AUTOMATED ITERATIVE GAME DESIGN

A Dissertation
Presented to
The Academic Faculty

By

Alexander Zook

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology

December 2016

AUTOMATED ITERATIVE GAME DESIGN

Approved by:

Dr. Mark Riedl, Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Brian Magerko, Co-Advisor
School of Literature, Media, and Communication
Georgia Institute of Technology

Dr. Charles Isbell
School of Interactive Computing
Georgia Institute of Technology

Dr. Ashok Goel
School of Interactive Computing
Georgia Institute of Technology

Dr. Michael Mateas
Computational Media
University of California, Santa Cruz

Dr. Jeff Orkin
Giant Otter Technologies

Date Approved: October 17, 2016

A delayed game is eventually good, but a rushed game is forever bad.

Shigeru Miyamoto

To my family, friends, and sometime rivals.

ACKNOWLEDGEMENTS

Acknowledgments are always a challenge: it is impossible to recognize all the people who have directly or indirectly influenced your work. Obviously this work would not be possible without Mark's generosity in providing me opportunities to do the work I am passionate about, patience to tolerate my tangents, and intellect to constantly challenge me to better refine the ideas that have driven my research. Boyang 'Albert' Li and Brian O'Neill have both been fantastic mentors throughout my early graduate career, guiding me through the bureaucracy, posing challenging thought questions, and being wonderful collaborators. And Kristin Siu and Matthew Guzdial were wonderful support in completing my work, coding up *Cardonomicon*, and drawing up (multiple) walls of *Pokemon*.

Any academic career is not limited to just those at your home institution, and there are more people than I can track to thank for their wisdom and insight. Mike Cook has been a constant partner in crime, starting from our first days presenting procedural game generation work with ANGELINA and GAMEFORGE at CIG in 2011. Between helping organize the Experimental AI in Games workshop, to driving me to organize the AI Game Jam, to joining me (with Gillian Smith and Tommy Thompson, too!) in the *Plus Four to Science* AI and Games podcast, I could not ask for better non-academic academic collaboration. Adam Smith pried me from the grasp of genetic algorithms School of Thought (sorry Julian!) to think harder about how to represent games and consider where computation can fit in the game design process as a whole. Ian Horswill served as a role model for re-evaluating how AI can work in games; his constant support helped keep EXAG going and showed me the potential for genuine change in the ways we approach research in games. Without filling pages, there is quite a list of co-conspirators I would be remiss to not thank:

- Tommy Thompson and Gillian Smith for helping with Plus Four to Science and general mentorship
- Eric Fruchter for showing me the potential of bullethell games for PCG and building

the platform and ideas behind automated iteration. Wish I could share where those have gone.

- Stephen Lee-Urban and Michael Drinkwater for helping realize GAMETAILOR. Michael especially for the long nights on POKEPOCALYPSE (RIP).

- Ken Hartsook and Sauvik Das: bringing GAMEFORGE to life not only started my academic career, but the experience of showing Grace Hopper attendees that RPGs could break story tropes opened my eyes to the range of possibilities for PCG.

- Antonios Liapis for making EXAG keep happening, staying on time, and fostering such a fantastic community with Mike and I!

- Stephano Allesina for my first publication and shift from neuroscience into computational work (and eventually games). What an odd transition that's been.

My work has also been *heavily* influenced by my experiences (and ongoing support) from my partners in the games industry. Randy Ramelb has been there since the beginning, hammering out some of my bad academic habits to give me a sharper and clearer mind for how game development works. Craig Fryar set me on the road to working on 'real' games by pulling me in as 'the math guy' for my first (and second) internships—the experiences at Bioware Austin on *Star Wars: The Old Republic* showed me what it means to analyze games to support game development. Jeremy Ballenger 'the Challenger' drove my growth as a strategic thinker during my time with him at Bioware Austin and (along with Craig) has been a wonderful ongoing mentor to help me navigate my experiences in the games industry.

Finishing my thesis and seeing some of my work brought to bear in live games would not be possible without the support of my Blizzard family. Chaitanya 'Chaitown' Chandra Chemudugunta first made the dream of working at Blizzard a reality through my summer internships, and has since given me the support and freedom to drive game design features

and changes I never imagined possible. His support has shaped me not only as a data scientist, but also grown my opportunities to lead teams, build products, and turn whistful ideas about the potential of AI and machine learning in games into reality. Of course, those efforts were not alone. Yuan Cheng has somehow managed to drill the ability to stop and carefully examine an analysis into me, imparting skills in thinking critically and analytically about projects. Dylan Wang set me on the path to actually working on matchmaking systems that would eventually culminate in me presenting Blizzard's worldclass efforts to many audiences, converting a topic I had pitched as important for game AI in 2013 into reality by 2016. And Xiaoyang 'XY' Yang gave me the opportunity to deploy the first live machine-learning system touching millions of players I was involved with. And then there's my fellow Blizzterns. Shawn Baskin helped do the impossible: see my work realized in the game I have spent the most hours of my life in (*World of Warcraft*). Not only that, but then see that work recognized as a key factor driving design and business decisions and me signing boxes of the game for the public and Blizzard employees. Mabel Lin fostered the major avenues of my matchmaking efforts in *Heroes of the Storm*, giving me the chance to shape the redesign of the ranked play system and building the relationships that would support public dissemination of our work. Finally, there is our department director, Ken Drake: you can stop using 'Mr.' Zook—somehow you got me to make my thesis deadline happen.

Lastly, it goes without saying that the final push to realize my thesis would not be possible without Haram. I would likely have starved to death, not booked my defense flight (or hotel), nor stayed on task were it not for her support in the final year of writing up my thesis. And I cannot wait to add more weird citations to my CV when we start co-authoring in public policy and management.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Iterative game design is a process for refining the design of a game through a process of: (1) creating a base game; (2) playtesting the game to gather examples of people playing the game; (3) evaluating playtest outcomes to assess how well the game meets design goals; and (4) choosing a way to iterate on the game design to better achieve desired design goals. Developing computational models of this process holds great potential value for informing our understanding of iterative game design and automating aspects of this practice. In this thesis I develop a set of systems to automate the iterative game design process.

The central statement of this thesis is:

> Explicitly modeling the actions in games as planning operators allows an intelligent system to reason about how actions and action sequences affect gameplay and to create new mechanics. An intelligent system facilitates human iterative game design by learning design knowledge about gameplay and reducing the number of design iterations needed during playtesting a game to achieve a design goal.

I demonstrate *general game generation* through developing a modular, mechanic-centric representation for games across genres that allows a system to reason about how players are able to achieve a variety of outcomes. This approach enables a system to generate games given only a specification of success and failure criteria for a genre and a modular specification of the mechanics for a genre. To enable *general game playing* I apply Monte-Carlo Tree Search (MCTS) as a domain-agnostic game playing algorithm, using the computational bounds of the search as a proxy for varying human capabilities to play games. To *evaluate the space of play in games* I develop a taxonomy of four types of metrics for actions taken in games, showing how these metrics reveal strengths and defects in the design of two games to support differentiation among the general game playing agents using MCTS. These evaluations showcase how these metrics can reveal where games sup-

port differentiation of player skills through design, in turn demonstrating their utility for design evaluation. This evaluation approach for design iteration is further supported with *evaluation of the design space of a game* by generating a range of game design variants and evaluating hypotheses about how different design choices influence player behavior in terms of the action metrics. The range of design variants supports direct optimization to choose the best design variant to achieve a design goal. A system is also able to learn predictive models for how changes to game design features in a card game result in changes in how actions are used, as measured by the previous action metrics. Finally, I apply techniques from optimal experimental design to show how a system can choose new design variants sequentially to balance the trade-off between optimizing the quality of a design against a design goal and exploring alternative designs to seek out the generally best design. By comparing a variety of techniques across two design optimization goals I illustrate the general applicability of this approach to enabling efficient design iteration.

CHAPTER 1
INTRODUCTION

Artificial intelligence (AI) research has long sought to enable computational systems to reproduce human cognitive capabilities. Creativity is considered a distinguishing (if not unique) human capacity for generating novel ideas and artifacts. Among the many domains of creativity, games are uniquely interesting. Games are fundamentally interactive artifacts. In visual art, music, stories, or cooking audiences react to an artifact by forming some judgment of the artifact. In games, audiences must interact with the artifact itself, forming their judgment in response to this particular interaction.

Games thus pose a fundamental challenge to computational creativity. Game designers decide on the structure of a game among many possibilities, but any game design enables a breadth of possible ways to play and designers are typically interested in what exists in that space of possible ways to play.

For example, in the *Super Mario Bros.* games a designer can choose how high to allow Mario to jump, but cannot directly decide where players jump or whether it is possible for players to reach the end goal of a level and ultimately how players feel about this experience. More generally, designers are often interested in understanding what behaviors are possible given design decisions for a game. What strategies are possible (or impossible) in this game? What is the expected way players will interact with the game? Can players even win the game? Can players lose? These aspects of a game do not lie directly in the created game, but instead are secondary effects of the game's design. Player experiences, in turn, develop in response to how they may (or may not) act within a game.

Games also have value as a domain of study for their growing cultural and practical relevance. Games are increasingly considered more than an entertainment medium [17]. Designers use games to express ideas and experiences directly through shaping the ways

players may or may not act in a virtual world. Games are being used for training and educating: providing basic job training, drills for military skills, or lessons in math. As motivational systems games are increasingly used to help people achieve goals ranging from picking up healthy eating habits to sticking with workout regimes. Games have even been used as means of advertising and tools in political election campaigns.

Despite these widespread applications, our understanding of how games work and our tools for designing games remain primitive. The bulk of scientific knowledge of game design to date draws from behavioral psychology studies [104]. These works highlight how game systems of rewards can shape behavior. Yet this knowledge falls far short of understanding how game designs shape the ways people may play a game. Our tools for designing games are similarly limited. While tools for creating the *content* in games have grown increasingly powerful—from sophisticated 3D modeling tools to complex music composition systems—our tools for *designing* games (with some exceptions discussed later) have advanced little since digital games first became common in the 1980s. That is, we still lack tools that enable designers to understand what behaviors a choice in game design may enable. Lacking such tools, there is little opportunity for designers to build knowledge of how their game works and ultimately generalize and share this knowledge to improve the practice of game design as a whole.

Why enable computers to participate in iterative game design? Building computational systems that can design games and acquire game design knowledge stands to benefit practitioners in a number of ways. Computational systems can help designers understand how different design decisions will influence player behavior in a game by simulating player behaviors. Using this knowledge, people can better navigate a range of design decisions by comparing the expected impact on game players. As these computational systems observe people designing, they can learn how designs influence people. This design knowledge can then be offered to people to guide them when making design decisions and even be used to highlight design alternatives people may inadvertently overlook. As computational systems

learn from many design cases they can use this knowledge to help people more effectively iterate on designs, enabling designers to create games faster and explore new design alternatives. At an extreme, computational iterative design systems can even automate the process of creating games, yielding new types of games.

In this thesis I present computational systems that model aspects of an iterative game design practice. To date, most computational game design work has emphasized models of the first step in iterative design—creating a game of interest—while ignoring or downplaying the role of evaluating the game and using those evaluations to refine the game design while learning about that type of game as a whole. I develop models of iterative game design with a specific focus on the problem of evaluating and refining a game design to drive player behavior. I show how computational systems can follow human processes for creating games and demonstrate the potential these systems have for augmenting the practice of game design. By developing these systems I fill gaps in our understanding of the process of designing games, addressing methods for genre-agnostic creation of games, genre-agnostic generation of behavior in games, general metrics for analyzing strategies in games, and a general method for efficiently iterating on game designs.

1.1 Iterative Game Design

Game design, like all creative processes, can take many forms. The game design literature is home to a breadth of advice on everything from techniques for conceptualizing a game to best practices for paper prototyping (that is, testing a simplified version of a game using paper, die, &c.) [61, 69, 110, 167, 171]. While much of this advice conflicts, there is a general consensus around the value of an iterative game design process [69]. *Iterative game design* emphasizes a process where designers have people play their game, evaluate how people play the game against design goals for the game, and use that feedback to make a change to the game (Figure 1.1). This process repeats until the game eventually accomplishes a designer's goals in terms of how people play the game. Iterative design

stands in contrast to waterfall design methodologies that emphasize extensive pre-planning and development before testing the game with players.

Iterative game design is considered a best practice among game design practitioners that is used to facilitate the creative process of game development [69]. First, create a (potentially simplified) game of interest. This game is designed to enable players to perform basic desired behaviors in the game. At this time the designer will have one or more theories about how the design will shape player behavior. Second, test the game with players to gather information on how people play the game. Playtesting provides the designer with examples of how people may interact with the game, informing their understanding of what behavior is possible in the game. Third, evaluate those example behaviors against gameplay goals. Designers can evaluate the behaviors seen in the game to decide if desired behaviors are occurring and may also check for emergent behaviors that may be of interest. At this time the initial design theories will be tested and refined as needed. Fourth, iterate on the game design by picking the most promising candidate (set of) change(s) to the design intended to better achieve gameplay goals. Iterations typically use evaluations of playtests to choose which changes are most likely to improve the design in terms of gameplay goals. Summarizing, iterative game design is a process of creating a game, playtesting to get user behavior, evaluating playtest outcomes, and iterating to choose the most promising changes to make to a design.

In this thesis I present systems that model this iterative game design process. To date, most computational game design work has emphasized models of the first step in this process: creating a game of interest. These efforts strive to enable computers to create games of a particular type, generally guided by some notion of what makes those games high quality. Rather than enable computers to evaluate the games they make, these systems use hand-crafted notions of quality that eliminate the need to iterate on a design. Relatively little effort has gone into understanding how a player might interact with a game to inform game design changes. This work addresses these gaps by explicitly targeting the full pro-

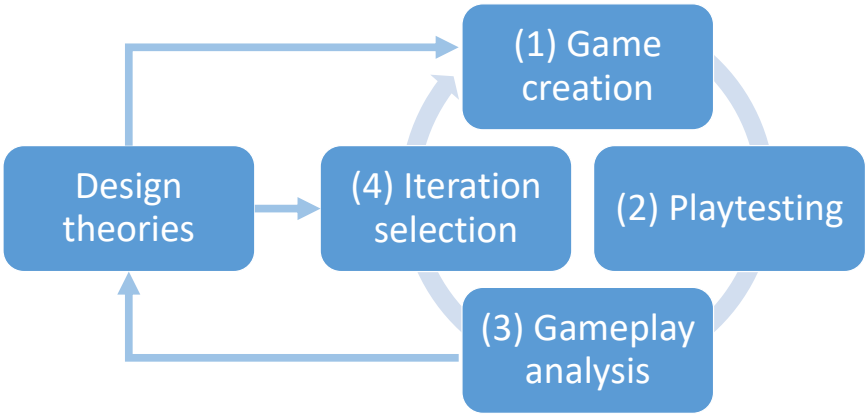

Figure 1.1: Iterative design process (for games) schematic.

cess of iteration, with a focus on the problem of understanding how players respond to a game and ways to use that information to guide understanding and further refining a game design.

1.1.1 Game Generation

A core requirement for any computational game design system is the capacity to generate games. Generating a game requires some means of representing the structure of a game in computational form. These representations for games define the *design space* of games a system may create. By defining what kinds of games are represented we define what games a system is capable of generating. Any representation must make trade-offs between the range of games possible and the computational costs of searching for good games within that design space.

Work in the area of game generation has developed systems that are able to represent and generate games in a variety of genres. Representations have been developed for card

games [65], 2-player board games [20], strategy games [123], adventure games [126] and subsets of 2D arcade games [193, 219]. While powerful as ways of enabling a system to be creative within a given game genre, these models tightly constrain the range of games a system may create. By favoring tractable design spaces these models limit our ability to understand the more general aspects of game design inherent in combining elements across genres or defining new genres altogether.

As an alternative I explore the notion of generating games in a way that is agnostic to the specific game genre by developing a representation that captures a broad class of discrete, turn-based games. Discrete games are those that do not use real values (floating point values), imposing a level of coarseness characteristic of a paper prototype or tabletop game. Turn-based games impose the constraint that actions do not occur in real time. These two constraints enable the system to generate games capturing a broad spectrum of tabletop games (board games, card games, or role-playing games) while also serving as a simplified model for games requiring continuous variables.

I take the approach of representing the game based around a model of game mechanics. Games can be broadly broken into mechanics and content. Mechanics define the actions possible in the game, serving as the rules for the game. Content captures the various assets used in a game, ranging from level structures to visual art.

I show how to generate game mechanics and level content across a variety of genres using this mechanic-centric representation. The system is able to reason about how mechanics enable players to achieve a variety of outcomes and uses this information to generate games that ensure winning conditions can be met. The representation is modular, enabling the system to combine genres and generate mechanics and content appropriate to these genre blends. By providing a genre-agnostic approach to generating games I provide a key step toward general game generation systems.

1.1.2 Playtesting

Any given game design affords a plethora of ways to play. The *play space* is the set of ways a player may behave in a game. When creating games a designer typically has some goals for the play space afforded by a game: the ways players may win or lose the game or the kinds of strategies players may take in a game. Designers (human or computer), however, are only able to choose a design from the design space. The challenge of game design lies in choosing designs in a way that shapes the play space of a game in desired ways.

Behavior sampling is the problem of gathering examples of behaviors from the play space. The most common approach to behavior sampling is simply to test the game with people. Playtesting with people provides clear evidence that (at least some) people will actually play the game in a specific way. As long as the playtester population is similar to the population of people intended to play the game, this can provide useful guidance on the actual play space people will use. In this work I use human playtesting as a base case to evaluate design iteration, presented in detail below.

Human playtesting, however, has a number of drawbacks. Running playtests with people can be time consuming. People may not sufficiently try alternative ways of playing the game, missing strategies (or bugs) that a larger group of players might find. Playtesters may not cover the full range of diversity of players of the final game.

How can computational systems address these limitations of human playtests for behavior sampling? One option is to use a small enough (or abstract enough) play space that it is feasible for a computer to enumerate all possible behaviors from the space of play [192, 206]. This approach enables a designer to have guarantees on the properties of the play space, by explicitly checking whether desired behaviors are present (and undesired behaviors are absent). By fully exploring the space of play, a system can cover the possibilities available to any group of players—this approach is used by the game generation system above. This comes with the caveat that there is no guarantee people will actually discover the ways of playing found through exhaustive enumeration. Even if a game *could*

be won, it does not mean that will be possible for any person to accomplish.

In many cases, however, exhaustive exploration of the play space is infeasible: the game may be too complex to simplify in a way that maintains fidelity to core elements of the design or the time required to search all ways to play is too great. An alternative approach is to simulate the ways people play the game to get a sense of the range of possibilities in the game. Simulation-based models of play enable a designer to get examples from the play space, limited primarily by their fidelity to human play.

I address this gap by developing a general approach to simulating play in discrete, turn-based games. This approach captures aspects of human skill through adjustable parameters of the simulation AI agents—these parameters allow alterations of agent planning capabilities to model a range of skills, from weak to strong game players. This work provides a general approach to sampling behaviors in a wide variety of game genres without custom algorithms for each game. At the same time, this approach proxies the notion of player skill, enabling the system to provide examples of how gameplay may differ between amateurs and experts in the game.

Behavior sampling provides examples of behavior to support evaluating a game. Designers typically come to a game with expectations for how players will play the game. Beyond simply being able to complete a game, designers may desire that certain strategies are possible in a game, or that various actions are balanced so that no single action is favored to the exclusion of alternatives.

Game analytics is a field dedicated to studying how people play games and using that information to inform game design [176]. Game analytics techniques enable designers to understand different types of players in games, summarize the outcomes of players playing, or predict player behavior including quitting or purchasing goods. These techniques provide designers with an understanding of possible behavior in a game when faced with vast amounts of play data from a running game.

Game analytics methods to date have primarily been concerned with understanding the

game *states* occupied by players, with relatively little attention given to *actions* taken by players. Modeling the state space of a game provides insight into what parts of a game are used, or what players may be expected to typically achieve. Modeling the play space of a game, however, requires analytics on the actions taken by players. Analysis of play can yield a deeper understanding of the strategies players employ (or lack thereof) and help guide changes intended to shape the play space of a game.

I develop a taxonomy of metrics for play actions to study the play strategies enabled by a game. *Summaries* are high-level metrics that summarize play behavior in a game; e.g., the typical number of actions taken in a game. Summaries provide a basic understanding of the larger structure of the play space of a game. *Atoms* are the frequency of individual actions used in a game, grounding an understanding of what actions are taken (or not). Atoms help determine to what extent actions in a game are balanced, by assessing how well the frequencies of actions align with those intended by a designer. *Chains* are common action sequences in a game, covering both typical ways of chaining together actions by one player and typical ways one player responds to the actions taken by an opponent. Chains enable designers to discover both expected and unexpected patterns of play in a game. *Action spaces* are metrics on the number of actions available to players (or taken by players) over the course of a game. Action spaces help designers understand whether a game supports an intended degree of breadth over the duration of a game. The metrics I develop enable designers to compare a design to design goals for a game, understand the kinds of strategies enabled by a game, and potentially discover unexpected patterns of action in a game.

1.1.3 Game Evaluation

Game evaluations serve two ends for designers: (1) discovering designs that best accomplish design goals and (2) providing knowledge about how a game design works. Discovering the best design in a design space requires a system to evaluate a breadth of designs and select those that best achieve design goals. I show a system that systematically generates a

wide variety of game design variants in a card game and uses behavior sampling and action metrics to evaluate these designs. The system can then choose among these designs to find designs that enable desired player strategic opportunities.

Evaluations of a game give information not only on that game, but also on similar games that share parts of their design. Designers begin a design process guided by a number of assumptions about how design choices will influence player behavior. This design knowledge shapes the initial design and is subsequently altered over the course of playtesting as designers learn more about how the design works with real players in practice. In this work I show a batched approach to acquiring design knowledge using the action metrics above. After generating a set of designs with minor changes to design features, a system uses behavior sampling to explore the space of play of each design variant. Design knowledge is learned by evaluating gameplay metrics while comparing design variants, yielding knowledge of how changes to a design change player strategic behavior. This work showcases how a computational system can gather knowledge about a design that can inform human designers and guide automated generation.

1.1.4 Iteration

In many cases design will not use simulated behaviors, but require human playtesting. Iteration in these cases is challenging due to the cost of playtesting each design considered, leading to a fundamental tension. On the one hand, it is valuable to explore very different designs to better understand the range of possibilities in a design space. On the other hand, it is also important to refine a design by considering similar, but slightly different, variants within the design space. The tension between exploration and refinement to minimize the cost of playtesting is a core problem for iteration.

This problem is not unique to game design: medical researchers (and scientists in general) are faced by the same challenge. Specifically, medicine often needs to study how a drug or treatment might help (or hurt) people. Testing a drug is expensive and risky—

researchers would like to design the optimal experiment (or set of experiments) that minimizes the number of patients tested before a conclusion can be drawn about the efficacy of a drug. This problem of *optimal experimental design* is a broad topic widely studied in statistics.

In this work I demonstrate how techniques from optimal experimental design can be used to choose design iterations based on playtest data. I study the case of online changes to a design: a game is deployed, data on how a small number of people play is collected, and a change is made to improve the game. I compare a number of models from optimal experimental design that make different trade-offs between exploring possible designs and refining the highest quality designs to explore different approaches to iteration. Through simulations and human studies I show these methods can reduce the number of playtests needed to achieve a design goal, opening new avenues for the application of computational methods to iteration.

1.2 Thesis Statement

The central statement of this thesis is:

> Explicitly modeling the actions in games as planning operators allows an intelligent system to reason about how actions and action sequences affect gameplay and to create new mechanics. An intelligent system facilitates human iterative game design by learning design knowledge about gameplay and reducing the number of design iterations needed during playtesting a game to achieve a design goal.

In the following chapters I will discuss the systems that realize this thesis statement.

In chapter 3 I demonstrate *general game generation* with a modular, mechanic-centric representation for games across genres that allows a system to reason about how players are able to achieve a variety of outcomes. This approach enables a system to generate games

given only a specification of success and failure criteria for a genre and a modular specification of the mechanics for a genre. To *simulate players to explore play spaces* I apply Monte-Carlo Tree Search (MCTS) as a domain-agnostic game playing algorithm, using the computational bounds of the search as a proxy for varying human capabilities to plan ahead when playing games in chapter 4. To *evaluate the space of play in games* I develop a taxonomy of four types of metrics for actions taken in games, showing how these metrics reveal strengths and defects in the design of two games. These evaluations showcase how these metrics can reveal where games differentiate among players of differing skill through design, in turn demonstrating the value of action metrics for design evaluation. This evaluation approach for design iteration is further supported in chapter 5 with *evaluation of the design space of a game* by generating a range of game design variants and evaluating hypotheses about how different design choices influence player behavior in terms of the action metrics. The range of design variants supports direct optimization to choose the best design variant to achieve a design goal. A system is also able to acquire design knowledge in the form of how changes to game design features in a card game result in changes in how actions are used, as measured by the previous action metrics. Finally, in chapter 6 I apply techniques from optimal experimental design to show how a system can choose new design variants sequentially to balance the trade-off between optimizing the quality of a design against a design goal and exploring alternative designs to seek out the generally best design. By comparing a variety of techniques across two design optimization goals I illustrate the general applicability of this approach to enabling efficient design iteration.

Together these systems support the thesis statement claims for modeling an iterative game design process. The next chapter provides a background on efforts in game generation, game analytics, and computational creativity as context for this work. The following chapters address each topic in turn: genre-agnostic game generation, human-like play sampling, game design evaluation, and efficient game iteration.

CHAPTER 2
BACKGROUND

Building computational iterative game design systems requires modeling the intertwined component processes involved in iteration. *Game generation* is required to create games of interest to iterate on. *Behavior sampling* is needed to gather information on the space of play enabled by a design. *Game analysis* is required to convert raw examples of behavior into an understanding of how well player behavior aligns with design goals. *Design iteration* is needed to choose how to proceed to the next game design during the iterative design process. More generally, iterative design is a creative practice that relates to broader concerns in *computational creativity* around how computers can be creative. In this chapter I review work in these areas as background for the contributions made in this thesis.

2.1 Game Generation

Game generation is the problem of creating a game from a description of what constitutes a valid game. Game description languages (GDLs) are representations of game domains to enable game generation. Procedural content generation (PCG) is a related area devoted to algorithms for synthesizing the content in games, such as visual assets, music, or level designs [180]. PCG is directed at elements of the game domain that are traditionally assets in a game engine; game generation also includes the behavior (rules) of the game that is traditionally governed by the game engine itself. A game generation system uses a GDL to define a search space of possible game designs to consider, thus coupling the choice of domain representation to the creative range of the generator. A central problem for any GDL is balancing between the power to express a broad range of domains of interest while remaining tractable to use for generating games. Researchers have studied game description languages primarily for three applications: (1) general game playing, (2) game

authoring support, and (3) automated game generation. While the former two approaches often yield overly expressive games that prohibit direct generation of games, the latter have often been too limited in scope to support generation of games across a variety of domains. In this work I bridge this gap through a representation that is able to capture a broad range of domains with low-level generative control. This effort, however, comes at the cost of limiting the scale of the games being generated to smaller puzzle-like games. Further, there remain open questions around generation for genres premised on large, complex content such as open-world 3D games or games with large numbers of players competing with one another.

2.1.1 General Game Playing

General game playing (GGP) researchers study how to create AI agents able to play games in arbitrary domains. GGP research strives toward algorithms that capture general AI capabilities by emphasizing the ability to use the same AI agents to play games in many different domains. To facilitate creating these agents GGP researchers have developed description languages for shared primitive language elements that can be used to create a variety of games with different challenges.

The Stanford Game Description Language [121] models turn-based, competitive games in a declarative language. This captures a broad class of adversarial board games including examples like chess and checkers. Extensions to the language have introduced randomization of events in games and agents with incomplete information about the world state [215]. These extensions allow for capturing card games such as Poker, where players do not have direct knowledge of which cards the opponent possesses. The Stanford GDL has been used in an ongoing series of AI competitions and has facilitated developments in AI agents that can play this class of games.

Video game AI researchers, however, have often been interested in the computational challenges associated with traditional video games. The Video Game Description Lan-

guage (VGDL) [169] was developed to better model arcade-style 2D games—popular examples include *Frogger* [109], *Space Invaders* [212], and *Pacman* [138]. The VGDL codifies a space of games that allows for continuous movement and physics, creating and destroying game entities, win/loss states, and a variety of other elements common to arcade game design. Recent competitions have begun to use the VGDL to test general game playing agents in this new environment that is less amenable to logic-based reasoning [153].

These efforts highlight the potential for representing broad classes of games in a way that is amenable to automated game playing. These languages, however, are of limited use for automated generation due to their emphasis on very low-level representations of game domains. An emphasis on low-level representations creates a vast space of possible games with only minor differences between them. This broad space creates creates a computational bottleneck for any search algorithm navigating the space of possible designs within these languages. Generation in these spaces only becomes feasible by further constraining the parts of the language used, effectively trading off the expressive power of the language for computational tractability [142, 127]. Adding further constraints on the language is undesirable as it limits the range of games possible in the language.

These languages are often not modular: they do not support the ready recombination of elements from different genres [140]. Modularity helps languages address an expressive goal of supporting the common creative process of mixing elements across genres. While the Stanford GDL and VGDL can both theoretically model games across many genres, their lack of representational modularity stymies practical uses of these description languages to generate games in a domain-agnostic fashion. Ideally a representation tailored to generating games should support combining domains while also enabling direct search for new kinds of games in the domain.

2.1.2 Game Authoring Support

Developing games is a challenging task, particularly for non-programmers who lack the ability to author code that realizes their goals for game systems. To address these needs researchers and game developers have created a variety of game authoring tools that embed specialized languages for authoring game content and behavior. These languages vary in their capabilities to support authoring, ranging from tools simply intended to provide a high-level language for describing content to systems that support running simulations or checks on how the game functions. Unlike game playing description languages, these tools are typically developed to facilitate the creation of games that are directly playable by people, rather than used as abstract interfaces for computer agents.

Authoring tools for interactive fiction—text-based games where players make decisions influencing the course of a story—have become popular as introductions to game design that remove the need for skills in creating visual art, music, or complex game systems. Twine[1] provides a simple visual interface for creating branching narratives and retaining some state variables. Inform7[2] enables English language-like authoring with more advanced capabilities including parsing text-based input. ScriptEase II [172] supports authoring stories using a graph structure to connect plot points and allows exporting stories to different game engines.

AI researchers have developed a number of systems that formalize social norms and interactions to enable authoring social systems. *Façade* [128] uses a specialized programming language—A Behavior Language (ABL)—for reactive planning to control how in-game agents respond to player choices, including behaviors that coordinate multiple agents. The Kodu AI Lab [67] provides authoring support for social game mechanics—attitude, learning, and fuzzy reasoning—in the Kodu Game Lab authoring tool. Versu[3][63] is a storytelling platform that provides a rich social model of small-group interactions, capturing

[1] `http://twinery.org/`
[2] `http://inform7.com/`
[3] `http://versu.com/`

abstract, context-specific and universal individual motivations; emotions; beliefs; social norms; and inter-personal relationships using a novel logical formalism. *Prom Week*[131] includes a "social physics" model that allows authoring individual character traits, feelings, and relationships and enables characters to form intents, take actions, relate to a shared cultural space, and remember and refer to past events. While these tools enable authoring content they provided very limited feedback on how authoring choices may alter potential player behavior in the games.

Authoring tools have also been developed to capture other broad classes of games. Puzzlescript[4] was designed to facilitate creating puzzle games, using a grammar-like front-end language. The EGGG [149] and Ludi [20] both model a class of adversarial board games. The Machinations [2, 55, 57] and micro-machinations [107, 162] frameworks were designed around modeling arbitrary game economies using Petri Nets as an underlying model and visual authoring tools. Machinations support running simulations of to allow authors to visualize potential runs of a game. BIPED [195] supports authoring game prototypes using the event calculus, allowing for logical checks on possible playouts by using model checking to test whether particular states may hold in the game. Gamelan [150] supports authoring a variety of turn-based board and card games along with the ability to author game 'critics' that check for the presence of behaviors in games. Potential behaviors are checked using static analysis of the game code or dynamic analysis based on pre-authored agent behavior patterns. Ceptre [126] models a similar class of games with a grounding in linear logic, supporting checks on game playouts using proofs from the game's core language. Gamika [157] models a class of simple 2D, physics based games using numeric vectors. These authoring tools readily support human authoring, but have been of limited value for game generation due to their great expressive power—something of value to human creators but an obstacle to computational exploration of the space of games possible with these tools.

[4] `http://www.puzzlescript.net/`

2.1.3 Automated Game Generation

Game generation systems require a GDL that can be efficiently searched for valid games while still capturing a wide variety of possible games. Definitions of a complete game vary from a specification of game rules to requiring full running code and choice of visual and auditory assets. For the purposes of this work I will focus on running code specifications of game entities and rules, without the need for selection of visual, auditory, and other game representational assets. While choosing representational assets is a central component of game design, the work in this thesis is primarily concerned with the mechanics of games (and the spaces of play they enable), rather than these aesthetic elements.

Approaches to game generation have broadly been classified into three types: constructive, search-based, and constraint-based [180]. *Constructive* game generators assemble content according to a predefined grammar of elements; grammars are defined in such a way that all generated games are valid [178]. *Search-based* generators iteratively enumerate games from the description language, evaluating generated games against a continuous evaluation metric and using these outcomes to guide search [220, 222]. *Constraint-based* generation techniques employ a form of model checking to guarantee that games generated from a specification do or do not have certain properties [141, 197]. These methods vary in how generation occurs, but share a common need for domain definitions to use in generation. To date, these models have emphasized assembling elements of game content, eschewing generation of the actions possible in the game. The work in this thesis addresses this gap by providing a low-level representation for game mechanics that can be algorithmically generated to create new game mechanics.

Several GDLs have been developed to model a desired genre of games. By focusing on a domain of interest GDL authors can tailor the language to capturing the nuanced systems and gameplay of that genre and develop ways to check generated content to appropriately match genre norms. Efforts in this vein have developed languages for strategy game units [123], card games [66], adventure game puzzles [50], 2D game bosses [188],

simple 2D physics-based games [157], and role-playing game actions and settings in 2D dungeons [115].

Alternatively, researchers have explored more general representations intended to capture a range of game genres, primarily board games and arcade games. Ludi [20] uses search-based generation to generate games in a class of 2-player, adversarial board games. Games generated by Ludi have subsequently been published, demonstrating the efficacy of this approach.

Early work in arcade game generation used fixed sets of potential definitions for rulesets that generated *Pacman*-like games that varied in agent behavior for collisions. These efforts included search-based approaches that evaluated games for the ability of playing agents to learn to play [219] and constraint-based methods that validated game playability (that is, the ability to complete game objectives) [193]. Game-O-Matic [226, 225] generates a broader set of arcade games using a constructive approach with subsequent filtering. Game-O-Matic is unique in taking as input a human author's definition for the intended semantic message for a game—represented using a concept network—and using this to guide choices of game content and rules.

Other efforts have semi-automated generation by enabling automated search within a prescribed design space. Powley et al. [157] tweak simple 2D arcade game levels using an automated player and a set of heuristics. Danesh [40] searches a space of generated content using pre-defined metrics for what space to search and how to evaluate the generated content. These efforts show initial efforts to generalize procedural generation techniques by plugging into user-specified generation tools or evaluation functions—my work applies this philosophy to game generation, playing, and evaluation in a generic manner.

ANGELINA [38] generates 2D platformer games using a combination of generation techniques. A search-based approach evaluates playability based on completing game levels and using game systems. A constructive method chooses game assets based on a human author's input phrase, using this to guide online searches for game content including

visual and auditory components. This approach has been extended in later iterations of ANGELINA [36, 37] to support 3D walking game generation; the approach based on taking input phrases has enabled ANGELINA to participate in the Ludum Dare[5] game jam, including being rated by other human creators.

GameForge [83] combines multiple approaches to generate computer role-playing games similar in style to the *Final Fantasy* [204] series of Japanese role-playing games. GameForge is unique in personalizing games to individual player preferences on plot and world layout [52]. A planning algorithm modifies an input story (represented as a series of connected pre-authored plot points) in response to player choices to include or exclude plot points. A search-based approach lays out the game world to embed plot-relevant locations while evaluating world structure for meeting player-given preferences for world size. Constructive methods are used to script the behavior of in-game agents to follow the plot.

Nielsen et al. [143, 142] have made initial efforts to generate games using the VGDL using search-based techniques. Games are evaluated by comparing how well agents of differing game playing capabilities fare in generated games [143, 156]. Generation proceeds either by starting from a seed example game that is altered or by random initialization from a more constrained subset of VGDL. Nielsen et al. demonstrate the potential for general domain generation and evaluation, an effort continued recently by Khalifa et al. [106] for general level generation using the VGDL.

Game generation from a GDL is a top-down technique: a system starts from an abstract definition for a domain and its elements to specify concrete entities in the abstract definition. *Mechanic Miner* [39] takes an alternative, bottom-up approach to mechanic generation. Rather than define a domain that can be converted into code, *Mechanic Miner* directly manipulates game code using program reflection. This approach directly exposes (a subset of) the materials of the game engine to a game generation system, rather than depending on a domain author's conception of how game structures should function. Di-

[5] `http://ludumdare.com/compo/`

rect code manipulation affords nuanced changes to a game's systems (here the mechanics, rather than full game code), but massively increases the design space an AI technique must navigate. *Mechanic Miner* illustrates the potential for low-level control of game systems while exposing the challenges of directly altering game engine code.

The work in this thesis takes an approach to game generation that generates game while functioning directly as the game engine. The model I use can be directly played, bypassing the need for authors to create snippets of code that realize high-level logical specifications. Working at the low level of game primitives that define elements of the game state provides nuanced control over the possible mechanics and levels in a game and enables the use of model-checking to verify game properties. By providing explicit control over the properties of generated games (e.g., winning conditions or the number of actions needed to complete levels in a game) my work presents an approach that bridges between human-readable expressions of design goals and guarantees of game properties.

2.2 Behavior Sampling

Game generation provides a functional game that meets a set of design specifications, but design iteration requires understanding the space of play afforded by a game to inform changes to the game design requirements. Behavior sampling addresses the problem of providing examples of player behavior—often called playtraces—from a play space. Samples of player behavior allow evaluating games in terms of the behavior they allow (or do not allow), in turn supporting iteration on the base game. Two approaches to behavior sampling are common: *exhaustive* techniques that provide information on all possible behaviors in a game and *simulation* techniques that provide samples of behavior from a space. Exhaustive approaches offer the benefit of capturing features of the entire space of play, but come at a steep computational cost. In many games the space of potential behaviors is too complex to exhaustively explore, leading to the need for simulation techniques. Simulation approaches offer the capability to play a game in a way intended to represent

particular types of behavior. In particular, believable agents are simulations that are designed to replicate aspects of human behavior as a way to provide playtraces that resemble those humans could be expected to provide. My work uses both approaches to show how exhaustive verification can enable cross-domain authoring while simulations can function across domains to sample playtraces in highly complex games.

2.2.1 Exhaustive Behavior Sampling

Exhaustive behavior sampling methods provide theoretical guarantees that certain properties hold for a game's space of play. When games (or their relevant subsystems) can be fully represented in a logical language, forms of model checking or logical proof can be used to validate properties of those games [139]. Smith et al. [22, 24, 191, 192, 196] have used Answer Set Programming to define abstractions of discrete game systems. With this approach, an answer set solver can use model checking to guarantee that a generated set of game content has playtraces with desired properties, and even guarantee that undesirable playtraces are not part of the play space. A similar approach (with a different logical formalism) is used in the computational critics in Gamelan [150]. Ceptre [126] uses a language based on a modification of linear logic and examines traces by instead generating logical proofs about the game in question.

In some cases the combination of increasingly powerful contemporary computing resources and efficient storage techniques enables brute force enumeration (and storage) of all possible playtraces in a game. Sturtevant [206] has shown how breadth-first search is a viable way to enumerate all possible ways of playing a puzzle game.Using this approach allows design analyses of all possible solutions to puzzles. By using this analysis across a number of puzzles with different structures it is possible to query potential playtraces that employ certain subsets of the possible mechanics in the game.

