R: 3| TC: 4| MK: 4
Fitness: 0.62

(a) Runner Submissive Levels

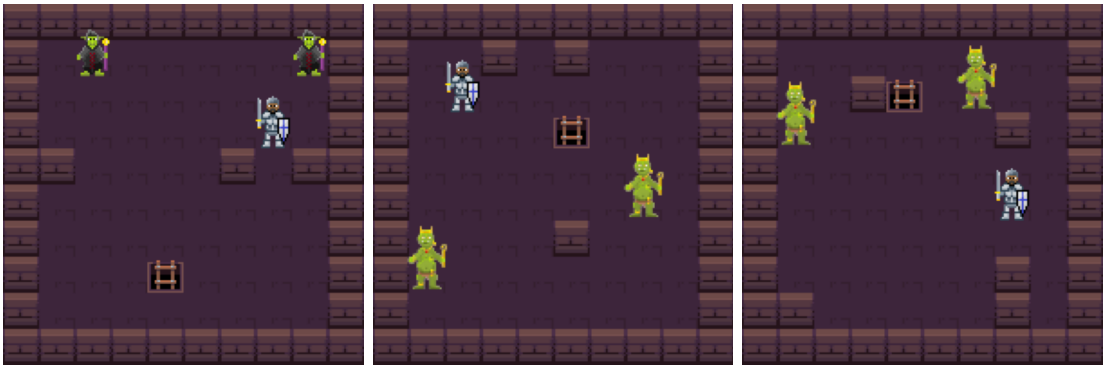


R: 4| TC: 1| MK: 4
Fitness: 0.62

R: 4| TC: 2| MK: 4
Fitness: 0.62

R: 3| TC: 2| MK: 4
Fitness: 0.6

(b) Treasure Collector Submissive Levels



R: 4| TC: 4| MK: 3
Fitness: 0.63

R: 4| TC: 4| MK: 2
Fitness: 0.63

R: 4| TC: 4| MK: 2
Fitness: 0.62

(c) Monster Killer Submissive Levels

Figure 12.6: Simplest generated submissive levels where a specific persona finishes with lower health than the other two.

runner submissive ones. The treasure collector submissive levels (figure 12.6b) place treasure and wizards in such a way that the runner can go directly to the exit without taking damage while the treasure collector gets hit by a wizard. In the runner submissive level (figure 12.6a), an enemy is placed close to the shortest path to the exit so it damages the runner on its way to the exit. Sometimes treasure is placed so that a treasure collector is guided away from the shortest path. The monster killer submissive levels (Figure 12.6c) place monsters in such a formation that the runner and treasure collector do not have to engage them to win the level. The monster killer will engage them and take damage.

12.4 Discussion

In this section, we discuss the results through the lens of the motivation for this paper: tutorial levels. Our original goal is to generate levels which encourage or discourage a player in following the behavioral patterns of a certain persona. Below, we analyze levels that spotlight situations in which some persona tactics end in success and others more likely failure. We also discuss the correlation of behavioral characteristics and difficulty, arguing a level's difficulty for a certain persona is not determined solely from behavioral characteristics but that it does play a role.

12.4.1 Tutorial Levels

A notable subset of elites — especially levels where the runner finishes with the most health out of the 3 personas — are generated levels that place the player right next to the exit but have multiple monsters and/or treasures placed throughout the map (see Figure 12.4a). These levels present a great example of encouraging certain playstyles for players - particularly for the runner persona. The monster killers will attempt to kill all of the monsters on the map and the treasure collector will try to get all of the treasure. But because of the overwhelming number of enemies, both personas either make it to the exit with very low health or die. However, the runner persona will only have to move a few steps to reach the exit, typically without facing any danger from traps or monsters. While these levels may not necessarily kill a player who plays however they want, they show that not every persona's strategy is easy, and they encourage a specific playstyle while punishing others.

Some of the levels encourage the player to use specific mechanics. For example, the monster killer dominant level shown in figure 12.4c nudges the player towards throwing the javelin to avoid losing health. These levels may encourage the player

to perform these mechanics but does not outright force them. If a designer wanted to force mechanic usage, they could simply change the player starting health. For example, if the player begins with only 2 HP in the monster killer dominant level (Figure 12.4c, then the level will force the player to learn that the only way to win is to throw the javelin.

Similarly, submissive levels are good at discouraging players from certain behaviors. For example, treasure collector submissive levels (Figure 12.6b) may teach the player that not every treasure is safe to acquire since that there might be enemies protecting it. We can make any submissive level an extreme variant by decreasing the player's starting HP to make these situations even more dangerous. The player will either need to learn to kill the enemies with the javelin to acquire the treasure (learning to be a monster killer) or avoid the treasure and run to the exit (learning to be a runner).

12.4.2 Correlation to Difficulty

Having a large amount of HP leftover for a persona does not always necessarily mean the level is "easy" for that persona. It does, however encourage the player to adopt the mindset of that persona style in order to maximize their ending HP. On the other hand, having a small amount (or hardly any) HP leftover for a persona typically does mean that playing as that persona is difficult.

One way to think about HP in MD2 is as a currency the player can spend. Some levels may require a player to spend more HP than others. On levels that require more spending, the player often has less leeway in choosing when and how to spend that currency. In rogue-like game design literature, this concept is often referred to as "strategic headroom" [148]. Typically levels that have more headroom are easier for the player to win (more options for the player to use) than levels that have less headroom. In MD2, levels that require less HP spending may offer the player more options to take (Figure 12.2b) and therefore provide more headroom. Levels like the ones in Figure 12.2a afford little headroom and are therefore difficult.

12.5 Summary

This chapter describes the development and results of a persona-driven tutorial level generation system that uses quality diversity in order to find and improve levels catered towards a specific playstyle. A set of generated elite levels are found in the solution space which address a range of playstyles. These levels often push a player towards trying a specific strategy ranging in difficulty. Sometimes

multiple playstyle types may succeed in a single level through their own strategy. While this system uses the domain of MD2, a deterministic dungeon crawler game, our pipeline could be applied to other games so long as it has a clearly defined mechanic space allowing for a range of different playstyles. This system could be incorporated into a persona-adaptive tutorial sequencer so that players may learn to adapt to different playstyles as the level context and difficulty changes. Similar to the previous chapters on level generation (Chapters 10 and 11), entropy pressure drives evolution to create simple levels. This, combined with a low A* tree size to restrict agent planning horizons (500 nodes per turn), causes levels to be easy when all agents beat them with high HP.

The work discussed here and in the previous two chapters describe systems which generate scoped challenges for the player to overcome. They can be either mechanic-driven via single-objective optimization (Chapter 10), mechanic-driven via quality diversity (Chapter 11) or persona-driven (Chapter 12). These generated scenes can be considered encapsulated tutorials which contain their own curriculum ("Learn this playstyle or mechanic"). If played sequentially, these levels can allow a player to practice skills in different circumstances and difficulties. The next chapter's work is born of this sequential order idea: what if scenes are combined to create a longer level?

