Influencing Game Event Sequences in Procedurally Generated Levels for Roguelike Games

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Masters of Computer
Science in the College of Engineering
at the University of Kentucky

By
Michael P. Probst
Lexington, Kentucky

Director: Dr. Brent Harrison, Professor of Computer Science
Lexington, Kentucky
2021

Copyright© Michael P. Probst 2021

ABSTRACT OF THESIS

Influencing Game Event Sequences in Procedurally Generated Levels for Roguelike Games

Procedural content generation in video games is a means for rapid development and increasing the game's replayability. Often, using machine learning to procedurally generate levels caters to a specific player's experience, and in doing so relinquishes the level designer's control over the events that occur in a generated level. We present a system for which procedurally generated levels are influenced by a predetermined sequence of events to be experienced by a player. We use a genetic algorithm which evaluates levels based on the intended and agent-experienced game events for a 2D, Roguelike game. When comparing the average fitness of our generated levels to the average fitness of a set of 1,000 randomly generated levels, it is shown that our system generates levels that are more likely to guide a player to experience a specific series of game events, defined by a level designer, than levels which are randomly generated.

KEYWORDS: Procedural content generation, genetic algorithms, video game design, artificial intelligence, authorial control

Michael P. Probst

December 7, 2021

Influencing Game Event Sequences in Procedurally Generated Levels for Roguelike Games

By
Michael P. Probst

Dr. Brent Harrison
Director of Thesis

Dr. Simone Silvestri
Director of Graduate Studies

December 7, 2021
Date

Dedicated to my grandfather, Dr. Gerhard F. "Bapo" Probst, who inspired me to be passionate about learning.

ACKNOWLEDGMENTS

Thank you to my advisor, Dr. Brent Harrison, who inspired me to attend graduate school in the first place, and has continued to support and encourage all my works.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vi

List of Figures vii

Chapter 1 Introduction 1

Chapter 2 Background 4
- 2.1 Genetic Algorithms 4
  - 2.1.1 Overview 4
  - 2.1.2 Applications in PCG 5

Chapter 3 Related Work 7
- 3.1 Procedural Content Generation 7
  - 3.1.1 Player Modeling 7
  - 3.1.2 Design Patterns 7
  - 3.1.3 Generative Grammars 8
  - 3.1.4 Mixed-Initiative Tools 9
- 3.2 Experience Management 10
  - 3.2.1 Metaphysical Experience Management 11
  - 3.2.2 Physical Experience Management 12
- 3.3 Problem Statement 12

Chapter 4 Methods 14
- 4.1 Fitness Function 14
  - 4.1.1 Agent Evaluation 14
  - 4.1.2 Sequence Partial Orderings 15
- 4.2 Crossover 16
- 4.3 Mutation 18
  - 4.3.1 Mutation Rate Methods 19

Chapter 5 Experiments 20
- 5.1 Test Environment 20
  - 5.1.1 Motivation 20
  - 5.1.2 Simulated Environment 21
- 5.2 Agent 21
- 5.3 Genetic Algorithm 22
- 5.4 Evaluation 22
  - 5.4.1 Test Cases 22
- 5.5 Time Limits 23

Chapter 6 Results and Discussion 24
6.1 Notable Results 26

Chapter 7 Conclusion 28
7.1 Limitations 28
7.2 Future Work 28
7.3 Final Remarks 29

Bibliography 30

Vita 33

LIST OF TABLES

4.1 Descriptions of event character representations for the Scavenger game. . 17
4.2 Comparison event partial orderings of designer sequence FXEW and agent sequence XFEW. '-' notes that the agent does not share a particular partial ordering with the designer. Sequence event representations described in Table 4.1 17

LIST OF FIGURES

2.1 Example of Crossover for Scavenger Level 4

4.1 Conversion of text abstraction to graphical representation of a level that scored 100 percent fitness for author sequence: 'Eat Food', 'Attacked by Enemy', then 'Win'. 18

6.1 Average generational fitnesses of levels generated for sequences of length 2. 24

6.2 Average generational fitnesses of levels generated for sequences of length 3. 25

6.3 Average generational fitnesses of levels generated for sequences of length of 4 or longer. 25

6.4 Interesting level generated for events 'Eat Food' then 'Win' 27

Chapter 1 Introduction

The production of a video game is a long and complicated process; even a simple game requires code writing, artwork, music and sound effects, user interface, and level design. Modern games are considerably more complicated and heavily detailed, requiring many, sometimes hundreds, of people to produce a game. As a result, a high standard has been set in terms of quality and content for game players. For small organizations or individuals, it is extremely difficult to produce the sheer amount of content that a large team can; however, the use of procedural content generation (PCG) grants the ability to create massive amounts of content at a fraction of the cost and labor. PCG has become increasingly important over the past decade as indie games such as *Minecraft* (Mojang, 2011) and *Factorio* (Wube Software, 2016) have proven PCG's viability in making a popular game with fewer resources, which has led to a prominence of PCG in games and games research [21].

In practice, the majority of PCG content relies heavily on randomization. As a result, PCG content can be chaotic, relinquishing much of the control that the designer holds over the gameplay experience of the player. This is undesirable in many cases because having little control over the player's experience may allow players to have very poor experiences, which a game designer should avoid at all costs. Designers can prevent poor player experiences by taking full control over the game's design, however, to do so they must design it themselves which may result in a lack of content and a significant amount man-hours to complete. Ideally, a designer has a level of control over PCG in which the player experience has enough structure such that it the designer's vision is realized, but is stochastic enough to provide the player with a sufficient amount of content. This balance can be found by introducing artificial intelligence techniques to PCG. Understandably, different techniques enable different levels of control over the player's experience. A simple approach is to have a designer enforce rules which the procedural generator must obey when creating content.

For example, suppose a game in which a series of rooms must be connected by doors. These rooms must be selected and oriented such that a path exists from each room to every other room. This is easily achievable if we select a room one at a time and place it flush to the exterior wall of some other room and add a door between them. The rules in which rooms are connected can be more complicated and can create different paths such as cycles or branching paths leading to dead ends, which would be good for an exploration-type game. The granularity of such rules can range from this broad approach in generating the layout of the game environment, to enforcing rules on individually generated rooms, to generated items within the rooms. Designers also have the choice to use any combination of these levels of procedural generation. For example, a designer can hand-author templates of connected rooms then randomly select to fill the template, or hand-design the whole environment, and generate where key items are placed within it.

Generally, the rules which are imposed in the PCG process are constraints on the physical environment such as enforcing a number of monsters in a dungeon or shaping the layout of a room. However, these rules are not limited to being physical characteristics of a game environment; rules may be imposed on abstract concepts of a game, such as game mechanics, player experience, and narrative. With whichever strategy a designer chooses, as the degree of rule granularity becomes lower-level, the more content can be procedurally generated, but the less control the designer has over the environment. Tradeoff of designer control for some other desirable feature of a system is a common problem in PCG and other fields of games research. A key goal of experience management is to balance coherent story progression with user agency, however, these two goals are often at odds [17, 18]. Allowing a player to freely explore the game world and experience whatever they desire is clearly difficult to control from a narrative perspective because the player can ignore or destroy elements which are essential to plot points in the intended game story. However, giving the player free reign in a world allows them to craft potentially infinite story variations making for an enjoyable experience while granting great replayablilty, much like PCG. On the other hand, heavily designer controlled games ensure the player has a specific experience laid out by the designer; which may be a very enjoyable experience, but will be nearly the same every time which gives the game poor replay value. Experience management attempts to find a compromise such that the player has some autonomy in the game, but the overall experience does not stray too far from what the game designer originally intended. Over the past couple decades there have been very successful projects that achieve this such as the Automated Story Director [16], PaSSAGE [22], and Façade [13]. While experience management itself does not utilize PCG, it serves the same goal of giving players a unique game experience and grants the author variable control over narrative.