Adversarial games potentially allow for the direct application of results from the field of game theory. Jaffe et al. [99, 100] used analytical solutions from game theory to study bal-

ance in two-player, simultaneous move, adversarial games. Game theory allows studying game balance (under the assumption of optimal play) at equilibrium, showing theoretical limits to how a game is balanced. Jaffe et al. combine game theoretic solution analysis with putting restraints on the actions or reasoning abilities of an agent to study the balance of these adversarial games. Game theory thus provides an important understanding of the strategies possible in a playspace (rather than properties of single-player playtraces).

Exhaustive sampling methods have shown to be powerful tools for gathering examples and proving properties of a game's play space. These techniques, however, are often limited to abstractions of a game or representations of a subset of a game's systems due to the need to examine the full space of alternative plays of a game.

2.2.2 Planning

Most exhaustive sampling approaches share a grounding in logical models that exhaustively explore the space of possible outcomes. Planning representations were developed to scale traditional AI techniques to complex domains by providing additional problem structure knowledge to AI search processes. In planning, a problem is broken into a domain defining the actions possible in the world and properties of states of the world. Given an initial configuration of the world and desired final world configuration, a planner is asked to find a sequence of actions that transitions the world from the initial to the desired state. Plans are often evaluated in terms of various desirable criteria such as their length or the states visited along the path from the initial to the final world state. Planning representations can often be converted to other representations to improve the performance of other approaches through additional representational factoring (e.g., converting plans to SAT problems, part of the shared language of many model checking systems) [164].

STRIPS [64] was one of the earliest planning representation languages, modeling actions ("operators") with logical predicates. Logical predicates are used to represent the state of the world. For example, *at(Alex,home)* would represent Alex being at home as a

specific instance of the more general *at(A,P)* predicate used to represent an agent, *A*, at a place, *P*. Operators represent actions in the world. Operators have a set of preconditions that must hold before the action can be executed and a set of effects that add or delete predicates from the state of the world. Planning is a process of finding a sequence of operations that transform the world from an initial state into a state in which the goal situation holds.

The Planning Domain Description Language (PDDL) [132] is an ongoing project to extend planning representations to address more complex tasks while building a shared language for research competitions. PDDL extended STRIPS-like representations with non-equality constraints, numeric fluents to model continuous domains, operators with duration, and timed initial literals that modify the world state at fixed times regardless of agent actions. PDDL has since added features such as a type system and requirements on state trajectories (states passed through when executing a plan); separation between agent-initiated and environment-initiated operators (PDDL+); multi-agent domains (MAPL); and probabilistic effects and continuous rewards (PPDDL) [73]. These planning representations demonstrate the value of providing rich representation languages for problem domains as a way to exhaustively (or efficiently) find solutions to specific problems.

In my work I use a modified planning language to solve game levels using a set of game mechanics. On this analogy a game level starts a player in some state and asks the player to reach an ending state. By using planning I show how to enable agents to test whether levels can be beat (subject to constraints on what states the agent visits while solving levels) in a domain-agnostic fashion. Unlike planning research, I *generate* the operators used—the actions available to the player. This approach supports explicit generation of games based on the actions allowed in the game while ensuring that logical checks can verify properties of the generated games. Using a planning representation supports both the generality of this approach and potential for greater efficiency compared to brute-force search or raw model-checking through factored planning representations.

2.2.3 Planning in Games

Planning algorithms have often been used to control adversarial agents in commercial games. Orkin [147] introduced the application of planning technologies to commercial games in the first-person shooter *F.E.A.R.* [135]. STRIPS-like planning proved valuable in decoupling agent actions and goals to allow for more dynamic variations in enemy strategies compared to contemporary scripted approaches. Bartheye and Jacopin [9] extended this line of thought to using PDDL in games, providing more flexibility to linking agent decision-making to varied game environments. In these cases planning provides a core framework that can efficiently drive agent behavior in a variety of domains.

Moving beyond single agent control, more sophisticated planning techniques have been used to control groups of agents in game domains. Hierarchical task networks (HTNs) are a planning technology that abstracts individual states and actions into macro components to facilitate planning speed. Hoang, Lee-Urban, and Muñoz-Avila [87] and Gorniak and Davis [77] addressed challenges of scaling agent planning by applying HTNs to coordinate behavior of squads of agents. HTNs allowed the game agents to coordinate behavior and scale to more complex game scenarios that possible with simpler algorithms like STRIPS.

Together, these efforts demonstrate the viability of planning as a technique to efficiently control agent behavior in games. The work in this thesis develops a modified STRIPS-like representation to enable game agents to test game states, using the generality of planning algorithms to search a space of ways to play a game.

2.2.4 Simulation-based Behavior Sampling

Simulation-based behavior sampling allows non-exhaustive collection of playtraces to check aspects of a game or capture notions of expected player behavior. Search-based game generation techniques commonly use simple agent simulations to check whether games may be beaten. But simulations can do more than verify whether an agent can complete a game by providing examples of expected player behaviors that can be subsequently processed

to analyze the game. Togelius and Schmidhuber [219] use a learning agent to assess how important learning is to playing a game. Browne and Maire [20] use adversarial agent play to assess a variety of qualitative metrics on game aesthetics. Nielsen et al. [143] compare the level of success of agents with varying capabilities in games to assess game quality in terms of rewarding players for greater skill. Bauer et al. [10, 11] use rapidly-exploring random trees (RRT; a randomized search technique) to generate graphs of the connectivity (that is, the playspace) present in game levels. Using these graphs authors may specify constraints that add or remove reachability between positions in game levels, achieved by optimizing level layout to provide desired playspace properties. Tremblay, Borodovsky, and Verbrugge [227] compare A*, RRT, and Monte-Carlo Tree Search (MCTS) for solving platformer game levels, showing differences in the types of playtraces generated by different techniques. Tremblay, Torres, and Verbrugge [228] show how RRT can be used to simulate stealth game movement and combat, providing diagnostic information on possible solution paths in game levels. Isaksen et al. [95, 96] simulate human reaction and perception speeds to simulate play in a large space of variants of *Flappy Bird* [58] (an arcade-style game emphasizing reflexes), using analysis of agent survival rates to compare design variant difficulty. These simulation methods use agents as a way to sample example player behavior in a game to approximate properties of the play space.

Simulations can also encode models of human-like behavior to direct sampling toward the most valuable information for design. In these cases a model of agent behavior in a game uses human data to adjust the agent to fit human behavioral patterns. For movement behavior, Cenkner et al. [27] model human hiding and seeking behaviors in 2D environments in terms of selecting and moving to locations with adjustments made based on player data. Tomai et al. [224] model movement in open-ended domains using path-relative recursive splines that fit to player deviations from optimal movement toward a goal. Ortega et al. [148] train a variety of agent controllers in *Super Mario Bros.* to either directly mimic player actions in a context or indirectly maximize a playtrace similarity metric. van Hoorn

et al. [231] apply multi-objective evolution to racing agent behavior to optimize both for in-game performance and similarity to human playtraces.

Other work has explored human-like models for action selection behavior in a variety of genres. Laird and Duchi [112] parameterize a cognitive architecture (Soar) when playing a first-person shooter game and evaluate which parameters are most important to human judgments of human-likeness and objective measures of agent performance in competition. Holmgård et al. [88, 89, 90, 91] address activity in turn-based action role-playing games through *procedural personas* that mimic human goals in a game. Personas are trained to maximize designer-defined notions of utility (such as finishing a level quickly or slaying all the enemies in a level) using reinforcement learning [89], neuroevolution [88, 90], or Monte-Carlo Tree Search [91]. Young and Hawes [242] transform continuous data on actions taken in a real-time strategy game into a qualitative, symbolic representation and train several classifiers to choose actions similar to player traces. Chang et al. [29] model player behavior in making and accepting or rejecting offers in a social ultimatum game using data from distributions of human actions. Tomai and Flores [223] model player behavior in a game using behavior trees and adapt trees to match human action selections in given situations.

While the models used vary, these approaches share a common technique of converting desired human behaviors into a parameterizable model (e.g., reward weights or unit movement instructions) and using human playtrace data as a comparison to agent behavior. The work in this thesis builds off this approach by addressing how MCTS can model general human behavior patterns, specifically addressing variation in human skill in games. In this work simulations are primarily used as a tool to proxy variations in human skill, though the existing literature supports the notion that this approach could be trained to emulate human playstyles.

2.2.5 MCTS

Simulation-based sampling approaches share a need for domain agnostic ways to evaluate expected player behavior. Recently, Monte Carlo Tree Search (MCTS) has emerged as a popular technique in general game playing after the successful application of MCTS to the game of *Go* [21]. MCTS is a stochastic planning algorithm that builds a plan of action in a game by estimating the long-term value of different actions. MCTS balances exploring new actions and exploiting effective actions by trying alternative actions, rapidly simulating the outcomes of those actions, and using those outcomes to update estimates of the long-term value of different choices. Choosing actions based on a combination of their estimated value and uncertainty helps navigate games with many actions at any given point. Using randomizing simulations to estimate the final game outcomes of an action ensures the algorithm is not biased by near-term gains at the cost of long-term success.

Game applications of MCTS include: card selection and play in *Magic: The Gathering* [45, 54, 236]; platformer level completion [98, 227]; simulations for fitness function heuristics in strategy [124], card [66], puzzle [66], abstract real-time planning [154], dungeon crawler [91], and general arcade games [143]; and high-level play in board games including *Reversi* and *Hex* [12]. MCTS has been combined with deep neural networks to yield world-class play in the game of *Go* [185]. MCTS has proven effective even in open-ended domains, including playing a wide array of games designed in the VGDL [68, 143, 153, 156]. MCTS offers the advantages of being game-agnostic, having tunable computational cost, and guaranteeing (eventual) complete exploration of the search space.

Unlike previous uses for (near-) optimal agent play, I use MCTS to sample playtraces in a game while varying agent computational bounds as a proxy for player skill. By varying the resources available to the MCTS agents I model aspects of human skill in forecasting potential ways of playing out a game. Explicitly tuning these agents to produce playtraces that are similar to humans (against a subset of desired metrics, such as length of games) trains the agents to behave in a human-like fashion. This works complements efforts in

believable agents by presenting an approach to creating believable agents tailored to gather design-relevant metrics.

2.3 Analytics

Behavior sampling provides examples of behavior in the space of play, but design iterations require understanding what these examples mean to inform design changes. Game analytics applies techniques for statistical description and modeling of game-related data to help understand how games function [176]. Game analytics has broad applications spanning models that predict players quitting a game (to estimate revenue in, for example, a subscription-based game) to heatmaps that visualize common locations for player events like deaths [216, 245]. This thesis is concerned with enabling a machine to automatically design a game toward gameplay goals. As such, the most relevant area of game analytics is gameplay analytics: analyzing how players behave in a game.

Game behavior analysis is focused on a representation of the *progression of states* in a game—understanding how gameplay occurs through studying common states, actions, and progressions between states. A progression of game states can be represented as a sequence of player actions or as a sequence of game states. The two perspectives are complementary ways of understanding a game's design. Understanding what parts of a game the player visits can inform design decisions around what content is being used (or not) by players and can shape decisions to change specific content to increase or decrease player activity around that content. Understanding what actions in a game the player takes can inform design decisions around what strategies players do or do not use in the game and can identify potential flaws that imbalance a game—making the odds of success unequal for two opposing sides or making one action preferable in all cases (obviating the need for alternatives being designed).

Game visual analytics researchers have studied ways to visualize game metrics to understand play behaviors [234]. Visual analysis examines many game properties, including

spatial distributions of events (e.g., using heatmaps) [216], aggregation and identification of player types (e.g., using dimensionality reduction or clustering methods) [59, 217], and summarization of individual player behavior (e.g., through dashboards and in- or out-of game representations) [133]. Analytic methods have been applied to assess procedural content generators, to date examining properties of the aggregate space of possible levels generated for 2D platformer games[26, 92, 125]. Analyses of gameplay as a progression of states have aggregated regularities in the individual states or sequences of states visited by players in single-player games [6, 116, 234]. Analysis of gameplay as a sequence of actions is less common, but has also been applied to capture patterns in gameplay in single-player games [10, 151, 152, 235].

In this work I contribute to action analysis of gameplay with a specific focus on multiple levels of granularity. Game analytics methods are often used at a single level of abstraction: providing high-level summary statistics of states visited or low-level examples of traces that fulfill specific criteria of interest. Instead I consider the question of multiple levels of abstraction, providing four tiers of action analysis that are more or less granular in their model of player strategy. I illustrate how this can provide a holistic picture of the strategies available (or not) in a game to demonstrate the value of this approach, particularly toward informing systems that automatically evaluate gameplay possibilities afforded by a game.

2.4 Iteration

Iteration is concerned with intelligently choosing the next version of a design to try in a way that minimizes the number of designs tested. Iteration combines information from analysis with knowledge of a game design to choose the next point in a space of designs to try. In this thesis I approach design iteration through the lens of techniques from optimal experimental design (OED) and active learning (AL). Active learning methods enable computational systems to efficiently iterate on a design by using knowledge of how well prior designs have worked to intelligently choose new designs that better meet design goals or better

inform the system about the space of designs possible.

Optimal experimental design is the problem of designing an experiment to optimize the information gained from that experiment [28]. OED grew out of efforts to improve the efficiency of medical tests and costly scientific experiments; for example, choosing the appropriate population size and dosing of a drug to determine it's efficacy. OED considers an array of possible experiment configurations that vary in key parameters (e.g., a series of drug dosage levels) and builds a model to predict the outcomes of experiments when using these different parameters (e.g., how well the drug treats the condition for a given dosage). These predictive models are usually built from data from prior experiments or pre-existing theories regarding the experiment outcomes. With the predictive model, OED uses statistical tests to estimate the information gained from different configurations to find an optimal configuration for learning about the outcomes of interest (e.g., drug efficacy).

In machine learning, the field of *active learning* is concerned with the related problem of choosing input data to collect to improve the performance of a predictive model. AL techniques were developed to support machine learning algorithms in cases where there is a choice of data to collect, but gathering that data is expensive. For a given input data point either the outcome is known (labeled) or unknown (unlabeled) (Figure 2.1). AL techniques start from a set of labeled data to train a predictive model. Given a set of potential unlabeled data to add to the model, AL techniques choose the optimal next data to query to add to the model to improve the model. These data are added to the model and the process repeats. Active learning and optimal experimental design both arose in response to the cost of obtaining data in certain situations and apply statistical techniques to intelligently choose how to gather new information.

Techniques from OED and AL can be applied to the problem of design iteration by treating the choice of a design as the "experiment settings" or "input data" and the quality of the design as the "experiment outcomes" or "model quality." Using this analogy can help reduce "sample complexity" for design iteration—the number of data points (playtests)

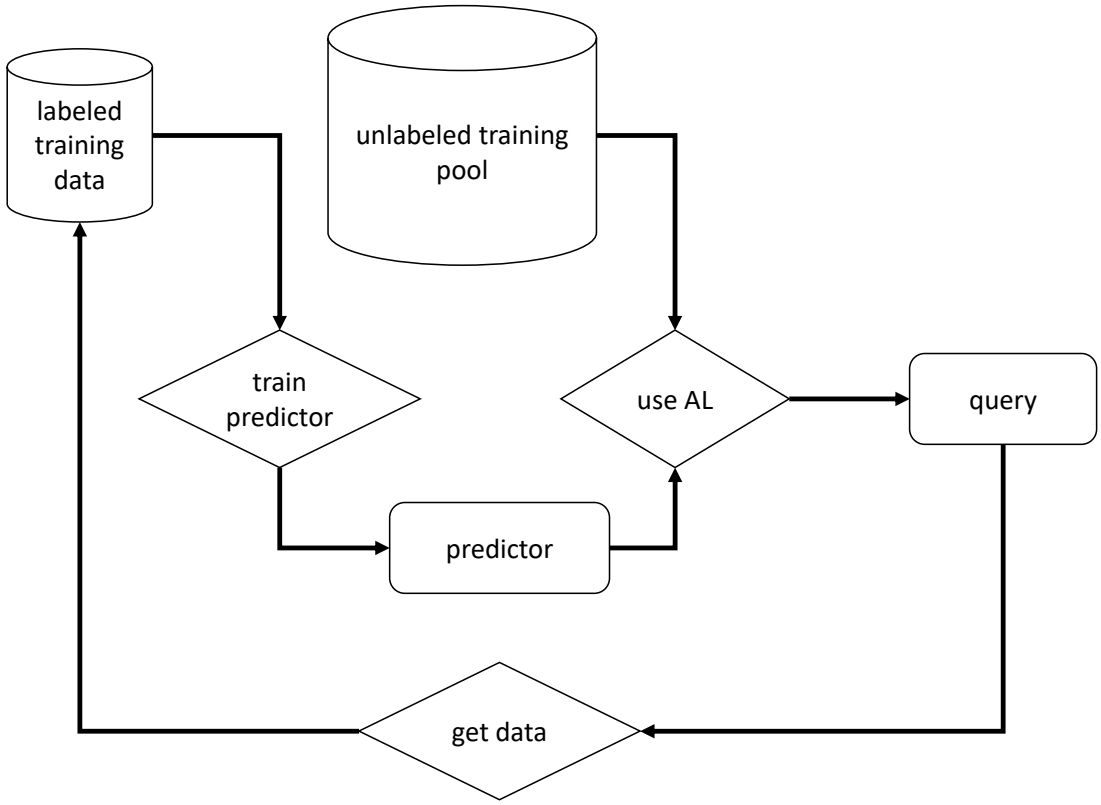

Figure 2.1: Diagram of active learning process.

needed to train a model. Active learning and optimal experimental design make explicit the trade-off between "exploring" potentially valuable game design settings and "exploiting" known good solutions with small changes.

To date, relatively little work has applied techniques from OED and AL to game design or game AI problems. In early work, Southey et al. [203] used active learning to test different ways an agent could make goal shots in a soccer game. A rule-learning system predicts the outcomes of shots made from different positions in the game (by a simulated agent) and active learning is used to guide tests of shots from new positions to improve how well the model predicts shot outcomes. Here, AL is used to reduce sample complexity and provide analysis of a game by guiding behavior sampling. For example, the rules learned can inform designers about the likelihood of shots from certain distances or angles scoring or not. The method used, however, is only targeting how to build a useful predictive model

of behavior, rather than optimize the design for desired player behavior.

Normoyle et al. [145] use AL to choose useful player metrics to track. In many situations it is possible to collect a vast amount of telemetry on player activity in a game, but hard to know which subset of these features are relevant to understanding player behavior in the game. A Markov Decision Process (MDP) models the relationship between gathered player metrics and in-game scenarios with active learning choosing which scenarios to collect data from to improve the model. Here, AL reduces the sample complexity of playtests, but again is limited to testing an existing game, rather than choosing optimal design variants.

Rafferty et al. [158] optimize the design of a cognitive ability testing game to gather the most accurate information about players. An MDP models player actions in the game and is used to infer features of player cognition (here concept learning). Optimal Experimental Design methods are used to choose among design variants to maximize the information gained about player cognition from the design. Here, OED optimizes the game design to gather information about players, but requires a detailed model of properties of how players are expected to learn and react to rewards. In this case playtesting sample complexity is not being directly minimized: OED is being used to find an optimal design based on offline models and tested against a non-optimal design. That is, information about prior playtest performance does not bear on the problem of design optimization.

Active learning has been applied to educational games both to optimize game designs for user learning and optimize for learning about game functionality. Lomas [120] use active learning to optimize engagement in an educational game when varying parameters controlling the challenge of the game. Typical engagement in a game design condition is tracked and a multi-armed bandit model (a type of active learning model) is used to select which parameters to direct new users to play. Liu et al. [117, 118] apply AL to the problem of testing scientific hypotheses about learning in games and balancing these tests against user benefits such as learning. A multi-armed bandit is used to balance between learning

how well different game configurations function and maximizing user learning. Here, AL serves to help explore a large space of game design configurations toward optimizing player learning and testing hypotheses about educational game design. In this case playtesting sample complexity is being reduced to improve a game design.

In this work I present a novel application of AL to reducing the sample complexity of playtesting for achieving desired design metrics. This work shows how to optimize a design for desired player responses in the game by varying features of the game's design. I evaluate the differences between many AL models for a given problem and contrast cases with design metrics for objective behavior or subjective feedback.

2.5 Creativity

Iteration is a core component of theories of general creativity and the creative process involved in design. Researchers studying human creativity agree that a process of iteratively developing and refining a creative product is typical—from a theory (Darwin's conception of evolution or Einstein's theory of general relativity) to an artifact (Picaso's *Guernica* or writing a book) [111, 163]. The work in this thesis develops a computational approach to game design iteration intended to understand the capabilities of AI systems to perform iterative design. By building models of creative processes we gain perspective on the capabilities and limitations of computational systems to perform human tasks.

2.5.1 Creativity Research

Kozbelt, Beghetto, and Runco [111] discuss a variety of overlapping frameworks for studying creativity. These perspectives range from psychometric theories intended to measure human creativity (like IQ) to developmental models that study how people develop creative abilities over their life. The most relevant theories for an AI system are grounded in *cognitive models* of the creative process and *systems views* of how creativity emerges through interactions between a creator and their artifact and environment. The theories falling un-

der this umbrella all embrace the notion of creativity as a process of creating an artifact, reflecting on the properties of the resulting artifact, and using that information to inform changes to the artifact to achieve creative desires.

Finke et al. [161] developed the geneplore *cognitive model* of creativity where creators alternate between thinking through phases of generating and exploring a creative product. During generation, a creator focuses on assembling the structure of a creative product, combining elements related to the concept or artifact to produce a product of interest. During exploration, the creator examines the resulting product to consider the consequences of creative decisions, particularly as they bear on the goal of a creative exercise. This cognitive model was developed and refined through laboratory studies and design tasks given to regular people as a way to understand the aspects of creativity shared by people [237]. Studies of writers' practices suggest a similar model of alternating between engagement in creation and reflection [183] and geneplore has also been applied to model digital filmmakers' creative practices [53]. Cognitive models like geneplore offer a valuable perspective on how people go about creating at the level of a detailed process account, informing how the monolithic problem of creative production can be broken into more tractable problems of generating and refining a creative product.

Systems theories of creativity emphasize how creative products are the result of interactions over time between an individual creator and their product and audience. Csikszentmihalyi's [48] systems view of creativity emphasizes that whether an artifact is deemed creative results from how a creator interacts with a broader community of creators. Creativity does not inhere solely in the creative product, but instead emerges from how the creator engages the audience of a creative artifact. For example, a painter can produce paintings that more or less adhere to stylistic norms among a painting community and it is only when the paintings are appropriate to the expectations of that community that the painter's work gains recognition. Csikszentmihalyi's theory was developed through studying the interactions among artistic communities and emphasizes the importance of creative

systems that acknowledge and respond to their audience. Gruber [233] developed a different systems theory through case studies of eminent creators, most prominently Charles Darwin's development of the theory of evolution. Gruber's work similarly emphasizes creativity as being interactive—based on interactions between a creator and their product and audience—while also putting a focus on the development of a creative product over time. Gruber's work helps dispell the commonplace view of creativity as emerging from a single 'a-ha' moment; instead, creativity results from the development of a product over time and through continuous development and interaction with responses to the product. Systems theories of creativity bring out the importance of considering the *audience* of a creative product and the role their reception of a product plays in shaping the product itself.

2.5.2 Design Research

Design researchers outside the domain of creativity research have also emphasized the importance of iteration and understanding audience reception of a creative product. Rittel and Webber [160] defined "wicked problems" to describe dilemmas encountered by designers where the nature of a problem shifts in response to solutions developed to that problem and where there are no fundamental objective notions of the value of a solution. Rittel and Webber recognized that then-current theories of social policies and planning were based on the assumption that a problem can be clearly defined and a solution planned based on shared values. In reality, in many situations the policy being designed required iterative development through meeting with key stakeholders who value the policy in development and adjusting the intended approach and plan in response to this new information. Schön's [174] theory of the reflective practitioner emphasizes a similar process of iteratively creating products and reflecting on the resulting product in conjunction with the intended audience of that product. These perspectives on the design process emphasize the central importance of iteration in developing an artifact and audiences in providing crucial feedback to shape the creative process.

2.5.3 Computational Creativity

Computational creativity researchers have developed some systems that model limited notions of the audience of a creative system and iteration to respond to that audience. A core challenge for computational creativity is capturing how the processes of generating and evaluating a creative product function. Sharples [183] proposed a model of creative writing as design, in which a writer cycles between stages of cognitive engagement and reflection (similar to the geneplore model). MEXICA [155] was developed based on this model, and iterates between phases of producing plot structure (guided by preset constraints) and evaluating that structure to guide refinements to the plot. During production MEXICA uses a memory model to retrieve content related to what is being created at the moment in a chain. When production stalls, due to lack of new content or violating constraints, refinement begins. Each phase of refinement can open the possibility of generating new structure by fixing potential flaws that prevented further plot construction. MEXICA is instructive in demonstrating that reflective processes interwoven with generation can help improve an artifact and expose potential paths for further computational generation. While this model captures the notion of iteration, it is limited to seeing creativity as purely driven by internal processes with fixed standards for how to evaluate a work.

How might a creative system evaluate it's work against changing and potentially external expectations? Case-based reasoning (CBR) has been used as a way to model how creators can take inspiration from existing works and modify these works to produce novel artifacts. Kolodner and Wills [108] first proposed modeling creativity using case-based reasoning—a model where problems are solved by finding similar older problems and adapting their solutions to new situations. By using a set of examples to guide creative processes these models capture how creators can manage a set of expected approaches to a problem, while evaluating these against constraints of a given situations. Gervás [72] extended the CBR approach with *dynamic inspiring sets*: evolving sets of examples used independently to construct or evaluate a work. A *learning set* of examples is used by a cre-

ative system to guide production of creative works. But a separate *reference set* of examples grounds evaluation of creative works. This distinction captures the notion of judging a creative artifact against an external set of norms similar to Csikszentmihalyi's systems view of creativity requiring creative products to be evaluated against existing accepted examples of creative work. Crucially, these inspiring sets are dynamic and can change over time as new examples are considered canonical (or not) for evaluation or construction of new works. Dividing creative processes into generation and evaluation components has proven valuable both in the domain of poetry generation [72] and the domain of recipe creation [136]. These efforts, however, still focus on internal evaluations of a creator, rather than explicitly addressing feedback from how an audience may receive a creative work.

Computational creativity research has increasingly embraced the notion of importance of how a product is *presented* to an audience. Creative systems are often challenged by people unwilling to accept products of computer creators as creative. Ventura [232] explicitly acknowledged this challenge by arguing against the notion that creativity evaluations could be based purely on metrics of an artifact. Instead, Venutra proposed that creative systems would need to explicitly persuade audiences to produce desired outcomes in terms of how a creative artifact was received. Colton et al. [31] developed this perspective further by arguing that creative systems need explicit ways of explaining their decisions, actions, and creative products to mitigate the bias may have against computer-generated products. The FACE model [32]—framing, aesthetic measure, concept, and expression of a concept—of creative generation includes a process in generation for framing an artifact for an audience. All of these efforts rest on the argument that audience reception is paramount to how a creative work is received. None of these works, however, explicitly uses audience reception itself—instead these models focus on ways to mitigate audience perception or persuade audiences to evaluate a work in a certain way.

Creativity support tools have developed in parallel to computational creativity. While computational creativity research is primarily interested in enabling computers to be au-

tonomously creative, creativity support research has focused on ways computers can augment the creativity of people [184]. Creativity support tools have taken many approaches to supporting human creative processes. Lubart [122] proposed four major categories for computers in the creative process: helping manage creative work, facilitating communication among people on a creative project, guiding use of creativity enhancement techniques, and mixed-initiative involvement in creative projects. O'Neill and Riedl [146] proposed the additional role of acting as a surrogate audience to provide feedback to authors without requiring people. The work in this thesis touches on two of these areas: autonomous computational creativity and modeling audience reactions. Building systems capable of generating games and iterating on their design moves toward full automation. However, these techniques also require ways to simulate audience reactions to guide iteration, in turn generating the type of audience feedback needed in creativity support. While some of the systems in this thesis may ultimately serve in mixed-initiative systems, the work here does not explicitly address how the computer interacts with a human creator.

As an example of a creativity support tool for games, Goel and Rugaber [76, 103, 229] developed a tool for designing game-playing agents. The system represents the knowledge structure of game playing agents in a turn-based strategy game. After a person designs a way for the agent to play, the agent simulates play in various game scenarios. Information from agent successes and failures when playing these game scenarios is then used in a meta-reasoning process to update the agent play strategies. This system illustrates how feedback from agent interactions with an environment can be used to update the agent, in this case for designing agent play strategies. By contrast, the work in this thesis focuses on the design of games, rather than the agents that play these games. Despite this difference, many conceptual elements of these approaches are similar: both Goel and Rugaber's system and the work in this thesis emphasize the need to represent the structure of designed systems and adjust that structure in response to learning from feedback when the system is tested.

In this work I focus on the challenge of iteration by developing techniques to gather

and model audience feedback (player in-game behavior) and use that information to guide iterations on a creative product (a game). Adjusting a game's design to produced desired player behavior has clear parallels to adjusting a creative work to produce a desired audience evaluation of creativity. By taking this approach I help bridge the gap between developed theories of creative practices and the capabilities of computational game design systems. Working in the domain of digital games demonstrates how these theories can bear on real-world creative practices while addressing a unique type of audience feedback in the form of direct interaction with a creative product.

CHAPTER 3
GAME GENERATION

3.1 Introduction

Iterative design depends on the ability to create a functional game that meets specifications on how the game functions. The goal of iteration is to adjust these specifications in response to player behavior. In this chapter I discuss how to generate the core functional systems of a game in a way that ensures they allow for basic game outcomes like winning or losing. By "functional" I refer to systems that dictate what behavior is or is not possible in a game, setting aside elements that alter the game experience without shaping the space of possible actions (e.g., the visual art style or audio choices). A key emphasis of my approach is to enable modularity and recombination of game content: a general game creation system should be able to combine elements from disparate game genres (as humans do), rather than requiring specialized logic to handle extensions to new game domains. The approach I take enables the combination of elements from game genres while ensuring generated games allow for basic game outcomes like winning or losing the game.

Game genres span a broad range, compassing 3D first-person shooting games (e.g., *Half-Life* [230]), 2D platforming games (e.g., *Super Mario* [144]), and tile-based puzzle games (e.g., *Sudoku*). Creating a representation that encompasses all these games is a daunting task that is unlikely to be computationally tractable [40, 218]. Yet there are still shared elements of how these games are designed worth considering outside the unique elements of designing games of a specific genre. To study these shared design challenges I focus on a constrained subset of game features that is computationally tractable: turn-based games in discrete worlds with deterministic actions and full observability. Using turn-based games simplifies the logic of how action effects are resolved in a game—it removes the need

for detailed considerations of second-by-second action duration and timing. Discrete game worlds simplify game generation by removing the need to reason on real-valued spaces of possible entities—this drastically shrinks the space of games possible while still remaining expressive enough to capture central details of a game. Deterministic actions remove the need to reason on probabilitistic outcomes—this simplifies exploring the space of actions possible in a game. Using fully observable game state simplifies the model of player knowledge, perception, and memory—this shrinks the space of games to remove varying degrees of observability while also removing the need to model additional aspects of agent reasoning. By confining generation by these limitations I am able to focus on generation that addresses elements common to this class of games. Fortunately, these constraints still allow models of a broad range of games, from the battles in turn-based role-playing games to the structure of movement puzzles in platforming games.

Most game designs are defined in terms of gameplay goals and failure states [69, 171] (although Cook and Smith [41] offer a countervailing perspective). Players are given the task of figuring out how to use the available actions in a game to reach a goal in a game, while avoiding obstacles along the way. The key insight to my approach is that this formulation closely mirrors the standard form for AI planning problems. Planning problems define a domain in terms of states, actions, and goals [164]. States consist of logical conditions using positive literals—definitions of aspects of that world that are true. Goals provide a full or partial specification of the target end state of the problem using a conjunction of positive literals. Actions (also called 'operators') are defined in terms of preconditions and effects. Preconditions are conjunctions of positive literals defining what must be true to take an action. Effects are conjunctions of literals defining state changes: positive literals define what becomes true as a consequence of an action while negative literals become false. A given planning problem in a domain provides an initial state that defines the start of the problem. A solution to a planning problem is a sequence of actions that moves the world from the initial to the final state, often subject to constraints on the use of actions. By

analogy, players are given an initial game state (the initial set-up of a platformer level) and tasked to find a sequence of game actions that moves to the win condition (the end of the level) of the game without hitting any of the loss conditions (colliding with an enemy) along the way. Classical planning technologies were designed specifically to address the subset of game features I use above, allowing for efficient solutions and transfer of techniques.

Using this perspective I treat game playing as a planning problem. Game generation then becomes the task of choosing the elements that define the planning problem, as opposed to assuming the problem elements are given and generating the plan. Winning a game is constructing a valid plan. Generating a level is defining an initial world state and goal state such that there exists a valid plan. From this perspective, the initial world state includes all relevant elements of the level, such as placement of the player, ground tiles defining the terrain, or enemies and items. Goal states define objectives for the player avatar such as reaching a location or collecting all items in the level. Generating allowed player actions in the game (game mechanics) is defining operators in a game that allow valid plans for a game instance. It is even possible to combine game genres by combining their planning domain definitions (more below).

In this chapter I detail this planning perspective on game generation. I will show how to adapt traditional AI planning representations to provide a general and reusable representation for functional elements of discrete, deterministic, turn-based games games. With that representation I show new game mechanics (planning operators) can be generated to create playable games and how this same approach can generate new game instances (levels). Unlike prior work on game generation this provides a general framework for representing action in a broad class of games, enabling extensions and further development of techniques for general game generation. I discuss mechanic generation in two example domains—role-playing game battles and platformer movement puzzles—along with a domain that combines these two domains. After this I present a number of extensions to game generation that use the planning perspective to control other elements of game de-

sign: ways to allow additional designer control through explicit specifications of costs or benefits of design choices; adaptation of game mechanics in the face of changes to level designs; levels with multiple characters; progressions of levels; and button-based controls to input actions to a game. I conclude by discussing limitations of this work and possible extensions that further develop the planning perspective on game generation.

3.2 Game Representation

Game generation requires a representation of game structures, in turn requiring a formalization of the elements of a game's design. In this section I present a formal description of functional elements of a game design. Using this formalization I define the problem of generating game mechanics and define a representation for game domains to support generating mechanics and game instances. Here I first describe the concepts and formal structures of this representation and in the following section I ground these concepts with an example game domain. The approach presented here assumes a game engine that exposes these elements of game state that can be manipulated.

Taking a planning perspective on gameplay, the two core functional components of a game design are the state model and transition model. The *state model* defines what makes up the game world through a collection of logical positive literals. This representation uses only first-order state descriptions: using propositions ($player$) or first-order literals ($Health(player)$). Any given state in a game domain must be ground and function free, though the definition of a domain may include non-ground terms ($Health(x)$). In a planning representation of state we track changing aspects of state using fluents. These fluents represent variables that exist in a game engine that runs the implemented game.

The *transition model* defines how the state evolves using logical assertions that define how states change (or remain the same) from one time step to the next. Actions taken by the player or other game entities are represented as planning operators (a subset of the game mechanics that are also called actions). Operators are defined by preconditions on what

must hold in the state to be executed and effects for how the state fluents change. In this planning representation (as in PDDL) we assume inertial state and circumscription: any states not explicitly changed by the transition model are assumed the same unless these states are derived from other logical assertions that were affected by the transition model.

A state model defines the design space of game instances (levels). A playable game instance provides a fully grounded state model that defines an initial game state along with additional logical terms that define the goal and failure states of the game. These goal and failure states are the *playability criteria* that create the challenge of playing the game.

Mechanic generation is the problem of constructing a (set of) game mechanic(s) such that they meet playability requirements to create a desired range of player behaviors (allowing and forbidding action sequences) while meeting design requirements on mechanic structure. Mechanic generation is thus the problem of constructing the transition model. *Playability requirements* are used in mechanic generation to ensure the resulting games are possible to finish according to their goals and maintenance goals. Game designs, however, are often subject to a number of constraints from a designer intended to focus design around a subset of behaviors. *Design requirements* specify high-level constraints on how mechanics work in a game. For a fully autonomous creative system design requirements are unnecessary; for design tools or mixed-initiative systems design requirements provide additional input to guide game generation. Note that both playability and design requirements may be specific to a game genre or domain-independent. Below I detail the state and transition models used by my system and a process to use these models in game generation.

3.2.1 State Model

In this representation a game state model (Table 3.2.1) is a set of positive terms defining the entities of a game world, parameters of entities, and the allowed values for those parameters to take in the game. To illustrate concepts we will consider two game domains: movement in a platformer game and the battle system of a role-playing game. In an platformer game

like *Super Mario Bros.* [144] the world consists of entities (Mario and his enemies) at locations on a 2D plane. Platformer mechanics define how the player avatar may move in the world, controlling the arc of movement for different player actions like walking or jumping. In the battle system of a role-playing game (RPG) like *Final Fantasy* [204] or *Dungeons & Dragons* [82] the world consists of entities (the player and opponent teams) with stats tracking for health, mana, or traits (like attack power). RPG mechanics define what actions players may use to affect allies or enemies; for example, these "spells" in the game fiction can be used to reduce enemy health or heal allies.

$Entity(e)$ defines the existence of an in-game entity, such as the player ($Entity(player)$) or an enemy ($Entity(enemy)$). $Parameter(p)$ defines that a parameter can be used in the game, such as positions along the x-dimension in a platformer ($Parameter(x)$) or health in a RPG ($Parameter(health)$). $Has(e, p)$ defines which parameters represent the state of an entity, such as player's having a position in a platformer ($Has(player, x)$) or health in a role-playing game (RPG) ($Has(player, health)$). Parameter value constraints come in two forms: values possible in the game world ($AbsRange$) and values possible for mechanic changes to a parameter ($RelRange$). A RPG player may only be allowed to have health values in the range $[0, 5]$ using $AbsRange(player, health, [0, 5])$. Spells (a type of game mechanic), however, may be limited to only changing player health by at most 1 point at a time using $RelRange(player, health, [-1, 1])$. By separating the world state from changes made by mechanics, it is possible to constrain generation of mechanics to "sensible" values. In many design situations the changes to parameters are limited to small, local alterations, which can be specified by using smaller allowed ranges for $RelRange$. For simplicity parameters currently range over integer values.

Referring back to the example RPG spell system player can be defined with the predicates in Table 3.2.1 ($RelRange$ relates to the transition model). This definition specifies the existence of a player, two game parameters for health and mana, and that the player has both of these parameters. The player's health is allowed to range from 0 to 3 while the

Table 3.1: State model definition

Entity(e)	e is a game entity
Parameter(p)	p is a game parameter
Has(e,p)	e has p as part of its state
AbsRange(e,p,r)	e has p limited to in-game values in the range r
RelRange(e,p,r)	e has p that may be changed by the values r

Table 3.2: Partial RPG domain definition.

Entity(player)	
Parameter(health)	*Parameter(mana)*
Has(player, health)	*Has(player, mana)*
AbsRange(player, health, [0,3])	*AbsRange(player, mana, [1,5])*

player's mana is allowed to range from 0 to 5.

Any concrete game instance can be defined through initializing the values for all entities in the game instance. I represent all changing state using a logical fluent $Holds(t, P(e), v)$; where t is the time index of interest, $P(e)$ defines the entity parameter of interest, and v gives the parameter value. We track historical state using time indices to facilitate non-Markovian mechanics that may reference state other than the current state. Game state (the planning problem) is initialized using $Initial(P(e), v)$, which the planner uses to create fluents for the initial time step. In the RPG example, we can set player health to initially be 3 using $Initial(Health(Player), 3)$, which becomes $Holds(0, Health(Player), 3)$. Events or actions may change the fluent values as defined below.