Chapter 13

Sequential Tutorial Level Generation

In this chapter, we present a method of experience-driven level generation for levels in which a player triggers the same game mechanics as a target level while being structurally different. Experience-Driven PCG [175] is a type of PCG that generates content so as to enable specific player experiences. This definition includes the production stylized generated content such as levels of specific difficulty [7], different playstyles [99], or even for playing ability [175].

Evolving large levels to be individually personalized is not trivial, therefore we propose a method of multi-step evolution using mini-level "scenes" that showcase varying subsets of game mechanics. By breaking this problem down into smaller parts, the system is able to focus more on the bigger picture of full level creation. Reis et al's generator [134] is a similar system which stitches together human-evaluated/human-annotated scenes to create longer levels. In contrast, our scenes are automatically generated using the Constrained MAP-Elites generator from Chapter 11 which creates the scene corpus with simplicity and minimalism in mind. An example of such a scene is displayed in Figure 13.1. These scenes are stored in a publicly available library[1]. The system uses these scenes with an FI2Pop [102] optimizer to evolve the best "sequence of scenes," with the target being to exactly match a specific player's mechanical playtrace. The levels that the system generates reflect a type of minimalist design that can be used as a tutorial to learn a target mechanic sequence. The work in this chapter is from "Mario Level Generation From Mechanics Using Scene Stitching" with myself as first author published at the Conference on Games, 2020 [68] and uses the Mario AI Framework described in Section 4.2.

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

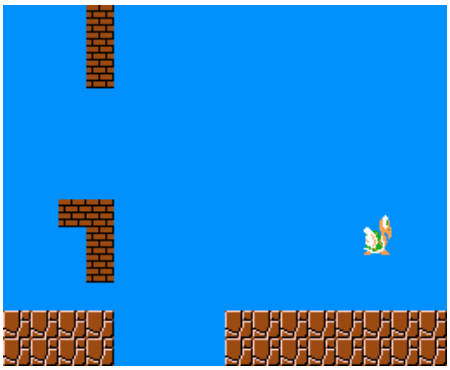


Figure 13.1: A example a scene from the system's scene library

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

13.1 Methods

Using a target mechanic playtrace, our system is able to generate levels that are mechanically analogous to the input.

Every scene is labelled with the mechanics that an agent triggered while playing it. Table 13.1 shows all the game mechanics in the Mario AI Framework that scenes may be labelled with. The scene library used in this system, evolved using a MAP-Elites algorithm to heavily promote the use of sub-sets of game mechanics in Chapter 11, encapsulates many feasible mechanic combinations (see Table 13.1). The table also shows the percentage of scenes from the corpus that contains 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 [102] that our system uses to stitch pre-generated scenes into playable Mario levels that an agent plays by triggering a specific game mechanic sequence.

13.1.1 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.

13.1.2 Genetic Operators

The system uses mutation and crossover as operators. Two-point crossover allows 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. The generator uses five 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 number of mechanics using rank selection.
- **Split:** split a scene in half and replace it with a left and new right scene, randomly selecting half the mechanics to go in a new left scene and the rest to go in the right.
- **Merge:** add the mechanics of a scene to the left or 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 number of mechanics using rank selection.

The system selects a scene to apply one of the operators, with a higher likelihood to select scenes with higher numbers of mechanics.

13.1.3 Constraints and Fitness Calculation

FI2Pop 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.

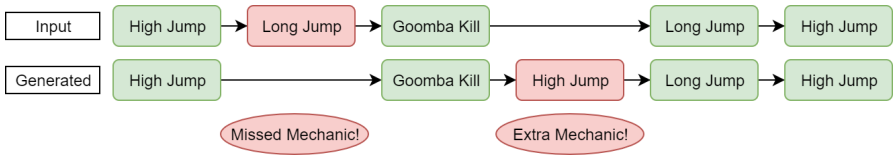


Figure 13.2: An example of missing and extra mechanic faults. The Long jump in the input playthrough is missing in the newly generated map's playthrough. The generated playthrough also contains an extra high jump not included in the input.

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

$$C = \begin{cases} \frac{1}{N}\sum_{i=1}^{N}\frac{d_i}{d_{level}} & \text{if } \frac{1}{N}\sum_{i=1}^{N} w_i < p \\ 1 & \text{if } \frac{1}{N}\sum_{i=1}^{N} w_i \geq p \end{cases} \tag{13.1}$$

where d_i is the distance traveled by the A-Star 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 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, which is the target mechanic playtrace, 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 13.2 displays an example of both. The system uses a sequence matching algorithm to calculate fault counts, as shown by Algorithm 2.

Algorithm 2 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:

$$\begin{aligned} P_{missed} &= W_{missed} \cdot M_{missed} \\ P_{extra} &= W_{extra} \cdot \tanh(b \cdot M_{extra}) \\ F &= S - (P_{missed} + P_{extra} \cdot (S - P_{missed})) \end{aligned} \tag{13.2}$$

Algorithm 2 Calculating fault count in a chromosome

```
GIVEN: generatedSeq, targetSeq
pTarget = 0; pGenerated = 0
extraMechs = 0; missedMechs = 0
for pTarget < len(targetSeq), pTarget = pTarget + 1 do
    targetMechanic = targetSeq[pTarget]
    genSubList = generatedSeq.subList(pGenerated, len(generatedSeq))
    mechanicIndex = genSubList.indexOf(targetMechanic)
    if mechanicIndex == -1 then
        missedMechs += 1
    else
        pGenerated += mechanicIndex + 1
        extraMechs += mechanicIndex
    end
    if pGenerated >= len(generatedSeq) then
        pTarget += 1
        break
    end
end
return extraMechs, missedMechs
```

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.

13.2 Experiments

To test the algorithms, we generate levels using three original Super Mario levels as targets (1-1, 4-2, 6-1 in Figures 13.3a, 13.3d, and 13.3g). The Robin Baumgarten A-Star algorithm, which was developed for the first Mario AI competition [160], is run once on each of the three levels. The resulting mechanic sequences are collected and used as targets.

Our system uses 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 and 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} equal to 1.0, b equal to 0.065, and S initially equal to 100. These values were picked based on some preliminary experiments that proves that they lead to

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 13.2: The frequency of each mechanic in the input playtrace.

better performance.

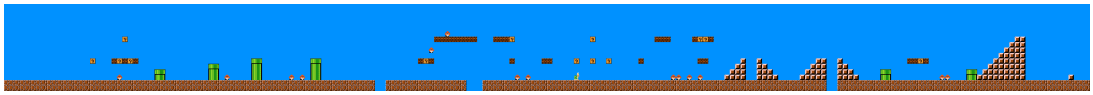
The scene corpus is taken from the results of *Intentional Computational Level Design* [96] which we review in Chapters 10 and 11. The corpus [2] contains a total of 1691 Mario scenes, with an average of 5.45 mechanics in a scene and a standard deviation of 1.74. The library is generated using the Constrained MAP-Elites generator created in the aforementioned project, with the dimensions corresponding to the mechanics shown in Table 13.1. The game playing agent in the MAP-Elites generator is the same agent we use: the winning A-Star agent from the 2013 Competition[162].