Whereas experience managed systems tend to have the narrative created prior to play in an environment which is already created, we propose a system that takes a given series of events and generates an environment such that those events are experienced by a player in that particular order. Doing so could be trivial if the environment is designed in such a way that the player is forced to experience the events in the given order. For example, if the generated environment is a hallway, placing events along the hallway in the order in which they appear in a predetermined sequence would accomplish our goal. However, in this trivial solution, the player has lost their autonomy and placing items in the order which they are given is not a very intelligent approach for an AI system. We wish to generate environments in which the player is guided toward experiencing a designed sequence of events, but has the freedom to explore the environment in any manner. Additionally, environments which are non-linear are much more interesting and allow for more unique designs which better serves the goal of creating unique experiences because player may be experiencing the same sequence of events, but they will be presented with variations on how to experience that sequence due to the uniqueness of the environments themselves.

Though there has been success in using experience management to balance player autonomy and designer control in games, it has been primarily focused on generating narratives which does not translate to physical asset generation with PCG. Some research has produced systems which grant designer control while allowing player autonomy, but the extent of the designer's control is over the physical appearance of the level [1] and not concerned with controlling the player's experience in an ordered sequence of events. However, some systems have come close in preserving control over game event sequences, but fail in granting player autonomy due to the nature of the 2D platformer environment [8, 20].

We address the issue of using PCG to generate levels which grant player autonomy in an explorable environment while preserving designer control over game event sequences. We do so with a genetic algorithm that selects and uses levels by how well the events experienced by a game-playing agent fit a designer-defined sequence of events. After several generations, the fitness of the generated levels and a large set of purely randomly generated levels were measured for how well they fit a designer's constraints. Our system was able to consistently outperform our baseline, showing that it is possible to procedurally generate a level without sacrificing event sequence control and that a genetic algorithm is a reasonable method to do so.

Chapter 2 Background

2.1 Genetic Algorithms

2.1.1 Overview

As the name suggests, genetic algorithms are inspired by genetics in that when two parents create children, they will reflect the characteristics of their parents. In genetic algorithms, these parents are data and their children reflect desirable traits which are described via a fitness function. Typically, the probability of being chosen for reproduction is directly proportional to the fitness value [19].

Once all individuals in a population have been evaluated by the fitness function, two individuals are randomly selected using the weights of their fitness scores to generate two new individuals. Then, a crossover point is randomly selected from the positions in their data. Both parents use the same crossover point and split themselves into two sections at that point. The resulting pairs are combined into new individuals, called children, such that the second section of the second parent is added to the first section of the first parent, and similarly for the remaining sections [19]. It is important to note that the parents are selected with replacement so that parents with greater fitness values contribute more to the succeeding generation. The process of crossover is illustrated for a level in our test environment in Figure 2.1.

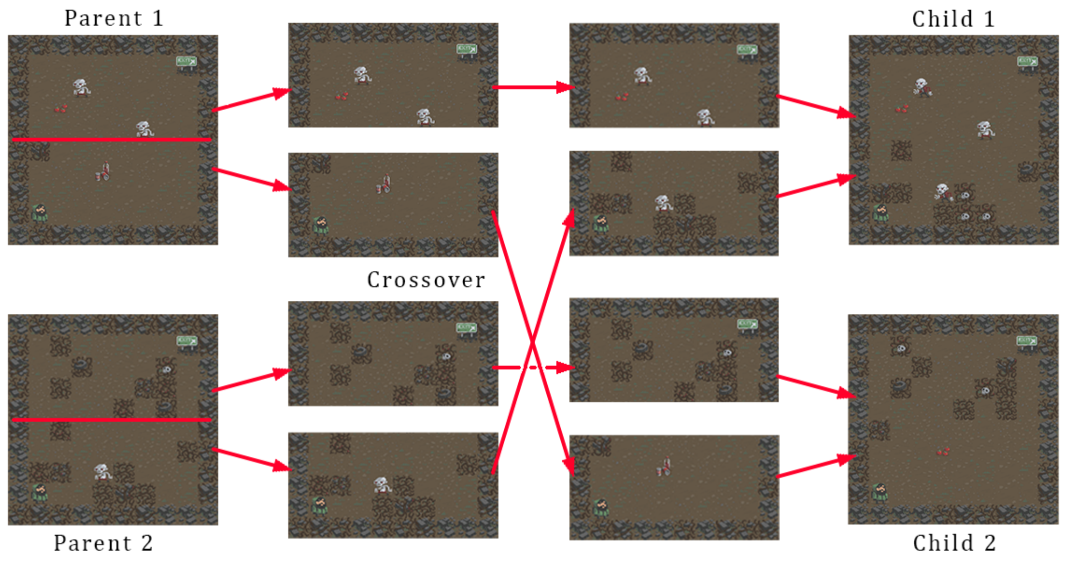


Figure 2.1: Example of Crossover for Scavenger Level

Combining data which score the highest fitness values optimizes the fitness of generated data to a maximum value with the intent to generate a piece of data which perfectly reflects desired model However, if one or more of the desired traits are not present in any of the individuals in the current generation, then no matter how many times the highest fitting individuals are crossed over, the perfect output can never be achieved. This issue is addressed by introducing new traits into the crossed-over data by mutating portions of it into some new value. Doing so adds variance and introduces values which may not have been present in any of the individuals, which may very well be a desirable trait that was non-existent in the population so far. Mutation allows the algorithm to explore different approaches of generating the data such that desirable traits are found then exploited by crossover. An implementation of a genetic algorithm is shown in Algorithm 1.

```
GeneticAlgorithm(population Fitness-Func)
repeat
    $NewPopulation \leftarrow$ empty set;
    for $i = 0$ to SIZE(population) do
        x $\leftarrow$ RandomSelection(population, FITNESS-FUNC);
        y $\leftarrow$ RandomSelection(population, FITNESS-FUNC);
        child $\leftarrow x, y$;
        if RandomValue < MutationProbability then
            child $\leftarrow$ MUTATE(child)
        end
        add child to $NewPopulation$;
    end
until $NumGenerations < MaxGenerations$;
```

Algorithm 1: Genetic Algorithm pseudocode

The combination of crossover and mutating two, well-fitting individuals provides a manner of working towards the maximal fitness through individuals that exhibit desirable traits, while exploring other patterns which are not congenital.

2.1.2 Applications in PCG

Genetic algorithms provide a means to generate many data which are bound to constraints defined by a programmer, but still vary from one to the other, making it a desirable approach for PCG. In particular, a game designer may use a genetic algorithm to generate levels that must contain certain elements of a level such as obstacles, enemies, and treasure. A simple genetic algorithm could ensure any number and combination of game elements are present in a level, but in varying locations such that the requirements are met, but the levels vary such that they are unique. Of course, this is not the extent of the use of genetic algorithms in PCG as they can identify traits as patterns that exist within the level and generate levels which exhibit these patterns [1, 2, 5]. Similarly, genetic algorithms can be used to generate levels which contain game elements which invoke a particular experience such as difficulty. Moghdam et. al. use a genetic algorithm to generate sequences of game elements which will reflect a particular difficulty rating that follows a difficulty curve as the player progresses through the game [14].

Genetic algorithms are a viable method to generate levels which imbue features and rules defined by a designer whether it be a physical constraint or one that is less tangible.

Chapter 3 Related Work

3.1 Procedural Content Generation

Research regarding PCG spans over many facets of game content such as: items, mechanics, stories, and quests; however, there is a particular emphasis on level generation. A prominent community of PCG researchers participate in the PCG workshop, hosted by the conference on the Foundations of Digital Games (FDG). In its first 10 years, 51 percent of the 95 published papers concerned level generation [11]. Throughout the breif history of PCG research, there have been several techniques developed which alter the game experience through the physical generation of an asset. Some approaches, such as design patterns and generative grammars, influence the game experience by replicating patterns and rules laid out by the designer, which can in turn be used to influence the player's experience be more aligned with what the designer intends. Others techniques, such as mixed-initiative tools and player modeling, use input from the designer or even the player to generate levels which alter the game experience to align with the designer's or player's desired experience.