Game states may be defined in terms of coordinate frames of reference. *Coordinate frames* distinguish between traditional world-state terms and "perceived" avatar-relative versions of world terms. *Absolute frames of reference* model requirements on the state of the world. *Relative frames of reference* capture the intuitive notion that many game mechanics have preconditions and effects relative to an avatar, rather than absolute world state (e.g. adjacency as relative position). Requiring 1 or more mana to cast a spell is an absolute constraint in an RPG; requiring the player to be directly above a solid object to jump is a relative constraint in a platformer. Absolute state in the game is tracked using the $Holds$ predicate defined above; relative states are tracked using $Senses(t, a, P(e), v)$, where a defines

the focal agent, $P(e)$ defines the target entity parameter, and v is the difference in values between the two agent and entity. $Senses$ predicates are all derived from $Holds$ predicates at each time step. In the platformer domain the player might sense a block below them, inferring from $Holds(1, Location(Player), (1, 2))$ and $Holds(1, Location(Block), (1, 1))$ the predicate $Senses(1, Player, Location(Block), (0, -1)$.

Goal and failure are defined by specific game situations—conjunctions of terms that may not include all terms in the state model. In this formalism each goal is considered to be a conjunction of the logical terms making up that goal. Each goal term takes the form $Goal(P(e), v)$. In the platformer example, reaching a given 2D location would be defined by two terms in the game instance definition: $Goal(xPos(Player), 3)$ and $Goal(yPos(Player), 2)$. Failure terms take a similar logical form, though each failure term is interpreted as a logical disjunction: failure occurs if any failure criteria is met. This is not necessarily limiting: parameters of entities can be defined through assertions based on other parameters. Thus, in the platformer failure can be defined for reaching the same position through an additional parameter specifying player location that is derived from the $xPos$ and $yPos$ parameters: $Holds(t, xPos(e), x) \wedge Holds(t, yPos(e), y) \implies Holds(t, Location(e), (x, y))$. Failure can then be defined by $Fail(Location(Player), (3, 2))$.

3.2.2 Mechanic Model

A set of mechanics define a transition model that allows forward simulation and playability checks as planning. Game mechanics take many forms, from high-level rules governing the order of turns in a game to low-level rules for resolving outcomes of simultaneous actions. Mechanics generally consist of fixed update rules defined by the game engine and actions that agents in the game may take. This work focuses on the class of *avatar-centric* mechanics—actions taken by the player (or other in-game agents) in the process of controlling an avatar. Avatar-centric mechanics define the actions that are possible in the game and form the core component of direct player control of games.

The action schema defining mechanics draws from standard PDDL to define an avatar-centric mechanic as a tuple: $\langle i, P, E \rangle$ where i is a unique identifier for a mechanic, P is a set of the preconditions needed for mechanics to occur, and E is a set of effects of performing the mechanic. Here preconditions and effects extend traditional PDDL action schemas with time-indexing and coordinate frames of reference. *Time-indexing* allows preconditions to reference the game state at times other than the present and allows effects to reference states other than the next game state. Allowing preconditions to check state at different times allows mechanics to check for historical conditions; for example, testing for player state in a previous turn. Games also often incorporate delayed effects or effects with a duration over multiple turns. Time-indexing enables mechanics to have limited ability to bypass the traditionally Markovian structure of game description languages. By constraining how time indexing is used it is possible to limit the additional computational overhead incurred when generating mechanics. For example, constraints can be a small range of allowed values or a design constraint indicating a preference for little use of time-indexing. Coordinate frames of reference (introduced above) enable more concise expression of mechanics that make relative checks or update date by relative amounts.

The planner here implements semantics for a subset of PDDL with extensions appropriate to this definition. $AbsRange$ is used to specify valid absolute frame of reference values while $RelRange$ is used for relative frames of reference. Preconditions test game state; we allow tests for equality, inequality, and lesser-than and greater-than relations. All preconditions and effects are tuples of the form $\langle frame, time, condition \rangle$; where $frame$ indicates a coordinate frame of reference, $time$ specifies a time-index, and $condition$ specifies a game state value to check for (or update). In this formalism, a condition takes the form $F(parameter(entity), value)$ where F is a logical function that either tests two values and returns a boolean value (for preconditions) or updates an entity parameter value (for effects). Testing for the avatar currently being alive would be $\langle Absolute, 0, GreaterThan(Health(Player), 0) \rangle$.

Effects update game state. Effects may reference current or future state; we forbid historical changes (time-travel paradoxes may result). For absolute frames of reference, updates set state to a particular value (constrained within $AbsRange$); for relative frames of reference, updates change state values by a given amount (constrained within $RelRange$). Effects take the same form as preconditions and are interpreted as logical rules for updates or setting appropriately. A spell that checks for the enemy being alive and reduces enemy health by 1 on the two next turns is:

$\langle DamageOverTime,$

$\{\langle Absolute, 0, GreaterThan(Health(Enemy), 0)\rangle\},$

$\{\langle Relative, 1, Update(Health(Enemy), -1)\rangle,$

$\langle Relative, 2, Update(Health(Enemy), -1)\rangle\}\rangle$

Not all mechanics are dependent directly on game state—many reference particular actions or events that occur in the game. Mechanic *recombination* occurs when one mechanic references another mechanic. Fighting game or rhythm game combo systems exemplify avatar-centric mechanic recombination for preconditions: the ability to use an action in a combo depends on having previously executed some other action. Mechanic recombination naturally encodes event-relevant mechanics, rather than being limited to mechanics that reference state. Mechanic recombination also supports modularity in mechanic effects: a mechanic may execute another mechanic on top of other modifying effects.

For mechanic recombination we allow preconditions and effects to reference the *event* of a mechanic occurring with $Performed(i)$. Semantically, a mechanic as a precondition requires that mechanic to have (or not have) occurred at a time index. For example, a double-jump may require a player to have jumped at the previous time-step: $\langle Absolute, -1, Equal(Performed(Jump), Player)\rangle$ When $Performed(i)$ appears as an effect the preconditions and effects of that mechanic are applied. The mechanic using $Performed(i)$ as an effect indicates the time to apply the performed mechanic. Note that frames of reference are not relevant for mechanic indexes (these are provided by the indexed

mechanics themselves) and are ignored. In the planner we track all mechanics that occur using the $Occurs(t, i)$ predicate and derive $Performed(i)$ for relative time step checks. To prevent circular mechanic recombination during generation, we only allow mechanics to reference previously generated mechanics. Unlike the base state fluents, $Performed(i)$ is treated as an *event* and not subject to inertial state.

3.3 Game Generation

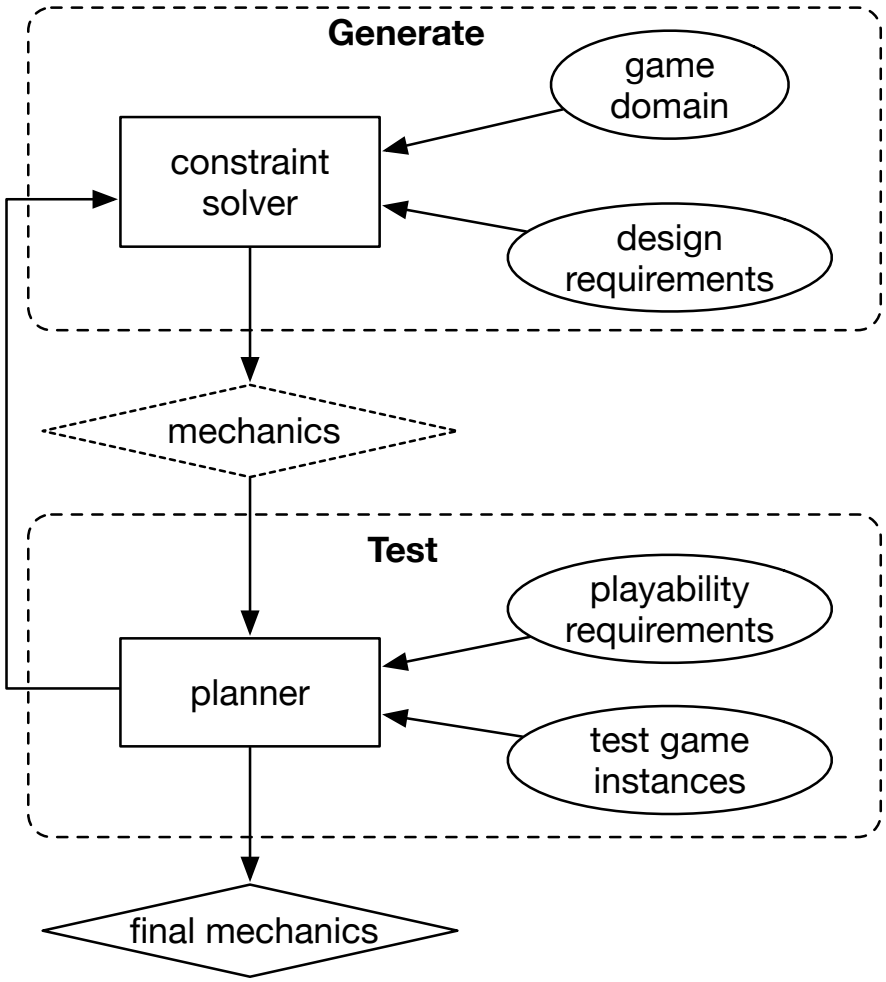

Figure 3.1: Process for generating games.

With a representation for the elements of games—the state model, transition model, and instances—we now turn to generating those elements to create games. In this work I focus

on generating both the transition model and accompanying game instances. Generating game mechanics requires specifying the precondition and effects for a set of mechanics. To make these mechanics semantically relevant, the system takes as input *design requirements* that constrain the types of mechanics allowed. To make these mechanics functional in the game, the system takes as input *playability requirements* in the form of goals and failure conditions as given above.

Conceptually, mechanic generation in the system uses a generate-and-test process (Figure 3.1). In generation, the system takes as input design requirements on the form of mechanics and a definition for the game domain to generate mechanics in. A constraint solver creates mechanics by choosing preconditions and effects for each mechanic while ensuring the mechanics conform to design requirements. These mechanics are fed into a planner that then checks whether mechanics meet playability requirements on given test game instances. If the mechanics pass the playability tests they are output as possible mechanics for the game instance.

3.3.1 Design Requirements

Two types of requirements are used to constrain the types of mechanics generated. *Design requirements* filter potential mechanics to avoid low-quality mechanics and guide the system toward the mechanic structures a designer is seeking (if any). Hard design requirements (as used by [197, 190]) enforce conditions on the form of mechanics or relations among a set of mechanics—e.g., not allowing a mechanic to have both equality and non-equality preconditions for the same game state or requiring no two mechanics to have identical preconditions and effects. Hard design requirements may require or forbid relationships between the preconditions and effects of mechanics. Hard design requirements are conjunctions of mechanic preconditions and/or effects that entail the $Invalid$ preposition. The constraint solver is required to always negate $Invalid$ to ensure specific conditions do not hold. The negation of a statement entailing $Invalid$ can be used to enforce presence.

This formulation draws from Answer Set Programming where forbidden conditions are expressed using an empty entailment [8]. For example, forbidding a mechanic from requiring preconditions for both equality check inequality check on the same parameter can be expressed as

$\langle i, \{Equal(P(e), v)\}, \{\}\rangle \wedge \langle i, \{GreaterThan(P(e), v)\}, \{\}\rangle \implies Invalid$ Note that matches are on partial mechanic structure, allowing these conditions to be met for any set of effects or other preconditions of the same mechanic, rather than strictly requiring no effects and the single preconditions.

Soft design requirements (as widely used in search-based procedural content generation [180, 222]) give optimization criteria for what makes (sets of) mechanics better or worse—e.g. minimizing the number of preconditions and effects used by a mechanic to favor simplicity. Soft design requirements take the form of predicates that assign an integer weight to mechanics (or parts of mechanics) in the form: $Weight(P, V(P))$, where P indicates a logical term of interest, and V is a function that assigns a value to that term. Weights can then be minimized or maximized by specifying this as a criteria for the constraint solver via: $Minimize(V)$ or $Maximize(V)$. For example, minimizing the use of preconditions on mechanics can be expressed as:

$\langle i, P, E\rangle \wedge Weight(P, Count(P))$

$Weight(P, V) \wedge Minimize(V)$

where $Count(P)$ indicates the site of the mechanic precondition set P.

Hard and soft design requirements vary in specificity to game domains. Some requirements apply across types of games: e.g., not requiring a state hold and not hold at the same time (rendering a mechanic unusable). Other requirements are domain-specific: e.g., minimizing preconditions on actions in a platformer to have more simple and general mechanics. Design requirements are intended to support human authoring by providing ways to shape the space of mechanics a system may generate. At the same time these requirements increase the efficiency of search for game mechanics by reducing the space of possible

mechanics generated through encoding common-sense knowledge about ways mechanics function.

3.3.2 Playability Requirements

Playability checking verifies that game mechanics allow players to win levels without losing along the way for every game instance. In planning terms this means the planner can use the mechanics (planning operators) to move from an initial game state to the goal game state for every initial and goal game state pair given. The goal game state is defined in terms of playability requirements provided as input.

Different constraints on playing the game are defined through three types of playability requirements (two discussed above): (1) goals, (2) maintenance goals, and (3) engine constraints. *Goals* define the game situation that must be possible for an agent to achieve. A planner must prove the existence of a plan that meets all goals. A game *situation* defines required values for a subset of all parameters in the game state; e.g., defining a target location for the player avatar in a platformer while leaving the locations of all enemies and items unspecified. Thus, the $Goal$ predicate above may only define a conjunction of entity parameter values of interest.

Maintenance goals define failure criteria in terms of game situations that must always hold in a successful plan. That is, maintenance goals negate failure criteria by specifying things the player must always keep true. Maintenance goals use the $Fail$ predicate above, where the planner is required to provide a valid plan that never causes the $Fail$ predicate to be true. In a platformer, failure occurs when the player and an enemy collide; thus a successful plan must always have the player an enemy occupying different coordinates.

Engine constraints enforce semantics for how the game functions outside the control of generation and are defined by pre-existing mechanics in the game. I call these engine constraints as they are intended to represent elements enforced by a game engine that implements a game. A planner must follow the engine constraints when making plans—these

mechanics are outside the scope of mechanic generation and are not included as mechanic indices the constraint solver may alter. There is no special syntax to represent these terms in the planner as they come with the domain definition as mechanics or derived logical terms that define game state. In a platformer, the engine may prevent overlap between the player and ground tiles through collision detection, thus the planner should always enforce this constraint in plans (and thereby in the mechanics being generated). Engine constraints can be more generally useful when using game generation in a design tool by representing fixed game systems intended to be outside the control of generation. In a platformer, this may mean that gravity (falling down at a fixed rate) must apply in the generated game, requiring the planner to always obey gravity and mechanics to be generated in a way to account for the effects for gravity (e.g., adjusting jump height).

3.3.3 Implementation

To implement the constraint solving and planner I used Answer Set Programming (ASP) [8]—a form of declarative programming. As a logic programming language (in the same class as Prolog), ASP supports the creation of the domain definitions and predicates above. ASP provides a declarative language for specifying constraint satisfaction problems and implements a variety of optimized constraint solving algorithms to solve problems posed in the language. Declarative programming languages emphasize defining *what* a computational problem is, rather than defining the algorithm for *how* to solve that problem. Thus, implementing the generation and testing process in ASP consists of providing a representation for the logical definitions above using the ASP syntax, rather than implementing a specific algorithm for solving the constraint satisfaction problem of generating mechanics or searching for valid plans. ASP in specific allows for a class of logical models where multiple solutions are possible: these "answer sets" are equivalent solutions to a problem and allow the generation process to produce multiple sets of equally valid games according to a problem definition.

While generating mechanics and testing for playability is a two-stage process, the shared implementation in a single constraint solver yields a single monolithic solving process that shares information about constraint violations between the planner and solver to improve efficiency. Internally to ASP, the process of generating mechanics using a constraint solver and testing those mechanics with a planner repeats until all hard requirements are met and all soft requirements are optimized. In this case, my representation of the planning problem in ASP is expanded into additional predicates and constraints that are part of the overall constraint solving process. While this is computationally expensive, I focus on small game domains to understand the challenges and limitations facing an AI system designed for domain-agnostic iteration. Note that many games use relatively small sets of mechanics (e.g., RPG spell systems, platformer movement mechanics, or card game rules), making this limitation less constraining than it may at first appear. For example, on a 2.66 GHz Intel Core i7 MacBook Pro with 8 GB of DDR3 RAM, generating the 2880 possible solutions to the combined domain took 16.427 seconds, adapting the platformer jump mechanics took 0.218 seconds to find 4 optimal adaptations, and generating the optimal model for the larger (due to the number of blocks) platformer domain took 30.857 seconds. For further details on the implementation of the definitions above refer to Appendix A.

3.4 Examples

In this section I illustrate the above game domain representation and process to generate both avatar-centric game mechanics and game instances. We consider how to represent combat systems in a simple RPG and movement puzzles a simplified platformer. To demonstrate the modularity of the approach we discuss combining the RPG and platformer state models to generate new games in this combined genre.

The genres here are intended to illustrate the versatility of this approach to game generation. RPG combat commonly involves two opposing parties taking turns to attack one another using various spells (mechanics) until one party is slain; the *Dungeons and Drag-*

ons [82] tabletop RPGs are a paradigmatic example. RPGs require a balanced and diverse set of character spells; in RPGs there is an expectation for a relatively large number of mechanics that function in similar but different ways. Platformers are games where a character navigates physical obstacles in a virtual space, exemplified by the *Super Mario Bros.* [144] games. Platformers require a finely tuned and widely reused small set of spatial navigation mechanics that are combined to solve movement puzzles. The system here generates spells in the RPG and movement mechanics in the platformer, demonstrating flexibility in meeting different playability needs from mechanics. Concatenating these two domains and generating combined mechanics illustrates how the model supports cross-genre mechanic generation.

3.4.1 Role-Playing Game

RPG combat mechanics can be specified in terms of a set of entity attributes and resources (here health and mana for the player and a set of enemies). The earlier RPG spell example defines this basic domain. The RPG has playability requirements for: a player goal situation of having all enemies dead, a player maintenance goal of not being dead; and an engine constraint preventing negative mana. Together, these playability requirements encode the basic notion of an RPG battle as killing an opponent without being killed while having bounded resources. Two domain-independent design requirements also apply: a hard requirement to prevent mechanics from having preconditions that force a predicate to equal more than one value and a soft requirement to minimize the number of preconditions and effects of all mechanics to produce the simplest set of mechanics. Many domains have a notion of actions having costs; a third hard requirement gives a domain-specific version of costs by requiring all actions incur a mana or health cost.

The system generated a variety of RPG spells using the game domain, a game instance with two enemies, and the playability and design requirements above. Plans in the RPG domain are a series of player actions (spells) used to damage each of the enemies while

costing the player health or mana. One example spell was given above, others typically perform simple effects such as inflicting damage at a single time point or affecting multiple targets. The system generated the following mechanic to damage both enemies:

$\langle damageAll, \{\},$

$\{\langle relative, 1, Update(health(enemy1), -1)\rangle,$

$\langle relative, 1, Update(health(enemy2), -1)\rangle,$

$\langle relative, 1, Update(mana(player), -2)\rangle\}\rangle$

where there are no preconditions and the effects damage both enemies while costing the player mana. Note that human-readable names have been given to the mechanics; internally i (the name) is an integer. Also note that the examples in this section were chosen to illustrate the most semantically sensible mechanics generated; by definition all mechanics achieve playability and design requirements.

3.4.2 Platformer

Two-dimensional platformers can be described in terms of a set of entities (here the player, blocks, and enemies) each assigned spatial coordinates corresponding to two spatial dimensions (Table 3.3). The initial state of the player for the example (see Figure 3.2) is $Initial(xPos(player), 1)$, $Initial(yPos(player), 2)$.

Table 3.3: Partial platformer domain

Entity(player)	
Parameter(xPos)	*Parameter(yPos)*
Has(player, xPos)	*Has(player, yPos)*
AbsRange(player, xPos, [1,8])	*AbsRange(player, yPos, [1,6])*

The platformer has playability requirements for: a player goal situation of reaching the end target location, a player maintenance goal of not overlapping with an enemy; and an engine constraint preventing the overlap of any entity and a block. Another engine constraint enforces gravity by requiring all entities to move down one unit each turn if that space is not occupied by a block. Two design requirements from the RPG example can be

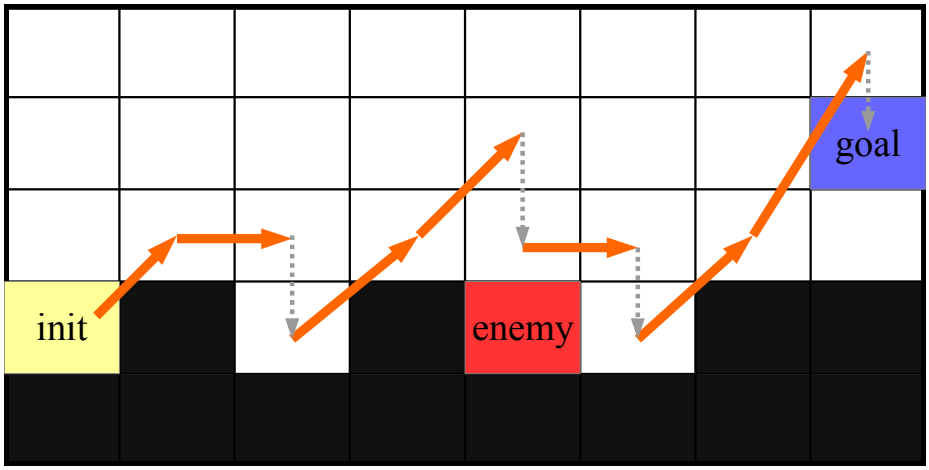

Figure 3.2: Platformer level showing a playtrace using a generated mechanic set. Arrows indicate generated mechanics, dotted arrows indicate gravity.

reused in this case: preventing exclusive pre-conditions and minimizing the number of mechanic preconditions and effects. A third soft requirement optimizes for as few mechanics as possible (to create a 'tighter' game system) and a fourth soft requirement minimizes the number of different entities referenced by mechanics (favoring motion of a single avatar).

Figure 3.2 illustrates a simple platformer level and shows one trace found by the planner that moves the player avatar to the goal position. The planner generated mechanics for moving forward, jumping, and double-jumping (indicated by arrows). Dotted arrows indicate the effects of gravity. The example shows a variety of movement mechanics the system generated, including two forms of jumping:

$$\begin{aligned}
&\langle jump, \\
&\quad \{\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle, \\
&\quad\ \langle relative, 1, Equal(xPos(e), xPos(block))\rangle\}, \\
&\quad \{\langle relative, 1, Update(xPos(e), 1)\rangle, \\
&\quad\ \langle relative, 1, Update(yPos(e), 1)\rangle\}\rangle
\end{aligned}$$

$\langle doubleJump,$

$\{\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle,$

$\langle relative, 1, Equal(xPos(e), xPos(block))\rangle,$

$\langle absolute, -1, Equal(Performed(i), jump)\rangle\},$

$\{\langle relative, 1, Update(xPos(e), 1)\rangle,$

$\langle relative, 1, Update(yPos(e), 2)\rangle\}\rangle$

$jump$ tests for the presence of a block to jump off of and, if so, moves the avatar diagonally up. $doubleJump$ does the same check while also requiring a jump to have occurred immediately before; the jump effect is slightly larger.

Two stranger mechanics resulted when using the system on a slightly simplified version of the above domain. The simplification removed blocks at even height with the player to create a level plain. The system generated a 'lift' and a 'ride' mechanic in two different solutions. $lift$ raises the enemy behind the player and was used to allow the player to move the enemy behind them while advancing to the goal:

$\langle lift,$

$\{\langle relative, 1, Equal(yPos(e), yPos(enemy))\rangle,$

$\langle relative, 1, Equal(xPos(e), xPos(enemy) - 1)\rangle,$

$\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle,$

$\langle relative, 1, Equal(xPos(e), xPos(block))\rangle\},$

$\{\langle relative, 1, Update(xPos(enemy), -1)\rangle,$

$\langle relative, 1, Update(yPos(enemy), 2)\rangle,$

$\langle relative, 1, Update(xPos(e), 1)\rangle\}\rangle$

$ride$ was a mechanic used to slide the player and enemy forward both by one unit and was used to have the player jump atop an enemy and 'ride' the enemy to the goal (shortening the jump distance needed):

$\langle ride,$

$\{\langle relative, 1, Equal(yPos(e), yPos(enemy) + 1)\rangle,$

$\langle relative, 1, Equal(xPos(e), xPos(enemy))\rangle\},$

$\{\langle relative, 1, Update(xPos(e), 1)\rangle,$

$\langle relative, 1, Update(xPos(enemy), 1)\rangle\}\rangle$

3.4.3 Combined Game

As a demonstration of the modularity of the game mechanic representation the previous two domains were concatenated to create a 'platformer-RPG' game. All game state definitions were unchanged: combining RPG resources and platformer location only makes entity state more complex. The previous playability requirements from both domains were retained with conjunctive (all criteria must be met) goals, maintenance goals, and engine requirements. With these simple changes the system generated mechanics appropriate to the domain such as attacking at a distance with a spell:

$\langle magicMissile,$

$\{\langle relative, 0, Equal(xPos(e), xPos(enemy) - 2)\rangle,$

$\langle relative, 0, Equal(yPos(e), yPos(enemy))\rangle\},$

$\{\langle relative, 0, Update(health(enemy), -1)\rangle\}\rangle$

where the preconditions check for an enemy two spaces in front of the player and the effect reduces enemy health.

3.5 Extending AI Design

The previous examples illustrate basic game generation using the representation described above. This section discusses extensions to the system that address more complex aspects of designing playable games: (1) adapting previously generated mechanics to instance changes, (2) cost-benefit balancing, (3) planning with multiagent games, (4) generating mechanics for multilevel progressions, and (5) mapping input controls to mechanics. These

extensions increase the scope of elements of design considered in game generation within a single representation, supporting the value of this representation for general game design challenges. By using a low-level representation of the manipulation of state variables this representation enables generation of elements of a game's design that are often overlooked.

3.5.1 Mechanic Adaptation

Designers often alter the instances of a game through iteration and tuning. Changes to a level design, however, can have unexpected consequences for the ability to use mechanics to complete other levels in the game. Addressing this problem requires a way to alter pre-existing mechanics to fit a new situation. In most cases these alterations should be minimal—as small of changes to the mechanics as necessary to still meet design and playability requirements.

Mechanic adaptation starts with a set of mechanics and produces a minimally changed set of mechanics (Figure 3.3). Mechanic adaptation uses the same process as mechanic generation, only now in support of human (or potentially computer) iterative design. When a set of mechanics are tested on game content the resulting insights about the game yield new criteria for the mechanics—adaptation requirements. *Adaptation requirements* specify additional playability or design requirements for mechanic generation. New playability requirements may indicate additional goal states for the player to pursue or identify unwanted states. New design requirements may control the amount of change to make to a set of mechanics. The definition of 'minimal change' varies by game domain and must be specified to adapt mechanics.

Mechanic adaptation takes the same input game state and transition models as mechanic generation, augmented with a pre-existing set of game mechanics. These pre-existing mechanics encode the mechanics to be adapted, providing the core systems from the game to adapt. Adaptation adds or removes preconditions and effects from existing mechanics and may also generate new mechanics. Changes to mechanics must meet designer-specified

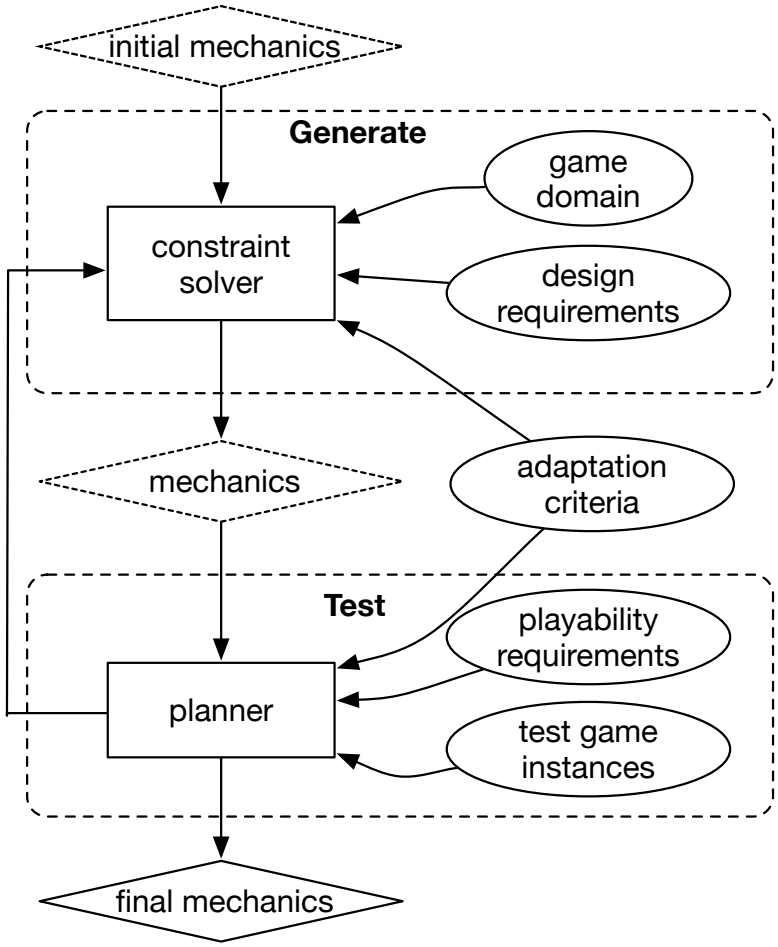

Figure 3.3: Mechanic adaptation starts with an initial set of mechanics and uses adaptation criteria to define minimal changes for mechanic generation to make to those mechanics. Testing uses the adapted mechanics in test game instances, requiring that any adaptation requirements for playability are also met.

criteria for minimality while adhering to all adaptation requirements. Minimality may constrain changes to small alterations of parameter values or may limit the addition or removal of preconditions.

The system adapts mechanics by performing the standard generation process but seeded with the additional mechanics (Figure 3.3). The previous set of design requirements are given along with new adaptation requirements and a definition of minimality (e.g. minimizing the total number of changes made). Mechanic adaptation performs the same generate-and-test loop as mechanic generation. The extension to the game generation system here

is trivial: rather than seeding the process with no mechanics the system begins with a pre-existing set of mechanics and a potentially different set of design goals that were used to guide initial generation. Adaptation may change ground terms from the mechanics as needed by variabilizing the mechanic predicates and selecting alternative ground values from the allowed ranges for the parameters in the domain.

As a test of mechanic adaptation in the platformer domain the system adapted mechanics generated without gravity to work in the same domain with gravity. First, the system generated a set of movement mechanics in the platformer domain, resulting in three mechanics: a long horizontal jump ($longJump$: 2 forward, 1 up), a short vertical jump ($highJump$: 1 forward, 2 up), and a dash forward (2 forward). Adding gravity requires the in-game agent to increase the amount of vertical movement when gravity is added. Gravity was added as an engine constraint and the system adapted the mechanic set above while reusing the same platformer domain and requirements. The resulting modifications made two changes: (1) the dash added vertical movement to now move 2 forward and 1 up and (2) the long jump added an initial lift phase moving 2 up, but at a time one step earlier than the rest of the mechanic. These results illustrate the flexibility to reuse the generation system for adaptation when baseline design considerations change. Adaptation only required seeding the generation with output from a previous generation step and specifying how many mechanics to use after adaptation (in this case preventing new mechanics from being added). In this case minimal change simply required the smallest total number of changes to the preconditions and effects of the provided operators.

3.5.2 Cost-Benefit Balancing

Game designers often employ intuitive notions of costs and benefits as a heuristic way to balance content in a game design. Schreiber [175] describes one such set of techniques, targeting examples like balancing cards in the card battling game *Magic: The Gathering* [71]. Card functions are assigned costs or benefits: a benefit per point of health of a card, a cost

per point of mana to play the card, or a benefit or cost for each of a card's abilities. In the case of *Magic*, using costs and benefits allowed designers to design specific cards to be balanced (by having equal costs and benefits) and also provide a progression of cards available at different levels of costs (or benefits). By assigning card functionality costs and benefits new content can be readily created by adding or adjusting functions until the costs and benefits of the functions of a piece of content are equal.

Schreiber's [175] concepts are readily incorporated into the design requirements of the mechanic generation system. Costs and benefits take the form $Cost(\langle E, P\rangle, C(V))$, where $\langle E, P\rangle$ indicates an entity-parameter combination, V is the value taken by the entity-parameter combination in a mechanic, and $C(X)$ is a cost function that maps from a given parameter value to its cost. Benefits take an analogous form, substituting the cost function with a benefit function. Design requirements may then specify constraints on the values of the costs and benefits: for example requiring no mechanic incur too large a cost or that all costs and benefits be equal.

In initial platformer mechanic generation runs $jump$ and $doubleJump$ lacked preconditions as this minimized the complexity of mechanics. To address this problem I added cost and benefit balancing based on the effects of movement mechanics. Each effect is assigned a benefit equal to the update effect absolute magnitude: $C(V) = |V|$. That is, larger changes to game state are considered linearly increasing benefits. Preconditions constrain mechanics and are assigned a cost of 1: $C(V) = 1$ The more limitations on using a mechanic are treated as a greater cost for the mechanic. A hard design requirement enforces 'balanced' mechanics by requiring the net costs and benefits of a mechanic are equal. Adding cost-benefit accounting led to the mechanics reported above.

While this cost-benefit model is simple, it allows humans to provide additional input to generation in form of additional design requirements. As with mechanic adaptation, the extension to the basic representation was trivial: a set of predicates that assign values to preconditions and effects were added as input and a design requirement for balancing

these costs and benefits was added. The low-level representation of the preconditions and effects of mechanics enables design constraints that reflect human heuristics for shaping the structure of game actions.

3.5.3 Multi-agent Games

The examples in this work have focused on games with a single active agent. Many game domains have the player face scripted opposition in the form of non-player characters. In a platformer these are enemies that patrol areas; in a RPG these are enemies that attack the player in battle. Enabling a game generator to consider the actions of these agents (when not fixed by a policy) allows for considerations of how opponents may alter the play space of a game.

The representation above only considers a single player agent, but the extension to multiple agents is straightforward. To account for the possible ways of playing out actions for each agent I augmented the planner to track state fluents specific to each agent (including actions performed and relative perceived state). Playability requirements also become agent-specific to account for the differing goals of agents. While the player goal in the RPG may be to kill the enemies, the enemy's goal is likely not suicide, but to kill the player.

Playability checks now must pass the conjunctive goal of all agent goals being possible with these extensions. In many cases these goals can be adversarial: the planner used here prevents explicit modeling of game-theoretic competition. There are a number of ways to circumvent this limitation by defining appropriate player and opponent goals. One approach (used in the platformer domain) is to ensure the player can achieve their goal situation before the opponents achieve their respective goal situations. The planner still ensures all agents may reach their goal situation, but by finding a plan where the player finishes first the game is guaranteed to have a way for the player to win without losing along the way. In the platformer this translates to the player being able to reach the end of the level while also showing the enemies could potentially overlap with the player (kill the

player) to achieve their goal.

A second approach (used in the RPG domain) is to define opponent goals in a way to improve player experience without directly negating the player's maintenance goals. The playability checks then optimize for all goals simultaneously. In the RPG this translates to the opponents aiming to minimize player health while keeping the player alive; in the ideal case it appears that enemies are fighting the player but fail to slay them.

The addition of multi-agent modeling is computationally costly (due to the increase in states being tracked and goals being optimized). These costs, however, enable modeling a wider range of game genres where their are other active non-player characters. Further, these models allow considerations of collaboration in games by considering two players as agents coordinating toward a set of goals. As with the extensions above, the changes to the base system were minimal: here the base predicates were merely extended to track one additional element indexing actions or relative state by the agent of interest.

3.5.4 Multi-instance Progressions

Platformers (and most game genres) typically introduce new mechanics to players over a sequence of instances (levels). Generalizing mechanic generation to include requirements on which mechanics are used along a progression requires two additions: planning across multiple levels and providing requirements on mechanic use. To implement multilevel progression I augmented the initial state and playability requirement definitions to be specific to levels with a level index. Playability checks must ensure the given mechanic set can yield valid playtraces for all levels provided, treating each as a separate planning problem with the same set of mechanics.

The constraint solver can enforce many types of progression across multiple levels. For example, requiring the number of mechanics used in each level increase over a level progression or requiring the mechanics used in each level reappear in all subsequent levels. In tests using these requirements the system has sequentially introduced the $jump$ and

$doubleJump$ mechanics above when provided a two-level sequence where $doubleJump$ is unnecessary to reach the goal in the first level, but both mechanics are required to reach the goal in the second level. In general the generated mechanic sequences often involve enveloping mechanics with a weaker and stronger (larger effect) version of the same mechanic. Progression requirements often encode a notion of training players by needing to master additional skills (see [23, 24, 56]). The more atomic mechanic representation used here can also require the progressive introduction of preconditions or effects (as in the $doubleJump$ introduction of an event precondition). These requirements allow more nuanced ideas of progression than previously done by using elements of the mechanics being introduced to the player. Again, the extension to the representation was trivial: adding an index to track the specific instance a state fluent was related to and then proving plans for all instances.

3.5.5 Control Generation

Platformers depend heavily on game controls. Control assignment can play an important role in how people play a game by making similar mechanics easier or harder to execute during play. While previously ignored in game generation systems the choice of controls can often play an important part in the realized space of play in a game [211].

To investigate this problem I considered a simple form of control assignment by mapping mechanics to inputs (as button or button combination presses). The system can assign controls by taking a set of control commands and adding these controls as additional preconditions for mechanics. One hard design requirement ensures there is always a single unambiguous mechanic for an input. This prevents control assignments where a single button press could trigger two mechanics simultaneously (even after considering whether the context is different via the check for mechanic preconditions). Another hard design requirement ensures all mechanics have at least one input and no two mechanics with the same preconditions use the same set of inputs. This ensures all mechanics can be triggered

by controls and no two mechanics could potentially fire in the same situation. One soft design requirement encodes simplicity by minimizing the number of inputs used in total and number of inputs used per mechanic. Another soft design requirement encodes a simple notion of 'intuitive' mappings by maximizing the use of the same buttons for mechanics with overlapping effects on the same entity-parameter-value settings.

In tests using this control mapping technique the system has generated (relatively) semantically sensible platformer controls. Inputs to the control mapping were the $jump$, $doubleJump$, $lift$, and $ride$ mechanics as above and a set of 6 input buttons for a 4-directional pad with two action buttons (A and B). Resulting control assignments used either 3 or 4 input buttons, trading off minimizing the total number of buttons used against minimizing the number of buttons used per mechanic (Table 3.4). Different assignments used different buttons for the same results. Note that no two buttons had identical preconditions, meaning a (non-optimal) assignment could have used a single button for all actions. This control assignment task illustrates the value of a low-level representation of mechanics for considering new game design elements previously overlooked in game generation research.

Table 3.4: Control assignment examples

	3 button	4 button
jump	↑	↑
doubleJump	A + ↑	→
ride	A	A
lift	B	B

3.6 Playable Game

The system described in this chapter generates definitions for playable games. Using these definitions a playable version of the game requires a way to store ongoing game state, gather player input, and present state to players. As a test case I implemented a platformer-style game domain (Figure 3.6). This domain has the player navigate to a goal state from

an initial state while avoiding enemies. Mechanics move entities in the world state, with different mechanics implementing different movements. For simplicity the UI shows the player, enemy, goal, and block positions. Mechanics are indicated by numbered circles, which show movement vectors of the player and other entities on mouse hover-over.

Figure 3.4: Game world state visualization for playable platformer domain. Player is represented by the wizard, enemy by the robot, and goal location by the green star. Numbered circles show possible movement vectors on hover-over for different mechanics (here three options with indices 1, 2, and 5).

To run the game I implemented a simple game engine in ASP to track ongoing game state, define valid actions a player may take, and update game state. The engine takes as input a game state and mechanics generated by the mechanic generation system above. At each time step the engine stores the current game state as the set of $Holds$ predicates in a flat text file. To provide valid player actions the planner portion of the generation system checks which actions have their preconditions met for the player agent at the current time step. These actions are presented to the player and the system waits to receive a player input in terms of choosing one of these actions. Once an action has been chosen the planner is then used to update game state to yield a new set of $Holds$ predicates. The planner also checks whether the player has failed or succeeded at this time. As an additional constraint

the game limits the number of player turns, with the player failing if they have not reached the goal state by the final turn. Note that the number of turns is defined as an input to the generation system, so the generated platformer levels are guaranteed to have valid solutions.