We compare the results from the 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 maximizes the number of matched mechanics based on each scenes labeled mechanics in the corpus. Each generator generates levels until it creates a total of 20 levels for each target level. Table 13.2 displays information about the mechanic makeup of the input playtraces. We can notice that all the playtraces have low frequency of killing enemies, hitting blocks, or collecting coins. This was not surprising as the used A-Star agent is designed to reach the furthest to the right in the least amount of time without caring about its score.

13.3 Results

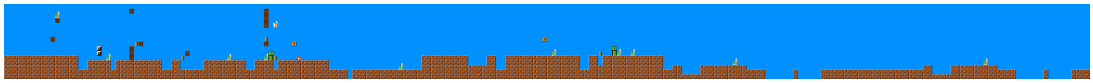
Figure 13.3 shows the original levels from Super Mario Bros (Nintendo, 1985) and its greedy and evolved counterparts. Both greedy and evolved levels have

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

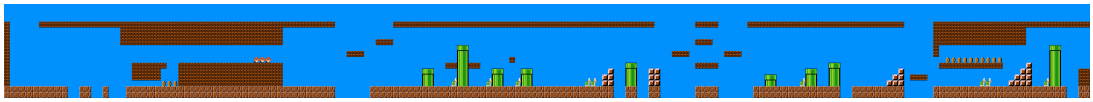


(a) Level 1-1 original

(b) Level 1-1 greedy



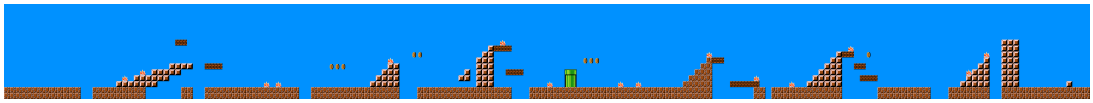
(c) Level 1-1 evolved



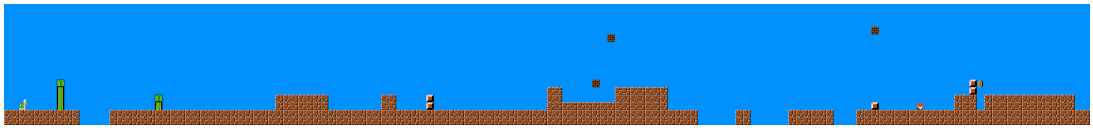
(d) Level 4-2 original

(e) Level 4-2 greedy

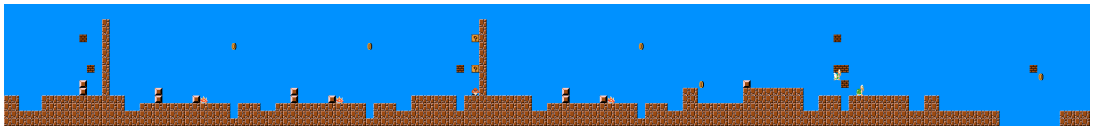
(f) Level 4-2 evolved



(g) Level 6-1 original



(h) Level 6-1 greedy



(i) Level 6-1 evolved

Figure 13.3: 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 13.3: Different level statistics calculated over 20 generated levels using different techniques and compared to the original levels as a reference point.

less graphical variance, a result of the lack of diversity in the scenes they stitched together. However, scene containing a 3-tile-high-pipe requires the same jump that 3 breakable brick tiles stacked on top of each other. The fitness function used to evolve these scenes [96] most likely negatively impacts level diversity, as it aims to create simple and uniform spaces. Most of the generated levels (greedy or evolution) seem flatter than their original counterparts. The fitness function used for evolving the scenes [96] implies a pressure for lower tile variance, and therefore impacts levels created with the scenes in a similar way.

13.3.1 Level Playability

To calculate playability for each group of levels, we run Robin Baumgarten's A-Star agent [160] 20 times per level and average the results over the whole group of levels. Table 13.3 shows the playability percentages of each of these groups. We find it notable that the A-Star 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-Star is not a perfect algorithm and not able to beat every level every time. Close to 100% of the evolved 1-1, 4-2, and 6-1 levels are completely playable. 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%).

13.3.2 Mechanic Similarity

Figure 13.4 displays the mechanic evaluation across all three target levels for all generators. "Matches" are the number of matched mechanics while "Extras" are the number of extra mechanics between the input sequence and generated agent

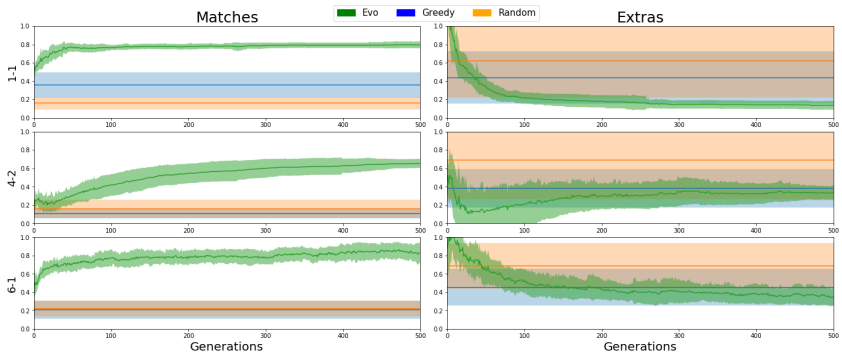


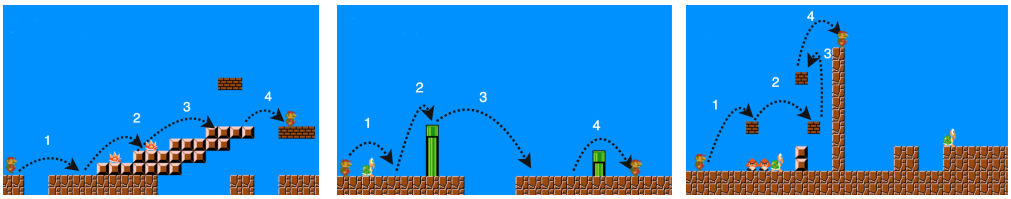
Figure 13.4: Tracking mechanic statistics throughout generations across all three target levels for all generators.

playtrace. "Matches" and "Extras" are normalized using the *total* value from Table 13.1 for each level. As fitness is impacted by matching mechanics it makes sense that the evolutionary generated levels go up over time, just as fitness does. Across all three levels, the evolution 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 evolutionary 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.

13.3.3 Structural Diversity

We use Tile Pattern KL-Divergence (TPKLDiv) calculations [112] to measure the structural similarity between the 20 generated levels which 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 are to 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 to the original level. We compute this value by calculating the TPKLDiv between each generated level and the original level and then we compute the average over all these values.

Table 13.3 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 random generated levels have a lower TPKLDiv value which indicates that the evolved levels have less diversity overall. Also, the evolutionary levels generated for World 1-1 have low diversity relative to the others. In the



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

Figure 13.5: An example of a 4 jump sequence in the original 6-1 level, and how the two generators try to copy it.

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 a 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 has a higher diversity than all the evolution levels, but at the same time they less likely to be playable.

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 13.3, 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).