3.1.1 Player Modeling

PCG using player modeling is a technique which generates levels based on observed player behavior [23]. These observations are used to classify the player, then generate levels which contain features that match the player's preferred way of playing. Data driven approaches of player modeling record game events and statistics from many players and create models which describe differing styles of play, then classify new players as one of the identified models and generate levels which were enjoyed by like-minded players [9, 15]. However, observations may be made on players individually to create intimate game experiences by, for example, learning a player's skill level and generating levels which are a suitable challenge [25]. Regardless of the approach, the generated level is tailored to each player, which accomplishes the goal of creating unique game experiences, but consequently the game designer loses control of the game experience. The designer may have some general control over the experience by dictating a difficulty curve they wish the player to follow as done by Holmgaard et. al. [9], but this will likely not be perfectly achieved because the generator will skew the difficulty based on the player's model.

3.1.2 Design Patterns

A method of PCG which gives the designer some control over the level design are through the use of design patterns. A design pattern is a is an observed sequence of recurring design elements in a game, gathered from several level examples, which can then be used to generate future levels [5]. The design patterns act as constraints which the level generator must follow such that the design elements described by the pattern are present in the level. The pattern can describe abstract features such as difficulty [14] and intensity [5] which are then translated to appropriate physical game elements which have been evaluated to provide the proper experience. With such patterns, a designer has broad control over the player's experience in that it is generally defined, but how the level is designed to provide the experience is still at the will of the generator. So, the designer may have control over the structure of the level, but the fine details of the game elements in the level are out of their control.

Though this is not the case for all generators using design patterns, some systems use very literal patterns describing the physical elements which must appear such as an enemy, treasure, or gap between platforms as in the Tanagra system [20] and others [8, 7]. Some of these approaches, [20, 8], could or do use a human designer's input to define the design pattern, which does in fact give the designer considerable control over the player's experience by defining the elements created by the level generator and the order in which the player interacts with them. In both of these system, levels are generated for a 2D platforming game, which are experienced linearly. For such environments, ordering the events which are experienced by the player is trivial because the player will always experience the events in the order in which they appear. For example if a designer wants the generator to create a level in which the player will encounter an enemy then a long jump, this is easily achievable by placing a partial-scene with an enemy followed by a partial-scene with a long gap between platforms. Additionally, there can be little variance in the space between events that would preserve the proper ordering of the events dictated by the design pattern because the game elements which are filling in the gaps may trigger events, especially in an environment in which any deviation from a flat platform is cause for a game event. Also, work by Green et. al. on scene stitching, their work reported the generated levels targeting specific event sequences were rarely completed by an A* agent, unless there was little variance in the generated level and the level from which the sequence was derived [8]. This goes to show that even for simple, linearly experienced games there is still work to be done to generate levels which may deviate from designer-curated examples and ensure a designer defined sequence of events is experienced which are also more consistently able to be completed by an A* agent and thus by human players.

Our work aims to satisfy the design pattern in a dungeon-type level in which the events experienced by the player are not guaranteed to be experienced in any particular order based on where they are physically located relative to the player, making the problem more difficult to solve. Also, we strive to have a much higher completion rate while allowing significant variance among our levels.

3.1.3 Generative Grammars

Generative grammars are formal languages which encapsulate the design of a level by imposing rules on the level's generation. Typically, such rules generally define game elements. For example, a room dungeon can be defined by room that has a monster and a piece of treasure. Therefore, on their own, grammars grant designers control over what elements can be in a level, but the order in which they are experienced by the player is not guaranteed. However, some order of the game elements can be controlled by using graph grammars [6]. Similarly to our problem, if the graph has multiple branching paths which all lead to a common terminal state, then the player will be given the opportunity to explore the level. However, if the designer intends the player to have a specific experience, each path to the goal must be the same. Then in translating the graph grammar to a level, there must be enough game elements to satisfy each path individually, meaning the player could experience more events on a factor of the number of paths to the goal, which is clearly not the experience the designer intended. Grammars are similar to design patterns in that grammars can be used to create a sequence of actions intended for a player to experience in the outputted generated level. PCG research using generative grammars grants designers the ability to constrain the individually generated game elements to a set of rules, which are in turn used to create mission structures for the player to experience. In work by Dorman et. al. [6], missions are generated based on the rules defined by the designer, however, the designer still has no input on the mission as a whole.

Graph grammars are similar to what we want to achieve in our work because it constructs missions consisting of a sequence of tasks, however, these missions are still random. Perhaps it could be modified such that we may prompt a designer to define the graph, but then we are still left with generative grammars describing each node in the graph. If the designer wants ultimate control over the player's experience, they must ensure each grammar produces only a single event, meaning there will need to be several definitions for each type of environment and each type of event, which is significant effort on the designer as mentioned in [6]. Therefore, our system must be capable of preserving sequence as in graph grammars, but must also ensure erroneous events are not added to the player's experience.

3.1.4 Mixed-Initiative Tools

Typically, PCG is not used to entirely generate a game and is merely a tool to assist a game designer in the creation of their game. Procedurally generated content must then be controlled such that it aligns with the themes of the game, otherwise players might think some generated content out of place. Mixed-initiative tools allow designers to have some control over the generated content by providing input that influences and/or constrains how the PCG system operates. Common workflows of mixed initiative tools are to generate to create several levels based on one provided by a designer [1, 4], and to have a generator create content on its own then have a designer provide feedback which influences the next generation [10, 20]. Like with many AI systems, there is a tradeoff of designer control for emergent level design with examples all over this spectrum.

Systems which emphasize variance in generated levels follows the theme of reqlinquishing some form of designer control, even if the designer provides examples of or specifies what they want the levels to look like. When a designer feeds a sample level into a generator, it can capture the patterns the designer knowingly or unknowlingly designed in the level, and can use that to learn and generate levels which reflect similar patterns [1, 4]. Though some control over the sequence of game events that occur in the level might be preserved in that patterns between two events, the randomness of PCG still leaves the levels subject to change that disrupts the gameplay experience. Because the gameplay events are not taken into account in a sequential manner, it cannot be reproduced. However, some approaches have preserved the designer's sequence of events while generating by nature of the levels being linear as described earlier in the Tanagra tool [20].

3.2 Experience Management

PCG is not the only manner in which a game designer may add replay value or control their game using artificial intelligence. In fact, there are means to do so that do not involve the generation of the physical assets of the game. Instead, there is interest in providing unique and plentiful game experiences through the generation of more intangible aspects of a game such as narrative. Though generation of narrative is quite different from the generation of physical assets, the methods discovered by research may be useful in designing PCG systems because the goals of PCG and experience management are more or less the same.

It is worthwhile to consider drama and experience management as methods for our work because it focuses on narrative generation and when an author writes a narrative, they structure their stories in specific sequences of linear events. Depending on the type of event, it may very well depend on a previous decision made by the characters in the story. For example in some story where travelers approach a fork in the road, choosing to go left will lead to a dramatically different series of events than if the travelers took the path to the right. In traditional storytelling, the author makes the decision for the characters and doesn't need to write any more story about the right path. However, if the characters have autonomy and are perhaps controlled by some entity other than the author, then the characters have the opportunity to experience the stories of the left or right path, or perhaps both if the story is read again and the previously unexplored path is taken. Clearly, introducing autonomy to the characters gives reason to the reader to read the story again, but to make different choices as is common in the popular choose-you-own-adventure (CYOA) novels. Because these decisions in CYOA novels lead to branching paths of other sequences of events, no matter what the characters decide, the author will have ultimate control over what is experienced. The drawback of CYOA novels is the burden on the author. For a single book, several stories must be written which usually sacrifices the length of any given story for the number of possible story-lines a reader could experience. With the integration of AI, the author's burden can be eased by generating many stories for readers to experience which all exist in an authored domain by implementing experience managers. Experience management is not contained to written story, it may be used to influence the narrative or other experiences of a video game such as dynamic difficulty, enjoyment, and player-catered experiences. Of the experience management techniques, the types of game elements which they control to influence the experience can be split into two categories as physical and metaphysical, both of which can be implemented using PCG.