I used the Unity3D[1] game engine to handle rendering the game state, gathering player input, and feeding that input into the ASP engine. In the Unity3D implementation, player input consists of mouse clicks on the numbered circles indicating mechanics. Rendering used the native Unity3D engine and all processing used external calls to the ASP solver. Two other versions of the game used different engines to run the game: a simple text-based interface and a Twitter[2] bot. The text-based interface simply outputs the raw $Holds$ predicates and gave players a numbered choice of action options. Input takes the player text value for an action. The Twitter bot used the Unity3D renderer to create screenshots and provided text to define the state changes defined by the mechanics. Player input was gathered through replies to tweets of a game state.

While making a playable version of these simple games is straightforward this highlights open problems in visualizing game state and game user interfaces. I chose the 2D platformer domain due to it's widespread familiarity and the relative ease of showing mechanics as movement in space. Purely numeric domains—such as RPGs—are also relatively straightforward as changes can be shown through text. Moving to more general classes of games, however, will pose new challenges in creating general approaches to rendering game state and providing interfaces for showing the effects of user actions. It is likely possible to capture broad classes of games—2D or 3D movement-based games, primarily numeric simulations, and so on—in common vocabularies, though this remains an open topic for research.

[1] `https://unity3d.com/`
[2] `https://twitter.com/`

3.7 Limitations and Future Work

In this chapter I presented a representation for game domains based on treating games as planning problems. The work presented here illustrates the value of taking this perspective for generating game mechanics, instances, and a number of related features. However, this model has a number of limitations that highlight challenges in the area of domain-agnostic game generation. Some challenges require extensions to the representation or algorithm: currently the game state model is not generated and design, playability, and adaptation criteria are used as input (rather than derived by the system). Other challenges will require new representations: generating game assets, modeling adversarial game domains, and representing domains outside the realm of deterministic, discrete, turn-based games.

3.7.1 Generating State Models

Generating the game state model is a conceptually straightforward extension to the game generation model here. Instead of treating the set of entities, parameters, and allowed parameter ranges as fixed, it is possible to allow the system to select these combinations from a larger pool of options. Generating the state model introduces two challenges, one computational and one semantic. As a *computational* challenge, exploring an unbounded space of parameter combinations requires a different algorithm for generation to appropriately limit and intelligently expand the state space model used in the game. Without any additional design or playability criteria the game state model could become arbitrarily large. This modification would also remove the ability to guarantee exhaustive or (near-) optimal choices of game designs from the design space. Addressing this side of the challenge would require different algorithms for generation that intelligently expand the search in the design space of game state models. New structure in the form of additional design constraints may also help constrain search to bound the space of alternative game parameterizations. This will still require additional research into ways to recognize structurally

equivalent parameterizations or decompose generation into sub-games that interlock (an approach taken toward general game playing agents [46, 79, 173]).

As a *semantic* problem, the current representation excludes any explicit notion of the meaning of the game state model to people. Any new predicates introduced into the game would lack any ready interpretation to people. In practice leads to new game elements lacking a human-legible name. The problem of generating new semantic content to fit in a known domain has received attention in the computational creativity literature and there is promising work based on hand-authoring domain semantics [226, 126] or mining pre-existing corpora of semantic content and using that to inform semantic labels [35, 36, 37]. Extending these approaches to learn the connection between a game state model and the semantics of that model is non-trivial, but may provide the additional constraint needed to render search computationally tractable.

3.7.2 Generating Design Goals

An alternative way to extend the scope of generation is to provide the system with explicit control over design and playability criteria or adaptation goals. Choosing these constraints faces similar problems as the choices in game state generation: the constraints serve to limit the search for potential games and embed a designer's concept of what should or should not be possible in a game. The choice of playability criteria is sometimes feasible: for example, the system currently generates goals and maintenance goals for game instances, but these are could be based on generalizing from examples in game instances to choose parameters for similar goal structures. Additional control over design and playability criteria will require ways to iteratively expand this search space or use hard-coded or learned semantics to constrain search. As with generating the game state model, this requires parameterizing the space of input parameters to the current generator. Guiding search in this space may be feasible by learning the consequences of design choices on the space of play and using that feedback to inform the creation of design and playability criteria or notions of optimality

for adaptation. Ultimately, generating all three types of constraints for the system can benefit from learning about the space of play enabled by designs and using this feedback to guide search. In chapter 6 I show one way to guide choices of design parameters in the late stages of design by using feedback from player behavior to guide the choice of design parameters.

3.7.3 Linking Semantics to Mechanics

The algorithmic creation of game assets is the primary concern of the academic field of procedural content generation in games [33, 180, 222]. Procedural content generation also has a history in commercial game development, from early examples including *Rogue* [4] and *Diablo II* [14] to recent growing popularity with contemporary examples including *Minecraft* [134] and *Spelunky* [137]. Commercial game developers often use procedural generation to give players variety in the content they experience on repeated play. At the finest level of detail, procedural generation of instantial assets such as trees [94] or rocks [51] has been used to reduce development costs for creating large game worlds. Randomization of level instances (e.g., *Rogue*, *Diablo II*, or *Spelunky* [243]) has served as a way to give players gameplay variety and encourage improvisational problem-solving over rote memorization of game levels. Procedural generation has also been used at a larger scale to construct full universes in games: *Spore* [130] and *No Man's Sky* [85] generate planets and their environments down to the individual creatures populating these planets, giving players the opportunity to explore seemingly endless new universes. Other efforts have used simulations to generate histories of game worlds, spanning the evolution of cultures and historical interactions among in game societies: Ryan et al.[166, 165], *Dwarf Fortress* [3] and *Ultima Regium Ratio* [102] use these rich simulations to provide a grounding social and cultural context for gameplay.

Tying choices of game content to the aesthetics and meaning of games has received little attention from either academic or commercial efforts. Treanor et al. [226] *Game-O-Matic*

take the approach of having players author conceptual content and using a pre-authored grammar to choose game assets and mechanics based on player input. Martens et al. [127] extend this approach to a bi-directional pipeline for interpreting and generating games using a static analysis of game mechanics using answer set programming. Human-authored knowledge informs the system as to how to derive gameplay dynamics from game mechanics and then derive the semantic meanings of the game from the enabled dynamics. Cook et al. [35, 36, 37] use a generation pipeline where a system takes as input a high-level semantic query in the form of a word or phrase and then searches databases for related game assets. In these systems aesthetics serve as a framework to guide choices of content for arcade-style gameplay—aesthetics dictate how to interpret an interaction as 'good' or 'bad' when progressing toward a high score or goal state. For Cook et al. and Treanor et al. this is used to guide choices of how to populate entities in templates for adversarial relationships common to arcade and platformer games. For Martens et al. these evaluations guide interpretations of what interactions are 'good' or 'bad' in a game. Outside these abstract frameworks of valuation, however, there is little work to derive more complex aesthetic or semantic statements from a game's structure resembling the critique people make of games [15, 16]. Creating modular and extensible valuable frameworks for generative systems has great promise to yield new types of games that ground different conceptual and aesthetic frameworks in (simple) game systems.

The mechanic generation framework in this chapter demonstrates the need for developing general models to constrain the semantics of generated mechanics. When combining domains, the system would often produce results that included unnecessary movement, such as $healBoth$:

$\langle healBoth,$

$\{\ \},$

$\{ \quad \langle relative, 1, Update(health(e), +1)\rangle,$

$\langle relative, 1, Update(health(enemy), +2)\rangle,$

$\langle relative, 1, Update(xPos(e), +2)\rangle,$

$\langle relative, 1, Update(yPos(e), +2)\rangle\}\rangle$

Or would combine movement and RPG statistic changes across entities without any clear pattern, such as $healthJump$:

$\langle healthJump,$

$\{\ \},$

$\{ \quad \langle relative, 1, Update(health(e), -1)\rangle,$

$\langle relative, 1, Update(xPos(e), +2)\rangle,$

$\langle relative, 1, Update(yPos(e), +2)\rangle,$

$\langle relative, 1, Update(xPos(enemy), +1)\rangle\}\rangle$

These cases illustrate the need for the system to possess knowledge of the human world to capture the ways people expect related entities in a game to function. Constraining mechanics to influence a single entity would prevent these cases, but also prevent mechanics like $lift$ above—simple constraints alone are unlikely to prevent many types of nonsense mechanics. Instead, the constraints on mechanic generation will need to capture the notion that relative movement is only possible along the lines of a naive physics, where proximity is required to have action (unless transmitted by some external force). Realizing a general form of human-like semantics will be a challenge, but can in turn greatly enhance the ability of this system to create mechanics that are readily interpreted by people. This will likely require a combination of intelligent authoring and means for automated systems to mine human examples, feedback, or pre-existing corpora for knowledge of how people expect the world to function.

Similar to the approach of mining corpora of semantic content, researchers are also

learning game structures. Dahlskog and Togelius [49], Snodgrass and Ontañón [200, 199, 201], Summerville et al. [210, 207], Guzdial et al. [81], and Jain et al. [101] learn the structure of platformer levels from corpora of platformer level sprites with different underlying machine learning techniques. Summerville et al. [208, 209] learn action-adventure game structures from similar corpora. Guzdial and Riedl [80] take an alternative approach by using computer vision to parse videos of people playing platformer levels and learn level design from the visual information; Summerville et al. [207] apply the same parsing technique to support player-tailored level generation.

To date, however, these approaches have not addressed the general problem of connecting game assets to game state models: i.e., learning how the choices of game assets are related to the choices of game state and transition models. Games studies researchers have considered this topic in terms of the *operational logics* of a game. An "operational logic defines an authoring (representational) strategy, supported by abstract processes or lower-level logics, for specifying the behaviors a system must exhibit in order to be understood as representing a specified domain to a specified audience" [129]. While the work on operational logics to date has been primarily driven by game studies analysis [15, 129, 238], there is a great opportunity to use the lens of operational logics to guide how a system learns to connect a game domain (the underlying logic) with the game's assets (representational strategy). Addressing this connection will require domain-agnostic game representations that allow learning how multiple games of similar and different genres function at the level of game state and transition model, while also capturing choices of game representation at a semantic level. Recent work has begun to approach this topic using hand-coded models of operational logics and the relationships among representational choices and game mechanics, demonstrating the potential for game analysis and generation in an automated fashion [127]. This work lays the foundation for future extensions that allow systems to automatically learn about these relationships and generalize them to generate novel content.

3.7.4 Adversarial Games

Games with more than one player are not possible in the current representation. This derives from the fact that planning represents the goals and intentions of a single agent, rather than searching the space of optimal strategies for multiple agents with differing goals. The approach taken earlier for representing the goals of enemy agents assumes agents that are ultimately aiming toward the player still being able to complete the game. That is, all planning is done by a single agent, rather than adversarial minimax planning between two or more agents.

Truly adversarial games cannot be represented merely by a single agent planning and instead require game-theoretic adversarial search for optimal play between agents. Capturing games with multiple competing parties requires replacing the planner with game theoretic search among agents. At the same time this search would still need to consider the space of possible plays between agents to guarantee conditions can be met in the game (e.g., both players can win the game). The generate and test approach taken here could be modified by replacing the planning with a game theoretic analytic solution (e.g., computing Nash equilibria if the mechanics allow) or search. But this replacement would be both computationally costly and require further consideration of how to define design and playability criteria that are sensible for an adversarial game. Browne and Maire [20] addressed this challenge by defining evaluation criteria over playouts between (not necessarily optimal) agents while Jaffe et al. [99] applied game theoretic evaluations to check win rate balance between optimal agents. Extending this system to handle adversarial situations will require similar definitions of the quality criteria of a game and done so in a way that accounts for a range of player skills. In the next chapter I show how to apply Monte-Carlo Tree Search (MCTS) to handle playing a broad range of adversarial games (an idea also suggested by Jaffe (Chapter 5, p. 60 in [99])) and use this to evaluate the design of these games.

3.7.5 Other Game Domains

The work in this thesis is specifically targeting deterministic, discrete, turn-based, fully observable games. These assumptions simplify both the design space being modeled and reasoning requirements on agent behavior in the play space. Lifting any of these restrictions can enable modeling different game domains with additional computational challenges. The use of a planning perspective on game play is valuable in suggesting ways to broaden to new domains. Continuous time domains can be addressed through planning technologies for real-time scheduling. Non-deterministic domains can be addressed with probabilistic planning and domains that are not fully observable can also be addressed. The primary challenge in most cases will be developing appropriate ways for plan failure to feed back into generation to guide design space search. Currently this problem is being addressed by using a single constraint solver to implement both search in the space of design and planning, thereby feeding back learned constraints from failed plans directly into design space search. Learning design space constraints and heuristics from play space search remains an open question that will be crucial to adopting more sophisticated planning techniques in a computationally tractable fashion.

3.8 Potential Impact

Game design research has the potential to change the way games are made and the experiences available to game players. In this section I briefly discuss how the game generation system in this chapter might influence game designers and players.

3.8.1 Game Designers

The system in this chapter works from an abstracted forward model of game mechanics that serves as a rudimentary game engine. In practice game designers typically use fully featured game engines to facilitate the creation of game instantial content (levels, areas,

gameplay systems, &c.). These engines provide useful tools for authoring content, but generally provide little or no support for understanding or evaluating the consequences of game design decisions. Designers would gain new tools to address these challenges by linking existing game engines to abstracted frameworks like the representation in this chapter.

Enabling a game engine to validate that players can accomplish a collection of goal states is a powerful tool for determining whether level designs are functioning as intended. Even if generation and adaptation are never used directly, planning in an abstract representation of game content can provide useful diagnostic information on what is or is not possible in a game. The system in this could provide all (abstracted) action sequences that achieve different game goal states, allowing designers to rapidly iterate on a level design or mechanic design to constrain the ways of completing content to the set of desired outcomes.

As a general game representation, integrating this tool into a new generation of game engines would provide an easy interface for AI agent play and design validation. General representations that can be programmatically defined for each game allow for a new class of tool for automatically suggesting content changes or providing alternative views of a design in terms of completion toward different game goals. In the future designers may be able to add a stage of design iteration without players that emphasizes abstract, general properties of play in a game. This in turn can enable new approaches to game design emphasizing desired ways games function, rather than placing an emphasis purely on the experience of using a game. In games that must enforce certain types of play (e.g., training games or games with a purpose), these tools can greatly accelerate the process of authoring content that produces intended outcomes.

3.8.2 Players

Players stand to benefit from generic generation systems by being able to experience new genres of games built around generation of mechanics and systems. New designs become

possible: for example, puzzles games where players pick an arbitrary goal on a game level and the system generates actions the player must use to reach that goal state. Other designs might leverage general domain combination to give players a modular "toolkit" of base game representations they can choose to combine and play example levels from. Alternatively, players might be given a goal to force a system to generate certain mechanics on a level, changing the goal states used as input to the system. In this game genre players would provide inputs to abstracted design tools in order to create desired game outputs. Flipping the roles of players and designers has already shown great promise both for entertainment (*Minecraft*) and games with a purpose (*Foldit* [42])

Simplified, abstract design tools can also increase the potential for people to use games as simple expressive media. People can far more readily explore ways of using games to express their experiences when game authoring becomes simplified (within a constrained genre) to authoring a handful of goals and levels. As these game authoring tools become trivial to use, a new form of game creation analogous to the relationship of Twitter to writing may emerge: a new class of ultra-streamlined game used to express an emotion or snippet of experience. Ultimately the growth of these games requires far more work in creating tractable small representations and robust user interfaces, but the strength of modular frameworks like in this chapter lies in modeling and generating games in these small-scale, well-defined designs.

3.9 Summary

In this chapter I presented a representation for games based on treating gameplay as a planning problem. The representation models game domains using a declarative model of the possible states in a game. Generating game mechanics and instances is solved using a generate and test cycle constrained by design and playability criteria that define desired mechanic structures and play behaviors, respectively. The low-level representation used enables a deep exploration of the game mechanics possible in games across many domains,

while the declarative domain model allows ready combination of game domains. Using a low-level, generic representation supports further game generation capabilities including adapting game mechanics to fit new content, incorporating designer intuition on the costs and benefits of how mechanics work, modeling games with multiple agents, controlling progressions of levels to introduce mechanics based on their functionality, and mapping mechanics to button-based controls. Together this work provides a backbone for a model of iterative game design by treating the problem of game generation in a low-level, domain-agnostic fashion.

Using a planning model for gameplay ensures games have desired properties such as victory or failure conditions. But designers are often equally concerned about the *typical* behaviors players may have in a game. Typical behaviors also often differ depending on individual differences among players, such as reaction time, ability to plan ahead, or preferred types of content to experience. Addressing these differences requires a different model for assessing the ways people play a game: simulated agents that can be configured to play in a variety of ways. In the next chapter I present the use of Monte-Carlo Tree Search (MCTS), a domain-agnostic stochastic planning algorithm, to generating examples of agent play at varying levels of skill. As many design goals for player behavior center on the *actions* players take, rather than the *states* they visit, I develop a framework of metrics to assess the choices faced by and used by players. I show how to evaluate game designs by combining this framework for action metrics with behaviors samples generated from MCTS agents of varying skill. The evaluations examine the classic word game *Scrabble* and a simplified card battling game mimicking the mechanics of *Magic: The Gathering*, showing how the metrics find *Scrabble* effective at differentiating agent strength while the simplified *Magic* game does not. This evaluation and framework provides the groundwork for automatically assessing the space of play afforded by a game design.

CHAPTER 4

ACTION SAMPLING

4.1 Introduction

Games typically afford a broad range of ways to play. Designers often have goals for that space of play, such as creating a core gameplay loop [167] and/or balancing the competitive elements of a game [61]. But reasoning on a static description of a game to understand the dynamics of play possible is a challenging task. This leads to two problems: (1) gathering examples of a variety of player behavior in a game and (2) evaluating those examples to determine the quality of the space of play in a game. *Behavior sampling* is the problem of gathering examples of expected player behavior in a game given the design of the game. *Gameplay analysis* is the problem of evaluating the quality of a space of play given examples of behavior in the game. In this chapter I present a general approach for behavior sampling that proxies player skill in turn-based games. The previous chapter illustrated how a system can automatically understand whether certain game states are possible in a game; this chapter demonstrates how to examine the range of choices players face (and act on) in a game while also accounting for differences in player skill.

Player skill plays a key role in game design: a single designed game must cater to players with different abilities to execute actions in the game. Designers use knowledge about player differences to tune a game design to allow for a desired range of differences in player skill. For many designers, a game design is only successful when player skills result in different outcomes, allowing higher-skill players to beat the weaker opponents. Thus, enabling designers to understand how a design differentially influences players based on their skill is important for informing iterative design.

Player skill comes in many forms: from reflexes to execute carefully timed actions to

social cunning to deceive other people. In this work I focus on player skill in the form of the ability to plan ahead in a game. Planning ahead involves planning an action, considering the next actions taken by a player (or their opponent), and planning subsequent sets of action in response. Planning ahead is a core component of gameplay in most turn-based games (and many real-time games), making it a useful general form of skill to consider in iterative design.

Behavior samples are only useful when they can be analyzed to inform design decisions. I present four metrics that use the *actions* agents of differing 'skill' take in a game to evaluate two game designs—this enables a general framework for evaluating the strategic space of a game. Unlike prior work in gameplay analysis this framework emphasizes the actions agents take in a game, rather than the states of the game agents visit, to provide perspective on the *strategic* space of a game. In many games, player experience derives primarily from the choices players can make in a game. Sid Meier is famously quoted as saying "A game is a series of interesting choices."—action metrics are designed to capture how well a game delivers on providing a series of interesting choices as a way to computationally formalize elements of this game design philosophy. In the next chapter I present methods for a creative system to apply this analysis to a space of game designs to find optimal design iterations and learn how those design choices influence the space of play.

4.2 Behavior Sampling

Computational techniques for analyzing games require means of generating examples of play and metrics to evaluate those examples to determine the quality of the space of play in a game. The notion of "quality" should reflect features of interest to game designers or game players. There are three main approaches to solving behavior sampling and gameplay analysis: (1) human playtesting, (2) model-checking, and (3) player simulation. Playtesting with humans can be effective for informing design questions sensitive to how people act, but these methods can be expensive and time-consuming while failing to check everything

needed of a game [176]. Any sample playtest population may not accurately reflect the full range of players in the game, skewing the evaluation of expected behavior. Further, a playtest group may not exhaustively test features of a game that a larger population will attempt, making it challenging to generalize from playtests to the full space of play in a game. Consequently, human playtesting methods emphasize generalizing early trends of human behavior as indicators for potential ways people may play, rather than measuring the space of play afforded by the game. Playtesting with people provides the ultimate answer to how players will experience a game—a behavior sampling algorithm provides cheaper alternatives to playtesting to augment the design process. Using automated techniques for behavior sampling offers the ability to direct playtesting to elements of design that do not directly require people to provide design guidance.

Model-checking methods (as used in the previous chapter) determine whether certain behaviors are possible in a game [139, 195]. The planning approach to game generation in the previous chapter is one example of a model-checking method: an automated system checks whether specific sequences of behavior are possible and whether certain game states may be reached. Model-checking is an effective way to address the limitations of playtesting in exhaustively searching for undesired behaviors in a game. However, the model checking approaches in the previous chapter fall short of on two counts: (1) modeling sets of *likely* behaviors; (2) scaling to complex or large games. First, model-checking techniques are designed around testing for the presence or absence of behavioral features. This means model-checking and proofs can show whether or not behavior is possible in a game—giving a sense of the bounds on a space of play. But design queries often involve the subset of a playspace players typically use. Or, design queries may concern aggregate properties of how people play across multiple sessions or as groups: e.g., which paths players typically take through a platformer level. These questions cannot be directly answered using model checking as the techniques do not represent notions of *likely* behavior.

A second problem of model-checking is the computational costs of checking large or

complex game designs. In many game designs the space of combinations of possible actions is prohibitively large to search exhaustively or prove properties on. In the card battling games *Magic: The Gathering* or *Hearthstone* players construct decks from a pool of cards and take turns drawing (in randomized order) cards from those decks and playing those cards. Exhaustively sampling this space requires considering all combinations of cards into decks (which may or may not have bounded size, depending on the game), all possible orders for drawing cards (which may have variable cards drawn depending on the cards played in the game), and all ways players might play cards when reasoning on hidden opponent state. Even just modeling agent play with decks known to both players (rarely the case in human play) requires considering all the ways a player and opponent might draw cards and make choices based on expectations about the cards held by an opponent. While games can often be represented more abstractly to reduce the search space, these abstractions require additional engineering to design for a game and lose fine-grained properties of player activity that may be of interest. In *Magic* or *Hearthstone* abstracting over card features loses important information about how players might be expected to act in a game. Model-checking is thus best suited to games with sufficiently abstract spaces of actions where the design queries of interest involve *possible* behavior, rather than *expected* behavior.

Simulation approaches to modeling human play in a game address both concerns about capturing how people are *expected* to play and concerns about game complexity. Simulated gameplay allows a sampling of behaviors possible in a game and often affords tuning the models to represent typical types of behavior in a game. Simulation-based playtraces can be generated and evaluated for both single-player games [91, 151, 152] and adversarial competitive games [100]. Simulation agents can address concerns about *expected* play behavior by emulating aspects of people's capabilities, such as reaction time [95] or memory and planning [19]. Researchers have trained simulated agents to reproduce human-like play behavior for movements in games [27, 148, 224, 231], and human-like ac-

tion choices in first-person shooter games [112], turn-based action role-playing games [88, 89, 90, 91], real-time strategy games [242], card battling games [54], and open world role-playing games [223]. The flexibility of parameterizable agents enables analysis in cases where games afford many levels of play: particularly when high-skilled players may pursue entirely different strategies to amateurs [61]. Modeling features of human capabilities in games can provide further ability to compare and contrast the expected play spaces when people have different capabilities. Simulated agents can also address concerns when games become complex by varying the number of simulations used or the complexity of the simulations to alter the extent of the play space explored. For complex game genres that do not afford ready abstraction—e.g., *Magic* or *Hearthstone*—this can be the only viable solution to understanding the play space. Agents can use randomized techniques and take probabilistic expectations to search the most important parts of the space of play [44, 45, 236]. Simulations make explicit the trade-off between how well (and how large) a subset of the play space is explored and the amount of computation required to explore that space. Simulation methods trade off the guarantees of exhaustive search of properties of a playspace given by model-checking for providing easier modeling of expected behavior patterns to offset the cost of using humans as playtesters.

Techniques for behavior sampling and analysis ultimately aim to provide an understanding of how people can play a game. To date, most analyses of gameplay have emphasized the *states* players visit in a game, to understand which content players consume [176, 234]. Yet playing a game is often more about the *actions* players take rather than the *states* players visit. Player skill most often manifests in the actions taken and strategies executed. Particularly in games where players compete with one another, the features of interest in the space of play concern the strategies players may pursue, rather than the particular game configurations they may visit. Analysis of player strategies necessitates representing a range of levels of abstraction for behavior: from granular individual action choices through chains of actions to execute a high-level strategy. I address this point by presenting four levels of

analysis for player actions in a game: summary statistics, atoms, chains, and action spaces. *Summaries* are high-level metrics of overall gameplay characteristics. *Atoms* are metrics of individual, context-free player actions. *Chains* are metrics about the relationships among sequences of player and inter-player actions [13, 25]. *Action spaces* address the range of possible actions over the course of a game [61]. These four levels provide multiple layers of abstraction for evaluating and comparing game designs.

To approach behavior sampling I focus on a specific subset of games and a specific type of player skill. The game subset I use are discrete, turn-based, fully observable games. By removing the restriction of deterministic outcomes (compared to the work in the prior chapter) I consider a larger set of games, including simplified models of domains like *Magic* or *Hearthstone*, without creating excessive additional computational complexity. I consider player skill in terms of capability to plan courses of action in a game—extending the planning approach of the prior chapter. In turn-based games the ability to choose an appropriate action typically requires modeling the game state several actions in advance, so I use agent search depth as a proxy for this human skill [19, 61]. Here I simulate play using a stochastic planning technique with demonstrated success in general game playing: Monte Carlo Tree Search (MCTS). Unlike prior work that has emphasized how well MCTS can play games to win, I use MCTS as a tool to sample the space of play. Varying the computational resources allowed to MCTS serves as a proxy for varying player skills. I use these agents and the four sets of metrics above to analyze the design of two games: the classic word game *Scrabble*[1] and a card game I developed as a simplified model of parts of *Magic: The Gathering* and *Hearthstone* called *Cardonomicon*. The *Scrabble* analysis shows how the metrics can identify balance in a game, while the *Cardonomicon* analysis reveals flaws in the game's design.

[1]The work on the *Scrabble* domain was joint work with Brent Harrison.

4.3 MCTS Background

Monte-Carlo Tree Search (MCTS) is a general game-playing technique with recent success in discrete, turn-based, and non-deterministic game domains [21]. MCTS is a sampling-based anytime planning method that can use additional computational resources to more fully explore a space of possibilities, allowing control over the balance between computational time and exploration of the full space of play. I chose MCTS as a behavior sampling algorithm for it's proven high-level performance, domain generality, and variable computational bounds. For simplicity, the study domain is perfect information (allowing players to see one another's hands) to facilitate use of MCTS.

MCTS's game playing success derives from modeling the quality of a space of actions over the course of a game. MCTS models game play using a tree to track the value of potential courses of action in a game. Actions to take are tree nodes and links between nodes indicate the next action(s) available after a prior action (Figure 4.1). Nodes for already attempted actions are *expanded* and not-yet-attempted nodes are *unexpanded*. Each leaf node in the tree tracks a reward value for the focal agent (the agent choosing an action, as opposed to the opponent) depending on if it won or lost the game. Typically, a reward value of 1 is assigned to wins and a reward value of -1 to losses.

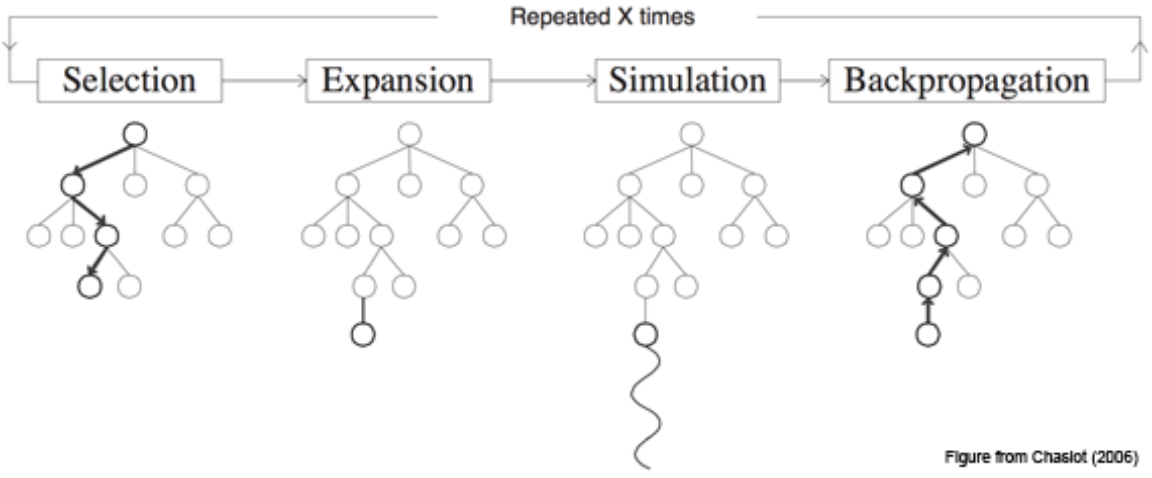

Figure 4.1: Diagram of MCTS algorithm steps from Chaslot (2006).

The MCTS algorithm has four steps (Figure 4.1):

1. **Selection** Descend a tree of expanded nodes until reaching an unexpanded node. Selection chooses the next expanded node to visit (among alternatives) based on a model of the expected value of taking a given action (visiting an expanded node).

2. **Expansion** Expand the set of actions available at an unexpanded node and choose a new node. Expansions visits new unexplored actions: MCTS algorithms (e.g., UCB1 [7]) typically ensure all nodes on a given branch of a tree are expanded before revisiting expanded nodes. This measure ensures the agent explores all alternatives at least once before honing in on nodes with high expected value.

3. **Simulation** Follow a fixed strategy (usually random choice) for how to act over all remaining unexpanded decisions until reaching the end of the game. Simulation is used to cheaply reach an end game state from a given point.

4. **Backpropagation** Use the reward from the end game state reached (e.g., win or loss) to update the expected value of the newly expanded node and all of its parent nodes in the tree. Backpropagation provides feedback on the value of nodes based on distributing credit for a simulation outcome among node choices.

MCTS balances between agents exploring alternative actions and exploiting known good actions. Typically selection uses the UCB1 algorithm, which picks a node using a combination of the average reward (eventually) received when taking the action and the proportion of all selections that used that node [30]. Note that UCB1 forces selection to first visit every possible move at least once before choosing among all visited nodes based on their value. I use UCB1 because this property ensures the agents fully explore the space of move options before continuing on to devote additional resources to better modeling the value of individual choices.

4.4 Skill-based Design Metrics

Gameplay analysis is often interested in aggregate properties of gameplay traces—sequences of player behaviors in a game. Traces can be viewed as a sequence of states or a sequence of actions. State analysis examines *what* players engage with in a game, yielding information about common states visited in game content or progressions used by players [5, 6, 116, 234]. Action analysis examines *how* players engage with a game, yielding information on what strategies players take and mechanics players do (not) use [61]. In this work I focus on action analysis metrics to understand player strategy and how they are sensitive to player skill. To date, researchers have emphasized understanding the states players visit in a game—this affords a sense of what *content* players engage with, but overlooks the choices players take (or consider) in the game. To many designers, the experience of a game revolves around the choices made in the game, famously summarized by Sid Meier as: "A game is a series of interesting choices." From this perspective, considering games in terms of the actions players take (or consider) is required to understand whether the game design is delivering on the intended experience(s) for a player. This is particularly true when designing competitive games, where a core component of the game design is whether players have strategic options available in a wide variety of scenarios [61]. Action analysis thus provides a complement to the typical state analysis applied to games, opening new possibilities for understanding how games create experiences for players.

Action analyses can be divided into four categories, with varying degrees of abstraction of the strategic space in a game:

- *Summaries* are high-level design metrics that aggregate playtrace features of interest. For example, the typical (median) length of the game or probability of the first-turn player winning in Chess.

- *Atoms* are metrics specific to individual actions in a game. For example, the frequency of playing a letter in *Scrabble*, potentially conditioned on a context like the

turn number in the game.

- *Chains* are gameplay patterns within or between players. *Combos* are regularities in actions taken by a single player: e.g., in *Magic*, tending to play a given pair of cards on the same turn (potentially due to positive synergies between the cards). *Counters* are action-reaction patterns in actions taken between a pair of players: e.g., in *Scrabble*, when one player spells "con" the opponent may often add "i" to form "icon."[2]

- *Action spaces* are sets of actions taken (or available) to a player, potentially over the course of a game. For example, in *Scrabble*, the number of valid words available to be played over the turns of a game or in *Magic*, the number of unique minions a player can play on each turn.

These categories are not intended to encompass all ways of analyzing playtraces, but instead to organize levels of analysis that share common techniques in terms of aggregating descriptive statistics and visualizing those results. Strategies for analyzing these metrics allow automating evaluation criteria for an iterative design system and also provide common analyses to support human design. These metrics only require sets of play traces as input and can equally apply to traces from humans or simulated agents. By only referencing actions taken in a game all of these metrics can be sub-divided by features of game players: here I consider player skill, though other features may be of interest (e.g., player gender or age [213]). The following sections clarify these definitions and provide examples for the *Scrabble* and *Cardonomicon* domains.

4.4.1 Summaries

Summaries overview features of gameplay to provide high-level summaries that guide further analysis and framing to interpret more granular analyses. Summaries are typically

[2]My definitions for 'atom' and 'chain' are distinct to those proposed by Dan Cook [34], but share the notion of distinguishing between single actions as atoms and patterned sequences of actions as chains.

single numbers that aggregate features of a game. Many game analyses for live games use summaries to track the health of an online game: the average duration of matches, number of users of the game, revenue earned per (paying) user, &c. For this work I focus on summaries applied to analyzing the design of the game in terms of strategic player behavior, leaving indicators of behavioral engagement and monetization aside. While these features are valuable when understanding live game performance, they are outside the scope of the skill-based modeling being done here, requiring new models to sample behaviors based on player engagement or monetization preferences.

Scrabble and *Cardonomicon* share summaries for typical game length, and the probability of the first-turn player winning. Other summaries include: game play duration, typical turn duration over the course of the game, number of actions taken in turns in a game (overall and split over the course of the game), probability of winning for players of different skill levels, &c.

4.4.2 Atoms

Atoms summarize the use of individual actions in a game, providing information on which game mechanics are (not) being used and are (not) available to be used. In *Scrabble*, atoms include the use of individual letters or the frequency of making or being able to make words. In *Cardonomicon*, atoms include playing cards on the board or using cards to attack other cards. Atoms form the core of actions players take in the game, revealing cases where actions may be too general and effective or never used.

Analyzing atoms can inform game balancing decisions around whether specific actions are over- or under-used in the game. Action analysis can consider both the actions *taken* by agents as well as the actions *available* to agents to use. Available actions are the actions possible at a point in a game: words to make in *Scrabble*, cards to play in *Cardonomicon*, plot choices to pursue in interactive fiction, or reachable locations to move to in *Super Mario Bros.*. In planning terms, an action is available in a given state if all of its pre-

conditions are met in that state. Understanding how often agents take actions provides information on the strategic appeal of an action to agents with a given level of strength. Understanding how often actions are available to agents reveals whether conditions for actions are too restrictive (or not restrictive enough). Further, the gap between action use and availability provides perspective on how relatively useful players perceive different actions when making a strategic choice. Slicing analysis of atoms by the strength of the player taking actions can reveal whether certain actions are more useful for players as they develop a deeper strategic understanding of a game. A lack of differences between player skill levels may indicate little advantage to greater learning in the game, itself a potential design flaw.

Descriptive statistics on atoms include computing the frequency of action use, frequency of action availability, and difference between the frequencies of use and availability. Visualizations of atoms typically use histograms to show these statistics across actions in a game.

4.4.3 Chains

Chains summarize recurrent play patterns in segments of traces. I consider two types of chains: combos taken by a single player and counters of one player responding to action taken by another. *Combos* are sequences of actions a single player commonly uses together. Combos are common in games with multiple actions per player turn or real-time action. In *Cardonomicon* combos include playing cards successively or using sets of cards to attack; *Scrabble* has no combos as players take a single move each turn. *Counters* are sequences of actions that occur when two (or more) players respond in similar ways to actions from other players. Counters are common in games with alternating turns or simultaneous turns. In *Cardonomicon* counters can occur when one player plays a card on the board and their opponent attacks it using a specific other card; in *Scrabble* counters occur when one player forms a word and their opponent builds a longer word from that base.

Analyzing chains can reveal emergent strategy within a game, including chains of ac-

tions that may exercise a skill [34] or ways players have discovered to thwart their opponents [25]. Understanding which combos or counters are common can inform decisions to alter the restrictions placed on using an action or alterations to how effective an action is. Segmenting analysis of chains by player skill can reveal how player strategies evolve with greater proficiency in the game and reveal balance concerns if specific actions disappear from chains used in high-level play.

Unsupervised learning techniques to identify chains include a wide array of sequence mining techniques, including itemset mining, rule mining, sequence analysis, and hidden markov models [84]. Analyzing combos requires traces consisting of single player actions within turns. Analyzing counters requires traces of player-opponent interactions over a desired number of turns. Visualizations vary by technique, but include histograms of short chain frequencies, graph visualizations highlighting common action-action transitions, and playtrace browsers highlighting trace subsequences matching a chain from a larger collection [176, 234, 133, 151].

4.4.4 Action Spaces

Action spaces summarize atom use over time or game states. In *Scrabble*, action spaces include the number of distinct tiles played or the number of distinct words available to complete across turns in a game. In *Cardonomicon*, action spaces include the number of distinct cards available to play or average number of cards able to attack across turns.

Analyzing actions spaces can reveal how a game progresses from the perspective of player choices. This analysis can identify cases where a game is too restrictive or overwhelming with too many options, informing decisions about the pacing and growth of game complexity over time. Considering differences in actions spaces between low- and high-skill players can reveal cases where skill allows better use of the game actions or where low-skill players fail to use actions commonly used by high-skill players.

Descriptive statistics on actions spaces are typically frequencies of actions used or avail-

able over the duration of a game. These statistics may condition on game conditions or player features, such as when a player has a specific cards in a deck in *Magic*[3] or whether the player is high or low skill. Visualizations of actions spaces can use line charts to show variations in frequencies over the duration of a game and to compare these frequencies across contexts (e.g., multiple lines for players of differing skill).

4.5 Metric Application Case Studies

Skill-based design metrics enable analysis of player strategies in games. To demonstrate how player simulation and skill-based metrics can aid in game design evaluation, I performed two case studies. The first case of the classic word game *Scrabble* explores how these metrics can evaluate a balanced game. The *Scrabble* case verifies these metrics can identify balance in a design and differences in player skill. The second case study of *Cardonomicon* shows how these metrics can assess a game with an intentionally flawed design. The *Cardonomicon* case shows how simulated agents and design metrics can identify game flaws and inform future design iterations.

4.5.1 Agent Design

To provide an even comparison across game domains I used MCTS agents to play both games, altering the number of rollouts used as a proxy for human ability to reason ahead [19]. Addressing generic approaches to behavior sampling I use the strength of MCTS to typically play well in discrete, turn-based, adversarial games and combine this with the ability to tune MCTS to have better or worse play [21, 143, 153, 185]. A key parameter to the MCTS algorithm is the number of *rollouts* used—the number of times the full cycle is repeated. By increasing the number of rollouts allowed to an agent, the agent can more fully explore the value of possible actions in the game and improve play (Chapter 5, p. 60 in [99]).