13.4 Discussion

Looking to results as shown by Figure 13.4, it is clear the FI-2Pop 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 presume the reason behind this influx in scene length is due to 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 occurred in the given length and thus favored levels with more scenes. Spreading the mechanics 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 13.5 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. [96] was built using a fitness function that encouraged simplicity, creating a minimalized pressure that is reflected on the levels built using the corpus. In Figure 13.5, 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 wall 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 13.3.3 and Table 13.3, 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 population of mechanics also makes it difficult to curate diversity (World 4-2). It is 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.

13.5 Summary

In this chapter, we explore a means to automatically generate sequential levels for tutorials by stitching pre-generated scenes in Super Mario. We compare the mechanical sequence on playthroughs of the Baumgarten A-Star agent using three unique levels from the original Super Mario (1-1, 4-2 and 6-1) to the mechanical sequences from playthroughs on generated levels to judge the success of the experiments. We use the FI2Pop evolution algorithm, with the focus of developing winnable levels that are mechanically analogous to the original level, and we find that it is able to match the sequence much better than either baseline. Although both baselines and the new method have lower diversity among themselves than the original Mario levels, this is most likely a result of the entropy pressure during the scene generation process.

We used mechanic-dependent scene generation to curate the library used in this project. However, these scenes could come from anywhere. They could be human-evaluated, like in the generator built by Reis et al [134], or they can be generated using personas like in Chapter 12. Some games may not even require "stitching", such as in MiniDungeons 2 where each level is its own encapsulated experience. In these types of games, the order that levels are presented to the player will be more critical to their engagement and learning experience.

Chapter 14

Conclusions

After the equivalent of 10 research papers, we are almost at the end. This final chapter is an attempt to wrap it all up into a short summary. We also present a few ideas about how to further the research presented here, ask some challenging questions, and propose some stretch goals.

At the beginning of this thesis, we started with a simple claim: *we can automatically generate tutorials and tutorial levels for video games using mechanics and play personas.* In the following sections, we summarize how we prove the validity of this claim with mechanic and player analysis. Through the use of these analytical methods, we can generate multiple types of tutorials for a variety of video games. These contributions are not without their limits, however, and we describe future projects which can expand upon this work in multiple directions. Figure 14.1 displays the project flow of this thesis as a reminder of how each project enables us to advance from game mechanics and playtraces to automatically generated tutorials.

14.1 Game Mechanics and Players

In the second part of this thesis, we review methods to analyze game mechanics and players. In Chapter 6, we introduce a family of "critical mechanic discovery" methods to automatically curate mechanic content for tutorials. Critical mechanics are the minimum set of game mechanics that the player **must** trigger in order to win a game or level. The discovery methods require the use of a directed "mechanic graph," which encapsulates mechanic relationships with game entities. Critical mechanic discovery methods can search the mechanic graph using uniformed search methods like breadth-first search or using informed search methods like best-first search by giving a cost to the mechanic nodes. We compare these two methods in an experiment and find that an informed search often finds a more complete

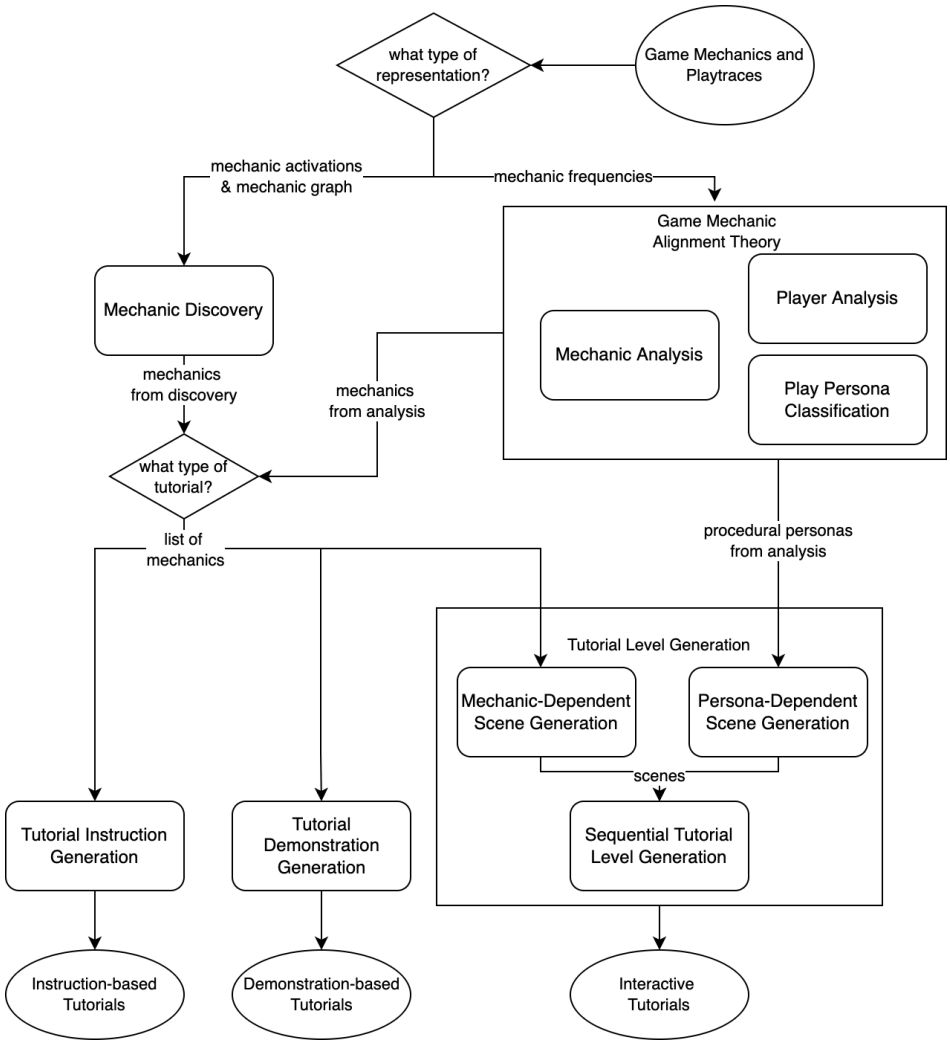


Figure 14.1: A map of how every project in this thesis fit together. First presented in Chapter 1.

set of game mechanics which help AI agents play better, although neither method always agrees with what humans believe are important mechanics. By the end of this chapter, we begin to wonder how we can build tutorials using this content as well as how we might find more personalized content for tutorials.

In Chapter 7, we introduce a theory of Game Mechanic Alignment (GMA), a framework in which mechanics can be analyzed in terms of their systemic rewards and agential motivations. We apply GMA to several well-known games (Super Mario, Minecraft, and Bioshock) to exemplify its usefulness. We also apply a form of GMA estimation to playtraces for Minidungeons 2, demonstrating its ability to articulate mechanical preferences and to enable developers to prioritize mechanic education based on playstyle. GMA is a powerful tool that developers can use to find personalized content for tutorials and analyze their player base. This theory does not define a "right" way to interpret results. For example, it will be up to the developer to decide if a player intended to use a certain mechanic more or less than other players or if it was an accident. However, it can help overcome some of the challenges identified for critical mechanic discovery methods, since it does not require domain knowledge of mechanic relationships to build mechanic graphs.