3.2.1 Metaphysical Experience Management

Metaphysical elements of a game are ones which are intangible That is, it is not something the player can see within the game environment, but can be experienced or alters how the game environment is experienced such as difficulty, game mechanics, and narrative. Each of these metaphysical elements may be managed during by adjusting the difficulty [6, 25], altering or adding mechanics [3], and by steering the narrative to certain plot points [13, 17, 22]. High-level structures of stories with branching paths are typically represented by directed graphs in which the path from the starting node to a terminal one represents the story experienced by the player. As a result, at any given time we know what the player has experienced and the options which are currently available to the player that will alter the trajectory of the story. With drama management, the option the player takes may be influenced such that they take an action that puts them on a path in the story graph which aligns with what the author intends.

PCG Approaches for Metaphysical Experience Management

There have been several methods to preserve an author's preserved story when the player takes actions that undermine it such as the accommodation and intervention techniques employed by the mimesis system [24]. When a player takes an action that would make the author's narrative impossible, such as killing an integral character to the plot, mimesis can accommodate by replanning the story graph such that the deceased character's later actions are carried out by another, or may intervene by disallowing the player from killing the character in the first place. While this approach preserves the overall narrative, control over the detailed narrative is lost because the author did not write in that the character should die or some other unfortunate action should happen, instead the manager simply countered the player's move. For our goals, we wish that the player stay on the exact path laid out by the designer without having to make adjustments that undo or restrict the player's actions, especially because the actions to accommodate or intervene are limited and if done many times, the player may not be immersed as pointed out by Magerko [12].

Another strategy to guide players to a specific event is to bring significant attention to it as in ALT-SIM. While our goal is to influence the player towards a specific event similarly to ALT-SIM, we want to do this by generating assets that guide player's to the event. Bringing significant attention to the item of interest is an obtuse way of influencing the player, and we want to employ a more subtle approach.

Dynamic Difficulty

Perhaps the most simple form of experience management is done by dynamically adjusting the difficulty of the game. This could be in service to the player to make the game easier if it is clear they are struggling, or to make the game more difficult to engage and entertain the player when the game is too easy [25]. It could also be in service to the game designer to create experiences or missions that follow an authored difficulty curve [6, 14].

3.2.2 Physical Experience Management

Physical elements of a game are those which are tangible assets such as the interactable objects and the layout of the level or game world. The objects which exist in a game greatly influence the experience had by the player. Some game elements are directly related to triggering a game event when the player interacts with it such as unlocking a lock with a key, grabbing an instant health pickup, or encountering an enemy. Where such items are placed within the game world directly influences the order in which they are experienced, and if an experience manager can determine which event should occur next and it has a correlating item, that item can be moved or spawned in front of the player to ensure the next event is experienced. Alternatively, the structure of the game world can be altered or generated to make paths on which the existing interactable objects lay such that they are experienced in a particular order.

PCG Approaches for Physical Experience Management

Rather than adapting to the consequences, an ideal technique to prevent actions which undermine the story would be to prevent the player from doing so in the first place. For example, if a player exhibits behaviour which makes it more likely that they would kill an important character to the plot, the targeted character could be instructed to avoid the player or other characters may come to convince the player to not kill them. Such concept was realized by the Interactive Drama Architecture (IDA) [12]. IDA observes scenes and determines the likelihoods of events that act against the intended plot and introduce subtle elements which reduce and eliminate the probability of such events. Doing so better hides the underlying AI system from the player, making the experience more believable while preserving authorial control. While the goals of IDA align very well with ours, it uses player modeling to classify the player and counter's the player's perceived play style to guide them toward the next event in the story. Also, IDA, like all experienced managed systems, must take actions during gameplay to guide the player along the narrative they wish to experience, however, we are concerned with generating levels which do so without the need of a runtime director adjusting the world around the player to force them into specified events independent of the play style of each player.

3.3 Problem Statement

From the previous sections, we have seen that control over a player's experience is a topic of interest to many respected researchers, however, much of this work focuses on controlling the experience through generated narrative and other intangible features rather than purely through physical asset generation. Of the research done on controlling player experiences though PCG, ones that have successfully controlled an ordered sequence of game events have been done in side-scrolling environments which are linearly experienced, making this problem particularly trivial. In environments in which the player has the freedom to explore and potentially break from the designer's intended experience, influencing an ordered sequence of events is more difficult to enforce. A clear means of solving this problem would be to generate levels which forces the player along the path which trigger the defined game events, however, doing so would trivialize the problem, so we must ensure that the player always has the opportunity to break from the designer's curated experience, but the levels are designed such that the player will be guided toward the intended experience. Therefore, we want to design a system which procedurally generates levels for a Roguelike game that influences players to experience a defined sequence of events authored by a game designer.

Chapter 4 Methods

In our work, a designer specifies a series of events which are used in the fitness function of the genetic algorithm. Then, the genetic algorithm produces several generations of levels such that the average fitness of the generation is maximized. The primary elements of our approach are discussed in greater detail below.

4.1 Fitness Function

The core of our fitness function is a comparison of events experienced by an optimal game-playing agent and the events defined by the designer.

4.1.1 Agent Evaluation

As mentioned previously, our fitness function relies on comparing experienced behavior against a target experience defined by the designer. To make this comparison, we simulate human behavior using an automated search agent. This was achieved using a synthetic agent that utilizes A* search to find an optimal path through each game level, however, it should be noted that any agent behavior could be used to test the levels; the most important factor is that the agent's behavior is consistent. One particular advantage of utilizing an A* agent is that if the level is impossible to complete, then the A* agent will not find any valid path to the goal, allowing us to label and discard impossible levels.

Before each move, the agent would plan a path to each terminal game state. Terminal states occur when the player has a score less than 0, or is in the goal state. Once all terminal states have been found, the path yielding the lowest cost is returned as the path taken by the agent. Each simulated game state is scored by the sum of heuristic values of all actions that led to that state, and the current action that leads to a terminal state with the smallest value is chosen.

Each game state leads to other succeeding game states depending on which action the player takes, each of which have additional succeeding game states until a goal state has been reached. The connections create a search tree of game states. Each game state on the path to the terminal state is scored using a heuristic that is, for the most part, directly related to the change in player's score when advancing to the next state with the exception of when the agent is adjacent to a breakable wall, in which case the cost of the action is equal to the health of the wall to have the agent choose the wall with the lowest health when adjacent to more than one wall. The heuristic values of all states on each path to a terminal state, including the current state, are summed and the agent begins on the path with the lowest value.

Because the agent searches each path of the search tree, it often took a long time due to the size of the search tree. After all, each game state has a branching factor of 4, and the agent is allowed 20 moves to start, and can earn additional moves leading to a tree of at least $4^{20} \approx 1.1 * 10^{12}$ game states. In order to speed up the search process, several procedures were implemented to reduce the size of the search tree. Our test environment is an 8x8 grid, so the maximum distance to the goal requires 16 moves, but it is likely that the path to the goal is takes more than 16 actions due to obstacles in the level. Therefore, we limited the height of the search tree to 18 because.Additionally, a sub-tree would be pruned if it was inevitable that the agent would lose the game if it continued that path. Inevitable defeat is determined by comparing the Manhattan distance of the nearest goal to the number of remaining potential actions. Both of these strategies dramatically reduced the runtime of the agent's evaluation of a level.

4.1.2 Sequence Partial Orderings

During the agent's play, it records game events that it experienced which are then compared to the designer's series of desired events. If the agent's sequence is an exact match to the designer's, then the fitness value is 1, otherwise, the longest, best-fitting, matching partial ordering of the agent's and designer's sequences is used to score the level's fitness. This strategy allows levels to receive some fitness score so that they can contribute their imperfect, but potentially useful traits to future generations of levels. For example, if the designer defines the sequence of events ”FXW” (see Table 4.1 for event descriptions) and one level in the population returns the sequence ”FW”, then it clearly partially satisfies the designer's constraints because it contains some game elements which leads to the ”F” event. Suppose there is some other level in the population which returns the sequence ”XW”, then it may be that the elements in the two levels are combined such that when evaluated it produces the events defined by the designer.