[3] *Cardonomicon* does not include deck choices to make game generation and play space analysis more tractable.

I use MCTS rollouts as a proxy for player skill. Modeling the effects of player skill enables many opportunities for applying behavior sampling to design questions:

- Many games are designed to reward more skilled players with greater rewards or higher win rates [13]
- Designers are often concerned with differences in play style dependent on player skill [61]
- Games (including adversarial games) are often designed to enable a smooth progression of skill as players learn over time [110]

I use rollouts as a proxy for player skill, specifically the ability to consider choices and plan ahead, with more rollouts simulating a player that is better able to consider the outcomes of actions in the game. In adversarial games, varying the rollouts used by two MCTS agents can compare how gameplay looks when two agents having varying levels of skill, as well as compare the effects of relative differences in skill between two agents; e.g., comparing high-level play between two strong agents or comparing games between a weak and strong agent. This is an improvement over human testing as it affords designers the ability to explore many different skill combinations, including some that may be difficult to examine using human playtesting alone.

4.5.2 Experiment Design

Both studies sample playtraces using MCTS agent pairs of varying computational bounds as a proxy for varying player skill. I varied agent reasoning to consider roughly one to two moves ahead in the game. Two moves ahead is an upper bound potentially relevant to human play; research in reasoning on recursive structures suggests people are able to reason to roughly two levels of embedding. Models of deductive reasoning on logic puzzles support this claim [19]. The MCTS selection policy (UCB1) I used forces trying all child

moves of a given move once before repeating a move: thus all rollouts will first explore options for a single move before exploring two-move sequences.

To set computational bounds I approximated the average number of moves available to an agent and used this number to estimate the number of rollouts an agent would need to consider one or two moves ahead in the game. To examine a range of agent capabilities I initially created three agent computational bounds (number of rollouts allowed):

- A *weak* agent with enough rollouts to explore the all moves on a given turn, but lacking resources to explore to two moves ahead
- A *strong* agent with enough rollouts to fully explore moves on the current and the next turn
- A *moderate* agent with rollouts halfway between these two.

Initial testing revealed little difference between the latter two agents; my results report agents that halve the number of rollouts of the two stronger agents as these more clearly illustrate the outcomes of variable player skill. The lack of differences may derive from marginal returns for greater computational resources in the case study domains, likely due to their large branching factor.

For each game domain I ran a pair of agents where each agent was set at one of these three levels. For each agent pairing I simulated 100 games to get aggregate statistics on agent performance and visualized these results to examine relevant design metrics in both game domains. In *Scrabble*, I approximated the number of rollouts for a single level deep by looking at the median number of possible words an agent could complete on a board: 50. Thus, the weak agent used 50 moves. Initially the strong agent was allowed 2500 rollouts (50^2 for two moves ahead) and the moderate agent 1250 rollouts. After halving, this resulted in a moderate agent with 650 rollouts ($(1250 - 50)/2 + 50 = 650$) and a strong agent with 1250 rollouts. In *Cardonomicon*, I approximated the number of choices of playing cards as choosing 2 cards to play each move out of a hand of 6 cards ($\binom{6}{2} = 15$

moves). I modeled attack choices assuming the player (and opponent) have approximately 3 cards on the board and one hero card, yielding 3 source card choices for 4 targets ($3^4 = 81$ moves) Together this yields a total of approximately 100 moves considered for the weak agent, 10000 for the strong, and 5000 for the moderate. After halving this resulted in 100, 2500, and 5000 rollouts for the weak, moderate, and strong agents, respectively.

Note that an alternative strategy to sampling up to two levels deep would be to have agents explicitly model a selection policy with pure exploration up to one or two levels. In this case, search bounds would vary over the course of the game. I chose to use a fixed number of rollouts to capture the notion of agents of fixed 'capability' in terms of resources to devote to the problem.

4.5.3 Scrabble

Scrabble[4] is an adversarial game where players take turns placing tiles onto a game board to create words (Figure 4.2). Players have a rack of seven tiles each with a single letter. While the rack is normally hidden from the opposing player we simplified *Scrabble* so agents have perfect information about one another's states and perfect knowledge of all legal words. On each player's turn, they select tiles from their rack and place them on the game board such that: (1) at least one of the tiles is placed adjacent to one of the other player's tiles and (2) the tiles create dictionary words either left to right, top to bottom, or both. The player that goes first, however, only needs to play a word that goes through the center space on the board.

Moves in *Scrabble* are typically considered as tiles being placed on the board. This representation, however, makes it difficult for the MCTS agent to play the game as it requires knowledge about whether the tiles being placed form legal words. Instead, the MCTS agent represents moves on a given turn as the word that was formed on that turn, using a dictionary to choose valid words. Thus, the space of possible moves on a given turn is all

[4]The *Scrabble* domain and analysis was done by Brent Harrison in joint work on applying MCTS to behavior sampling citezook2015:mcts-strategy.

Figure 4.2: A digital recreation of the word game *Scrabble*.

possible words that can be made on that turn. The search tree builds as sequences of words played by the agent and its opponent, with leaves alternating between words formed by each agent.

Players earn points for forming words on a turn. Each letter tile has a score associated with it; a word's score is the sum of the score values of the letters used to make that word. The board is also populated with *bonus spaces* that increase the value of a word. Bonus tiles available on a typical *Scrabble* board can double or triple the value of either a specific letter tile or of the word that the letter tile is part of.

Once a player receives points for a move, that player draws tiles at random until their rack is refilled with seven tiles and the turn ends. Normally, the game ends when a player cannot draw new tiles and the winner is the player with the highest score at that point. In

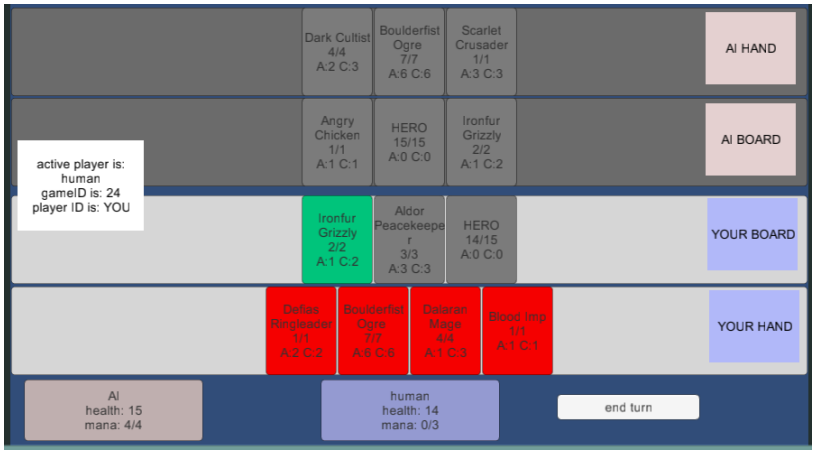

Figure 4.3: *Cardonomicon*, a minion-based card game.

our implementation, however, the the first player to meet or exceed 150 points wins. This simplification improves search performance in the domain (and does not adversely affect the ability to ask design questions, as seen in the study evaluations).

4.5.4 Cardonomicon

Cardonomicon has the core elements of a class of game mechanic-heavy adversarial card games, exemplified by games like *Magic: The Gathering* and *Hearthstone* (Figure 4.3). From a design perspective, games like *Magic* and *Hearthstone* are difficult to balance due to the difficulty of predicting the strategies players will develop to play the game. The design is highly sensitive to interactions among mechanics: each card must be balanced with respect to all other available cards; e.g., a single overly powerful card can make all other cards irrelevant. Further, the random order of card draws and non-deterministic effects of actions introduce a large space of non-deterministic outcomes to sample over. While *Magic* and *Heartstone* have hidden information, for simplicity *Cardonomicon* is perfect information. In addition, I fix the decks used by players, rather than allowing deck construction—this drastically reduces the search space for behavior sampling while preserving properties that make this a domain of interest.

In *Cardonomicon*, two players start with an identical deck of 20 cards representing

minion creatures (see Appendix B for the list of cards). Gameplay consists of drawing cards, spending mana to place cards on the game board, and using cards to attack one another and the opposing player's hero. Cards are parameterized by health, attack power, and mana cost. Players start with a single hero card on the board with 20 health and 0 attack; a player loses when their hero's health is reduced to or below 0. Each turn, players may play any combination of cards for which they can pay the mana costs. A player's mana starts from 1 on the player's first turn and increases by 1 each turn up to a cap of 10. Cards on the board may attack any other opposing card once per turn after the turn the card is played. When a card attacks, the opposing card's health is reduced by the attacker's attack; attacking cards receive counter damage. I designed a set of cards to allow the player to play one of multiple cards on each turn (with differing parameterizations), assuming they have drawn a playable card.

For any given game there are multiple ways to represent the actions available to an MCTS agent. Here, I take the approach of representing each choice the agent makes individually, rather than aggregating sequences of choices that occur together during a single agent turn. The MCTS agent represents possible moves as either playing a card or using a card to attack another card on the opponent's board. One turn may involve multiple moves in a row. The agent has one move for every card that can be played in the agent's hand and one move for every pair of their card attacking a target opponent card. Only cards that may attack are represented and no attacks on the agent's own cards are permitted as this has no purpose in the *Cardonomicon* domain. One additional move to end the turn is always available. Thus, MCTS agents reason at each turn about whether to play a card, use a card to attack the opponent, or end their turn.

4.6 Results

For the two game domain cases I examined the four skill-based design metrics above: summaries, atoms, chains, and action spaces. In the *Scrabble* domain these metrics highlight

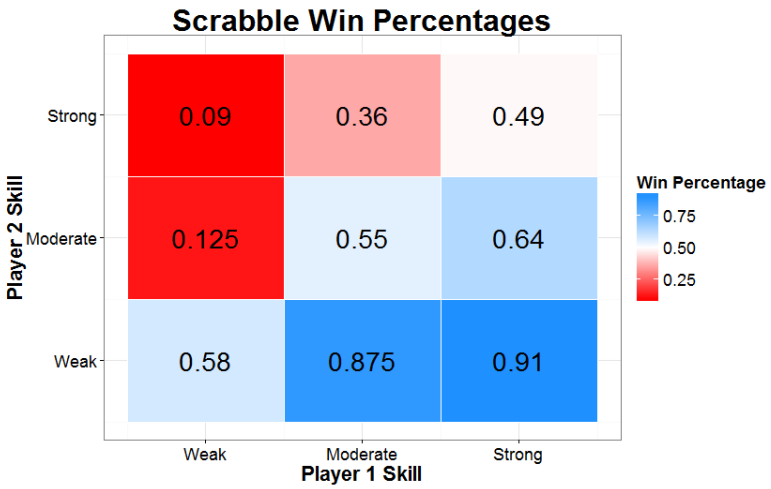

Figure 4.4: Win percentage based on agent skill. Win percentages are calculated from the perspective of Player 1. Blue regions correspond to win percentage greater than 50%. Red regions correspond to a win percentage less than 50%.

how the game is balanced and illustrate how player skill differences manifest as differences in skill-based metrics. In the *Cardonomicon* domain these metrics reveal imbalances in the design of the simplified game. Together, these studies illustrate that skill-based design metrics can help inform designers about the strategic space of play in a game.

4.6.1 Scrabble Metrics

The *Scrabble* domain shows how skill-based metrics reveal balance and player skill differences despite changing the game to end at 150 points. The study shows these changes did not upset the game balance and demonstrate that *Scrabble* rewards high skill play.

Summaries. The summary statistics that we choose to examine in *Scrabble* are win percentage (Figure 4.4) and the length of a game based on turns. Ideally, players with higher skill will consistently defeat lower-skilled opponents; however, it is unclear how skill will affect game length.

Comparing agents of varying rollouts shows the game is balanced with higher skilled opponents consistently defeating lower skilled opponents (Figure 4.4). This difference is

large when the strong agent plays against the weak agent and becomes smaller as the skill difference between agents decreases. First turn players had no difference in win rates, meaning there is no first turn advantage.

Games played against skilled opponents are typically slightly shorter. When weak agents play against each other games last 26 turns on average; this decreases to 22 turns when strong agents play against each other. This is likely because skilled opponents make moves worth more points, reaching the 150 point ending criteria sooner. As shown below, stronger agents typically play longer words, corroborating this conclusion.

An alternative explanation for the shorter games is that stronger agents make more use of the bonus tiles on the board. Effective use of bonus tiles increases individual word scores, speeding the game toward the 150 point end. However, different strength agents did not differ in their use of bonus tiles. Thus, the main source of score difference between agents seems to come from the length of words played.

Atoms. In *Scrabble*, the main atom metric is the rate of using words as moves. Figure 4.5 shows the word usage distribution separated by word length and grouped by agent skill. Weak agents tend to favor playing shorter words, while stronger agents play a wider variety of word lengths. However, skill has little effect on the specific words played. Figure 4.6 shows the most popular three-letter words and how often each agent used each one. There is no strong trend in the specific words an agent plays (while the figure shows three-letter words, these findings were consistent across word lengths).

Chains. In *Scrabble*, *counters* are the words played by the opponent after a word has been played by the other player. To determine what common counters in *Scrabble* were, I used frequent itemset mining on itemsets comprised of the words played on a given turn and the words played on the next turn. Among the top itemsets of words created across two turns, most counters either add to the previously played word, or build a two or three-letter word off of the word that was previously played. For example, one of the top counters to

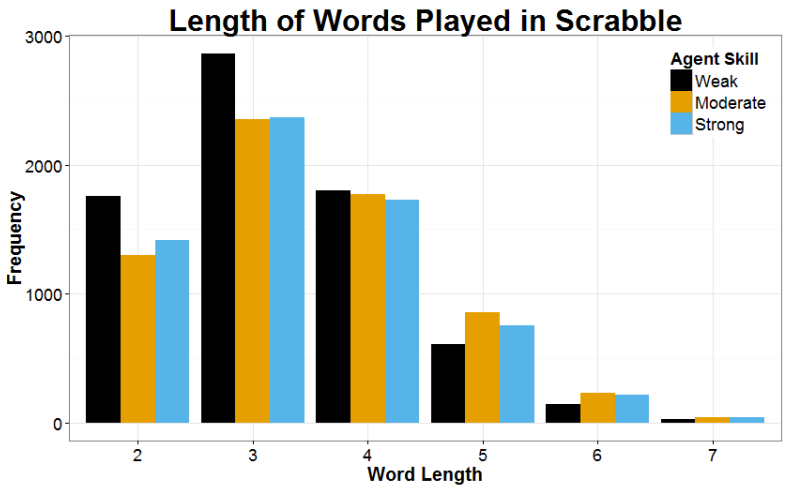

Figure 4.5: Word length frequency in *Scrabble* by skill.

a player playing the word "con" on a turn was to add an "i" to the beginning of it to make the word "icon." This is not unexpected as building off words that were previously played will typically result in a higher point total since the player is playing a longer word than the opponent.

Action Spaces.

The action space in *Scrabble* can be characterized by the number of possible words that can be played and were actually played. Figure 4.7 shows the median number of *possible* words that could have been played on a given turn based on skill. This conveys how the complexity of the action space changes over time. Figure 4.7 shows that the space of possible actions shrinks over the course of the game, likely because valid word placements become fewer later in the game. The figure also shows that stronger agents have more possible actions on a given turn than weaker agents.

Figure 4.8 shows how much of the action space was actually explored over the game. This figure shows that the space of words played shrinks faster for stronger agents than weaker agents, likely because stronger skilled agents successfully identify moves worth more points and avoid the rest of the action space.

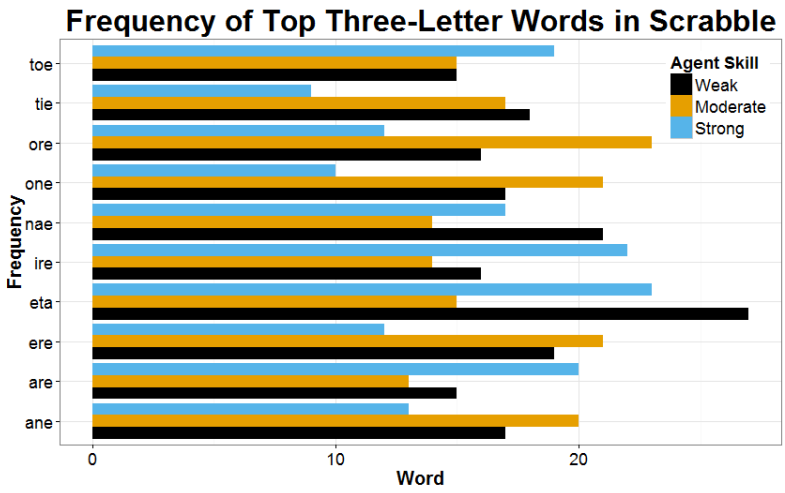

Figure 4.6: Frequency of the top three-letter words in *Scrabble* by agent skill.

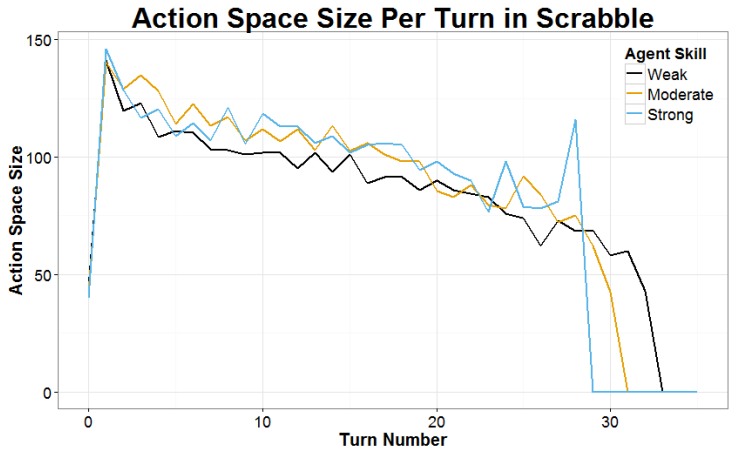

Figure 4.7: Median number of words that could be played per turn based on skill.

4.6.2 Cardonomicon Metrics

The *Cardonomicon* domain shows how skill-based metrics can identify design flaws. Recall that *Cardonomicon* is highly constrained in terms of the types of cards that are available to use and the types of decks that players can use. These major alterations to the typical structure of a card game negatively impacted the balance of the game.

Summaries. A key design flaw in *Cardonomicon* is the player going second has a large win rate disadvantage. Figure 4.9 shows the win rates for the player who starts second.

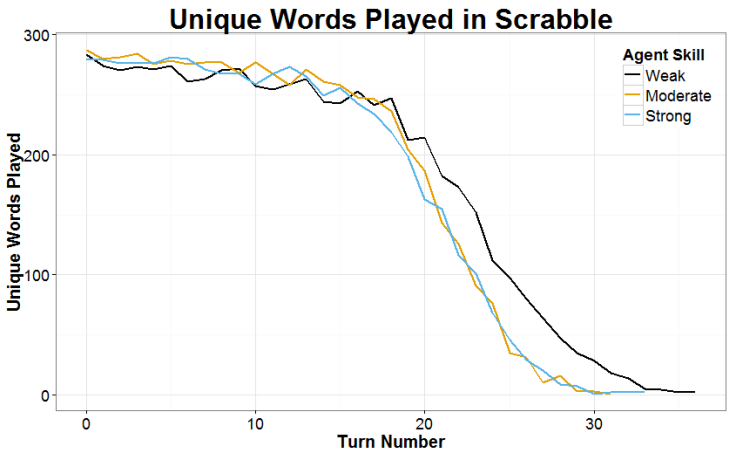

Figure 4.8: Number of unique words played per turn based on skill.

Regardless of agent strength, the player going second has a win rate substantially less than 50%. That said, win rates increase for the agent going second if they are more skilled than the agent going first. Thus, while agent skill influences player win rates in *Cardonomicon*, the game is flawed in giving a strong disadvantage to the player taking the second turn. This is expected due to my partial adoption of mechanics from *Hearthstone*: in *Cardonomicon* cards are able to attack and receive damage in retaliation, but the second player has no advantage in being able to play more cards on their first turn. As such, the second player will always deploy cards after the first player, but lacks a mechanism to catch up to the player who acts first.

Stronger agents have (slightly) longer games when matched to evenly skilled opponents: median 16, 17, and 18 turns for the weak, moderate, and strong agents, respectively. I attribute this trend to stronger agents being able to better counter one another while retaining enough cards to play until the end of the game.

Atoms. *Cardonomicon* atoms consist of actions to play cards or use cards to attack. Based on the frequency of playing different cards, stronger agents generally play more cards, but show no large differences in their use of specific cards. Stronger agents manage their mana to play more cards, but do not seem to favor specific cards to play. This likely indicates the

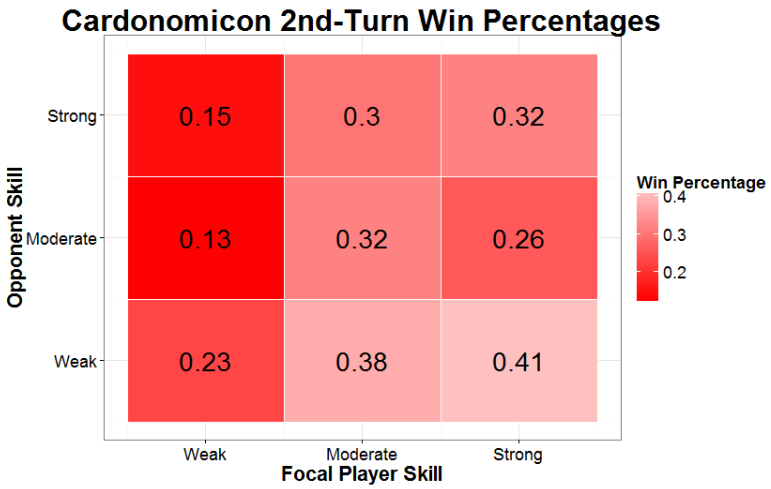

Figure 4.9: Win rates for second turn player in *Cardonomicon*. The x-axis indicates agent strength for the second turn player; the y axis indicates the opposing agent's strength.

deck size in *Cardonomicon* is too small: agents will play all of their available cards faster than they draw new cards and thus have no opportunities to favor playing specific cards against others.[5]

When examining the frequency of using cards to attack, stronger agents also tend to use cards to attack more overall. Three cards showed disproportionately greater use by stronger agents compared to weaker agents: these three cards all had large amounts of health but low attack for their cost. Strong agents use these cards to destroy multiple weaker cards by intelligently trading off card attacks and retaliations. That is, stronger agents recognized the value in using a card with low attack (but high health) to remove several cards with lower attack and health over multiple turns. This confirms *Cardonomicon* allows for a limited form of strategic variety and supports the notion that MCTS rollouts can help detect these potential strategic variants dependent on player skill.

Chains. Chains in *Cardonomicon* are primarily combos: sequences of actions taken by a single player in a turn of the game. As expected from the atom analysis, there were no significant combos in terms of playing or attacking cards. This is likely due to the lack of

[5]To reduce redundancy I have suppressed images that illustrate simple trends of the same form as shown with *Scrabble*.

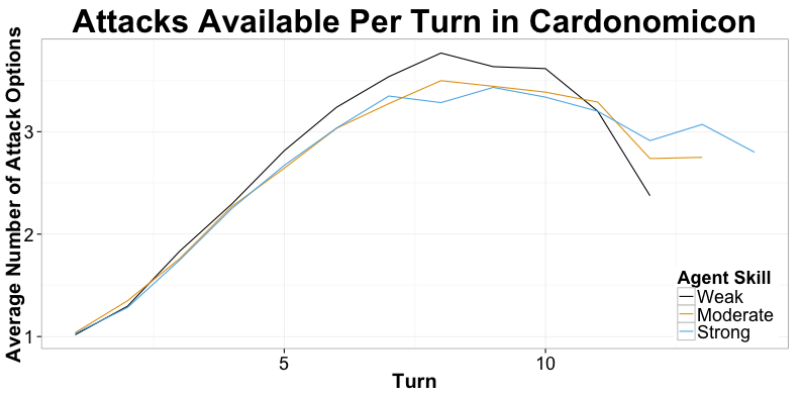

Figure 4.10: Average number of possible attacks per turn based on skill.

any strong synergy among cards in *Cardonomicon*: no pairs were particularly outstanding as no pairs had effects that would be advantageous to use together. This highlights another way to detect design flaws through these metrics: the absence of chains indicates no strong synergies exist in the design for players to use in combos.

Action Spaces. As with *Scrabble*, stronger *Cardonomicon* agents have a larger space of cards they may play (not shown) and use to attack (Figure 4.10). Specifically, stronger agents have more options to play cards late in the game, while having fewer mid-game attack options with more late-game attack options. These results align with intuition: in the early game both weak and strong players have a similar range of options constrained primarily by the amount of mana players have. By mid-game stronger players will have fewer attack options as they retain cards they may play for the late game. Playing these cards in the late game leads to more options to attack. Aligning with these analyses of the number of *possible* plays, more skilled players both play and attack with a larger number of cards on average. Thus, skilled players also actually use this larger set of options. Overall, these results demonstrate that more skilled players in *Cardonomicon* will open more plays in the mid-game by intelligently retaining cards before using these cards in the late game; in sum, these players are more efficient in their use of mana.

4.7 Limitations and Future Work

Skill-based design metrics provide tools for analyzing the strategic play space of a game to enable designers to understand how designs influence the *actions* available in a game, rather than the *states* players reach. The solution to behavior sampling and analysis in this chapter is MCTS as a model of players of differing skill. Yet MCTS (and other agent play algorithms) have parameters that are not necessarily related to human skill. This begs the question: what makes for human-like agent play? Or rather, how can agent parameters be tuned to achieve human-like play relative to a set of skill-based design metrics?

4.7.1 Design Space Generalization

Researchers are beginning to address human-like play by gathering human play data and optimizing agent control parameters to match human behavior on desired metrics across a wide variety of game genres [54, 88, 89, 91, 112, 207, 223, 242]. When an agent is trained to match human play the training data is derived from one or more instances of a game's design. The agent then is used to play other design variants to provide human-like play metrics for those games. In these cases we assume the design variants are close enough in terms of how people play them that information on how people play one variant can be smoothly mapped over to another variant. For example, in the studies above the costs of cards smoothly alter the rates of playing those cards; small changes in the cost of a card result in relatively small changes in the rate of play. The problem arises that in some cases this smoothness will be violated: a change in game parameters can result in sudden, sharp changes in the observed skill metrics. For example, introducing the 'taunt' mechanic from *Hearthstone*, where a card with taunt must be destroyed before other cards can be attacked, would create a discontinuity in play metrics. When this kind of change occurs the mapping of agent play to human play become uncertain at best. This presents a number of open questions: what kinds of smoothness assumptions are made of the space of play (in

terms of different metrics measuring that space)? How can these assumptions be modeled and tested? How can violations of the models be detected? To date we still have little understanding of where and when simulated play will accurately reflect human behavior, and little understanding of how strongly agent behavior must correlate with human behavior to be useful for different design objectives. Modeling 'skill' in play is a useful subset of design problems that emphasizes the competitive design of a game without necessitating models of more subjective aspects of human experience engendered by games. Until agents can be automatically created to generate human-like play across a variety of games, each new game will require a separate process of data collection and agent tuning. Identifying ways for agents to rapidly learn human-like behavior (or make a 'best guess' at human-like behavior) within a broad genre will be crucial to enabling broader adoption of these techniques in automated game evaluation.

4.7.2 Single Player Games

Behavior sampling in this chapter and the next chapter is only used in two-player, adversarial games. The behavior sampling framework, however, is more general and allows for examining designs even in single player games. For single player games behavior sampling offers the ability to rapidly get examples of ways of playing in a game space and can help quantify the diversity of options available in that space.

Consider a role playing game (RPG). MCTS agents can provide information on the strategic depth of the combat system in the game, ease of navigation in game dungeons, or breadth of customization from a statistics system. In a turn-based combat system MCTS agents can proxy how well players can learn to use combat skills to defeat enemies.[6] Limiting the number of rollouts of the agents would provide information on how much planning players might need to address the strategies used by scripted opponents. Evaluating action

[6]The MCTS agents as created in this chapter are designed for turn-based combat with discrete actions. Continuous action values or continuous time can be addressed by discretizing the action or time space into sufficiently small units for planning.

atoms can reveal the balance of abilities in terms of frequency of use, potentially revealing subsets of actions that are always or never needed. Combos in action chains could reveal synergies in abilities used, or the absence of any synergy in terms of realized player strategies. Action spaces can be used to measure the breadth of actions available to players over combat encounters as well as cases where players recognize simple dominant strategies—these cases would emerge as having many action options but few distinct actions realized at certain points in combat. Designers can then use this set of metrics to visualize and adjust combat encounters to provide the desired level of strategic complexity, being guided by expected amounts of player action variability.

Similarly, applying the MCTS model to discretized dungeon maps would reveal ways players might navigate the dungeon or become stuck and need backtracking. Note that the model presented in this chapter assumes perfect information for an agent, so this would only apply when agents knew a dungeon map perfectly in advance of navigating it. MCTS algorithms for problems with imperfect information are needed to model players navigating a dungeon not known in advance [44, 45]. Action metrics on dungeons could quantify backtracking, running into dead-ends, and other metrics of movement through space to proxy actions that may be tedious to players. For dungeon designs combos would reveal whether players are making similar navigational choices repeatedly, and action spaces could be visualized to understand how linear a dungeon is navigated when players have varying abilities to plan ahead for movement in the dungeon.

Action metrics can also provide useful insight when comparing agents that are given differing levels of in-game power, such as avatar levels in the game or the stats given to an avatar. Action metrics from agents with varying levels of power can reveal how much different in-game statistics impact player strategic decisions. In most RPGs greater in-power power leads to less variation in actions chosen as the player avatar overpowers opposition. This type of progression can be readily quantified by comparing the diversity of actions chosen by agents with varying power in a given encounter. Diversity in these cases can

be quantified using metrics such as entropy of the Gini index, which quantify dispersion within a population, here treating the actions taken as a population to measure.

This RPG example illustrates the ways in which behavior sampling and action metrics can support design decisions around ability systems, dungeon designs, and statistics systems in a single player game. Many other single player game systems are amenable to a similar approach, with the key (current) limitation that the system have no hidden information.

4.7.3 Types of Skill

Constructing agents that mimic human play in terms of skill-based design metrics assumes that the notion of 'skill' is known for a given game. Skill is often reduced to the ultimate metric of winning or losing a game [62, 74, 75, 86]. Yet this coarse definition obfuscates what makes for skillful play of a game, overlooking how play behavior evolves over the course of a game. How should skill be defined in a given game? For example, in *Cardonomicon* I examined skill in terms of the actions of playing and attacking with cards, along with the option to play or attack with cards. Yet the choice of these actions was informed by prior knowledge on how this genre of card game works, and does not necessarily translate to other game domains. To date we lack any clear taxonomy delineating the ways skill manifests in games and how these bear on metrics related to how people play those games. Lacking a notion of the space of metrics to apply, using skill-based design metrics will remain an ad-hoc process of constructing definitions for each application game. Beyond limiting human uses of skill-based design metrics, this will also inhibit automated approaches to assessing design variants, as automated assessment will be contingent on humans providing the relevant metrics to be evaluating.

To address this limitation we will need to develop ways for agents to construct and evaluate potential skill-based design metrics. One avenue will be to develop general metrics for skill that can be applied across games, similar to the kinds of metrics being developed to

assess game-playing agent strength across games [143] or metrics for level design quality across genres [114, 202]. Alternatively it may be more effective to develop ways to assess game differences that relate to known, high-level metrics of skill. For example, metrics like Elo [62] provide an abstract notion of skill that is (relatively) agnostic to game structure. An agent could then compare behaviors of agents with high and low Elo ratings to discern which features of those behavior traces are predictive of high or low Elo. In this case, agents playing the game need only optimize for winning and losing to provide Elo scores, with skill-based design metrics providing a set of levels of granularity of actions to evaluate. The assumption here is that agent Elo scores against one another will be indicative of the same scores when playing against humans—an assumption about the similarity of agent and human populations. Automating the evaluation of skill in games is both a general challenge for any competitive game and one with direct relevance to supporting automated game design, presenting promising future opportunities.

4.8 Potential Impact

Game design research has the potential to change the way games are made and the experiences available to game players. In this section I briefly discuss how the action-based metrics and behavior sampling technique in this chapter might influence game designers and players.

4.8.1 Game Designers

Behavior sampling provides designers with a sample of ways agents might play any game. Using algorithms like MCTS adds to this value by providing an array of different potential ways of playing based on player abilities to plan. MCTS only requires a forward model (a model of how actions transition game states forward to function) to be applied to a new game. Thus, MCTS can be readily linked to the core logic of a game engine to simulate many ways people play and automatically provide designers with this feedback. In the

future game engines may incorporate general behavior sampling techniques like MCTS into the core systems and interface to provide designers with ready access to examples of ways people play a game. In this scenario a designer could arrange a design, press a button, and automatically get feedback on how people could play the game (split by levels of ability to plan). Having these overlays directly supported by a game engine allows designers to rapidly tweak a design in response to these variations to tune the game.

The action-based metrics in this chapter highlight another avenue for improving game design practices. The hypothetical game engine integration above could readily provide evaluations of some of the pre-defined metrics above along with designer-provided additional action metrics. Using these a designer could easily explore design alternatives and gain a sense of how simple design changes alter the design. Design practices would in turn grow to use action metrics as a way to quantify some aspects of the strategic depth in a game [113], spurring development of useful design metrics that quantify the way people play games.

4.8.2 Players

Players stand to benefit most when game designs more generally incorporate agents with smoothly varying capabilities to plan. In most competitive games agents have only one or a handful of difficulty settings. With algorithms like MCTS the notion of a smoothly varying notion of opponent strength is possible by tuning the number of rollouts free to an agent. For players this allows agents with smoothly varying challenge levels as a setting.

Going further this could allow for automated creation of agents that match the strength of certain human players. Similarly to how racing games allow 'ghosts' that replay a game course as another person did, MCTS agents could be trained to play adversarial games in a way similar to another person. In this scenario, a player would face a series of MCTS agents each with differing predefined numbers of rollouts. These training rounds would be used to tune the agent rollout parameter to produce a desired win rate against the player

(e.g., 50% to be perceived as an evenly matched opponent). This pre-trained agent could then be given to other players as a mimic of the training player, providing gameplay against a set of exemplars of other players (or friends) of a given player. These mimic players could transform leaderboards from tracking high scores to providing portable players to face.

Rollouts, however, are only a single parameter that coarsely represents the playstyle of a person. Developing further parameters of the MCTS agent would allow this model to replicate human-like behavior in a number of other ways. The key benefit afforded by the MCTS model in this case is a generic method that can readily be applied to a wide class of games, allowing this to serve as a general functionality across games to improve player experiences.

4.9 Summary

In this chapter I presented MCTS as a technique for general behavior sampling to proxy variations in human skill and four metrics for evaluating player strategies of varying levels of abstraction. MCTS is a general technique for simulating agent play in games that can be tuned to vary computational resources and was used as a proxy for varying capabilities at playing a game. The levels of strategy evaluation metrics assess how well a game supports different levels of strategic depth and variety—quantifying aspects of the choices a game supports, rather than the content players see. Applying these techniques for behavior sampling and analysis to *Scrabble* demonstrated the metrics can detect how a game allows for variable player skill and performance; applying these techniques to *Cardonomicon* illustrated the metrics are able to detect design failures as well. Together, these techniques provide a general set of tools for evaluating a game design in terms of the levels of strategy and differential player performance the design supports.

The strategy metrics here provide a way to consider the space of play in a single design. But iteration requires comparisons of the space of play between different designs. How can a system compare spaces of play to pick the best option for a design goal? What

can a system learn from evaluating a sample of designs from a design space? In the next chapter I present work on automated gameplay analysis to choose the optimal design from a design space to achieve design goals. I also show how an automated system can use data from designs in a design space to generalize to hypotheses about how a design works that supports optimizing the design toward various design goals.

CHAPTER 5
GAMEPLAY ANALYSIS

5.1 Introduction

Game designs are often intended to induce particular behaviors in players, through shaping the space of play available. Designers choose design features based on their theories of how particular design decisions will influence expected player behavior. Design choices are often informed by high-level, abstract theories about how players are expected to respond to a game design, drawing from examples from prior game designs [17, 168], human-computer interaction [97], or psychology [47, 104]. Design knowledge is used to predict how specific changes to a design will change player behavior. For example, design knowledge may be that increasing the power of an attack in a game will decrease the typical length of the game. Design knowledge provides a framework to guide choices of how to iterate on a design when attempting to achieve a design goal. By accruing knowledge of how changes to a design may alter player behavior a designer can target future design iterations on designs that move toward a design goal. In addition, an explicit model of design knowledge can reveal aspects of a design that are poorly understood, revealing design alternatives to test to learn move of a design.

Automated game generation and optimization models to date have largely overlooked design knowledge for guiding generative processes. Systems will typically produce content according to hard-coded processes [178], or optimize for a design goal without accruing any knowledge regarding how design choices made along the way influence player behavior [220, 222]. While this process can optimize a design for a particular player or outcome, the system loses all information from iterations on the design about how design choices alter a space of play in a game. This in turn limits the capability of the automated system to

understand how a design works when making future changes and prevents the system from providing useful information to a human or system interacting with this system.

How could a system learn design knowledge to support automated game generation and inform human designers? The previous chapter demonstrated how MCTS can be coupled to skill-based design metrics to evaluate a *single* design. These methods evaluated how agent strength was (or was not) different in a game in terms of the strategies pursued. In this chapter I extend these methods to a space of designs that are evaluated using the same skill-based design metrics, focusing on the *Cardonomicon* card game domain. As a base case of design optimization, the system generates a space of design variants and uses this space of alternatives to automatically optimize a design iteration for a desired skill-based design metric, such as game length or frequency of using cards to play. The system is also able to learn a set of statistical models of the form:

> 'X card parameter influences the rate of performing Y action' from this space of designs.

These models show how changes to card cost parameters can increase game length and reduce the space of cards available to play, while altering card health or attack can drive greater use of cards to attack, providing knowledge of how design features impact player behavior. Together these analyses illustrate how an automated iterative design system can use a design iteration both to optimize a design and learn to predict how design changes influence player behavior.

5.2 Game Design Knowledge

This chapter is concerned with two goals for gameplay analysis: (1) choosing an optimal game design iteration in a design space and (2) learning to predict how the features of a design influence the space of play in a game. Optimizing iterations uses the play space metrics from the previous chapters and uses these to select designs from a space of

designs—applying the skill-based design metrics from the previous chapter to a space of many design variants, rather than the single variants assessed in that chapter. Learning how design features influence play features concerns the problem of automatically learning game design knowledge.

Game design knowledge predicts how features of a game design result in player behaviors. Most evaluations of game design knowledge to date have been comparisons of a few variants of a given design by humans to test a psychological theory. Examples include assessing the influence of reward systems on player behavior [187, 189] or the influence of game parameter tuning on player learning and engagement [120, 118, 117]. Design knowledge can guide automated choices among design alternatives and is also valuable to human designers when making design decisions for games.

Automated systems have been used to support game design optimization by searching a space of possible designs to optimize for a given set of design outcomes [181, 197]. These approaches, however, have sought simply to optimize the outcomes of a design without modeling *how* a design produces those outcomes. As systems are used and reused for iteratively refining a design it is no longer sufficient to merely produce the single best outcome. Instead, design iteration requires the ability to learn about a space of designs to support subsequent choices of designs to compare. Without the ability to acquire knowledge about a design space and use that knowledge to guide future design choices a system is limited to blindly searching for desired outcomes. When exploring large spaces of possible designs it will be crucial to learn from the search process itself to inform future design iterations [194, 239].

In this chapter I explore automated design space modeling by generating a wide space of possible design variants and acquire knowledge about how design parameters influence the space of play. I use behavior sampling using MCTS coupled with behavior analysis using summaries, atoms, and action spaces from the previous chapter to learn design knowledge. This allows the same form of design optimization as done in prior work, but grounded on a

general framework for generating behaviors in a game and evaluating them with respect to *actions* available. The learned design knowledge extends this work to illustrate the potential for automated design space learning for automated iterative game design.