In Chapter 8, we present a way to rapidly classify players by their play personas using their mechanic behavior. Building off of Chapter 7, we can utilize information gleaned by mechanic preference to understand more about players. The more we know about a player, the more personalized a tutorial for them can be. Action Agreement Ratio is a method that uses procedural persona action matching to label playtrace similarity to known personas. But AAR can be slow (taking over 7 hours in our experiment). In contrast, an SVM using a mechanic frequency vector representation takes only seconds to train and infer and agrees with AAR 70% of the time. We also train an LSTM which uses a game state representation and obtains 72% accuracy, demonstrating that mechanic frequency vectors are a good data augmentation method. Overall, this representation allows developers to compress the data needed to identify player personas from a long state-trace to a small multidimensional vector *and* allows personas to be identified quickly.

14.2 Tutorial Generation

In Chapter 9, we present a set of experiments which use mechanics found using critical mechanic discovery (Chapter 6) to build instruction- and demonstration-based tutorials explaining how to win, lose, and gain points. In Chapter 10, we introduce two single-objective optimization methods to build small levels or "scenes" which spotlight the use of specific game mechanics. Chapter 11 extends this research by exploring the use of quality diversity algorithms to generate

mechanic-dependent scenes. In either chapter, the input mechanics can come from anywhere. Perhaps a mechanic discovery method is used to identify a list of critical mechanics. Maybe GMA is used to calculate the relevance of each mechanic to winning and losing. Or maybe GMA is used on groups of players to identify their mechanic preferences using a ranking method similar to the one in Chapter 7. In any case, the resulting list of mechanics can be used to generate instructions, demonstrations, or interactive tutorials. In Chapter 12, we investigate a technique to build small tutorial levels which encourage or discourage the use of various play persona strategies. After a user's or set of users' play personas have been identified, those personas can be used to generate scenes. The technique presented in Chapter 13 allows us to stitch together scenes from either Chapters 10, 11, or 12 into a larger tutorial level.

14.3 What's Next?

Depending on what information you have available to you and what representation you wish to use, you can build tutorials directly from mechanics using mechanic discovery or from mechanics using Game Mechanic Alignment, with the latter allowing for a more personalized result if personas are used. These two representations and their mechanic curations come with their own sets of strengths and weaknesses. Mechanic discovery is fast and seemingly accurate for 2D arcade games like those in the GVGAI framework. However, it requires insider knowledge of mechanic relationships, which are relatively simple for these types of games compared to more complex games (i.e. Minecraft, Skyrim, or No Man's Sky). Mechanic graphs may not be readily available for games built commercially and/or building them might be expensive and difficult. Game Mechanic Alignment Theory and its resulting set of mechanic and player analysis methods do not require this kind of knowledge, but it may mean making some assumptions about your players' intentions. How can we negate some of these challenges? Can we build mechanic graphs by simply watching a playthrough? Is it possible to measure player intent or gauge individual skill from past behavior?

We've demonstrated the capability to build instruction and demonstration tutorials for arcade games, but we have yet to apply our methods to more complex ones. One path going forward is to try mechanic discovery and/or GMA on games like Bioshock or Minecraft. These games do not provide logs of mechanic activation, so an overlaying system able to track mechanic usage may be necessary. Such a system could be integrated at the video-level, meaning that it could be able to single out mechanics as they appear on screen using computer vision. Methods for automatically identifying maps, mechanics and other characteristics of games given

only an executable version of the game show promise for this kind of project [118].

Persona-dependent scene generation requires the use of procedural personas, which in our experiments came with the Minidungeons 2 framework (Chapter 12). However, what if there are no personas? Or what if there is not a definition of what the personas are, such as if you clustered similarly behaviors together? In situations such as these, we would like to explore persona evolution: searching for personas using MAP-Elites neuroevolution with game mechanic usage as behavioral characteristics.

Three distinct types of tutorials have been identified and scoped for in this thesis. These three types are present in a wide-variety of games/genres but are not intended to be an all-inclusive list. There could be less-popular additional types which could be good candidates for generation. This is tangentially related to difficulty, which is another topic not studied in this thesis. Tutorials can increase in difficulty to help the player gain better skill mastery. Another future project would be to incorporate increasing difficulty into tutorial level generation using agents with varying planning horizons to mimic different player skill levels [86, 99]. In several projects, entropy pressure is used to drive evolution toward simple levels. Therefore, we would like to propose future work in which we experiment with a "targeted entropy" instead of a minimizing one. This may allow us to evolve different level simplicities with ranging difficulty.

These chapters define self-contained projects which demonstrate small parts of a tutorial generative system. The AtDelfi system is the first attempt at an end-to-end tutorial generation system, where game mechanics are served as input and instruction/demonstration tutorials are built as output. The next step would be to expand upon AtDelfi to do GMA analysis and interactive tutorial generation in real-time, adapting itself to the needs of the player while they play, feeding them text, examples, and levels to keep them engaged and teach them new skills. This level of personalized content generation has yet to be realized for any type of content or game.

What about domains outside of games? Can the theories explored in this thesis be applied to teach science, mathematics, or even language? Students are people, and people are unique. They learn at different paces and gravitate toward the topics that they enjoy learning. I dream of a world in which students can receive a truly personalized education, with a curriculum that engages them and with content that they find exciting, that can adapt to their strengths and weaknesses as they grow. I hope that ideas in this thesis may serve as a part of the foundation for personalized learning.

I believe that fully automatic tutorial generation is possible. I've attempted to prove that here, but there is still a lot of work to be done. I, for one, look forward

to it.

Bibliography

[1] Ryan Alexander and Chris Martens. 2017. Deriving quests from open world mechanics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 1–7.

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

[3] Erik Andersen, Eleanor O'Rourke, Yun-En Liu, Rich Snider, Jeff Lowdermilk, David Truong, Seth Cooper, and Zoran Popovic. 2012. The impact of tutorials on games of varying complexity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 59–68.

[4] Damien Anderson, Matthew Stephenson, Julian Togelius, Christoph Salge, John Levine, and Jochen Renz. 2018. Deceptive games. In *International Conference on the Applications of Evolutionary Computation*. Springer, 376–391.

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

[6] Daniel Ashlock. 2010. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. 289–296. `https://doi.org/10.1109/ITW.2010.5593341`

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

[8] Batu Aytemiz, Isaac Karth, Jesse Harder, Adam Smith, and Jim Whitehead. 2018. Talin: A Framework for Dynamic Tutorials Based on the Skill Atoms Theory. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 14.

[9] Gabriella AB Barros, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2018. Data-driven Design: A Case for Maximalist Game Design Paper type: Position paper. In *9th International Conference on Computational Creativity, ICCC 2018*. Association for Computational Creativity (ACC), 169–176.

[10] Gabriella Alves Bulhoes Barros, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2018. Who killed Albert Einstein? From open data to murder mystery games. *IEEE Transactions on Games* 11, 1 (2018), 79–89.

[11] Debosmita Bhaumik, Ahmed Khalifa, Michael Green, and Julian Togelius. 2020. Tree search versus optimization approaches for map generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. 24–30.

[12] Debosmita Bhaumik, Ahmed Khalifa, and Julian Togelius. 2021. Lode Encoder: AI-constrained co-creativity. In *2021 IEEE Conference on Games (CoG)*. IEEE, 01–08.