The method used to obtain these partial orderings achieves the same goal as the longest common subsequence (LCS) algorithm. The key difference in our approach is that rather than obtaining the longest common subsequences, we find all common subsequences so that as many with varying length can be used to score the event sequence, which is described in detail later.

Event Sequence Scoring

Desirable traits of levels are not only found from the entire event sequence, but could be found as subsequences. Continuing the example, some partial orderings of designer sequence ”FXW” are ”FX”, ”FW”, and ”XW”. Suppose a level returns the sequence ”EFEFXFW”, which contains the subsequence ”FXW”. This example generated much more events than the designer wanted, but the desired sequence of events did occur somewhere in the level. Therefore the level possesses the traits required to deliver a proper level, and perhaps in future generations, those traits will be reflected and the others may fade. If the agent's and designer's sequences are not exact matches, then sets of partial orderings of both sequences are generated and compared. If there are any matches in the sets of partial orderings, the longest match is chosen to score the fitness of the current level. The chosen partial ordering is awarded points equal to its length and awarded additional points for each instance in which two events occur in the correct order relative to the designer's sequence. Scoring the events and their orderings with the same weight makes the order in which events occur as important as the events themselves. If there are multiple partial order matches of equal length, then the one with the greatest score is chosen to score the level. Additionally, if the chosen partial ordering is less than half the length of the original sequence, the scores of multiple partial orderings are summed to score the level. If multiple partial orderings contribute to the fitness score, then the sum of their lengths must not exceed the length of the designer's sequence. To constrain the fitness value between 0 and 1, the score of the partial ordering(s) is divided by the maximum possible score, the score of the designer's sequence. In the event that the agent cannot complete the level, it is assumed that the level is impossible and is given a fitness value of 0. However, if a level is simply not completed due to a lack of time, then it is still considered useful and is evaluated, but is clearly not capable of achieving a high fitness value. In the event that a level is concentrated with objects, the agent will return a very long sequence of events, which may likely contain the designer's sequence. It is obviously good that the level contained the correct events, but must still be penalized for generating too many events because the designer should receive levels reflecting exact sequence matches. This is mitigated by subtracting 0.5 times the difference between the length of the designer's sequence and the agent's if the agent's sequence is longer. We chose 0.5 because a reduction of any more score caused many levels to calculate a fitness value of 0. Also, permitting 2 extra events for each correct event seemed like a reasonable trade-off, and performed well in testing.

Scoring Example

To demonstrate the fitness function, consider an example where the designer defines the sequence FXEW and the agent returns the sequence XFEW for a particular level. See Table 4.1 for an explanation of these sequence representations. The partial orderings of the designer's defined sequence (XFEW), the agent's matching partial orderings, and how they are scored are shown in Table 4.2. The greatest agent score would be chosen being event FEW with score of 5: 1 point for each of the 3 correct events, and 2 points for correct orderings FE and EW. However, that is the scoring relative to the agent's sequence, and to truly score the sequence we must compare the score to one relative to the designer's sequence. The desiger-relative sequence FEW only scores 4 points: 1 point for each of the 3 correct events, and 1 point for the correct ordering EW (note that the ordering XE does not score any points because that ordering does not occur in the designer's original sequence). Because a score of 4 is the score given relative to the designer's sequence, the overall score for this level would be 4. The maximum possible score for this sequence is 7: 2 times the length, 4, minus 1, thus the fitness value is 4/7 or 0.57.

4.2 Crossover

Crossover in genetic algorithms involves splitting two different data at some arbitrary point and using these broken pieces of data to create two new data. In this experiment,

Table 4.1: Descriptions of event character representations for the Scavenger game.

Character	Event
F	'Eat Food'
E	'Attacked by Enemy'
X	'Destroy Wall'
W	'Win'
L	'Lose'

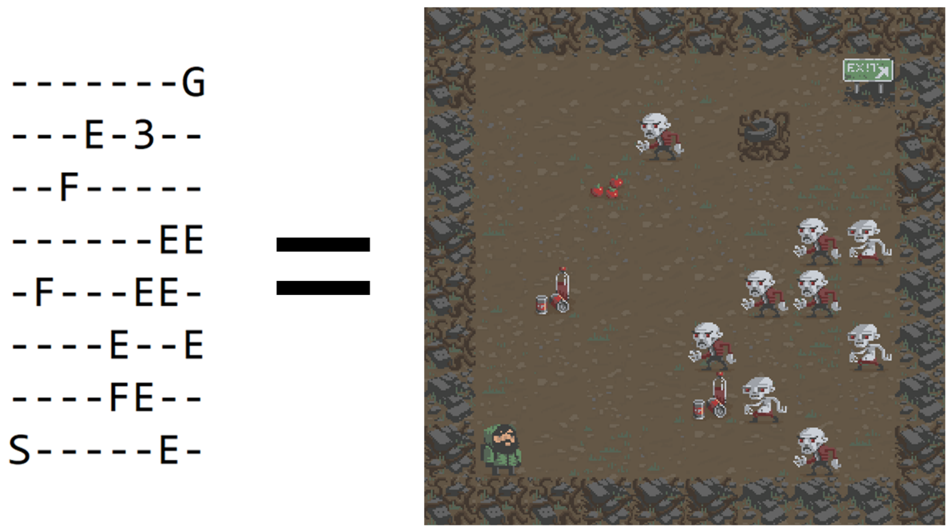
Table 4.2: Comparison event partial orderings of designer sequence FXEW and agent sequence XFEW. '-' notes that the agent does not share a particular partial ordering with the designer. Sequence event representations described in Table 4.1

Sequence	Designer Score	Agent Score	Fitness
FXEW	7	-	-
FXE	5	-	-
XEW	5	4	4/7
FXW	4	-	-
FEW	4	5	4/7
FX	3	-	-
XE	3	2	2/7
EW	3	3	3/7
FE	2	3	2/7
FW	2	2	2/7
XW	2	2	2/7
F	1	1	1/7
X	1	1	1/7
E	1	1	1/7
W	1	1	1/7

the data is saved as a string of characters that describe the level, and the point at which this string is separated is determined at random. An example of the character abstraction can be seen in Figure 4.1.

The two new levels are then generated by adding the front part of the first level to the back part of the second level, and vice versa.

The interesting part is in deciding which levels are chosen to crossover. Each level has already been evaluated by the fitness function, so the fitness value of each level is used for random weighted sampling with replacement. This allows levels with higher fitness values to be more influential on the future generations of levels.



```
-------G
---E-3--
--F-----
------EE
-F---EE-
----E--E
----FE--
S-----E-
```

Figure 4.1: Conversion of text abstraction to graphical representation of a level that scored 100 percent fitness for author sequence: 'Eat Food', 'Attacked by Enemy', then 'Win'.

4.3 Mutation

After two new levels have been generated via crossover, they are mutated to create some variance in the levels. In this experiment, each position in the level has some probability of being mutated. This probability is determined by an exponential decay function that is determined by the current generation number, shown below:

$$\epsilon = (1 + \epsilon_0) - 0.99^{-numGens}$$

Where *numGens* is an integer describing how many generations have passed, and ϵ_0 is the initial probability of mutation, 0.15. This exponential decay of probability of mutation yields significant exploration for the desired events in the beginning of the test, and allows the final levels to exploit the previous generations variance. The probability of mutation is set to never be less than 0.03 to prevent the final levels from being too similar.

If a position is to be mutated, what it is turned into is determined by the frequency of events the agent is producing compared to the events that the designer desires. For example if an agent is producing a lot of 'Eat Food' events, but the designer defines no or few 'Eat Food' events, then the position will have a reduced probability of mutating into food such that the mutations are always reflecting the needs of the designer.

4.3.1 Mutation Rate Methods

We tried several other approaches for mutation such as a static mutation rate and linear decay before ending with the exponential decay strategy. Using a high static mutation yielded levels with a lot of variance in early generations, which was good for finding traits of desirable levels, however, those traits could not be preserved due to the high mutation rate. Using a low static mutation rate prevented desirable traits from being phased-out, but did not allow for enough variance to find the traits. Clearly, we needed a means for high variance in early generations and low variance in later generations, so we tried a linearly decaying mutation rate. This worked fine, however the mutation rate had to start high for significant variance to find desirable traits, which led to higher mutation throughout generations, which would sometimes mutate unfavorably in later generations. We then expanded on this idea by using an exponentially decaying mutation rate which performed better than linear decay, so we stayed with this approach for our final experiment.