5.3 Game Domain

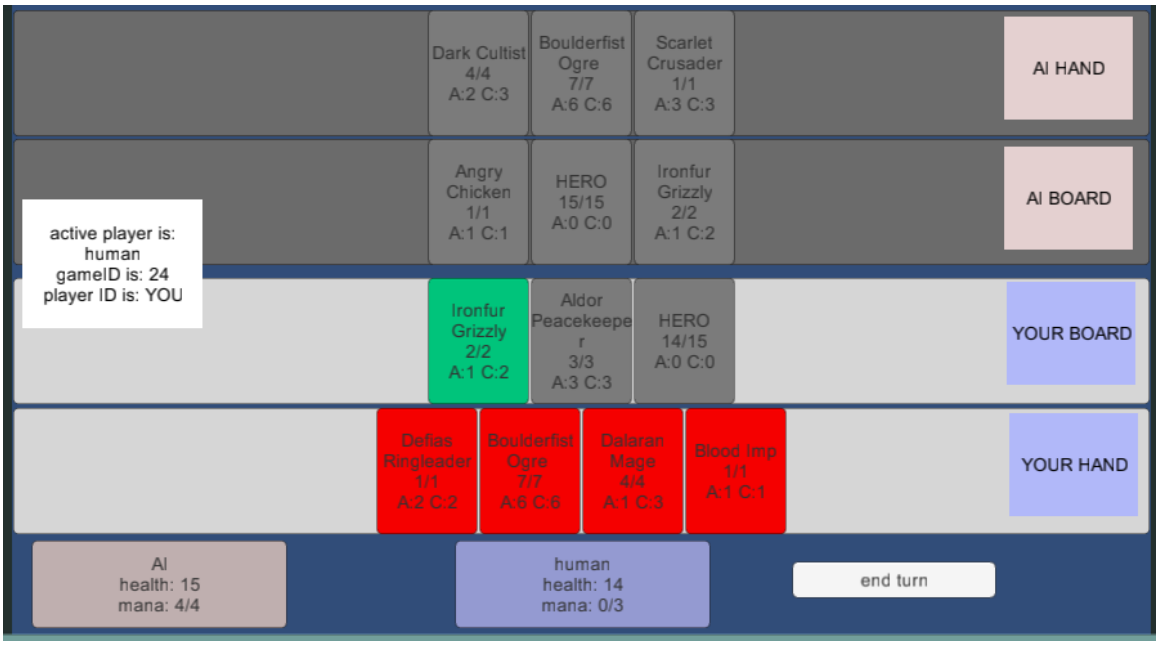

Figure 5.1: *Cardonomicon*, a minion-based card game.

Evaluating game design knowledge requires a base game design domain—here I use the *Cardonomicon* card game domain introduced in the previous chapter[1]. *Cardonomicon* has the core elements of a class of game mechanic-heavy adversarial card games, exemplified by games like *Magic: The Gathering* and *Hearthstone* (Figure 5.1). From a design perspective, games like *Magic* and *Hearthstone* are difficult to balance due to the difficulty of predicting the strategies players will develop to play the game. The design is highly sensitive to interactions among mechanics: each card must be balanced with respect to all other available cards; e.g., a single overly powerful card can make all other cards irrelevant. Further, the random order of card draws and non-deterministic effects of actions introduce a large space of non-deterministic outcomes to sample over. While *Magic* and *Heartstone*

[1]I repeat the description below for separation of chapter content.

have hidden information, for simplicity *Cardonomicon* is perfect information. In addition, I fix the decks used by players, rather than allowing deck construction—this drastically reduces the search space for behavior sampling while preserving properties that make this a domain of interest.

In *Cardonomicon*, two players start with an identical deck of 20 cards representing minion creatures. Gameplay consists of drawing cards, spending mana to place cards on the game board, and using cards to attack one another and the opposing player's hero. Cards are parameterized by health, attack power, and mana cost. Players start with a single hero card on the board with 20 health and 0 attack; a player loses when their hero's health is reduced to or below 0. Each turn, players may play any combination of cards for which they can pay the mana costs. A player's mana starts from 1 on the player's first turn and increases by 1 each turn up to a cap of 10. Cards on the board may attack any other opposing card once per turn after the turn the card is played. When a card attacks, the opposing card's health is reduced by the attacker's attack; attacking cards receive counter damage. I designed a set of cards to allow the player to play one of multiple cards on each turn (with differing parameterizations), assuming they have drawn a playable card.

For any given game there are multiple ways to represent the actions available to an MCTS agent. Here, I take the approach of representing each choice the agent makes individually, rather than aggregating sequences of choices that occur together during a single agent turn. The MCTS agent represents possible moves as either playing a card or using a card to attack another card on the opponent's board. One turn may involve multiple moves in a row. The agent has one move for every card that can be played in the agent's hand and one move for every pair of their card attacking a target opponent card. Only cards that may attack are represented and no attacks on the agent's own cards are permitted as this has no purpose in the *Cardonomicon* domain. One additional move to end the turn is always available. Thus, MCTS agents reason at each turn about whether to play a card, use a card to attack the opponent, or end their turn.

5.4 Design Space Evaluation

Applying the same design metrics to a set of related designs can answer questions about how different features of a design influence player behavior. Automating this process with simulations enables a system to assess a range of design variants to (1) optimize for a target design goal across designs and (2) learn to predict how design changes will alter player behavior. In this section I illustrate this approach with *Cardonomicon*, considering a design space of changes to individual card parameters and evaluating the influence of card parameters on the skill-based design metrics presented in the previous chapter. I show how card parameters influence game length, agent actions (using attack with cards or playing cards to the board), and the number of actions available to agents over the course of the game. These examples illustrate how an iterative design system can automatically choose design iterations and use behavior sampling to learn models to predict how a design will influence player behavior.

5.4.1 Experiment Design

For the experiment I generated 27 variants of a single card—"Stonetusk Boar"—in *Cardonomicon*. Varying a single card allows a focused study of how a minimal design change can influence the space of play in a game. Each variant altered the attack, health, or mana cost of that single card in the game. For each variant I simulated play between agents of differing strength, gathering data on which actions agents chose during the game. The metrics derived from these playtraces form the basis of the analysis below.

Card variants were: different attack, health, and cost parameter settings for a single card in the game ("Stonetusk Boar"). Each design variant altered the card's parameters to the value 1, 4, or 7. The 1, 4, and 7 values span the range of low, middle, and high values for each of the given parameters in this game. This yielded a grid of 3 attack values $\times$ 3 health values $\times$ 3 cost values $=$ 27 card variants. Each of these *Cardonomicon* variants were used

for simulations with varying agent pair strengths.

Simulations paired agents of differing strength with balanced first turn assignment. Agent combinations covered all cases where agents had different strength: weak vs moderate, weak vs strong, and moderate vs strong. To compensate for non-deterministic game mechanics 100 simulations were run for each card variant, agent configuration, and first turn agent combination. Together this required: 27 card configurations $\times$ 3 agent configurations $\times$ 2 first turn players $\times$ 100 simulations $=$ 16200 playouts. Each playout tracked the same actions and action options as used in the *Cardonomicon* case study in the previous chapter: each choice the agent made to either play or attack with a card, allowing multiple choices to be made in a given turn (subject to the mana cost constraint). Each playout also tracked a set of metrics on game state for each turn: hero health, number of cards in the agents' hand, number of minions on the player's board, and total health of minions on the player's board. These playouts were used for two models: (1) finding an optimal game design within the design space and (2) modeling how design parameters influenced player behavior.

5.5 Design Optimization Results

Given a space of game designs, finding an optimal iteration requires identifying the design configuration that yields optimal values for desired play space metrics. When the full design space can be generated, this amounts to searching the space for the optimal configuration for a given play space metric. For these results the playspace had already been explicitly generated and search amounted to database queries for design variants meeting desired features.

One example design goal for the system is controlling for game length: minimizing game length can give quick games while maximizing game length allows for more opportunity for play in a session. To find design variants that met these different goals for the summary metric of game length the system aggregated all playtraces using the same

card parameter configurations for attack, health, and cost, evaluating the average length of games for those parameter configurations. Using these aggregates the maximal game length configuration was for an attack of 1, health of 7, and cost of 4, yielding an average game length of 17.18. The minimal game length was achieved with an attack of 7, health of 7, and cost of 1, yielding an average game length of 15.03. These results both align with expectations: low attack and high health should prolong a game, while higher attack should allow for faster games (particularly when the card has high health and thus remains a threat for longer).

An alternative metric to optimize for is the frequency of using a card or having a card available. For these analyses the system aggregated across the same playtrace features, averaging the frequency of using the "Stonetusk Boar" card. Maximal use of the card to attack occurred with an attack of 1, health of 7, and cost of 1, yielding an average rate of 1.44 attacks per game. Maximal plays of the card to the board occurred with an attack of 1, health of 4, and cost of 1, yielding an average rate of 0.57 plays per game. Maximal frequency of having the card as an option to attack occurred with an attack of 1, health of 7, and cost of 1, yielding an average of 18.56 attack opportunities per game. Maximal frequency of having the card as an option to play occurred with an attack of 7, health of 4, and cost of 4, yielding an average of 4.01 play opportunities per game.

Together these results demonstrate how the general MCTS behavior sampling model coupled with the action metrics of the last chapter enable optimization of a design iteration toward design goals. The next section discusses methods to learn design knowledge about how changes to design features change player behavior using the same data.

5.6 Design Knowledge Results

To assess the impact of design variants on play I had the system learn models of how card parameters influence skill-based design metrics. The system learned that increases in card attack power reduced game length. The system also learned card variants impacted how

often agents played cards or used cards to attack. This also held true for the options of cards available to attack or play. Examining the space of cards played (or that the agent had the option of playing), the system learned changes to card cost (but not health or attack) impacted how agents played cards. Together the results illustrate that automated learning about design knowledge is feasible and can provide information about how changes to design parameters alter a game's playspace.

To test the impact of design changes on card use the system learned using a statistical model of count data to estimate how features of a data set alter the proportions of outcomes that are count data: Poisson regression or negative binomial regression.[2] Here the count data was the frequency of using a card to attack or playing a card; the features were the card parameter settings and/or agent strengths. Note that negative binomial regression is used when a data set is overdispersed relative to the Poisson model, meaning there is more variability in the data than is assumed by Poisson regression. The system first checked for overdispersion, choosing a Poisson model when overdispersion was not detected and the negative binomial model otherwise.[3] Poisson and negative binomial regression models both provide an estimate of the statistical significance of a feature impacting count volume (frequency of use) and an estimate of the magnitude of this effect. Below I report the learned model coefficients *after* exponentiating them—these values are 'incidence ratios,' interpreted as the proportional change in rate of the counts when comparing a group to the reference group. For example, a coefficient value of 0.5 for $attack = 4$ indicates that the attack feature value reduces the frequency of counts by 50% relative to the default value of $attack = 1$. For agents, p1 had a default strength of 'weak' (100 rollouts) compared to a 'moderate' (2500 rollouts) agent, while p2 had a default strength of 'moderate' compared to a 'strong' (5000 rollouts) agent. The next sections demonstrate this learning approach applied to action summaries and action atoms, showing the system learning how changes

[2]I used R's glm function for both:
`https://stat.ethz.ch/R-manual/R-devel/library/stats/html/00Index.html`

[3]Checks used the dispersion test provided by R's AER package:
`https://cran.r-project.org/web/packages/AER/index.html`

Game length vs card parameters	
feature	coefficient
attack = 4	**0.97**
attack = 7	**0.94**
health = 4	1.00
health = 7	1.00
cost = 4	**1.04**
cost = 7	**1.04**

Table 5.1: Effect of card parameters on game length. Bold values indicate significance ($p < 0.001$)

to card attack, cost, and health parameters change game length and card use frequency, respectively. The appendix includes an additional set of alternative models built by the system; for brevity I present a single key model for each metric considered.

5.6.1 Summary Metric

Game length is important for card games like *Cardonomicon*: if games are too short players may feel they have no choices, while overly long games can feel tedious. The system learned that of the three card parameters (treated as factor settings), attack and cost impacted game length relative to a baseline value of 1 ($p < 0.05$, the value used for 'significant' in these models), while health settings did not (Table 5.1). Poisson regression was used as game length was not overdispersed relative to card parameters. Greater attack parameter values reduced game length while greater cost parameter values increased game length. Increasing the card's attack will allow agents to more quickly defeat one another. Increasing the card's cost will slow down how quickly the card can be played and put to use, in turn lengthening the game.

These effects are readily discernible when examining the average game length in each of the card parameter configurations. Figure 5.2 shows the average game length for different parameter configurations. Columns divide configurations hierarchically: first splitting by attack, then by health. Rows divide configurations by card costs. Red lines indicate values averaging over health values (the average length for a cost and health configuration).

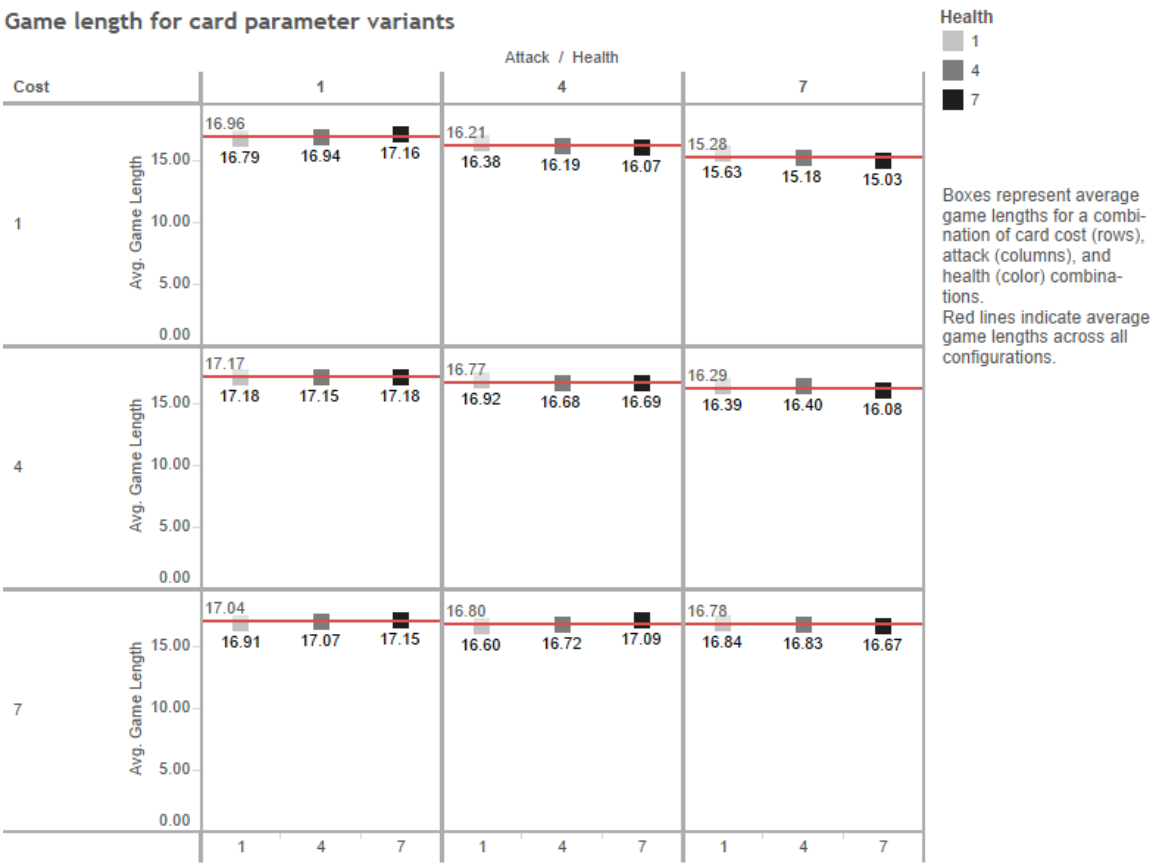

Figure 5.2: Average game length based on card configuration.

The decreasing average lengths for attacks (red lines) indicates that as attack increases, game length decreases. The higher average lengths for costs (rows) indicates that as cost increases, game length increases.

5.6.2 Atom Metrics

Varying game lengths can be primarily due to two factors: how often cards are used to attack and how often cards are played (making them available to attack). In games like *Cardonomicon* the features of a card govern (human) choices to use different cards. Typically, cards are discussed in cost-benefit terms: costs (e.g., mana cost) are based on the resources needed to use a card while benefits (e.g., card attack power or health) are based on the potential efficacy of the card when played. Predicting how changes to card parameters alter the frequency of using a card to attack or playing a card provides useful

Attack frequency vs card parameters	
feature	coefficient
attack = 4	**0.81**
attack = 7	**0.74**
health = 4	**1.55**
health = 7	**2.18**
cost = 4	**0.61**
cost = 7	**0.27**

Table 5.2: Effect of card parameters on card attack rates—coefficients give proportional change in card attack frequency; values above 1.0 indicate increased frequency. Bold values indicate significance ($p < 0.05$).

information on how to alter a design to increase or decrease how often different choices of cards are made in the game. Below I show models learned by the system to predict how changes in card parameters alter the frequency of using the card to attack; the next section predicts frequency of playing the card to the board.

Card Attack Rates

The system learned a model to predict how changes to the "Stonetusk Boar" card parameters change the frequency of using the "Stonetusk Boar" card to attack. All data were from the same set of playtraces generated by behavior sampling described above—for all the models learned the system is employing the same set of data on the design space of *Cardonomicon* variants of the "Stonetusk Boar" card. Counts for attack frequency were overdispersed ($p < 0.001$), so the system used negative binomial regression as the predictive model. All card parameter settings significantly altered attack rates. Greater cost parameter values reduced attack frequency and greater health parameter values increased attack frequency. This can be seen by incidence ratios below 1.0 for higher attack settings and above 1.0 for higher cost settings (Table 5.2). The system thus learned that more expensive cards will be played less often and used less to attack as a result; higher health allows cards to be used to attack more.

Figure 5.3 shows the average number of times the "Stonetusk Boar" card was used to

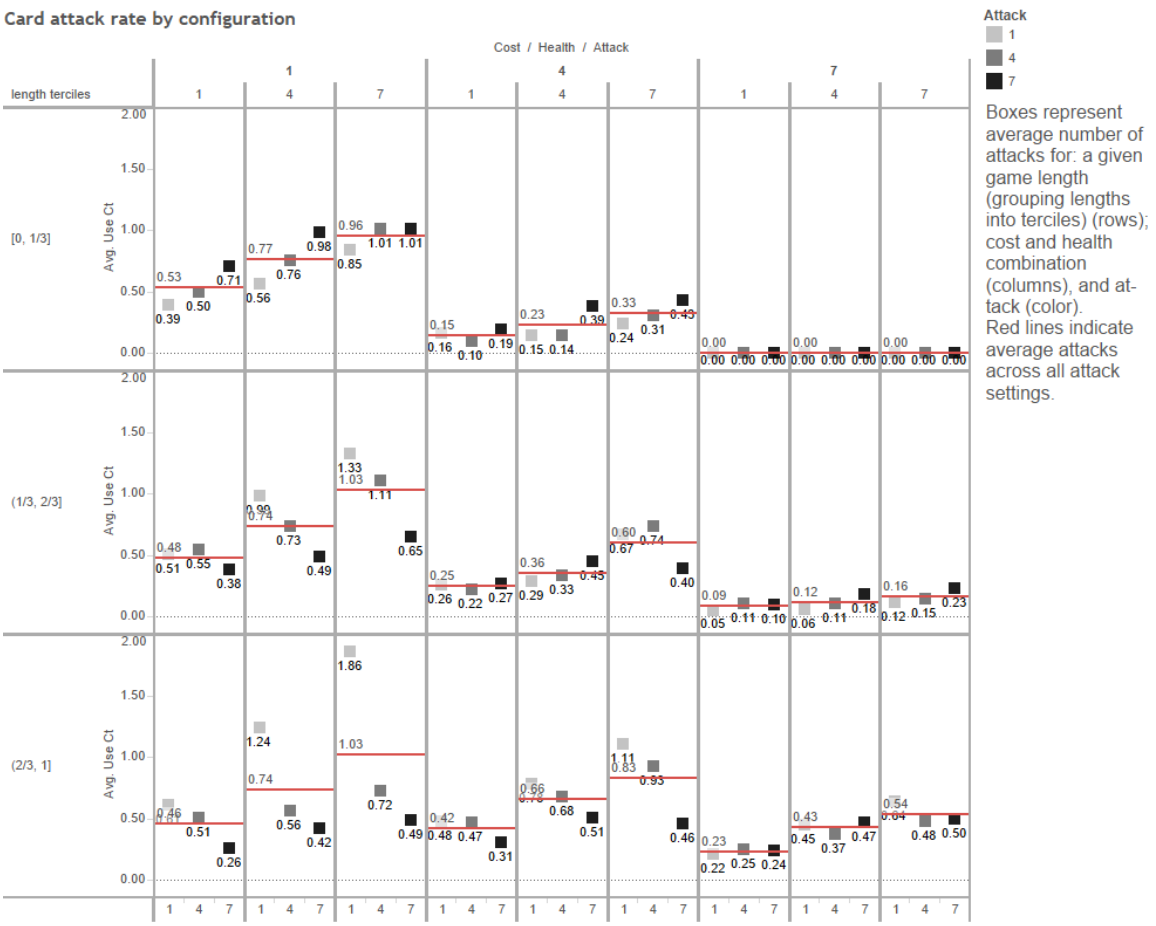

Figure 5.3: Average number of times "Stonetusk Boar" is used to attack given a card parameter configuration.

attack over games for different parameter configurations. Columns divide configurations hierarchically: first splitting by cost, second by health, and third by attack. Rows divide configurations by the game lengths, grouping games by length terciles (below 14 turns, between 14 and 16 turns, and more than 16 turns). Red lines indicate average card attack rate across attack configurations (marginalizing to game length, health, and cost combinations). High cost cards (far right triple of columns) reduce the frequency of card attack rates to near zero except in long games (bottom row). Increasing health (red lines) also increases card attack rates. These results visually corroborate the model learned by the system for human designer consumption.

Play frequency vs card parameters	
feature	coefficient
attack = 4	**0.92**
attack = 7	0.94
health = 4	1.04
health = 7	1.00
cost = 4	**0.83**
cost = 7	**0.48**

Table 5.3: Effect of card parameters on card play rates—coefficients give proportional change in card play frequency; values above 1.0 indicate increased frequency. Bold values indicate significance ($p < 0.05$).

Card Play Rates

Predicting card *play* rates used the same data to learn the model, replacing the predicted action metric of card attacks with a predicted action metric of card play while still comparing variants of the "Stonetusk Boar" attack, health, and cost parameter values. "Stonetusk Boar" play counts were not overdispersed ($p < 0.001$), so the system used Poisson regression as the predictive model. Higher cost values significantly reduced card play frequency ($p < 0.05$), with higher costs decreasing play frequency more (Table 5.3). Thus, the system learned that increasing cost reduces the frequency of agents being able to play a card.

Figure 5.4 is similar to Figure 5.3, only now displaying the average number of times the "Stonetusk Boar" card was played (rather than used to attack). This figure provides a visualization of the statistical relationships learned above. Longer games allow more opportunities for play, seen by comparing the average play rates across the three game lengths (rows), especially in the highest cost scenario (far right column). Cost clearly reduces play frequency, seen by comparing the three sets of columns (cost is the outmost grouping of columns). Conversely, neither attack nor health appear to have a directional effect, seen by inconsistent relationships among play rates for different attacks (colors) or healths (red lines).

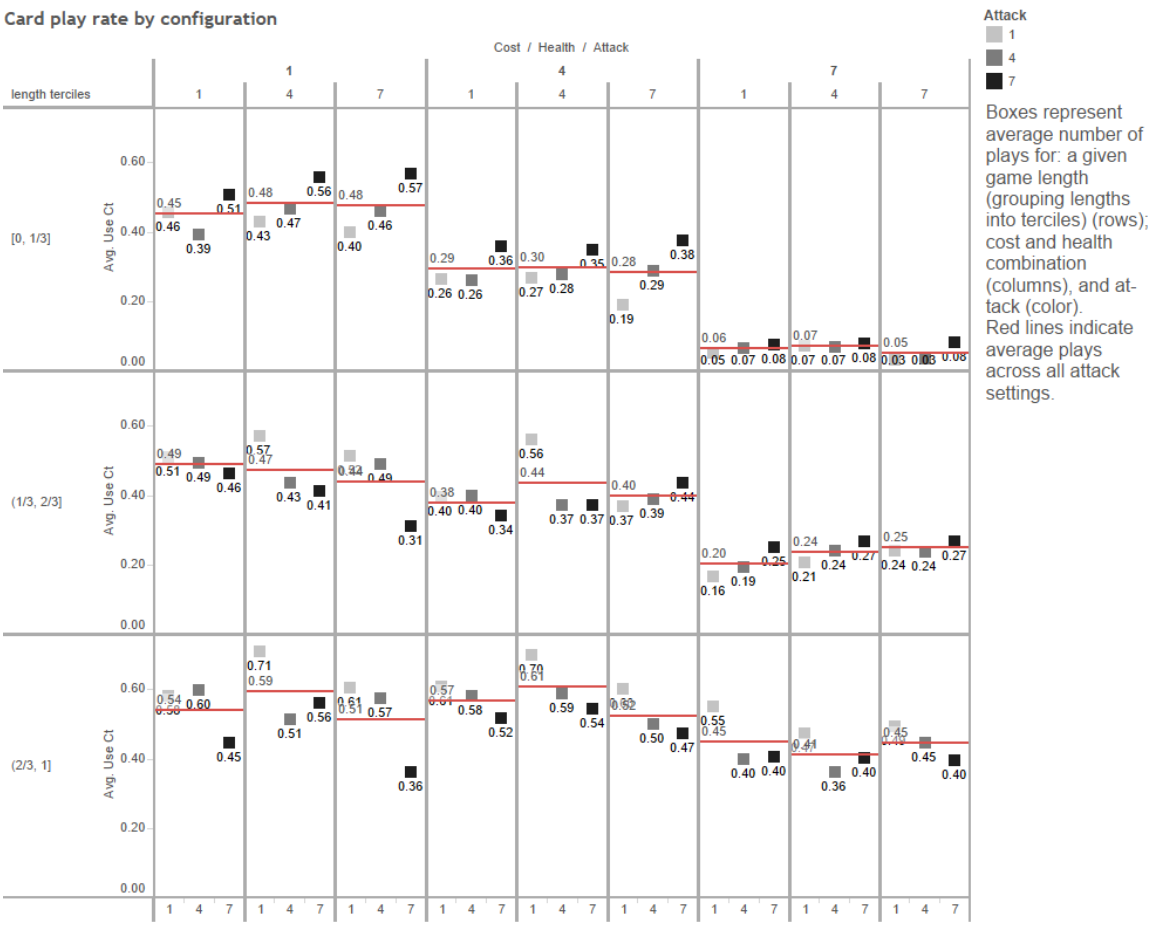

Figure 5.4: Average number of times a card is played for a given "Stonetusk Boar" card configuration.

Card Attack Options

Card *use* is not the sole indicator of the influence of design parameters on play: alterations in how often a card is an *option* for use can indicate how card parameters influence player behavior to use or hold on to cards. Card *options* are the cards an agent has the choice to use, either to attack an opponent or to play to the board. Unlike card actions (examined above), card options give a sense of the strategic possibilities an agent has at hand. As before I had the system learn how card parameters influenced card use, only now targeting the outcomes of the frequency of having the option to attack with or play the "Stonetusk Boar" card.

Similar to card attack *action* rates, the system learned a model of the frequency of card

Attack option frequency vs card parameters	
feature	coefficient
attack = 4	**0.71**
attack = 7	**0.53**
health = 4	**1.84**
health = 7	**2.49**
cost = 4	**0.71**
cost = 7	**0.31**

Table 5.4: Effect of card parameters on card attack option rates—coefficients give proportional change in card attack option frequency; values above 1.0 indicate increased frequency. Bold values indicate significance ($p < 0.05$).

attack *option* rates with varying card parameters, focusing on the "Stonetusk Boar" card. Card attack option counts were overdispersed ($p < 0.001$), so the system used negative binomial regression as the predictive model. As with card attack counts, all parameters had a significant effect ($p < 0.05$), with greater health values increasing the rate of attack options and greater cost values reducing the rate of attack options (Table 5.4). Greater health values have strategic implications: when a card has more health it is not only useful for the act of attacking, but also as an option for attacking later. Greater cost values reduce the rate at which a card is available to attack as the greater cost gates use of the card. Thus, the system learned a model demonstrating how to alter card design to change how often a card is (or is not) available for action in the game.

Figure 5.5 provides a visual overview of the card action option outcomes in a similar manner to Figure 5.3. The similarity to Figure 5.3 supports the conclusion that card parameters have similar effects on attack actions and attack options.

Card Play Options

Similar to card play *action* rates, the system learned a model of the frequency of card play *option* rates with varying card parameters, focusing on the "Stonetusk Boar" card. Card play option counts were overdispersed ($p < 0.001$), so the system used negative binomial regression as the predictive model. As with card play actions, card play options were

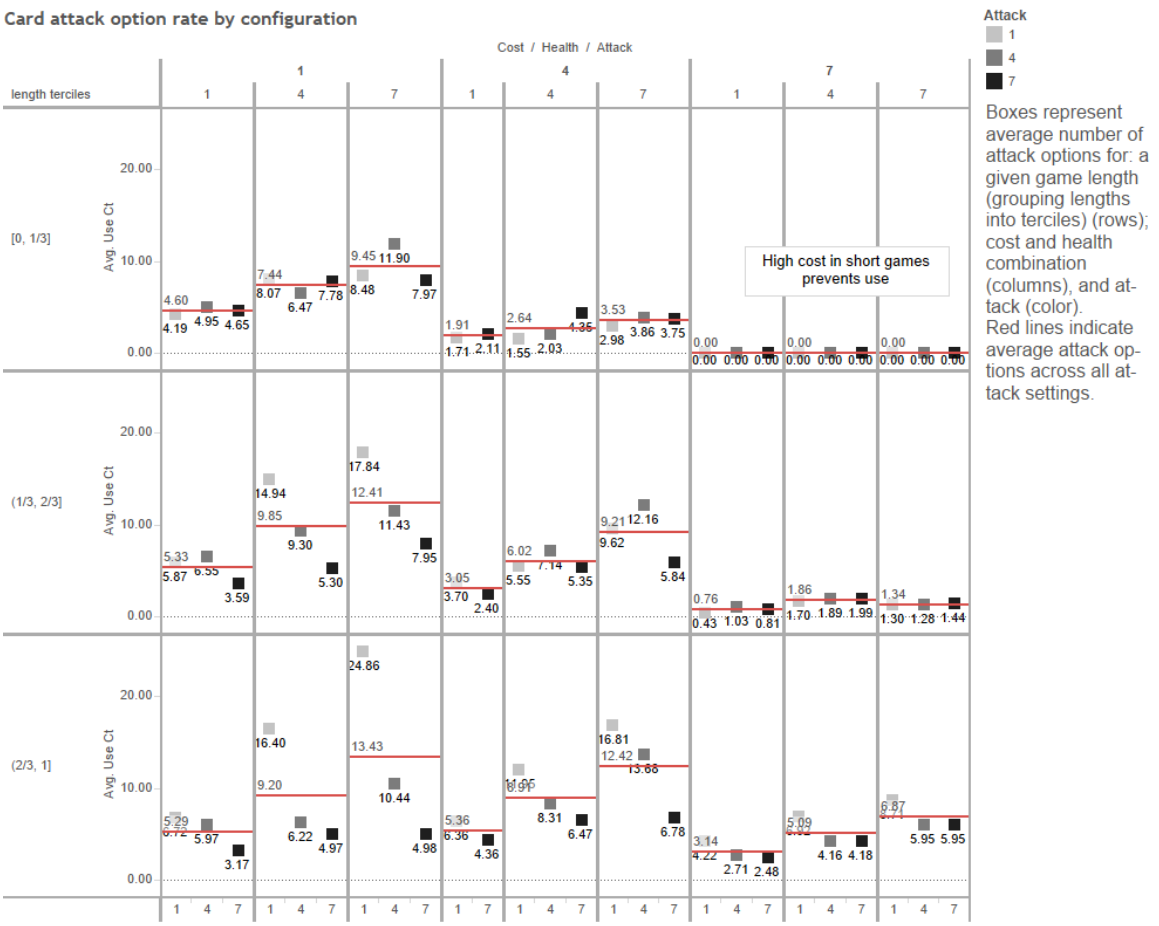

Figure 5.5: Average number of times a card is played for a given "Stonetusk Boar" card configuration.

significantly influenced by all card parameters (Table 5.5). Greater cost values increased the frequency of play options, reflecting the way cost prevents a card from being played. Both greater card attack and greater card health led to small reductions in the rate of card play options. Thus, the system learned a model predicting that as the card benefits (attack and health) when played on the board increase, the card becomes more attractive to play, leading to fewer turns where the card is retained as an option.

Figure 5.6 is similar to Figure 5.4, only now displaying the average number of times the "Stonetusk Boar" card was an option to play, rather than being played. Longer games allow more play options, seen by comparing the average play rates across the three game lengths (rows), especially in the highest cost scenario (far right column). Cost clearly reduces play frequency, seen by comparing the three sets of columns (cost is the outmost grouping of

Play option frequency vs card parameters	
feature	coefficient
attack = 4	**0.94**
attack = 7	**0.91**
health = 4	0.98
health = 7	**0.95**
cost = 4	**1.45**
cost = 7	**1.08**

Table 5.5: Effect of card parameters on card play option rates—coefficients give proportional change in card play option frequency; values above 1.0 indicate increased frequency. Bold values indicate significance ($p < 0.05$).

columns). Conversely, neither attack nor health appear to have a directional effect, seen by inconsistent relationships among play rates for different attacks (colors) or healths (red lines).

5.7 Limitations and Future Work

5.7.1 Automating Design Knowledge Structures

The studies in this chapter demonstrate how simulated play data from design variants can be used to automatically iterate on designs and learn models to predict how aspects of a game design influence player behavior. The models learned, however, were all human-provided—the system was given a set of features (card parameters) to examine. Machine learning techniques provide a suite of tools for automated feature selection, often with strong mathematical foundations. Yet these methods typically work from the model of having a predefined, relatively small space of features to consider, relying on human intuition to choose the appropriate subsets of features to consider. Automating the modeling of a design space will require ways to bias a system toward iteratively exploring increasingly complex models of how a design works to acquire (and re-test) knowledge about how that design works. In this case this would entail a loop of using a model to select a sample of new design variants to test, sampling behavior from those variants, and updating the learned model with outcomes from those design variants. Recent efforts have begun exploring how

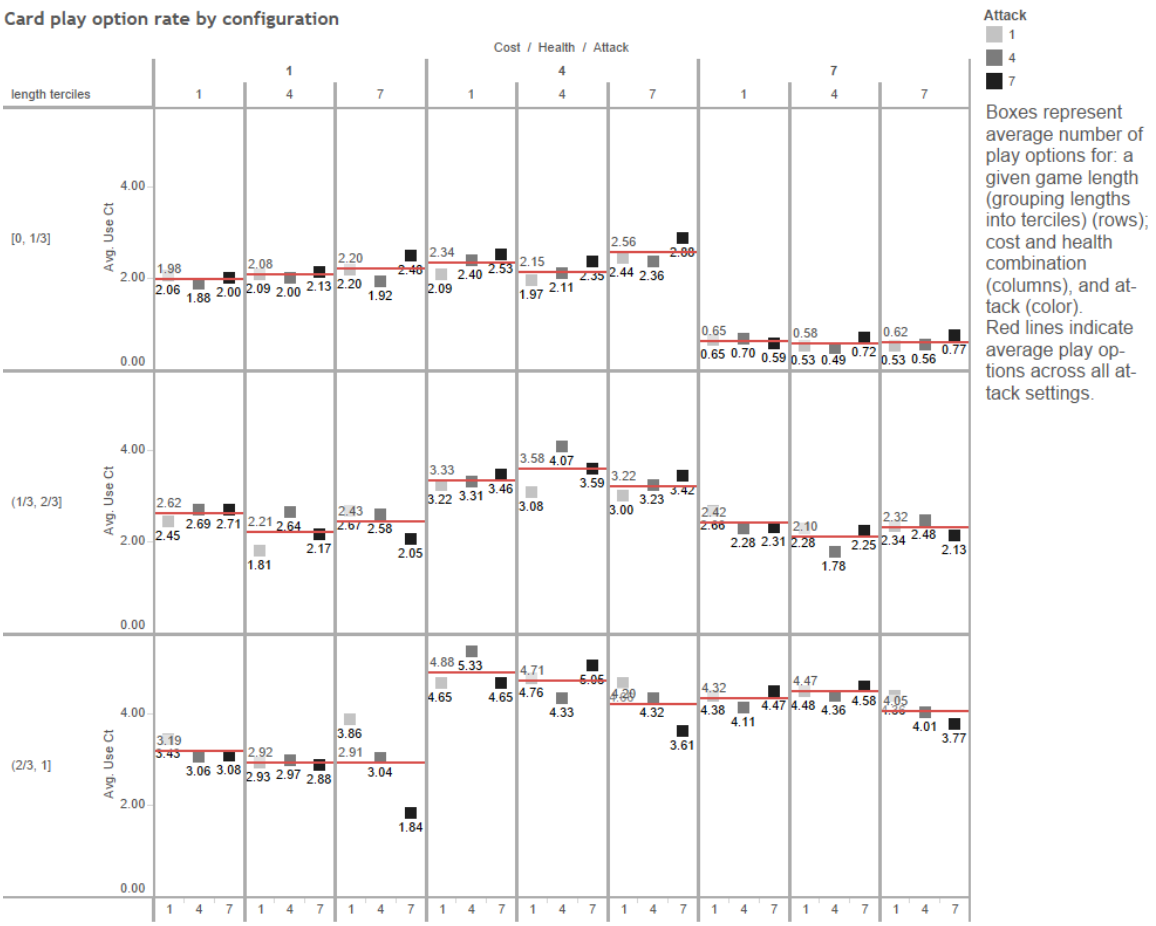

Figure 5.6: Average number of times a card is played for a given "Stonetusk Boar" card configuration.

search processes combined with Bayesian statistical hypothesis testing frameworks can be used to learn human-like models of the structure underlying observations of real-world phenomena [60, 78, 105, 214]. These models may be amenable to extension and application to enabling the automated learning of design knowledge about game designs as well.

Fully automating the process of analyzing and generating games requires techniques to use the models built to inform future design iterations. Ideally the models built from play data can inform the choice of new design variants to test: not only checking different parameter values, but choosing new game design parameters to vary in a way that maximizes knowledge acquired about the space of designs. For example, the agent from this chapter could consider changes to the rate at which agents gain mana over turns in the game, rather than only manipulating card parameters. This approach would support an

agent gradually accruing knowledge about a game design domain, though it will require complex modeling of the relations of components of the design space to one another and testing (and re-testing) the models built over time. Effectively searching this space of alternative knowledge will require careful modeling of how different axes of design changes may support learning toward desired design goals and which axes of change may be less valuable for future iterations. In the next chapter I show how optimal experimental design techniques can improve the process of seeking design variants to optimize for design outcomes (the first portion of this chapter); developing the appropriate techniques for learning predictive models of the design space remains an open question.

The design knowledge learned in this chapter is readily described in human-legible terms (as done throughout the results section). Ideally an automated system could present these outcomes to humans directly through a combination of text and relevant imagery. Enabling the system to choose the appropriate output and the subset of all learned knowledge most relevant to a human remains an open challenge. Recent efforts have begun developing ways for machine learning systems to render their output in human-understandable language [119]. Extending these efforts to appropriately filter from a large space of disproved or inconclusive information, however, requires further modeling of which outcomes are truly useful or interesting and which are not.

5.7.2 Scaling

The data in this chapter evaluated *Cardonomicon* variants in terms of changes to a single card. Game design spaces typically involve a wide variety of features to alter that have many potential interactions. This poses a challenge: how well can these techniques scale with a (1) large number of variations of a feature or (2) complex interactions among features?