[13] Philip Bontrager, Ahmed Khalifa, Andre Mendes, and Julian Togelius. 2016. Matching games and algorithms for general video game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 12.

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

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

[16] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on CI and AI in Games* 4, 1 (2012), 1–43.

[17] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[18] Edward Byrne. 2005. *Game level design*. Vol. 6. Charles River Media Boston.

[19] Charles B Callaway and James C Lester. 2002. Narrative prose generation. *Artificial Intelligence* 139, 2 (2002), 213–252.

[20] Megan Charity, Michael Cerny Green, Ahmed Khalifa, and Julian Togelius. 2020. Mech-Elites: Illuminating the Mechanic Space of GVG-AI. In *International Conference on the Foundations of Digital Games*. 1–10.

[21] Megan Charity, Ahmed Khalifa, and Julian Togelius. 2020. Baba is y'all: Collaborative mixed-initiative level design. In *2020 IEEE Conference on Games (CoG)*. IEEE, 542–549.

[22] John William Charnley, Alison Pease, and Simon Colton. 2012. On the Notion of Framing in Computational Creativity.. In *ICCC*. 77–81.

[23] Zhengxing Chen, Christopher Amato, Truong-Huy D Nguyen, Seth Cooper, Yizhou Sun, and Magy Seif El-Nasr. 2018. Q-deckrec: A fast deck recommendation system for collectible card games. In *2018 IEEE conference on Computational Intelligence and Games (CIG)*. IEEE, 1–8.

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

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

[26] Kate Compton, Johnathan Pagnutti, and Jim Whitehead. 2017. A shared language for creative communities of artbots. In *Proceedings of the 2017 Co-Creation Workshop. Eighth International Conference on Computational Creativity, Atlanta, Georgia, USA*.

[27] Dan Cook. 2006. *What are game mechanics?* `http://lunar.lostgarden.com/labels/skill%20chains.html`

[28] Dan Cook. 2007. *The chemistry of game design.* `http://www.lostgarden.com/2006/10/what-are-game-mechanics.html`

[29] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *European Conference on the Applications of Evolutionary Computation*. Springer, 284–293.

[30] Michael Cook and Azalea Raad. 2019. Hyperstate space graphs for automated game analysis. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–8.

[31] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[32] Rémi Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*. Springer, 72–83.

[33] Extra Credits. 2014. Design Club - Super Mario Bros: Level 1-1 - How Super Mario Mastered Level Design. `https://www.youtube.com/watch?v=ZH2wGpEZVgE`.

[34] Steve Dahlskog and Julian Togelius. 2012. Patterns and procedural content generation: revisiting Mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games*. ACM, 1.

[35] Steve Dahlskog and Julian Togelius. 2013. Patterns as objectives for level generation. (2013).

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

[37] Isaac M Dart, Gabriele De Rossi, and Julian Togelius. 2011. SpeedRock: procedural rocks through grammars and evolution. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, 8.

[38] Fernando de Mesentier Silva, Aaron Isaksen, Julian Togelius, and Andy Nealen. 2016. Generating heuristics for novice players. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*. IEEE, 1–8.

[39] Fernando de Mesentier Silva, Julian Togelius, Frank Lantz, and Andy Nealen. 2018. Generating Beginner Heuristics for Simple Texas Hold'em. In *Genetic and Evolutionary Computation Conference*. ACM.

[40] Fernando de Mesentier Silva, Julian Togelius, Frank Lantz, and Andy Nealen. 2018. Generating Novice Heuristics for Post-Flop Poker. In *Computational Intelligence and Games*. IEEE.

[41] Omar Delarosa, Hang Dong, Mindy Ruan, Ahmed Khalifa, and Julian Togelius. 2021. Mixed-initiative level design with rl brush. In *International*

Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar). Springer, 412–426.

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

[43] Joris Dormans. 2011. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, 2.

[44] Joris Dormans and Sander Bakkes. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 216–228.

[45] Sam Earle, Maria Edwards, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. 2021. Learning controllable content generators. In *2021 IEEE Conference on Games (CoG)*. IEEE, 1–9.

[46] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. 2013. Towards a video game description language. In *Dagstuhl Follow-Ups*, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[47] Egoraptor. 2011. Sequelitis - Mega Man Classic vs. Mega Man X. `https://www.youtube.com/watch?v=8FpigqfcvlM`.

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

[49] George Skaff Elias, Richard Garfield, and K Robert Gutschera. 2012. *Characteristics of games*. MIT Press.

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

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

[52] Raluca D Gaina, Diego Pérez-Liébana, and Simon M Lucas. 2016. General video game for 2 players: framework and competition. In *Computer Science and Electronic Engineering (CEEC), 2016 8th*. IEEE, 186–191.

[53] Euro Gamer. 2015. Miyamoto on World 1-1: How Nintendo made Mario's most iconic level. https://www.youtube.com/watch?v=zRGRJRUWafY. Last Accessed: February 1, 2021.

[54] GameSpot. 2011. *How to Beat Mr. Freeze Boss Fight - Batman: Arkham City - Gameplay Movie (PS3).* https://www.youtube.com/watch?v=hocJDIJ0DqQ

[55] James Paul Gee. 2003. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)* 1, 1 (2003), 20–20.

[56] James Paul Gee. 2006. Are video games good for learning? *Nordic Journal of Digital Literacy* 1, 03 (2006), 172–183.

[57] Tomas Geffner and Hector Geffner. 2015. Width-based planning for general video-game playing. *Proc. AIIDE* (2015), 23–29.

[58] Michael Cerny Green, Gabriella AB Barros, Antonios Liapis, and Julian Togelius. 2018. DATA agent. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–10.

[59] Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. 2019. Two-step constructive approaches for dungeon generation. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 1–7.

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

[61] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, and Julian Togelius. 2019. Automatic Critical Mechanic Discovery using Playtraces in Video Games. *arXiv preprint arXiv:1909.03094* (2019).

[62] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, and Julian Togelius. 2020. Automatic Critical Mechanic Discovery Using Playtraces in Video Games. In *International Conference on the Foundations of Digital Games*. 1–9.

[63] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Andy Nealen, and Julian Togelius. 2018. Generating levels that teach mechanics. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–8.

[64] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, and Julian Togellius. 2017. " Press Space to Fire": Automatic Video Game Tutorial Generation. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[65] Michael Cerny Green, Ahmed Khalifa, Philip Bontrager, Rodrigo Canaan, and Julian Togelius. 2021. Game Mechanic Alignment Theory and Discovery. *International Conference on the Foundations of Digital Games* (2021).

[66] Michael Cerny Green, Ahmed Khalifa, M Charity, Debosmita Bhaumik, and Julian Togelius. 2022. Predicting Personas Using Mechanic Frequencies and Game State Traces. *arXiv e-prints* (2022), arXiv–2203.

[67] Michael Cerny Green, Ahmed Khalifa, M Charity, and Julian Togelius. 2022. Persona-driven Dominant/Submissive Map (PDSM) Generation for Tutorials. *arXiv e-prints* (2022), arXiv–2204.