Chapter 5 Experiments

To test the effectiveness of using a genetic algorithm to produce levels that grant event sequence control by designing for designer-defined events, the average fitness of 25 generations of 25 levels each were compared to the average fitness of 1,000 randomly generated levels with a grid-shaped for a Roguelike game to measure if the generated levels are more effective at representing the designer's goals over time. 9 different sequences of events were tested to gauge the effectiveness of this system with a variety of event orderings and sequence length. Several different fitness functions and strategies for mutation rate were also tested, the best of which were described in previous sections and were carried out in our final results.

5.1 Test Environment

The Scavenger game was used as the environment to test our system. The game features a survivor who must traverse an 8-by-8, grid-shaped level while collecting food, digging through walls, and avoiding enemies to reach the exit. At any point in time, the level can be described as a game state recording the locations of every currently active entity. The level is comprised of tiles, and there may be an entity on top of each tile. Entities in the game include:

- player: The character controlled by the player. The player only has the action to move to an adjacent tile. If moving into a wall, the wall loses 1 health.
- empty tile: a tile in the level which has no item atop it.
- food pickups: Item on the ground that grant the player 20 food points upon collection.
- enemy: Characters that move toward the player after every other player action. If the enemy is adjacent to the player when it can move, it attacks the player instead subtracting 20 of the player's points.
- wall: An obstacle blocking the player's path. If the player moves into the wall 3 times, the wall is destroyed and replaced with an empty tile.
- exit: The goal state. When the player reaches this tile, they win the game.

To play the game, the player must move through the environment to reach the exit. Each time the player takes an action, they lose a point and if they run out of points they lose.

5.1.1 Motivation

Several factors led us to use the Scavenger game as a basis for our experiment. World time in this game is discrete, so time only advances when the player has made a move. At each time step, the player loses food, and every other time step, the enemies move or attack if they are adjacent to the player. This makes simulating game play much more simple because the data structures can represent discrete events rather than continuous time. Additionally, the game provided three assets that were easily represented as events: food pickups, enemies, and breakable walls, granting us the ability to test well varied event sequences. Due to the 8-by-8 grid structure of the levels in this game, it was simple to abstract the levels into small text files where each tile on the grid described each asset with a single character. This allowed a simulated environment to be used which did not require the rendering of the game's graphics, which rapidly increased level testing time.

5.1.2 Simulated Environment

In order to conduct testing in a timely fashion, the mechanics of the game were abstracted into a simulation program to remove the overhead of displaying the graphics of the game.

There is still some latency due to the exhaustive search requiring thousands of differing game states to be held in memory. Due to the dynamic nature of each of the level's objects, the agent had to recalculate its path after each action because food availability status, changing wall health, and enemy positions and their turn order would needed to be stored from state to state. Most of these factors in the game state could be represented by a character that described the object's location, and the health of the wall could be represented by a number character, but the turn order needed to be stored in a integer of its own. This 72-character long (including newline characters) was used to generate a new game state for each potential future game state in the A-star agent's search, and generated at the end of the agent's action, which can quickly require a large amount of system resources for both storing these game state representations and their generation into a new instance of the simulated environment. 4.1 shows how the levels are represented as characters and converted into the playable level.

5.2 Agent

The A* search algorithm has historically been used to simulate human gameplay behaviour as demonstrated by Green et al. [7], so we used this strategy to employ an agent that would play our levels and record the game events which it experienced. The heuristic which scored simulated game states in the search tree is calculated roughly based on the point system of the original game where if player earns 20 points for eating food, loses 20 points for being hit by an enemy, and loses a point each time they make a move. The only difference between the point system of the game and the heuristic function is that the value of digging into a wall is the current health of the wall so that the agent will always choose a wall that is more damaged than its neighboring walls.

Since the agent's heuristic is primarily centered around the point structure of the game, the optimal path would yield the best score. This also ensures that if an optimal agent cannot complete a level, then cannot be completed by a human and should thus be discarded.

5.3 Genetic Algorithm

This experiment used a genetic algorithm that generated 25 generations of 25 individuals. The mutation rate was calculated using an exponential decay function which started at 0.15 and did not fall below 0.03.

5.4 Evaluation

As a simple test of our levels' success, the average fitness of each generation is compared to the average fitness of 1,000 randomly generated levels based on the same designer defined sequence of events. To further test the flexibility of our system, 10 different sequences of varying length and events were tested. Over time, if the generated levels produce a higher average fitness than the baseline, it is an indication that our system is successful in generating levels that guide players to experience a defined sequence of events.

5.4.1 Test Cases

The test cases used for evaluating our system were organized by the complexity of the designer's input. Each test case ends with the 'Win' event because if the levels are designed for the players to fail, that would not be very helpful to the level designer and ensures the level is possible to complete.

The first set of tests are sequences of only two events. Each case in this set is one of the available events in 4.1 followed by a 'Win' event. This first set represent the simplest cases in the Scavenger environment and will allow us to directly compare how well the system we designed can directly design for each specific event. Additionally we tested generating levels for simply the 'Win' event to gauge if our system would optimize the level by not generating any event objects, or it would create levels with objects, yet the optimal path is to not interact with them.

The second set of tests are sequences of three events. Each case in this set explores combinations of events such that each type of event is combined with each other event in at least one sequence. For the sake of graph readability, every combination of events were not tested. There is one exception with the sequence FFW because we wanted to see how the system handled designing for the same event more than once in a level.

The third set of tests are sequences of four or more events. This set of test cases is meant to push our system to its limits because each sequence contains every possible event. We suspected that it may be very difficult for the system to generate levels that return large, ordered sequences of events in a relatively small environment.

5.5 Time Limits

Some generated levels become quite complicated during this process, which can require a significant amount of processing resources during level testing. Some of the cases result in the agent to spend several minutes to an hour to make a single move. To quicken this process, if an agent takes more than one second to make a decision about a move, it will take the current, best-known action. This also causes some levels take an a long time to process if each agent's move takes a significant amount of time, so the evaluation of a level is also be cut off if it takes longer than ten seconds and the events that the agent had experienced until that point are returned to be evaluated.

To test the effectiveness of this system, 25 generations of levels were generated fitting to 10 different designer defined sequences of varying length. The average fitness of the levels at each generation are plotted to compare to the average fitness of 1,000 randomly generated levels for the same 9 designer defined sequences. For reference, level events 'Eat Food', 'Attacked by Enemy', 'Destroy Wall', and 'Win' are represented by the characters F, E, X, W, respectively.

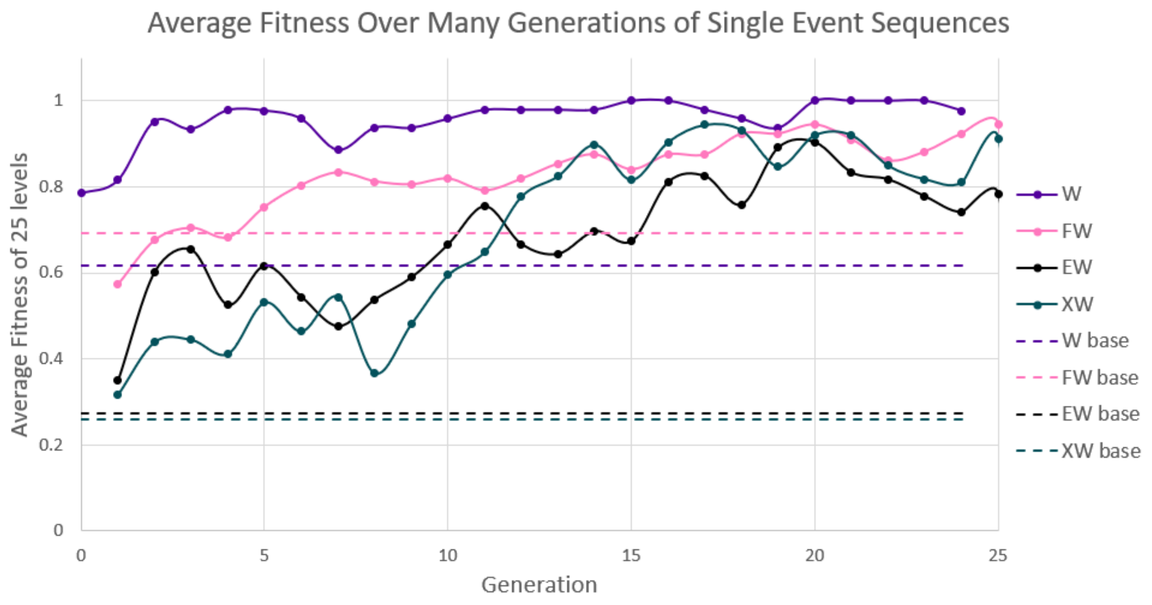


Figure 6.1: Average generational fitnesses of levels generated for sequences of length 2.