For the card example in this chapter a single card varied in three levels of three parameters. Generating and evaluating these variants required roughly two weeks of computer

time using a 64-bit machine with 16GB of RAM and a 3.40GHz processor. When examining more fine-grained variants of a design or change to more parameters at once this would rapidly explode the space of design variants possible. Two avenues would provide some ability to mitigate these growing computational costs: parallelization and efficient search. First, these examples are massively parallelizable, as each design variant, level of agent strength, and game played is independent of all others. As such, the evaluation of many designs can be readily made into a parallel process on a single machine using multiple CPU threads, GPU parallelization, or parallelized on a cloud computing platform. This does not remove any computational cost from the model, but instead provides practical means to reduce the time needed to run the system.

Second, the grid of parameters used was defined *a priori* and not adjusted at run time. In practice the grid search could use binary search and other simple search techniques to more efficiently iterate on a design when seeking to optimize the design. When learning design knowledge these iterations could be evaluated in terms of how well the provide information on the knowledge being learned. In the next chapter I present the application of techniques from active learning as one way to do this more intelligent search process.

The other challenge of evaluating this design space entails handling feature interactions. When more than one card is varied at a time the system must both sample more designs to vary both cards and learn more complex (and harder to learn) models of how card features interact. The problem of evaluating more design combinations falls under the same challenges as discussed above regarding parallelization. The problem of learning feature interactions with sparse examples, however, requires alternative techniques for sparse learning. To date the challenges of learning from sparse examples have largely been addressed using Bayesian statistical modeling, using prior knowledge to bias the models learned until sufficient information is gained. These techniques hold promise for helping mitigate the needs to generate many examples of behavior from games.

Alternatively, design systems may use a small set of initial examples to learn how

strongly related different features in the design are. In this approach a system would attempt to find weakly coupled game systems that could be sampled independently, borrowing ideas from Simon's [186] thinking on weakly coupled systems. In this case, the system would attempt to mitigate the cost of learning about a design space by learning additional meta-information on the structure of the overall space. Ultimately, however, automated search and learning will still face computational barriers that require alternative solutions such as search heuristics, constraints on the designs considered, or prior design knowledge. The approaches presented in this chapter provide a simple initial approach that exposes the need for further development of these automated learning techniques.

5.8 Potential Impact

Game design research has the potential to change the way games are made and the experiences available to game players. In this section I briefly discuss how the methods for game design optimization and design knowledge learning in this chapter might influence game designers and players.

5.8.1 Game Designers

Game designers are the primary audience benefiting from the automated techniques for optimizing a design and learning about its functionality. Automated design optimization has obvious use for maximizing a given design metrics for a game. In the future, this could change the practice of tuning games to emphasize designers developing metrics to quantify the quality of a game in terms of the behaviors possible in the game, rather than a practice of iteratively adjusting a game until observing the desired set of behavior in the game. That is, design practice would change from examining a few (hopefully representative) concrete examples of behavior to using a variety abstracted design metrics. Designers would then focus more on ensuring patterns of play are typically observed, rather than adjust a design to produce desired exemplar behavior. While concrete examples of behavior will never be

removed from design practice, automated gathering of summary metrics stands to alter the way designers iterate on a game.

The generality of the underlying MCTS framework and action metrics used in the system above provide a core set of systems that can readily be incorporated into a game engine. This in turn opens the potential for game engines that automatically generate potential game behaviors (in certain scenarios) and provide output metrics on these scenarios. The system can even automate the process of selecting candidate designs that optimize different design metrics to provide suggestions of extreme design examples for a designer to consider.

Having a game engine able to model a space of designs in a game opens the potential for a system to highlight what aspects of a design a designer has explored and what remains unknown. By quantifying certainty in different pieces of design knowledge a game engine could help a designer recognize what aspects of a design space she has explored well and what aspects of the space remain untouched. Even with limitations in how a system can automatically formulate design knowledge, this combined human-computer system could more effectively learn about designs and retain that knowledge. In the future this could lead to hybrid design approaches where designers use computational systems to model and learn about behaviors in the designs they explore, providing automated documentation of design iterations and learnings for later reference.

5.8.2 Players

Automated design learning can provide players new types of games built around meeting or violating models of behavior a system learns. For example, if a system learns that a weaker card shortens game length, players could be automatically challenged to find cases supporting or violating this learning. This in turn would create a feedback loop of a system posing learned knowledge and using players to improve this design knowledge by posing examples and counter-examples. Players might also be able to provide example knowledge of their own for the system to evaluate on other players. This kind of game

design provides players with automated goals in content—the knowledge to validate or violate—while providing data to a learning game design system.

5.9 Summary

In this chapter I showed how a space of design variants could be evaluated to find designs that optimize a given desired gameplay outcome. The same technique was also used to test design knowledge about how game design parameters influence the space of play in a game. For these studies I generated a set of game design variants that covered a range of card parameter settings in the *Cardonomicon* card game domain. Using MCTS to generate behavior samples I evaluated the game design variants in terms of the use of actions to attack with cards or play cards. Comparing across these variants allowed selection to optimize for game feature outcomes including game length and card use frequency. The evaluations also allowed for learning about how card cost, health, and attack influence agent actions.

Generating the space of design variants, however, is expensive. In this chapter the process required simulating playing 16200 games just to assess the influence of three parameters of a single card on the space of play in *Cardonomicon*. If humans were playing these games the testing required for a single iteration on the design would be even more time consuming to the point of preventing any practical use of these methods. How can design space evaluation be efficient (while remaining effective) for automated iterative design? The next chapter addresses this question by applying techniques from optimal experimental design to the problem of parameter tuning in a design space. Optimal experimental design techniques enable a system to trade off between exploring different design variants and improving on a given design variant when iterating on a design. This analysis shows how to extend the approach in this chapter to a different game domain, focusing on how to use optimal experimental design techniques with human playtesters to efficiently optimize for two major types of game design goals.

CHAPTER 6
DESIGN ITERATION FOR PARAMETER TUNING

6.1 Introduction

Game designers seek to shape player behavior, but only have indirect influence through choices of a game's design. Designers have a notion of what player behavior is expected of the game from the range of all behaviors that might be possible in the game: I call this the *play space* of the game. To learn about actual player behavior, designers playtest by having people play the game to see the distribution of possible behavior in the game. Playtest results inform designer expectations for how different choices of game elements might induce different player behaviors. Using these expectations designers then choose a new design to consider, balancing between trying radically different ideas to fine-tuning existing choices through small changes. I call the space of possible game designs the *design space*: an individual element of this space is a single game design. Design iteration is the task of navigating the design space to find a desired design, using playtesting information to evaluate individual designs in the space.

Design iteration, particularly when human playtesting is necessary or desirable, is an expensive process. Recruiting people to play a game (online or offline), having people play a game, aggregating and analyzing playtest results, and making design decisions to fine-tune the game are all time-consuming efforts. In many cases design iteration can be mundane: playtest results often lead to small design changes, yet become the primary task of a designer in the late stages of fine-tuning a game's parameters. Despite the ubiquity of design iteration for fine-tuning game parameters, we lack a computational model of how to perform this process.

Computationally modeling the process of choosing design iterations can alleviate the

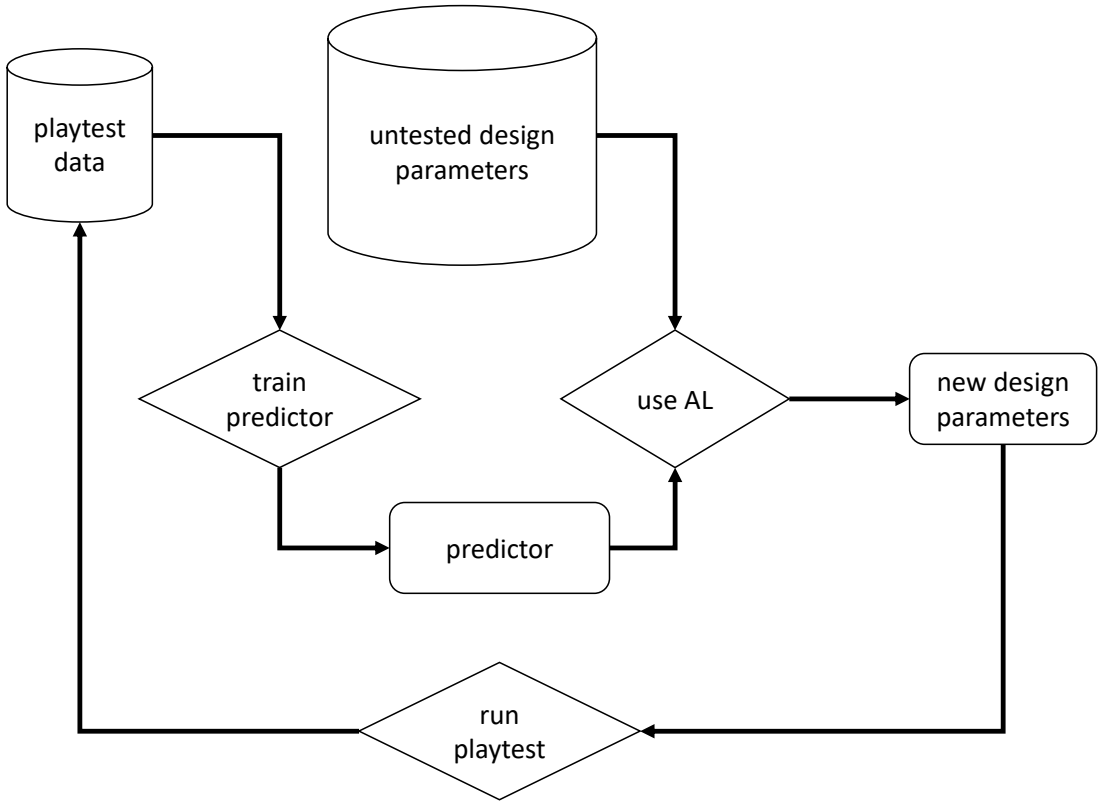

Figure 6.1: Diagram of active learning process.

burden of design iteration by enabling a machine to run the process of processing playtest data, choosing design alternatives to test, and deploying those design alternatives (Figure 6.1). By computationally modeling this process we can attempt to increase the efficiency of playtests at reaching a design goal by minimizing the number of playtests run. This design iteration model can further inform our understanding of the best ways for people to perform design iterations and contribute to improved mixed-initiative systems where computers help people design games.

In this chapter I present a system for design iteration where a computer deploys playtests online, gathers results from those playtests, and uses those results to optimize toward a designer-given goal for the game design. The approach I take treats the choice of the next game design to test as a machine learning *active learning* problem [177]. Active learning (AL) techniques were developed to improve the efficiency of training machine learning

models in cases where there is little data available, or gathering new data is expensive. AL techniques start from an existing set of labeled data to train a predictive model. Given a set of potential unlabeled data to add to the model, AL techniques choose the optimal next data to query to add to the model to improve the model. These data are added to the model and the process repeats.

From the active learning perspective, the design space is an space of possible games to use as input and design iteration is the problem of choosing the next game to test to optimize an objective for the game design (Figure 6.1). I show how this model can optimize for attributes of objective player behavior (like how much damage players take in a game) and attributes of subjective player feedback (like whether a set of controls were preferable). I show how a number of active learning models map to different approaches to navigating the design space in terms of balancing between *exploring* very different designs to learn about alternatives and *exploiting* similar designs for fine-tuning. By addressing the inherent cost of playtesting this model can improve human playtesting practices or enhance automated game generation systems that use some form of iteration (primarily search-based generation models [220, 222]). This is the first work (I am aware of) to address this component of game design and closes the loop on the final step of design iteration.

6.2 Design Iteration as Active Learning

The key insight of this model is to treat design iteration as an active learning problem. Active learning [177] is the machine learning problem of selecting among a set of possible inputs to get the best output while minimizing the number of inputs tested. Here, the potential inputs are the design space, with each input being a game design. The best output in this case is the output that most closely matches a designer-given design goal: e.g., having the player take (on average) a certain amount of damage in a battle or tuning player controls to those they prefer. Minimizing the number of inputs becomes the task of testing as few games as possible, reducing the number of people who must playtest the game and

time the playtesting process takes overall.

Active learning provides a variety of generic techniques that address different learning goals and different ways to optimize for achieving those goals cheaply. A core problem in active learning—shared by human design iteration—is the balance between exploration and exploitation. Exploration emphasizes testing unknown parts of the input space to gather new information, similar to a designer testing a radically different design to see if there are viable alternatives being overlooked. Exploitation emphasizes testing slight variants of known parts of the input space to attempt to improve existing results, similar to a designer fine-tuning a design to inch closer to an optimal result. Active learning provides a number of computational approaches to the exploration-exploitation problem that can prove more or less efficient in different contexts. I will show how different approaches fare in a test domain to illustrate how the choice of "best" approach differs by design context.

The active learning model of design iteration has three core components: (1) a design model, (2) a design goal, and (3) a playtesting strategy. Formally, a *design model* is a function that models how an input set of game design features maps to an output set of game metrics. Design models capture how a designer expects different design variants to work based on prior knowledge of design alternatives they have tested. A *design goal* is an *objective function* that specifies how to evaluate game metrics as being better or worse. Game metrics are any quantitative summary metrics from a playtest. In active learning, objective functions define the goal for a model, such as minimizing error; in design iteration, this becomes the desired metrics from playtesting the design. An *iteration strategy* is an *acquisition function* that uses predictions for game metrics from the design model and evaluations from the design goal to choose the next design to test among the alternatives in the design space. In active learning, acquisition functions balance exploration and exploitation of design alternatives to maximize for design goals; in design iteration, this is the human heuristic for choosing designs to playtest next.

There are a number of alternatives for choosing design models, each with many alterna-

tives for iteration strategies. Design goals can range more widely, with the only restriction that they provide the type of output needed by a design model. Design models come in two forms: regression and classification. Regression models take as input a set of game features and predict as output a *continuous* game metric. Continuous game metrics are common in games, from the time it takes to finish a level to the amount of damage taken in a battle. Classification models take as input a set of game features and predict as output a *discrete* game metric. Discrete game metrics are also common in games, from choices among text responses in interactive fiction to selecting normal or inverted look controls in a first-person game. Note that prediction models capture *expected* player behavior, aggregating over individual differences. In this work we are concerned with choosing a single design for all players, making the models purely depending on game features and excluding player features (such as age, gender, prior experience, &c.). This is not a limitation of the model presented: player features could be incorporated for games that adapt to audiences online or have default parameters that vary for different players.

Acquisition functions (iteration strategies) differ for regression or classification models. In the next sections I discuss a number of alternatives that balance exploration and exploitation in different ways. I emphasize the intuitive meaning of these models for design iteration, leaving the full mathematical treatment to the referenced materials.

6.2.1 Regression Models

Acquisition functions balance exploration and exploitation to minimize the number of inputs tested to optimize an objective function. In this work I consider models that span the exploration-exploitation spectrum, including models that are purely exploratory or exploitative and models that balance between the two in different ways. For regression models I used Gaussian Processes (GPs), the standard non-linear regression model used in the Bayesian experimental design literature [159]. Gaussian processes have a number of attractive features: they can model a wide class of non-linear relationships, they are com-

putationally inexpensive to update, and have been the most common model for developing acquisition functions. A non-linear model is appropriate for game design as non-linear relationships are common in games, where changing parameters is not expected to produce a directly proportional change in player behavior.

I consider four acquisition functions for regression models: (1) variance, (2) probability of improvement, (3) expected improvement, and (4) upper confidence bounds. These acquisition functions were developed in the field of Bayesian experimental design and apply generally to any regression model with a probabilistic interpretation [18, 28]. These four regression acquisition functions are:

- **Variance** Exploration by picking the input with greatest output variance (uncertainty) [18]. Corresponds to picking the design that is hardest to predict the outcomes from.

- **Probability of Improvement (PI)** Exploitation by picking the input most likely to have an output that improves over the previous best [18]. Corresponds to picking the design most likely to improve over the current best.

- **Expected Improvement (EI)** Balances exploration and exploitation by picking the input by weighting the output amount of improvement by the probability of output improvement [18]. Corresponds to picking the design with largest expected improvement.

- **Upper Confidence Bound (UCB)** Balances exploration and exploitation by picking the input with greatest combined expected output value and output uncertainty to gradually narrow the space of inputs [205]. Corresponds to picking designs that seem high quality but are poorly understood to gradually narrow the space of design to be known good or expected bad but uncertain.

6.2.2 Classification Models

Classification models are primarily concerned with increasing certainty in predicting outcomes—improving the model of how the design works. Unlike regression, there is not a single commonly used model for classification. I used three classification models that cover common approaches: Gaussian Processes (GP), Kernel Support Vector Machines (KSVM), and optimized neural networks ("neuro-evolution", NE). Kernal methods are a popular machine learning technique for dealing with input spaces (game features) with complex relationships and have been previously applied to player modeling [244]. Neuro-evolution has been widely applied to preference learning in games to handle the complex relationship between game features and player preferences [241].[1] Gaussian Processes can be applied to classification tasks and were included as a point of comparison to the regression modeling case.

I consider five acquisition functions for classification models: (1) entropy, (2) query-by-bagging (QBB) vote, (3) query-by-bagging (QBB) probability, (4) expected error reduction, and (5) variance reduction. These acquisition functions have been developed for classification models; several—entropy, QBB probability, and expected error and variance reduction—require probabilistic predictions. Since neuro-evolutionary models do not produce probabilistic predictions they cannot be used with some of these acquisition function; in evaluation tests below I include results only for valid combinations of classification model and acquisition function. The five classification acquisition functions are:

- **Entropy** Picks the input with greatest output uncertainty according to entropy—a measure of the amount of information needed to encode a distribution [177]. Corresponds to picking designs where player choices are most uncertain.

- **Query-By-Bagging (QBB)** Picks the input with most disagreement among copies of

[1] For computational reasons I use a gradient-based optimization method for network structure, size, and weights, rather than an evolutionary algorithm as used in most neuro-evolutionary approaches. I found no performance differences between the two optimization approaches in initial tests on the study data below, though the evolution approach required substantially more time to train.

a classification model trained on random subsets of known results [177]. **QBB vote** picks the input with the largest difference between its top two output options [177]. **QBB probability** picks the output with greatest average uncertainty across the models [1]. Corresponds to picking designs with greatest variability in expected outcomes when viewed based on different subsets of playtest data.

- **Expected Error Reduction** Picks the input that, if used to train the model, leads to the greatest expected reduction in classification error [177]. Corresponds to picking designs that will most improve prediction of the design outcomes over the design as a whole.

- **Variance Reduction** Same as expected error reduction, but seeks to reduce variance in output predictions rather than error [177]. Corresponds to picking designs that lead to greatest reduction in uncertainty about the design space over time.

6.3 Experiment Design

The question I seek to answer is: how effective can this active learning model be at reducing the number of design iterations needed to optimize toward a design goal? To study the efficiency of the alternative iteration strategies (acquisition functions) I used a simple shoot-'em-up game (Figure 6.2) that tested for optimizing two broad classes of design goals: (1) player game play behavior and (2) player subjective response. Player game play behavior goals cover cases where designers desire particular play patterns or outcomes—e.g., player success rates or score achieved. Subjective responses goals cover cases where designers desire specific player subjective feedback—e.g., getting good user ratings on the feel of the controls. Together these goals encompass a broad range of typical design concerns during playtesting, demonstrating the value of active learning as a generic approach to design iteration.

Evaluation used both a simplified simulation for player behavior and data collected

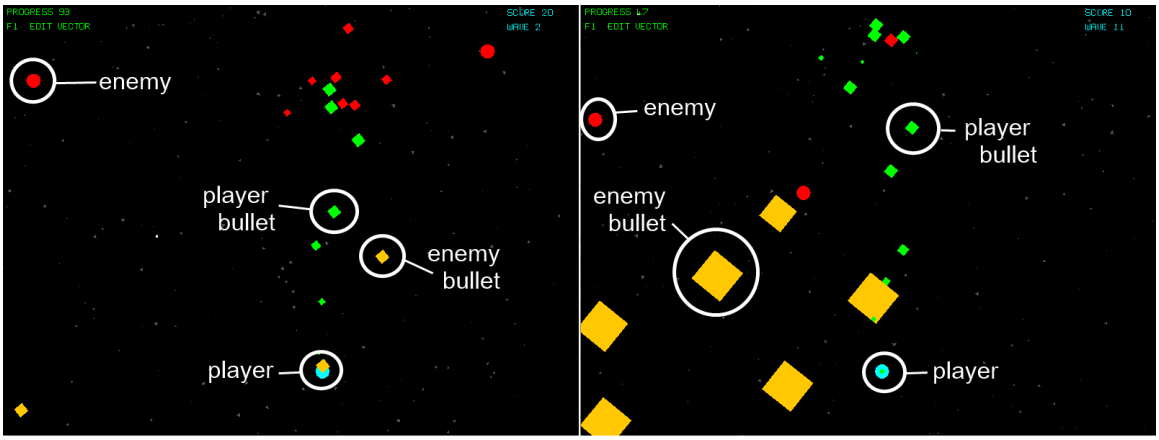

Figure 6.2: Study game interface illustrating player, enemies, and shots fired by both at two points along iteration process.

from people playing the game online. Simulations allow verification that the method works in principle and can test edge cases of player behavior that do not appear with among a given study population (but may occur in a larger population). Human studies demonstrate the method applies to real-world situations. In the following sections I describe the game domain used, simulation model, and data collection for human data.

6.3.1 Game Domain

While design iteration can alter any aspect of a game, for scope this study focuses on tuning the parameters of a game. Game parameters are values chosen by designers to tune existing rules or structures in a game. For example, in a platformer a design might tune the strength of gravity or height a player may jump in the game, keeping the mechanics for jumping and gravity fixed. Parameters can be set to a wide range of values, leading to substantially different games: a platformer without gravity will play very differently to a platformer with gravity. Parameter tuning is a useful setting for testing design iteration as the bounds on parameters can define a large design space with many uninteresting or impossible designs. At the same time, the space is likely to contain valuable games: the range of games afforded by low-level control over designs tests the efficiency from the active learning system.

As a domain of study I used a shoot-‘em-up game (Figure 6.2): these games are similar

to *Space Invaders*, where waves of enemy ships attack a player who controls a ship.[2] Shoot-'em-up games emphasize reflexes and pattern recognition abilities as a player maneuvers a ship to dodge enemy shots and return fire. Shoot-'em-up games, and arcade games in general, are an ideal domain to test low-level parameter tuning:

- There are a number of parameters that can potentially interfere with each other: size and speed of enemies and enemy bullets, rate of enemy fire, player speed, player rate of fire, &c.
- The game can be played in a series of waves, enabling our system to naturally test game parameter settings and gather player feedback.
- Action-oriented gameplay reduces the complexity of player long-term planning and strategizing.
- A scoring system makes gameplay goals and progress clear, unlike domains involving puzzle-solving or aesthetic enjoyment of a game world or setting.

Combat in the shoot-'em-up game occurs over a series of waves of enemies. During each wave a series of enemies appear that fire bullets at the player: the player's goal is to destroy the enemies while dodging their fire. To encourage players toward this gameplay they were shown a score that rewarded points for enemies defeated and penalized player score when hit by enemy fire. To test AL for regression I set a game play behavior design goal (objective function) of the player being hit exactly six times during each wave of enemies (output) and tuned enemy parameters (input). The goal was evaluated by squaring the difference between the score a player achieved on a wave of the game and the desired score (being hit six times). A squared difference more steeply penalizes games with greater differences from the ideal. I used three game parameters to tune enemy power: the size of enemy bullets, the speed of enemy bullets, and the rate that enemies fire bullets. Increasing

[2]The game was developed in conjunction with Eric Fruchter, who did the bulk of the game engine programming.

bullet size requires the player to move more to avoid bullets. Faster bullets require quicker player reflexes to dodge incoming fire. More rapid firing rates increase the volume of incoming fire. Together these three parameters govern how much players must move to dodge enemy attacks, in turn challenging player reflexes. These three parameters admit a number of potentially effective designs: e.g., a game with large, slow moving, and rarely fired bullets forcing players to plan a path between shots or a game with small, fast, and more rapid enemy fire requiring players to quickly react to oncoming attacks. Getting approximate settings for these parameters is easy, but fine-tuning them for a desired level of difficulty can be challenging.

To test AL for classification I set a subjective player response design goal (objective function) of the player evaluating a set of controls as better than the previous set (output) and tuned player control parameters (input). The goal was evaluated in terms of prediction quality from the classification model using the F1 score as a metric. I provided two parameters for player control over ship movement: drag and thrust. Drag is the "friction" applied to a ship that decelerates the moving ship at a constant rate when it is moving—larger values cause the ship to stop drifting in motion sooner. Thrust is the "force" a player movement press applies to accelerate the ship—larger values cause the ship to move more rapidly when the player presses a key to move. Combinations of thrust and drag are easy to tune to rough ranges of playability. However, the precise values needed to ensure the player has the appropriate controls are difficult to find as player movement depends on how enemies attack and individual player preferences for control sensitivity (much like mouse movement sensitivity). Some players may prefer ships that move and stop quickly (high drag and thrust), while others may find a moderate amount of drift more intuitive (with lower drag and thrust). After each wave of enemies a menu asked players to indicate if the most recent controls were better, worse, or as good/bad as ("neither") the previous set of controls. I provided a fourth option of "no difference" for when players could not distinguish the sets of controls, as opposed to "neither" where players felt controls differed but

had no impact on their preferences.

6.3.2 Simulation Models

Simulations used two simple models of how players might respond to different design parameters. Both simulations are models of expected player behavior in response to design parameters, rather than agents that directly play the game in-engine. The regression simulation treats players as having an underlying set of skills related to each enemy tuning parameter along with a cross-skill tolerance for differing challenge demands. Greater differences between player skills and enemy parameters lead to larger differences from being hit at a base rate. The classification simulation treats players as having preferences for each of the control tuning parameters with a cross-parameter tolerance for differences from preference. Preference choices are based on the difference between the ideal set of parameters and design control settings.

The regression simulation is a probabilistic model of player behavior in terms of underlying player skills and design parameters. Simulated players have three independent skills for responding to enemy bullet size, bullet speed, and firing rate. Values for all three skills are sampled from a normal distribution with a variance term capturing variability in skill. Taking the difference between the player-specific skills and the design parameters, then scaling by the error in player skills produces an estimated rate of being hit by enemies in our game. When using simulated players I generated a fixed ideal playtester and allowed the AL model to choose sets of enemy parameters among 10 bullet sizes $\times 10$ bullet speeds $\times 10$ firing rates $= 1000$ design variants. Each playtest generated a new playtester.

The classification simulation is a probabilistic model of player responses in terms of underlying control preferences and design parameters. Simulated players have two independent preferences for force and drag parameters; both are sampled from a normal distribution with a variance term capturing variability in preference. There is also an error threshold that sets a lower bound for parameter differences that a player may distinguish.

When given a set of design parameters the model performs a two-stage comparison process. First, each individual design parameter is compared to the desired parameter for the model by taking a cumulative normal distribution centered at the difference of the parameters and scaled by the player variance term. Differences below the player error threshold yield a "no difference" result; positive or negative differences above the threshold yield "better" or "worse" responses, respectively. Second, the individual parameter responses are combined into the final model response. If both responses are the same then that response is given. If one response is "no difference" then the other response is given. Otherwise the model responds with "neither." When using simulated players I generated a fixed ideal playtester as before and varied sets of controls using a grid of 5 current force $\times 5$ current drag $\times 5$ previous force $\times 5$ previous drag $= 625$ design variant grid for both current and previous wave control parameters.

Both these simulation models are intended as simplifications of players used to ensure the active learning model is robust to a wide range of player behavior and response patterns. While neither model is a strong representative of true human behavior, comparing simulation results to human behavior helps illustrate the reasons for specific cases being easy or hard for active learning to optimize.

6.3.3 Human Data Collection

The human study used two versions of the game deployed online. To recruit subjects I publicized the game through websites and local emailing lists; no compensation was offered for participation. Players were asked to try to play at least 10 waves of the game and were given no upper bound on the number of waves they played. The lower limit was not enforced through the game, but for analysis I discarded data from players with fewer than 10 waves played. This measure helped ensure players were able to reliably play the game (i.e., the game did not crash or they did not understand the game and quickly quit).

After filtering, the regression experiment collected data from 137 player and 991 waves

of the game in total. The classification experiment gathered data from 57 players: of these, 47 only provided the two binary responses of "better" or "worse" and analysis was limited to this subset of players to result in 416 paired comparisons. Preference responses were only used from the first 10 waves of the game—this avoids biasing the sample with data from players who were highly engaged and could provide very skewed positive responses. Preference data was not collected from the first wave of the game as players could not yet compare control settings.

6.3.4 Active Learning Experiment Design

The analysis used the data from the simulated models or collected from human players to study the effectiveness of different active learning models. The analysis tested whether AL could reduce the number of human playtests needed to tune design parameters compared to a random sampling approach. Random sampling is the standard baseline used to evaluate the efficacy of active learning models for improving an objective function for a fixed number of inputs [177]. Random sampling is similar to the A/B testing approaches used in game design that playtest a game with a large audience and then act on the playtest results. The primary difference in this case is that I use batches of a data from a single playtest, rather than many playtests, to update the active learning model.

I performed 10-fold cross-validated experiments to measure how well a playtesting strategy (acquisition function) could achieve a design goal (objective function) given a set of design parameters (input). For regression a Gaussian Process (GP) was trained to predict the number of times a player was hit during a wave of the game using the three enemy parameters. For classification one of the three classification models (GP, KSVM, or NE) was trained to predict control preference indication as better or worse using the two control parameters from both the previous and current wave of the game. In both regression and classification cases I tested all valid combinations of acquisitions with the design model.

For each cross-validation fold I first randomly selected 10% of the data and set it aside

for evaluating objective function performance. Next I randomly sampled 30 input–output pairs from the other 90% of the data set to create a training data set; the remaining unused samples formed the training pool. In the case of simulation these pairs were generated and there was no larger population pool to choose from. Within each fold I then repeated the following process:

1. Train the regression or classification model on the training data set.
2. Evaluate the objective function for that model on the testing data set.
3. Use the acquisition function to pick a new input sample from the training pool (without yet knowing the sample output) to improve the objective function.
4. Move the selected sample (including the true output) from the training pool into the training data.
5. Return to the first step and repeat the process until the maximum number of training samples are used.

I used a maximum of 300 training samples in both regression and classification. For simulation I used a fixed population of 500 testing points with responses generated by the simulation models.

6.4 Results and Discussion

Overall the study results found active learning is a promising approach to reducing the number of playtests needed for design iteration toward a design goal. In both regression and classification settings active learning techniques improved over the random baseline model. For enemy parameter tuning (a regression problem), acquisition functions that balance exploration and exploitation (especially UCB) have the best performance. The regression problem targeted a specific player behavior, making methods that gradually tuned the game to induce that behavior effective. For control tuning (a classification problem) we

found acquisition functions that tolerate high variance (e.g. QBB and entropy) have strong performance. The classification problem targeted a player subjective response, introducing greater variability in outcomes that require the ability to overcome the potentially conflicting feedback. It may be that there is no single optimal control configuration for all players, which would yield conflicting results that require methods that mitigate variance to perform well.

No single acquisition function, objective function, or acquisition-objective function pair was optimal across cases and number of playtests. These results align with previous work in AL showing that many data-specific properties impact AL efficacy [170]. This highlights the importance of investigating active learning techniques appropriate to different design iteration goals.

Below I provide further details with an emphasis on how active learning impacted the number of playtests needed for the regression and classification settings. I present results both in terms of the overall performance of the models and the performance of models relative to the random baseline. Overall performance evaluations describe how well active learning is doing in absolute terms: e.g., having all players hit exactly six times in the regression setting or always predicting player preference in the classification setting. Comparative performance evaluations describe how much value active learning is contributing by intelligently choosing playtests. Any design model improves with more data and the question of concern is whether the data being given to the model is providing the greater gains than randomly adding data. These analyses show model performance at varying numbers of samples to show trends in model improvement as more data is gathered.

6.4.1 Regression

The regression study found active learning can efficiently tune game design parameters for desired player performance, even with few samples. Having a clear behavioral objective (being hit a number of times in the game) was likely a strong contributor. Any variability

Table 6.1: Regression Gaussian Process mean squared error comparison of acquisition functions—in simulation. Sample sizes indicate values averaged over a range of ±5 samples (for smoothing). Lower values indicate better performance.

acquisition function	65 samples	280 samples
random	0.00280	0.00107
variance	**0.00126**	0.00110
PI	0.02889	0.02137
EI	0.00151	**0.00103**
UCB	**0.00131**	0.00116

in the data was primarily dependent on differences in player skill and prior experience with the genre—the earlier filtering of players with less than 10 waves of play likely removed players facing technical problems or who had little motivation to attempt to succeed at the task. Upper confidence bounds (UCB) was most effective at improving the performance of the Gaussian Process (GP) regression model (Table 6.2).[3] Upper confidence bounds balances exploration and exploitation over time by starting with an emphasis on exploration and gradually shifting to exploit effective parameter choices.

In simulation all acquisition functions performed well except for probability of improvement (PI). Figure 6.3 shows overall mean squared error values (higher is worse performance) for model predictions when trained with different acquisition functions, illustrating the better performance of most acquisition functions over random sampling (top, black line). Table 6.1 provides values at the highlighted regions. Note that PI is not shown as it distorts the graph scale. PI is a pure exploitation strategy it focuses on playtesting parameter settings that are highly certain. Because tuning used three parameters in a fine-grained space it is easy to find bad parameters and waste samples attempting to improve on a globally poor local optima. That is, PI would often find a relatively poor design and waste effort attempting to polish that design rather than find alternatives.

Roughly 50-70 playtests were needed to tune the three parameters related to player performance against enemies when using upper confidence bounds (UCB), expected im-

[3] All acquisition functions yielded significant improvements over the random baseline in all reported results tables.

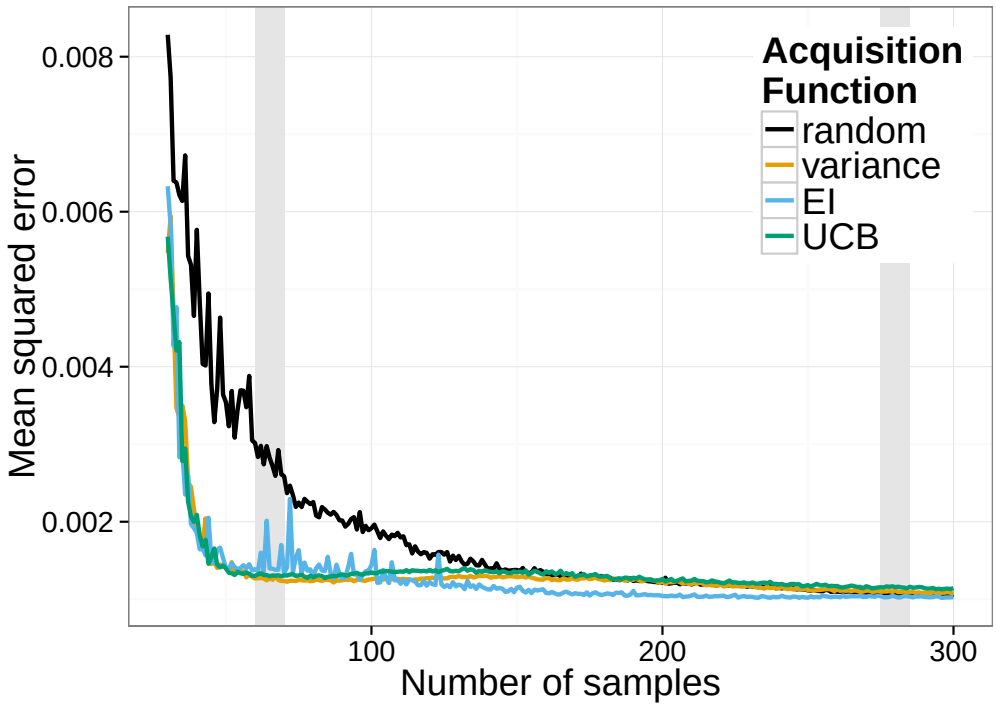

Figure 6.3: GP performance using different acquisition functions—in simulation. Shows MSE with an increasing pool of AL-selected training samples. Lower values indicate better performance. Bands indicate values that were averaged to produce Table 6.1.

provement (EI), or variance; random sampling required 175 playtests for comparable performance. More playtests marginally improved AL performance, though with diminishing returns. Figure 6.4 shows how much different acquisition functions reduced mean squared error compared to the baseline random sampling approach (larger values indicate greater improvements), demonstrating early large improvements. Improvements decrease with more samples as the amount of data gathered converges to greater coverage of the design space, diminishing the need for smart sampling. These results demonstrate the potential for active learning to improve simulation-based game generation systems, particularly those with expensive playtest simulations that would cost more computational effort than the relatively cheap GP-UCB model.

On human data the acquisition functions that did not balance exploration and exploitation had worse performance when limited to few samples or given many samples. Fig-

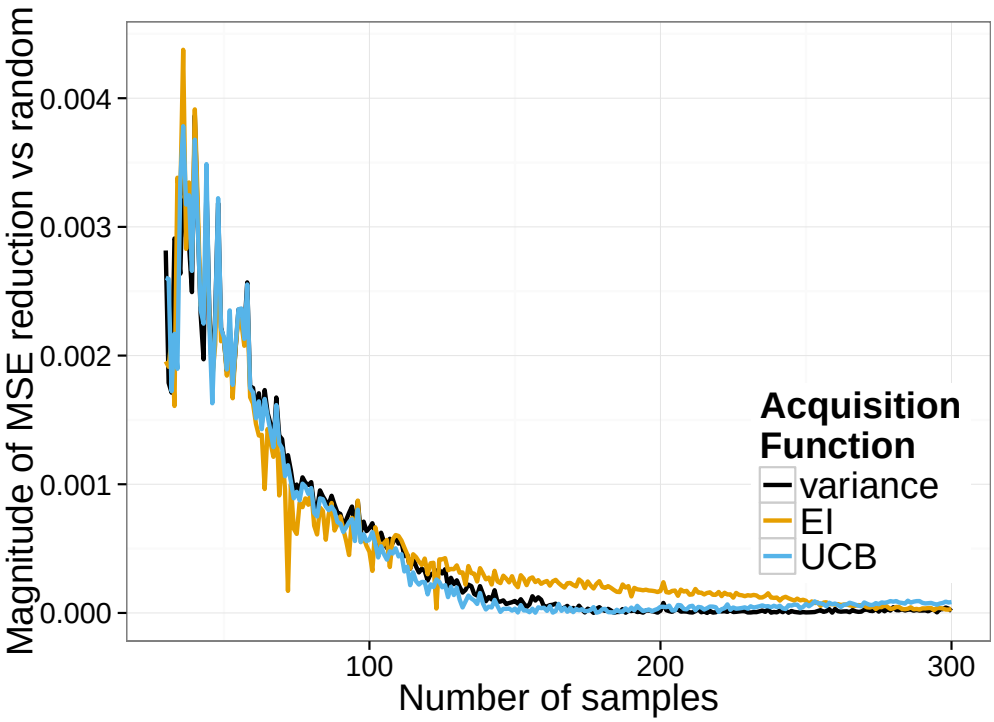

Figure 6.4: GP performance improvement over random sampling using different acquisition functions—in simulation. Shows amount of MSE reduction with an increasing pool of AL-selected training samples. Larger values indicate better performance.

ure 6.5 shows overall mean squared error values (higher is worse performance) for model predictions when trained with different acquisition functions, illustrating the better performance of all acquisition functions over random sampling (top, dark grey line); Table 6.2 provides values at selected regions. The two acquisition functions that balance exploration and exploitation—UCB and EI—show the best performance.

Figure 6.6 shows how much different acquisition functions reduced mean squared error compared to the baseline random sampling approach (larger values indicate greater improvements). From these figures it is clear active learning is particularly advantageous at small sample sizes, though most methods show improvements up to the maximum number of samples used. As in simulation, PI shows poor performance, barely improving over the baseline model, which is a fixed value of 0 improvement. All acquisition functions trend downward: this simply indicates that as the amount of data collected converges to

the full data set the value of intelligently sampling decreases. That is, eventually more data outweighs intelligently sampling in this task.