[68] Michael Cerny Green, Luvneesh Mugrai, Ahmed Khalifa, and Julian Togelius. 2020. Mario level generation from mechanics using scene stitching. In *2020 IEEE Conference on Games (CoG)*. IEEE, 49–56.

[69] Michael Cerny Green, Christoph Salge, and Julian Togelius. 2019. Organic building generation in minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 1–7.

[70] Michael Cerny Green, Benjamin Sergent, Pushyami Shandilya, and Vibhor Kumar. 2019. Evolutionarily-curated curriculum learning for deep reinforcement learning agents. *Deep Reinforcement Learning in Games Workshop at AAAI* (2019).

[71] Jason Grinblat. 2016. Markov by candlelight. *https://www.youtube.com/watch?v=3AjlsTtrfVY* (2016).

[72] Cristina Guerrero-Romero and Diego Perez-Liebana. 2021. MAP-Elites to Generate a Team of Agents that Elicits Diverse Automated Gameplay. In *2021 IEEE Conference on Games (CoG)*. IEEE, 1–8.

[73] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark O Riedl. 2019. 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*. 1–13.

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

[75] Matthew Guzdial and Mark Riedl. 2016. Toward game level generation from gameplay videos. *arXiv preprint arXiv:1602.07721* (2016).

[76] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9, 1 (2013), 1–22.

[77] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2018. Automated playtesting with procedural personas through MCTS with evolved heuristics. *IEEE Transactions on Games* 11, 4 (2018), 352–362.

[78] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. 2014. Evolving personas for player decision modeling. In *Conference on Computational Intelligence and Games*. IEEE.

[79] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2014. Generative Agents for Player Decision Modeling in Games. In *Proceedings of the Foundations of Digital Games Conference*.

[80] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. 2014. Personas versus clones for player decision modeling. In *International Conference on Entertainment Computing*.

[81] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2015. Evolving Models of Player Decision Making: Personas versus Clones. *Entertainment Computing* (2015).

[82] Christoffer Holmgård, Julian Togelius, Antonios Liapis, and Georgios N Yannakakis. 2015. MiniDungeons 2: An experimental game for capturing and modeling player decisions. In *Foundations of Digital Games*.

[83] Tobias Huber, Silvan Mertes, Stanislava Rangelova, Simon Flutura, and Elisabeth André. 2021. Dynamic Difficulty Adjustment in Virtual Reality Exergames through Experience-driven Procedural Content Generation. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1–8.

[84] Robin Hunicke, Marc LeBlanc, and Robert Zubek. 2004. MDA: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, Vol. 4. San Jose, CA, 1722.

[85] Aaron Isaksen, Dan Gopstein, Julian Togelius, and Andy Nealen. 2017. Exploring Game Space of Minimal Action Games via Parameter Tuning and Survival Analysis. *IEEE Transactions on Computational Intelligence and AI in Games* (2017).

[86] Aaron Isaksen, Drew Wallace, Adam Finkelstein, and Andy Nealen. 2017. Simulating strategy and dexterity for puzzle games. In *Computational Intelligence and Games*. IEEE.

[87] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. 2016. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, Vol. 9.

[88] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. 2010. Polymorph: dynamic difficulty adjustment through level generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 1–4.

[89] Tobias Joppen, Miriam Moneke, Nils Schroder, Christian Wirth, and Johannes Furnkranz. 2017. Informed Hybrid Game Tree Search for General Video Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games* (2017).

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

[91] Isaac Karth. 1984. Elite (1984). (1984).

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

[93] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. 95–101.

[94] Ahmed Khalifa, Fernando de Mesentier Silva, and Julian Togelius. 2019. Level Design Patterns in 2D Games. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–8.

[95] Ahmed Khalifa and Magda Fayek. 2015. Automatic puzzle level generation: A general approach using a description language. Institute of Electrical and Electronics Engineers.

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

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

[98] Ahmed Khalifa, Aaron Isaksen, Julian Togelius, and Andy Nealen. 2016. Modifying MCTS for Human-Like General Video Game Playing.. In *IJCAI*. 2514–2520.

[99] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet hell generation through constrained map-elites. In *Proceedings of The Genetic and Evolutionary Computation Conference*. 1047–1054.

[100] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. 2016. General video game level generation. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, 253–259.

[101] Ahmed Khalifa and Julian Togelius. 2020. Multi-Objective level generator generation with Marahel. In *International Conference on the Foundations of Digital Games*. 1–8.

[102] Steven Kimbrough, Gary Koehler, Ming Lu, and David Wood. 2008. Introducing a Feasible-Infeasible Two-Population (FI-2Pop) Genetic Algorithm for Constrained Optimization: Distance Tracing and No Free Lunch. *European Journal of Operational Research* 190 (10 2008).

[103] Steven Orla Kimbrough, Gary J Koehler, Ming Lu, and David Harlan Wood. 2008. 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 (2008).

[104] Raymond Lang. 1999. A declarative model for simple narratives. In *Proceedings of the AAAI fall symposium on narrative intelligence*. 134–141.

[105] Frank Lantz, Aaron Isaksen, Alexander Jaffe, Andy Nealen, and Julian Togelius. 2017. Depth in strategic games. *IEEE Transactions on Computational Intelligence and AI in Games*.

[106] Wei Li, Yuanlin Zhang, and George Fitzmaurice. 2013. TutorialPlan: automated tutorial generation from CAD drawings. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[107] Antonios Liapis, Héctor P Martínez, Julian Togelius, and Georgios N Yannakakis. 2013. Adaptive game level creation through rank-based interactive evolution. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE, 1–8.

[108] Antonios Liapis, Gillian Smith, and Noor Shaker. 2016. Mixed-initiative content creation. In *Procedural content generation in games*. Springer, 195–214.

[109] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. 2011. Optimizing visual properties of game content through neuroevolution. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[110] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. 2013. Sentient sketchbook: computer-assisted game level authoring. (2013).

[111] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

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

[113] Tiago Machado, Andy Nealen, and Julian Togelius. 2017. SeekWhence a Retrospective Analysis Tool for General Game Design. In *Proceedings of the 12th International Conference on the Foundations of Digital Games* (Hyannis, Massachusetts) *(FDG '17)*. ACM, Article 4, 6 pages. `https://doi.org/10.1145/3102071.3102090`

[114] Athar Mahmoudi-Nejad, Matthew Guzdial, and Pierre Boulanger. 2021. Arachnophobia exposure therapy using experience-driven procedural content generation via reinforcement learning (EDPCGRL). In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 17. 164–171.

[115] Hiroaki Mikami. 2014. *AUTOMATIC GENERATION OF TUTORIAL FROM UNIT TESTS*. Ph. D. Dissertation. University of Tokyo.

[116] Roxana Moreno and Richard Mayer. 2007. Interactive multimodal learning environments. *Educational psychology review* 19, 3 (2007), 309–326.

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

[118] Joseph Osborn, Adam Summerville, and Michael Mateas. 2017. Automatic mapping of nes games with mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 1–9.

[119] Joseph C Osborn, Adam Summerville, Nathan Dailey, and Soksamnang Lim. 2021. MappyLand: fast, accurate mapping for console games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 17. 66–73.