Figure 6.1 shows that the average fitness of the generated levels is consistently greater than the baseline for each of the respective sequences and steadily increases. By the 20th generation, the average fitness is its most stable and each test achieves a fitness score of 0.9 or greater, which is well above their respective baselines indicating the generated levels are consistently representing the exact desired sequence. The average fitness of the 'Win' only test is the only test that yielded an average fitness of 1 and did so for several generations, likely because the sequence is as simple as it gets. When observing the produced levels, ones from the 25th generation are quite bland, only including a few walls. This is due to variance becoming poor and the decreased likelihood of generating other types of objects with the more generations produced. However, with only 5 generations, we are still able to achieve several fitness scores of 1 while producing levels that still include interesting objects such as enemies and food, but do not trigger any events other than reaching the end of the level.

Figure 6.2 shows results of a series of slightly more complicated designer sequences, and similar results to the first set of tests are achieved. However, by the 20th gener-

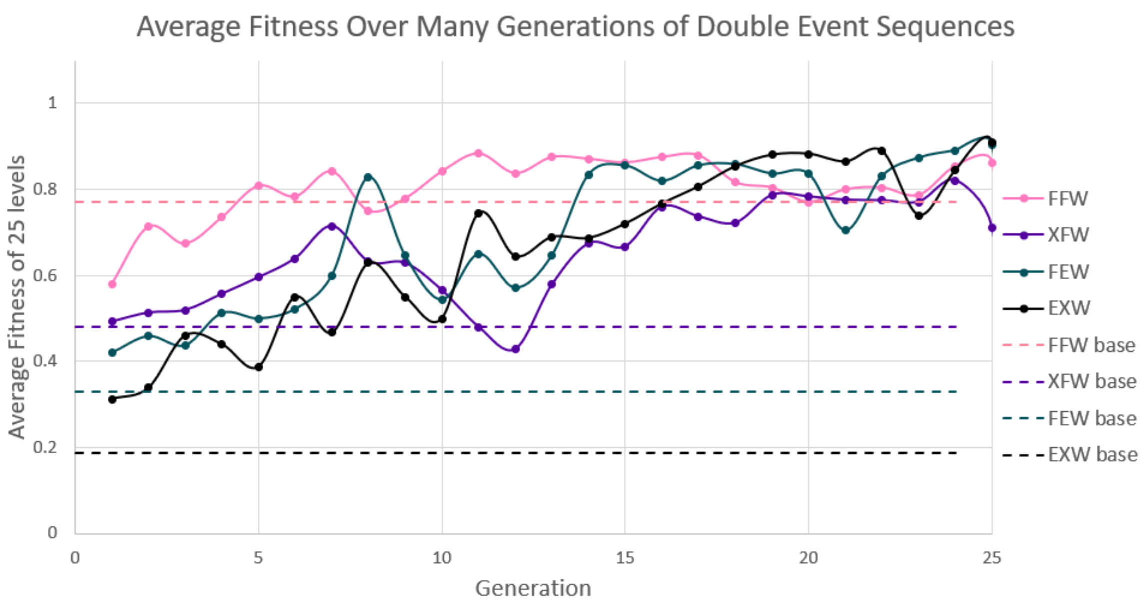


Figure 6.2: Average generational fitnesses of levels generated for sequences of length 3.

ation, the average fitness scores are again stable, but each test has a slightly lower fitness of about 0.8 or more. The lesser score is attributed to the difficulty of producing a level that is more complicated, however, several of the final outputted levels reported fitness scores of 1.

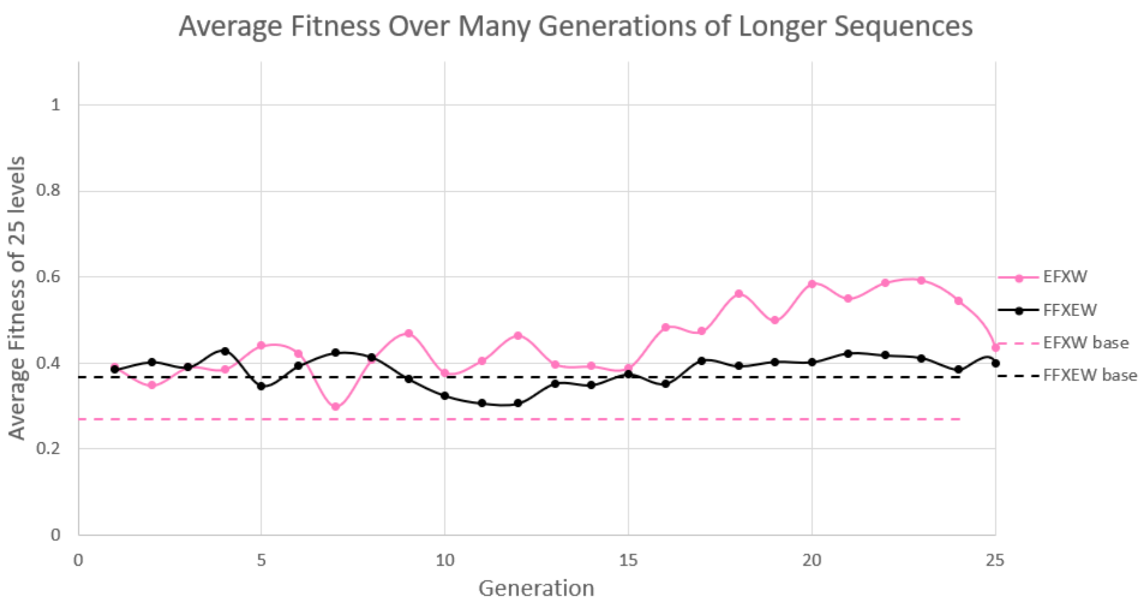


Figure 6.3: Average generational fitnesses of levels generated for sequences of length of 4 or longer.

Figure 6.3 shows the results of our genetic algorithm when used to create levels to realize designer goals containing four or more events. The four event sequence shows improvement and is again stable around the 20th generation, however only achieves an average fitness of 0.5, which is greater than the baseline, but not as significantly so as the results in previous tests. Additionally, no outputted levels yield an exact match to what the designer desired. The sequence of length five shows no improvement in fitness, therefore we conclude that in this environment, we cannot control the sequence of events if it is longer than 3. However, given the small size of the environment, it is reasonable that shorter sequences of events are more manageable.