[120] Johnathan Pagnutti, Kate Compton, and Jim Whitehead. 2016. 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*.

[121] Nathan Partlan, Erica Kleinman, Jim Howe, Sabbir Ahmad, Stacy Marsella, and Magy Seif El-Nasr. 2021. Design-Driven Requirements for Computationally Co-Creative Game AI Design Tools. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*. 1–12.

[122] Lyn Pemberton. 1989. A modular approach to story generation. In *Proceedings of the fourth conference on European chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 217–224.

[123] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. 2018. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv preprint arXiv:1802.10363* (2018).

[124] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. 2019. General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms. *Transactions on Games* (2019).

[125] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. 2016. General video game ai: Competition, challenges and opportunities. In *AAAI Conference on Artificial Intelligence*.

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

[127] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. 2016. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 3 (2016), 229–243.

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

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

[130] Jan L Plass, Bruce D Homer, and Charles K Kinzer. 2015. Foundations of game-based learning. *Educational Psychologist* 50, 4 (2015), 258–283.

[131] Edward Jack Powley, Mark J Nelson, Swen E Gaudl, Simon Colton, Blanca Pérez Ferrer, Rob Saunders, Peter Ivey, and Michael Cook. 2017. Wevva: Democratising game design. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

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

[133] Sheri Graner Ray. 2010. Tutorials: learning to play. `http://www.gamasutra.com/view/feature/134531/tutorials_learning_to_play.php?print=1`.

[134] Willian MP Reis, Levi HS Lelis, et al. 2015. Human computation for procedural content generation in platform games. In *CIG*. IEEE.

[135] Ruben Rodriguez-Torrado, Pablo Ruiz, Luis Cueto-Felgueroso, Michael Cerny Green, Tyler Friesen, Sebastien Matringe, and Julian Togelius. 2022. Physics-informed attention-based neural network for solving non-linear partial differential equations. *Scientific Reports* (2022).

[136] David E Rumelhart. 1975. Notes on a schema for stories. *Representation and understanding: Studies in cognitive science* 211, 236 (1975), 45.

[137] Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. 2020. The AI settlement generation challenge in minecraft. *KI-Künstliche Intelligenz* 34, 1 (2020), 19–31.

[138] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. 2018. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–10.

[139] Christoph Salge, Christian Guckelsberger, Michael Cerny Green, Rodrigo Canaan, and Julian Togelius. 2019. Generative design in Minecraft: Chronicle challenge. *International Conference on Computational Creativity* (2019).

[140] Kumara Sastry, David Goldberg, and Graham Kendall. 2005. *Genetic Algorithms*. Springer US, 97–125. https://doi.org/10.1007/0-387-28356-0_4

[141] Fabian Schrodt, Yves Röhm, and Martin V Butz. 2017. An Event-Schematic, Cooperative, Cognitive Architecture Plays Super Mario. *Cognitive Robot Architectures* (2017), 10.

[142] Noor Shaker, Mohammad Shaker, and Julian Togelius. 2013. Evolving playable content for cut the rope through a simulation-based approach. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

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

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

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

[146] Tianye Shu, Jialin Liu, and Georgios N Yannakakis. 2021. Experience-driven PCG via reinforcement learning: A Super Mario Bros study. In *2021 IEEE Conference on Games (CoG)*. IEEE, 1–9.

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

[148] Alex Smith. 2006. *Strategy headroom in roguelikes.* http://nethack4.org/blog/strategy-headroom.html?fbclid=IwAR1qUtRc3aK_K900zJ4WmYKdUEjhyRQSqsWs8EHhDZ7HVkX5rPDJD67w5k0

[149] Gillian Smith, Jim Whitehead, and Michael Mateas. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. 209–216.

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

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

[152] Paul Suddaby. 2012. *The Many Ways to Show the Player How It's Done With In-Game Tutorials.* https://gamedevelopment.tutsplus.com/tutorials/the-many-ways-to-show-the-player\-how-its-done-with-in-game-tutorials--gamedev-400

[153] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark O Riedl. 2016. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *Twelfth artificial intelligence and interactive digital entertainment conference*.

[154] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph Carter Osborn, Noah Wardrip-Fruin, and Arnav Jhala. 2017. From Mechanics to Meaning. *IEEE Transactions on Computational Intelligence and AI in Games*.

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

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

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

[158] Ryan Taljonick. 2014. *Shadow of Mordor's Nemesis system is amazing–here's how it works.* https://www.gamesradar.com/shadow-mordor-nemesis-system-amazing-how-works/

[159] Carl Therrien. 2011. " To Get Help, Please Press X" The Rise of the Assistance Paradigm in Video Game Design.. In *DiGRA Conference*.

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

[161] Julian Togelius, Noor Shaker, and Joris Dormans. 2016. Grammars and L-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*. Springer, 73–98.

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

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

[164] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2020. Bootstrapping conditional gans for video game level generation. In *2020 IEEE Conference on Games (CoG)*. IEEE, 41–48.

[165] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. 2012. Game-O-Matic: Generating Videogames that Represent Ideas.. In *PCG@ FDG*. 11–1.

[166] Mike Treanor, Michael Mateas, and Noah Wardrip-Fruin. 2010. Kaboom! is a Many-Splendored Thing: An interpretation and design methodology for message-driven games using graphical logics. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. ACM, 224–231.

[167] Mike Treanor, Alexander Zook, Mirjam P Eladhari, Julian Togelius, Gillian Smith, Michael Cook, Tommy Thompson, Brian Magerko, John Levine, and Adam Smith. 2015. AI-based game design patterns. (2015).

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

[169] Tony Veale and Mike Cook. 2018. *Twitterbots: Making machines that make meaning*. MIT Press.

[170] Ngo Anh Vien and Marc Toussaint. 2015. Hierarchical monte-carlo planning. In *AAAI Conference on Artificial Intelligence*.

[171] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the genetic and evolutionary computation conference*. 221–228.

[172] Vivek R Warriar, Carmen Ugarte, John R Woodward, and Laurissa Tokarchuk. 2019. Playmapper: Illuminating design spaces of platform games. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–4.

[173] Leonid Wasserstein. 1969. Markov processes with countable state space describing large systems of automata. *IEEE Transactions on Image processing* 5, 3 (1969). in Russian.

[174] G. Christopher Williams. 2009. the pedagogy of the game tutorial. `http://www.popmatters.com/post/111485-active-learning-the-pedagogy-of-the-game-tutorial/`.

[175] Georgios N Yannakakis and Julian Togelius. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2, 3 (2011), 147–161.

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

[177] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. 2020. Rotation, translation, and cropping for zero-shot generalization. In *2020 IEEE Conference on Games (CoG)*. IEEE, 57–64.

[178] Yahia Zakaria, Mayada Hadhoud, and Magda Fayek. 2021. Procedural Level Generation for Sokoban via Deep Learning: An Experimental Study. (2021).

[179] Yulun Zhang, Matthew C Fontaine, Amy K Hoover, and Stefanos Nikolaidis. 2021. Deep Surrogate Assisted MAP-Elites for Automated Hearthstone Deckbuilding. *arXiv e-prints* (2021), arXiv–2112.