It is apparent from each Figures 6.1 and 6.2 that sequences which include more 'Eat Food' events tend achieve a higher fitness more quickly. This is most likely due to the 'Attacked by Enemy' and 'Destroy Wall' events being much more difficult to achieve. Eating food is very likely to occur because the agent receives a large positive reward for doing so, and can do so easily by simply moving to the food's location, which is why these events were satisfied in earlier generations. Forcing the agent to destroy a wall is difficult because the agent won't go through a wall if it could easily go around it. For 'Destroy Wall' events to occur, the system must generate lengths of walls to block the character, or place them strategically such that it makes it possible to avoid an enemy's path, which is needless to say much more difficult than arbitrarily placing a piece of food. Lastly, the agent receives a large negative reward for being attacked by an enemy, so this event will only occur when it the agent has no other choice but to accept an attack from an enemy. It is quite impressive that this system can consistently design levels that ensures the optimal path requires an optimal agent to sacrifice some of their points. Additionally, the system could easily satisfy the designer's conditions by generating levels that are completely filled with whatever objects generate the desired events, however, this does not occur because it must adhere to the number of times and the specific order in which the designer dictates each event to occur.

6.1 Notable Results

Upon inspection of the final generated levels, it is clear that our system did not just generate levels that would most easily control the player's experience, but additionally created interesting levels where the optimal path is not clear. If the system were ordered to generate a level that causes the player to eat some food, then win, surely the easiest level to design would be to generate nothing in the level other than a single piece of food, directly in front of the player or directly in front of the goal. This is not the case with our system, instead the level designed for the events 'Eat Food' then 'Win' contains two food items which are guarded by enemies, see Figure 6.4.

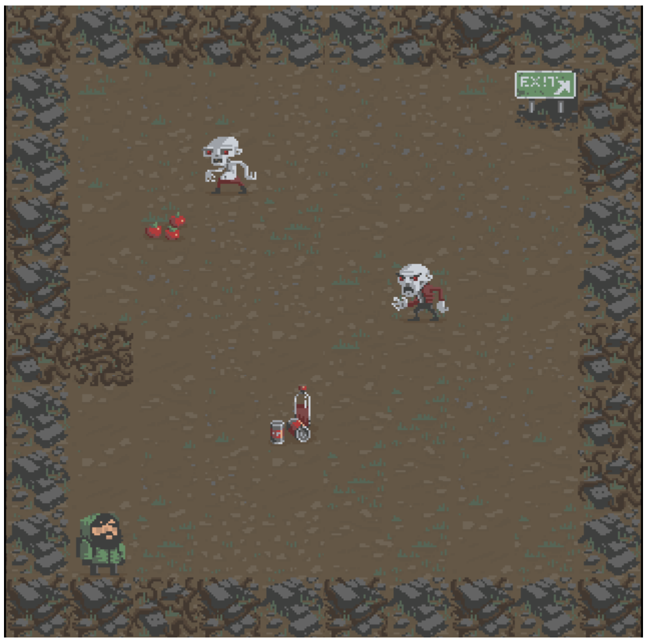


Figure 6.4: Interesting level generated for events 'Eat Food' then 'Win'

Chapter 7 Conclusion

7.1 Limitations

A significant limitation of this experiment is its run time. Due to the large state space explored by the game-playing agent, it can take quite a while for it to simulate playing the level. This causes a generation to take about 30 seconds to be created, on average. So, if many generations of levels are to be generated, then this process can easily take a long time, however, the results show that just 5 to 10 generations are necessary to produce well fitting levels in simple cases. Additionally, waiting 2.5 to 5 minutes for several level designs is presumably much quicker than if the level designer designed the levels themselves when considering the time it takes to design the level, test it, and redesign to better fit the desired sequence of events.

Since a typical player's performance is not optimal like our agent's, it is apparent that if a player were to play levels generated by our system, then they may still not experience the level in the intended manner. However, this is also true for hand-designed levels because it is impossible to know exactly what the player is going to do, and designers can only try to guide the player to play a level how they want if not forcing the events to occur. Thus, we think it is acceptable that levels generated with our system guide an optimal player to experience the level as intended because a player is more likely to want to play optimally than they are to play poorly.

The system created in this experiment is curated for the specific testing environment, which does not prove its viability for generating levels for games of other genres. However, with some effort, it would be possible to extend this system to accept other environments as long as the game can be simulated and the levels and all necessary data within them can be abstracted to text files.

7.2 Future Work

We would like to improve this project by reducing the run time of the generation process to make this a viable tool for game designers. This would likely be done by optimizing our code such that the agent's search space is much smaller and the search tree is pruned more effectively without degrading performance, or perhaps we could develop such a way to evaluate the levels without a gameplay agent at all. Since there are several techniques to randomly generate levels, we would like to explore these methods and compare them to see which best addresses this problem.

Also, it would be interesting to compare events that human players experience based on what the level was designed for to see how well this system works in practice. Additionally, it would be interesting to see if the players have any preference for our generated levels, or hand-designed ones.

We would also like to see how well this system can be applied to a variety of games of other genres and level size to test the flexibility of this system and determine if this is a reasonable approach for controlling event sequences for other types of games.

7.3 Final Remarks

The system we have created allows game designers to procedurally generate levels for a 2D, Roguelike game which will guide players to experience game events in a sequence that the designer authors, despite the inherit randomness of PCG.

Bibliography

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

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

[3] Calvin Ashmore and Michael Nitsche. The quest in a generated world. In *DiGRA conference*. Citeseer, 2007.

[4] Alexander Baldwin, Steve Dahlskog, Jose M Font, and Johan Holmberg. Towards pattern-based mixed-initiative dungeon generation. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–10, 2017.

[5] Steve Dahlskog and Julian Togelius. Patterns as objectives for level generation. In *Design Patterns in Games (DPG), Chania, Crete, Greece (2013)*. ACM Digital Library, 2013.

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

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

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

[9] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Evolving personas for player decision modeling. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[10] Isaac Karth and Adam M Smith. Addressing the fundamental tension of pcgml with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019.

[11] Antonios Liapis. 10 years of the pcg workshop: Past and future trends. In *International Conference on the Foundations of Digital Games*, pages 1–10, 2020.

[12] Brian Magerko. Evaluating preemptive story direction in the interactive drama architecture. *J. Game Dev.*, 2(3):25–52, 2007.

[13] Michael Mateas and Andrew Stern. Integrating plot, character and natural language processing in the interactive drama façade. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment (TIDSE-03)*, volume 2, 2003.

[14] Arman Balali Moghadam and Marjan Kuchaki Rafsanjani. A genetic approach in procedural content generation for platformer games level creation. In *2017 2nd Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*, pages 141–146. IEEE, 2017.

[15] Chris Pedersen, Julian Togelius, and Georgios N Yannakakis. Modeling player experience in super mario bros. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 132–139. IEEE, 2009.

[16] Mark O Riedl, Andrew Stern, Don Dini, and Jason Alderman. Dynamic experience management in virtual worlds for entertainment, education, and training. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning*, 4(2):23–42, 2008.

[17] Mark Owen Riedl and Vadim Bulitko. Interactive narrative: An intelligent systems approach. *Ai Magazine*, 34(1):67–67, 2013.

[18] David L Roberts and Charles L Isbell. A survey and qualitative analysis of recent advances in drama management. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning*, 4(2):61–75, 2008.

[19] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach Third Edition.* Prentice Hall, 2010.

[20] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on computational intelligence and AI in games*, 3(3):201–215, 2011.

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

[22] David Thue, Vadim Bulitko, Marcia Spetch, and Eric Wasylishen. Interactive storytelling: A player modelling approach. In *AIIDE*, pages 43–48, 2007.

[23] Georgios N Yannakakis, Pieter Spronck, Daniele Loiacono, and Elisabeth André. Player modeling. 2013.

[24] R Michael Young. Notes on the use of plan structures in the creation of interactive plot. In *AAAI fall symposium on narrative intelligence*, pages 164–167, 1999.

[25] Alexander Zook, Stephen Lee-Urban, Michael R Drinkwater, and Mark O Riedl. Skill-based mission generation: A data-driven temporal player modeling approach. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, pages 1–8, 2012.

Vita

Michael Philip Probst

Place of Birth:

- Lexington, KY

Education:

- University of Kentucky, Lexington, KY
  B.S. in Computer Engineering, May 2020
  magna cum laude

Professional Positions:

- Unity Developer, Enomalies Mar. 2020–present
- Graduate Research Assistant, University of Kentucky Summer 2021–present
- Electrical Engineering Intern, Mason and Hanger May 2016–Mar. 2020