Variance performed relatively better with many samples, explained by the need to explore heavily before having a good enough design model. When tuning many parameters at once it is easy to find many sets of uncertain (but bad) parameters, leading to poor performance with few samples. Over time EI gradually worsened while UCB and variance maintained better performance. As more samples are gathered UCB reduces exploration while EI eventually begins to make poor playtest choices. Approximately 70 samples were needed to train the successful AL methods for the largest peak performance improvements; random sampling never achieved this level of performance on our data set (Table 6.2). Overall this clearly demonstrates active learning can enhance playtesting efficiency by reducing the samples needed to match a randomized batch approach (A/B testing). This gain is perhaps beyond what would happen through simply A/B testing and collecting data: UCB and variance achieved asymptotically higher performance than random sampling.

The regression experiments show the power of active learning to reduce the amount of playtesting required and better achieve design goals. UCB's balance of exploration and exploitation had the greatest efficacy and suggests a gradual refinement design process is optimal. These results make a strong case for active learning applied to optimizing low-level in-game behaviors, such as difficulty in terms of in-game performance. Applying this approach to the ongoing development of a game with hundreds of parameters—for example, tuning over 100 characters with many different abilities and attributes in a multiplayer online battle arena game like *League of Legends* [70] or *Defense of the Ancients 2* [43]—stands to provide enormous benefit to reducing or removing the need for ongoing human intervention in tuning live games.

Table 6.2: Regression Gaussian Process mean squared error comparison of acquisition functions—with humans. Sample sizes indicate values averaged over a range of ±5 samples (for smoothing). Lower values indicate better performance.

acquisition function	65 samples	280 samples
random	268	239
variance	233	**228**
PI	255	236
EI	**210**	242
UCB	**203**	**224**

6.4.2 Classification

The classification study found active learning can improves models of subjective player preferences (classifiers) with both probabilistic and non-probabilistic acquisition functions. Compared to the behavior tuning task for regression, the preference optimization task was more challenging, evidenced by proportionately smaller improvements over the baseline model. The inherent subjectivity of control preferences likely contributed to this challenge, along with the potential for multiple distinct optimal configurations that differ by players. Note that unlike a game personalization or adaptation task, the design model here is for a single design for all players. Methods that tolerate high variance—entropy, query-by-bagging (QBB) using vote or probability, and expected error reduction—have the strongest performance (Table 6.4). These acquisition functions succeed by overcoming the noise inherent in human playtest data, particularly when using few playtests. Our results show active learning is effective even with more complex data and can improve a variety of baseline classification design models: Gaussian Processes (GPs), Kernal Support Vector Machines (KSVMs), and optimized neural network structures and weights (NE).

In simulation entropy and QBB vote showed strong performance gains across all three objective functions (Figure 6.7, Table 6.3). Previous work has found QBB models are best for high variance data sets [170] and entropy sampling is also effective for high-variance situations. The complexity of our simulation model made it a highly variable function to optimize and demonstrates the value of acquisition functions that are effective against high

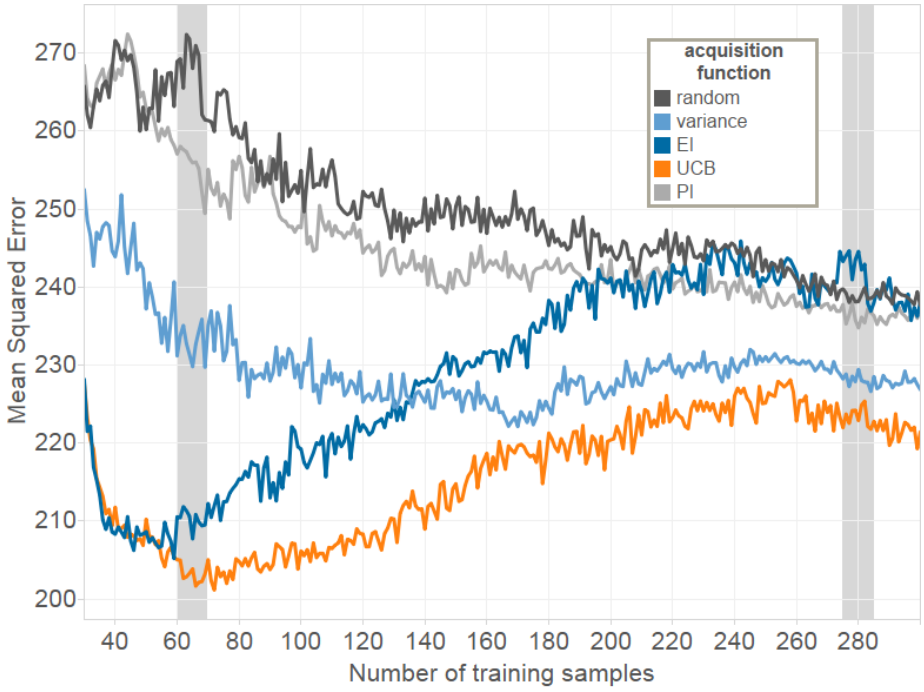

Figure 6.5: GP performance using different acquisition functions—with humans. Shows MSE with an increasing pool of AL-selected training samples. Lower values indicate better performance. Bands indicate values that were averaged to produce Table 6.2.

variance. Among objective functions NE showed the best performance; GPs had strong early performance before plateauing to similar performance as KSVMs. Researchers have also found NE effective for preference learning tasks in other game design contexts [182, 240].

Figure 6.8 shows how much different acquisition functions increased F1 scores compared to a baseline random sampling approach using the same classifier (larger values indicate greater improvements). Improvements decrease with more samples as the amount of data gathered converges to greater coverage of the design space, diminishing the need for smart sampling. These results reinforce the potential for active learning to improve simulation-based game generation systems.

On human data entropy, QBB vote and probability, and error reduction all improved classifier performance (as F1 score) over random sampling. Figure 6.9 shows overall F1 scores (higher is better performance) for the best performing acquisition functions for each

Figure 6.6: GP performance improvement over random sampling using different acquisition functions—with humans. Shows amount of MSE reduction with an increasing pool of AL-selected training samples. Larger values indicate better performance.

design model; Table 6.4 provides values at selected regions. GPs had the strongest performance when using random input data as an acquisition function. Compared to this baseline, pairing a GP with QBB probability yielded the best performance with few samples; pairing a KSVM with QBB prob yielded the best performance with many samples. NE performed worse than the GP model paired with randomized sampling.

Figure 6.10 shows how much different acquisition functions increased F1 scores compared to a baseline random sampling approach using the same classifier (larger values indicate greater improvements). All classifiers improved with some acquisition function, with KSVMs benefiting most, followed by NE. These figures illustrate active learning can provide substantial gains with few samples and maintain an improvement over random sampling up to the maximum number of samples used in this study.

Comparing the acquisition functions, QBB methods (especially vote) were effective at both few and many samples. Entropy was only effective with few samples while error

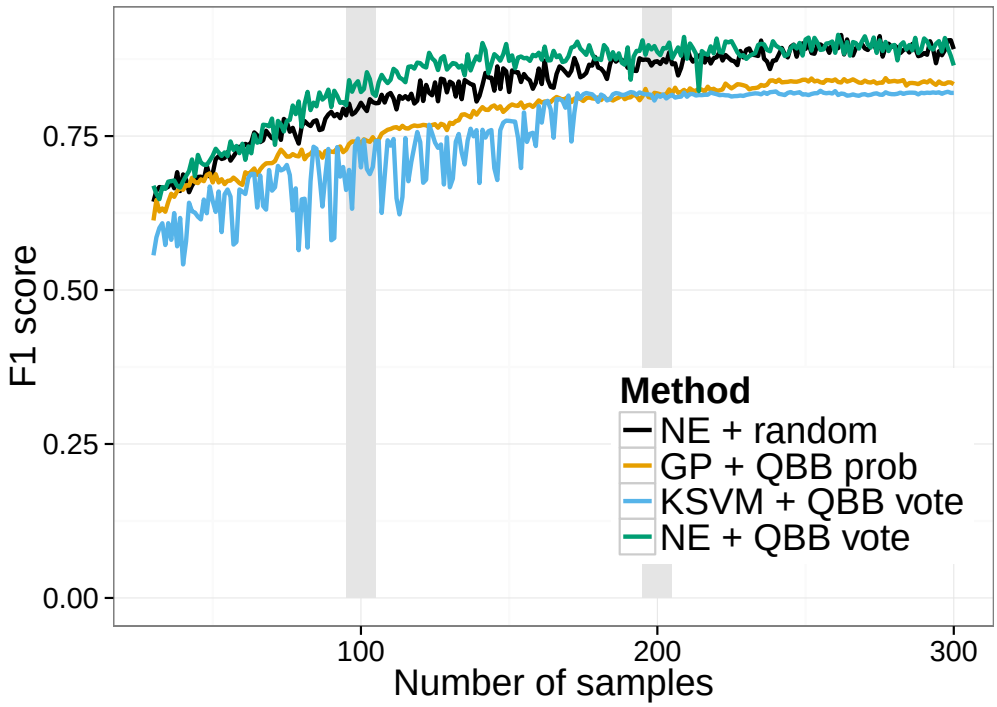

Figure 6.7: Classification performance with different combinations of classifiers and acquisition functions—in simulation. Higher values indicate better performance. Shows F1 score with an increasing pool of AL-selected training samples. Bands indicate values that were averaged to produce Table 6.3. Only the best-performing acquisition functions for each classifier are shown for clarity.

reduction was most effective with more samples. Expected error reduction must predict future outcomes and thus requires more initial data before becoming effective. Variance reduction had poor performance. As with the variance acquisition function for regression, a large number of possible parameters causes difficulty in effectively reducing variability in responses. Overall active learning can yield improvements even with noisy, subjective data, but these gains are likely mitigated by differences among players or shifting player preferences (e.g., coming to prefer more sensitive controls with experience in the game).

Comparing the design models, we found GPs had the best baseline performance (with random sampling), followed by NE and then KSVMs. Overall GPs with QBB probability or expected error reduction did best, followed by KSVMs with either QBB method and then NEs using QBB vote. Using active learning provided the largest performance

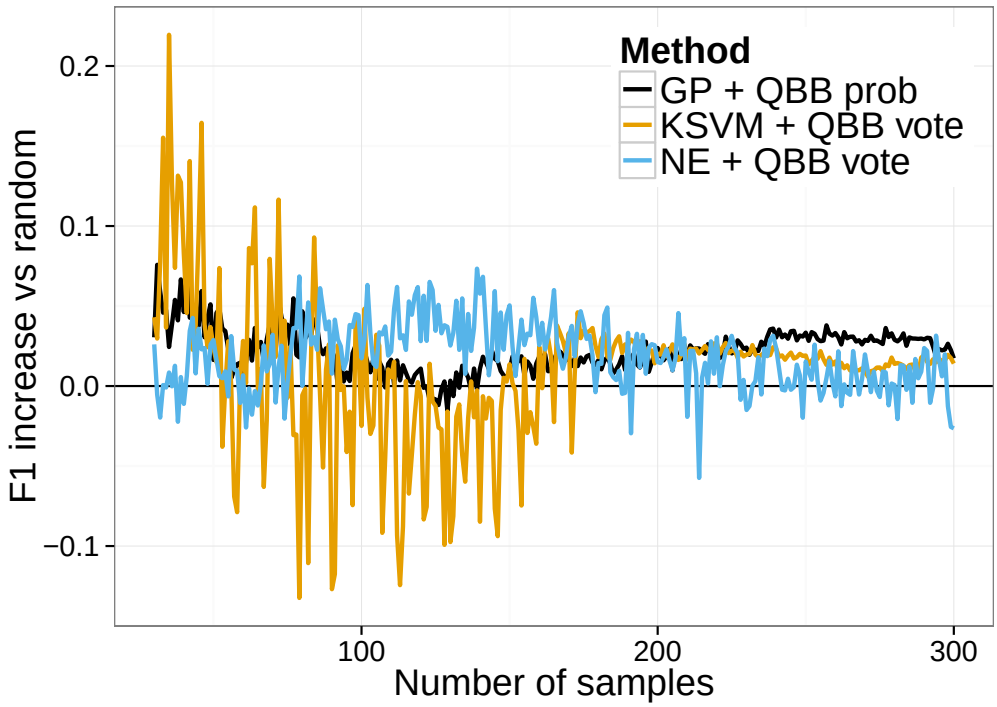

Figure 6.8: Classification performance improvement over random sampling with different combinations of classifiers and acquisition functions—in simulation. Higher values indicate better performance. Shows gains in F1 score with an increasing pool of AL-selected training samples. Only the best-performing acquisition functions for each classifier are shown for clarity.

boost for KSVMs, though GPs and NE also benefited. The performance trends of GPs and KSVMs mirrors that of the simulation results, where GPs perform well with few samples and KSVMs perform well with more samples. NE showed (compared to other classifiers) worse performance on human data than simulation: this may be due to greater noise in the human data or lack of a large enough feature space to be effective. GPs have traditionally been applied to Bayesian optimization tasks where there is relatively little data and few features, but rapid learning is desired. By contrast NE is commonly applied to tasks with more data and larger feature spaces. GPs may be better suited to smoothing over noise in the human data compared to NE. It may be that NE is better suited to design optimization when many parameters are being simultaneously varied and more data is available.

The classification study demonstrates active learning can reduce the number of playtests

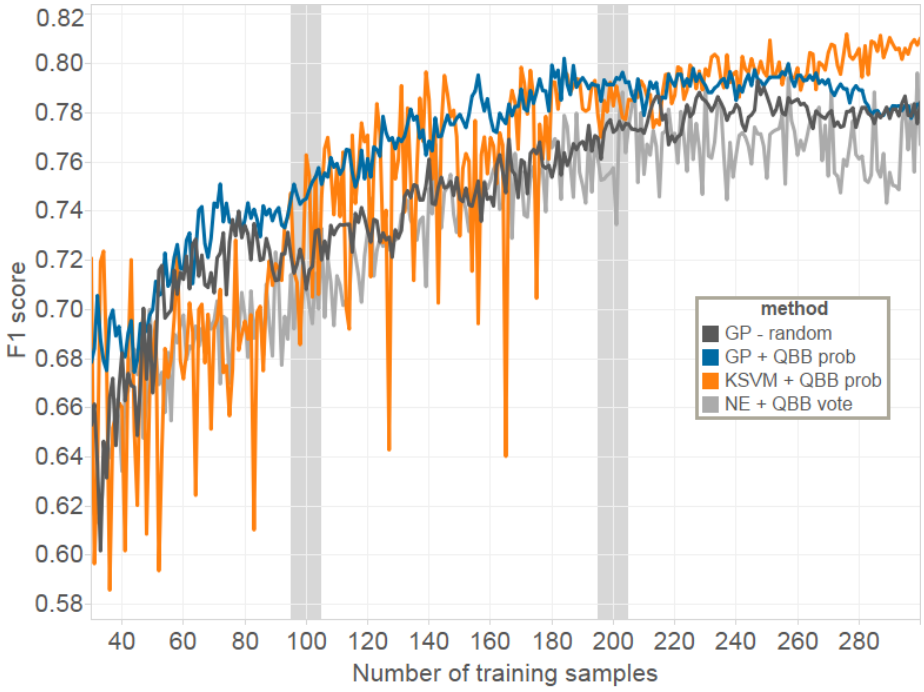

Figure 6.9: Classification performance with different combinations of classifiers and acquisition functions—with humans. Higher values indicate better performance. Shows F1 score with an increasing pool of AL-selected training samples. Bands indicate values that were averaged to produce Table 6.4. Only the best-performing acquisition functions for each classifier are shown for clarity.

needed even for subjective features of a design such as control settings. Reducing playtest costs requires acquisition functions (e.g. entropy, QBB, and error reduction) that mitigate the noise inherent in preference response data. Active learning always improved over random sampling across different design model approaches, though the best acquisition functions varied. These results make a strong case for considering active learning when optimizing a design toward player preference data and may apply to other discrete choice settings in design (e.g., branching specialization choices in a role-playing game).

[[tense!: experiments should all be past (or change to present)]]

6.5 Limitations

The active learning approach to design iteration presented here has a number of limitations that point to promising avenues for further research. The shoot-‘em-up game domain was

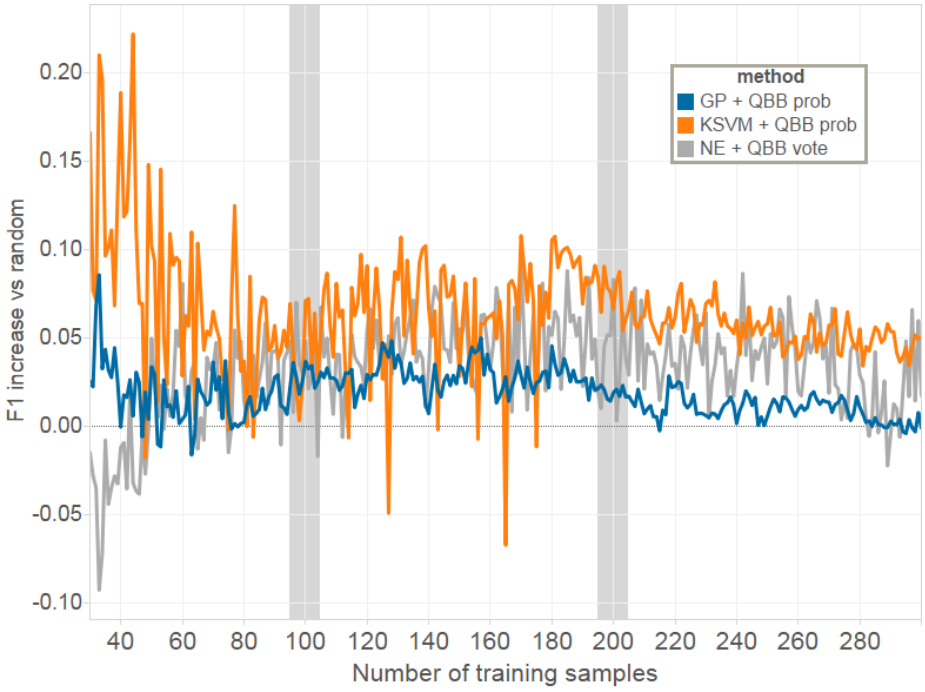

Figure 6.10: Classification performance improvement over random sampling with different combinations of classifiers and acquisition functions—with humans. Higher values indicate better performance. Shows gains in F1 score with an increasing pool of AL-selected training samples. Only the best-performing acquisition functions for each classifier are shown for clarity.

used to minimize the effects of player learning and long-term strategizing while maximizing the data gathered per playtest. This approach is likely effective for a large number of reflex-based arcade games, such as *Frogger* or *Flappy Bird* [95, 96]. But many design parameters influence complex systems with interconnected consequences (e.g., economic simulations in a game) or bear on player strategic choices (e.g., unit parameters in a strategy game). In these situations alternative strategies may be needed to handle credit assignment to specific design parameters in these large feature spaces.

The design parameter model here used a flat set of parameters with no explicit dependencies. That is, no choice of parameters would invalidate the use of other parameters. In tasks where parameters have structure—e.g., branching paths of choices in a narrative where eliminating one choice would remove downstream choices—alternative methods will be needed to handle the composition of modular elements.

Table 6.3: Classification acquisition-objective function F1 score comparison—in simulation. Sample sizes indicate values averaged over a range of ±5 samples (for smoothing). Higher values indicate better performance.

acquisition function	100 samples			200 samples		
	GP	KSVM	NE	GP	KSVM	NE
random	0.725	0.711	0.797	0.800	0.794	0.872
entropy	**0.776**	0.721	N/A	**0.815**	0.774	N/A
QBB vote	**0.784**	0.704	**0.833**	**0.815**	**0.815**	**0.891**
QBB prob	0.742	0.642	**0.807**	**0.819**	0.752	**0.890**
error red	0.716	**0.759**	N/A	0.799	**0.803**	N/A
var red	0.716	**0.735**	N/A	0.796	**0.810**	N/A

Table 6.4: Classification acquisition-objective function F1 score comparison—with humans. Sample sizes indicate values averaged over a range of ±5 samples (for smoothing). Higher values indicate better performance.

acquisition function	100 samples			200 samples		
	GP	KSVM	NE	GP	KSVM	NE
random	0.720	0.684	0.673	0.773	0.709	0.718
entropy	**0.763**	**0.731**	N/A	0.763	0.751	N/A
QBB vote	**0.758**	**0.746**	**0.703**	0.780	**0.777**	**0.760**
QBB prob	0.749	0.724	N/A	**0.792**	**0.782**	N/A
error red	**0.761**	0.702	N/A	**0.795**	**0.772**	N/A
var red	0.660	0.667	N/A	0.725	0.723	N/A

The classes of design objectives covered in this study is a small subset of the space of potential objectives. Choosing or developing the appropriate metrics for design goals or acquisition functions to optimize toward these goals remains an open topic. How can a system optimize for players spreading their choices among alternatives (rather than converging to a single choice)? How can a design optimize for long-term player outcomes, rather than immediate feedback? How can a system optimize the choice of playtests to maximize learning about the full design space with as few playtests as possible? How can optimization account for design constraints in terms of dependencies between parameters? Addressing these and many other topics will broaden the cases where machines can automate or support human design iteration practices.

6.6 Potential Impact

Game design research has the potential to change the way games are made and the experiences available to game players. In this section I briefly discuss how the methods for game design iteration in this chapter might influence game designers and players.

6.6.1 Game Designers

At its core, automated parameter tuning enables designers to offload fine-tuning of a design on a system. In practice this can alter design practices to focus on defining what metrics of behavior capture design goals. Alternatively, this may also lead to new crowdsourced design practices where player feedback is used to guide systems toward what players subjectively perceive as enjoyable. As a system tunes parameters to this notion of 'enjoyment' designs would learn what set(s) of parameters capture this experience, in turn potentially informing large-scale design changes. This future design practice will gradually train designers to think in terms of systems that are tuned when released, rather than concretely defined in the abstract without player feedback (as objective behavior or subjective response).

Game designers will also stand to benefit when releasing complex games that require ongoing maintenance and tuning. Currently these ever-evolving games require designers to continually adjust parameters for maps or characters in games as players change their strategies. The patch notes of games like *Starcraft* or *World of Warcraft* are a testament to the never-ending need for ongoing game tuning. With automated tuning designers would be relieved of burden, allowing them to let a game continue to evolve on its own as a system continually rebalances systems. Instead of continually optimizing a subset of design parameters, a designer's job would instead be to appropriately define parameters and their ranges of acceptable variation, moving tuning to a more abstract process of defining what should be fixed or variable at any time for a given design. In systems with sufficiently large

numbers of systems a single change can have rippling consequences on coupled game systems. Automated parameter tuning may be the only way for designers to focus their attention on the most challenging systems to alter, leaving an automated system to adjust related systems to dramatically reduce the workload needed to fully tune a game.

6.6.2 Players

Efficient design optimization presents players the opportunity for novel forms of dynamic difficulty adjustment that continually tune game parameters to challenge players in different ways [93]. When systems can intelligently choose new designs to optimize learning about a player they can provide challenges that are novel to players, rather than simply providing new content that is expected to not provide much difference to player behavior [221]. Different design objectives would allow the system to tune the game toward different goals: for example, increasing or decreasing rates of failure or the length of play sessions. Alternatively, different design parameters would allow the system to choose ways of altering player experience, altering controls in one case or avatar power in another case. Generalizing, a class of game built around probing player capabilities becomes possible, with an intelligent "AI Director" choosing game variants that provide players with game variants selected to optimize player performance toward some system objective (in the exploitation case) or optimize learning about how the player performs (in the exploration case). This design paradigm would shift games from providing abstract notions of fixed 'achievements' for all players, to individualized game designs that force players to achieve objectives in different ways.

6.7 Summary

In this chapter I cast design iteration as an active learning process and demonstrated how active learning can apply to two classes of design goals. Broadly speaking design goals concern regression—optimizing game parameters for a continuous game metric—and classification—

optimizing game parameters for a discrete game metric. As representative examples I considered the regression problem of optimizing for a desired difficulty in term of number of times a player is hit in a wave of a shoot-'em-up game. For classification I considered the classification problem of learning subjective player control preferences in the same game. Active learning provides a large number of potential acquisition functions to guide design iteration choices. In both simulation and human studies I show how these acquisition functions can reduce the amount of data needed to achieve a given level of model performance and show that active learning may even yield better models than achieved by a random sampling baseline. Together these results make a strong case for the value of active learning to improve iteration in game design.

For game generation systems that use simulations to evaluate content there is a clear value to using active learning to improve the efficiency of the generation process, potentially uncovering better results. Most automated search techniques used for procedural content generation or design optimization can benefit from an active learning wrapper to guide the learned model toward the most valuable parts of the search space [40]. This has the potential to generally improve the efficiency of these algorithms, at least in cases where model training is sufficiently cheap and data collection (behavior sampling) is sufficiently costly.

For human design iteration practices there is the potential for active learning to guide design to test games more effectively than the standard A/B testing approach. An AL model could inform designers about the potential value to testing different planned design variants by providing estimates on how player behavior would be altered in terms of design goals, along with an estimate of how uncertain the model is about those outcomes. Going further, the AL model could guide automated iterations of a game's design in cases where the balance of a game is constantly shifting. In these cases AL offers the benefit of minimizing the amount of change players experience, creating a more seamless experience in the game. For example, many competitive games are designed with the intent that players

may choose among equally viable alternative character configurations, such as in fighting game avatars, racing game cars, or shooter game weapons. However, these games often include a variety of areas to compete in (fighting arenas, racing tracks, or shooter maps), with changes to character traits often interacting with the tuning of parameters related to these areas. In these scenarios, designers can benefit from an automated system that tunes area parameters to maintain balance, even in the face of changes to characters. More generally, most online games where players compete or cooperate face the challenge of constant upkeep to maintain balance as player strategies evolve: in all these cases an active learning approach can be used to efficiently change the game to minimize the number of variants human players would experience. Together, these examples illustrate the potential for AL techniques to improve game development practices during early iteration through ongoing updating and refinement of a live game.

CHAPTER 7
CONCLUSIONS

The central statement of this thesis is:

> Explicitly modeling the actions in games as planning operators allows an intelligent system to reason about how actions and action sequences affect gameplay and to create new mechanics. An intelligent system facilitates human iterative game design by learning design knowledge about gameplay and reducing the number of design iterations needed during playtesting a game to achieve a design goal.

Over the chapters in this thesis I demonstrated a series of systems that provide general methods for modeling components of an iterative game design process 7.1. These systems addressed four core components of the iteration cycle:

1. Generating games in a domain-agnostic manner (chapter 3)
2. Generating example behaviors from games to explore the ways players can play the game (chapter 4)
3. Analyzing game designs in terms of how well the behaviors they afford meet design goals for the game (chapter 5)
4. Iterating on the game design to choose the next candidate design to evaluate (chapter 6)

These systems each provide general tools for automating game design iteration, emphasizing general algorithms over specific techniques tailored to an individual game design domain. Below I discuss each of the major systems in terms of their contributions and future areas for development.

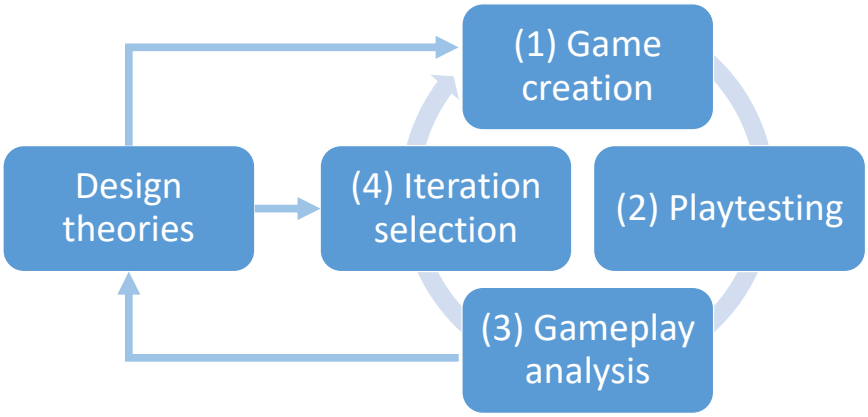

Figure 7.1: Iterative design process (for games) schematic.

7.1 Game Generation

Game generation took a mechanic-centric approach to enable generation agnostic to a specific game domain, using AI planning as the guiding model to ground generation. The system adapted traditional AI planning representations to provide a general and reusable representation for functional elements of discrete, deterministic, turn-based games games. This foundation allowed the generative system to create not only the mechanics in the game, but also the levels of the game, progressions of the levels, and controls players used in the game. Using planning as the underlying technology for agent behavior enabled the system to generate a wide variety of games, unifying multiple aspects of game content and mechanic generation within a single representational framework.

A shared foundational representation for game systems opens the scope for the creative autonomy of generative systems. Procedural content generators now can consider how pieces of content interact, allowing for broader variation in the types of games systems can generate. Generative systems stemming from this framework will be able to reason about

other games within this broad design space, allowing hand-off of creative products between systems. An open question remains as to how these systems can potentially interface with other existing generative systems, using their reasoning capabilities to improve artifacts not initially created in their realm [40]. Doing so will further extend the already wide range of novel creative artifacts encompassed by these models.

The mechanic generation and design iteration systems together provide ways to generate and refine the controls players have in games. This work begins to address how to give players control in games, tackling *how* players make choices in games, rather than *what* those choices are. As design is often about the 'feel' of a game, these advances hold the potential to give computational creators better control over core aspects of player experience [211]. Future work can build on these efforts to fully generate and test a variety of alternative control schemes, moving from choosing button mappings or control tuning to choosing the appropriate control framework (e.g., joystick vs mouse) for a game.

7.2 Behavior Sampling

Behavior sampling is the problem of creating examples of player behavior for a given game design. A stochastic planning algorithm—Monte Carlo Tree Search (MCTS)—was presented as a general solution to action planning in the class of discrete, turn-based games considered above. The key feature of MCTS used was the ability to parameterize agent strength in terms of rollouts used, allowing the algorithm to serve as a general tool for proxying varying player capabilities to plan ahead in turn-based games.

Many game design goals center on the choices player make in a game, requiring a framework for measuring the actions available in a game, rather than the content available for consumption. Four categories of metrics were presented that focus on the space of actions in a game, addressing a range of levels of abstraction. Summary metrics measure aggregate properties of the space of choices in a game. Atom metrics measure the use of single actions (or the opportunity to take action) in a game. Chains measure sequences

of actions in a game where players link together choices they make or respond to choices made by others. Action spaces measure the trajectory of choices over a game to understand how the space of choices evolves over the course of a game.

Combining action metrics with MCTS behavior samples from agents of varying skill allows for evaluation of how well a design differentiates among agents of different skill. Using this approach showed *Scrabble* effectively differentiates between agents of varying skill, while an intentionally simplified *Hearthstone* variant did not. Developing metrics centered around the actions in a game opens the way to shift the ways generative systems function, moving from an emphasis on what content players visit to how players act.

Existing metrics for evaluating generative systems typically emphasize features of the content produced by a system [26, 125, 179, 198, 207]. Action metrics provide a new set of criteria to evaluate the space of choices available in a game. Comparing generated content in terms of how well that content differentiates agents of varying skill further improves the ways generative methods can be assessed. Combining existing approaches with these new metrics allows generative methods to now better shape the choices players make at varying levels of skill. Extending this approach to more aspects of modeling audiences in the future will provide generative systems more ways to search for creative content of value to people. In the future these systems could also use agents tuned to have human-like behavior as a way to better proxy expected human reactions.

7.3 Gameplay Analysis

With the tools to generate and evaluate individual game design instances, design iteration becomes the process of navigating the space of possible design variants to find those most suited to design goals. An initial naive approach to choosing a design iteration generated a large space of design variants by varying parameters of elements of the design (here cards in a card battling game) and then sampling behaviors from each variant using MCTS. Picking the best design variant from this space amounted to: (1) evaluating the desired action metric

on behavior samples from every design variant and (2) selecting the variant with values nearest to the design goal. With a space of design variants and their associated action metric evaluations available, it was also possible to learn models to predict how changes in the design variant would alter action metric outcomes. This modeling provides a form of generalization around how design choices influence the space of play in a game. In the future these models could be made into systematic and general design knowledge, similar to the generality sought by the game generation system. Providing portable, general design knowledge has the potential to enable systems to gradually explore a massive design space, using accrued knowledge to find the areas of greatest value to explore. Design knowledge could also help new generative systems, providing heuristic information to guide initial generative efforts and improving the initial quality of generated artifacts.

Coupling models of how designs influence player behavior with the AL design iteration models allows generative systems to more efficiently generate content. Using these models, generative systems can effectively consider a broader range of generated artifacts by saving effort from generating and evaluating low quality products. In the future this approach could be integrated into generic tools for supporting other generative methods. A general wrapper could provide these efficient search capabilities to systems built on other frameworks, serving as a general way to amplify the creative efforts of other systems.

7.4 Design Iteration

Generating a space of designs and sampling behaviors from each design can be prohibitively costly. As an alternative, active learning (AL) techniques enable a system to quantify the trade off between expected improvement to a design and expected learning about a design space. Using active learning enables a system to efficiently search the space of design variants when optimizing for design goals including both goals for player objective performance and subjective preference. Active learning can improve a wide variety of baseline models for searching for a desired game among a space of game design variants, serving

as a general tool to improve the efficiency of automated generation.

The AL design iteration model highlights the potential for reconsidering how creative processes are modeled. Using AL for efficient iteration is a way to improve a wide variety of creative systems, as most systems implicitly or explicitly evaluate candidate alternative artifacts before creating a final product. This underscores the need for more general tools to augment creative systems: what other aspects of creative processes remain unexamined? For example, the systems in this thesis depend on explicit design goals. Are there common tools or frameworks for creating design goals? Evaluating whether goals are possible and when to pursue them (or not)? Developing these kinds of models will be crucial to continuing to extend the role computational creators play in creative processes.

Summarizing, the work in this thesis made a number of additional contributions to techniques for game design automation:

- A planning representation for domain-agnostic mechanic generation
- Monte Carlo Tree Search for general agent modeling with differing skill to plan
- Four categories of action metrics to quantify the space of choices afforded by a game
- Learning predictive models for how different design features across a design space predict different gameplay behavior outcomes
- Active learning algorithms to enable efficient selection of design variants to test

Each of these contributions lays groundwork for further research into general techniques for automated game generation and design iteration.

7.5 Computational Creativity

The work in this thesis sheds light on key questions around the knowledge, processes, and limitations of computationally creative systems. Creative systems will need to represent and gather knowledge about their audiences (both intended and actual) to create artifacts

of that are seen as valuable. MCTS allowed one form of audience representation, capturing the notion of skill in audiences and how that shapes the way an audience interacts with a game. The AL system also used information from live audience reactions to refine a game design, feeding back information about how individuals respond to the artifact to shape its adjustment to best meet the design goals given to the system. Creative systems will also need ways to represent their goals for created artifacts: the game generation system used information on required failure and victory conditions while the AL system used goals in terms of player behavior. The full spectrum of design goals from static—related to the form of the artifact itself—to dynamic goals—related to the behavior induced by the artifact—are necessary for creative systems to create products for an audience [81]. As creative systems assume control over larger parts of the creative process the representation of and reasoning about these goals will become increasingly important.

Predicting how artifacts will influence audience reactions is a core part of the iterative creative process modeled in this thesis. The AL system used learned models of anticipated design quality to guide choices of future design iterations. The gameplay analysis work showed the potential for learning models to predict how audience behavior is influenced by design features. Predicting audience behavior plays a key part in enabling creative systems to intelligently choose how to alter a creative artifact and may play a role in human creative processes as well. Integrating predictive modeling into computational creators will be important as a means of enabling these systems to create artifacts tailored to their intended audiences.

Computational limitations on creative processes played a key role in driving the design of the creative systems in this thesis. Behavior sampling using MCTS was used to address computational limitations on modeling the space of all possible ways to play complex games. The full generation of a design space for evaluation in the gameplay analysis chapter illustrated the need for efficient design space navigation realized by the AL system. In these and many other cases, creative systems are inherently limited by combinatorial

explosions in the space of design choices involved in a creative artifact. Humans certainly face similar limitations in their ability to consider a broad range of creative alternatives—the systems in this thesis provide tools for computational creators to begin to overcome these limitations. In the future the need for general ways to efficiently sample from spaces of artifacts and behaviors will be crucial to expanding the level of control computational creators have over the artifacts they create.

The systems in this thesis also have relevance to human creative practices. Game design iteration has many mundane aspects that these systems help automate and support. MCTS provides a way to automate the gathering of playtest samples for players of varying skill, in turn providing designers with an algorithm for gathering initial proxies for player behaviors in a game at different levels of skill. The AL system took an alternative approach to providing ways to automate the process of tweaking design variants even when humans are in the loop. In both cases these systems demonstrate the potential for computational systems to reduce the need for human creators to perform rote tasks, freeing time and attention for more complex parts of the creative process.

New kinds of creative artifacts are now also possible with these systems. Humans can use the AL system model to dynamically rebalance a live game, using human data to continually tune features of the game without manual intervention. Particularly in cases where games have many interacting systems an automated balancing system can provide value by minimizing the unintended negative consequences of design changes. Rather than treat a created game as a static artifact, these systems can enable an ever-changing game that maintains desired design goals in response to shifts in player behavior (due to new player behaviors, changes in the demographics of players, and so on). While existing techniques for content optimization can make these changes, they do so while ignoring the efficiency of their changes. Minimizing the number of changes enables creative systems to be minimize the changes felt by players, creating a more seamless experience.

Beyond the artifacts created, creative processes can benefit from the models in this

thesis as well. The action metrics provide new lenses on how players interact with a game at the level of the *choices* they make. Comparing the choices of players at different skill levels provides a new way to assess how well a design supports a space of alternatives (or not), giving game creators new ways to consider whether a design is meeting their goals. The gameplay analysis system learned models to predict how player behavior would (or would not) be influenced by design changes. Creators stand to benefit from using these models to understand the designs they have and can use the model predictions to guide design choices to consider. Having on-demand predictions for the effects of a design change can greatly benefit creators in finding the right design for their design goals.

Combined, the systems in this thesis provide tools to augment the craft of expert creators. To date, these systems all require creators sufficiently comfortable with programming to express their design goals in some computational form (whether programmatic definitions of success and failure or metrics for objective behavior). Many of the systems developed, however, do not require direct human intervention: automated parameter tuning can readily be defined by programmer-designers. To bring these systems to bear on challenges faced by amateur creators will require further work to refine the paradigms for expressing design spaces and goals to these systems.

7.6 Future Work

The systems in this thesis address core components of automated game design, but are not integrated into an end-to-end pipeline and make strong assumptions about the kinds of games to generate. While the systems each provide general tools for components of the iterative game design process they do not function together in a single system. Developing such a general system will require further integration of the output of different systems into one another.

Game generation research as a field regularly faces challenges in integrating components of large systems [37, 40]. Addressing integration requires shared platforms and

representations that are relatively agnostic to the systems interfacing to these infrastructures. While shared platforms come with risks of narrowing the scope of research done in a field, a lack of shared platforms can obstruct incremental improvement. In the future, game generation systems will need ways to generate a wide array of content with a shared underlying representation for knowledge of that content. Providing an extensible representation for such knowledge has the potential to greatly accelerate research into game generation in specific and computational creativity in general by allowing systems to work together on a shared artifact, rather than independently generate similar (but representationally distinct) artifacts.

The systems in this thesis consider the question of game design in terms of a development process where design goals are given and the task of design is to define the actions for players to take. Game design, however, involves a host of related aesthetic decisions about a game, ranging from the writing (if any) used, to how the user interface is shaped, to the color palette for a game. An important open question for future work is to enable systems to reason on these aspects of game design as well. This content will need a shared representational platform to allow a system to integrate reasoning on these choices with other choices such as the controls in the game or mechanics available. Human creators rarely build a game "from scratch" using a single tool: art assets are created with art creation programs, levels with level editors, music with sound creators, code in a programming language, and so on. Yet humans can reason about how these different content choices relate. This begs the question: how can a computational creator represent these diverse pieces of information in a way that affords general reasoning on how to combine content toward a broader aesthetic goal?

Questions of combining reasoning all pieces of game content highlight a key open problem from this thesis: reasoning on design goals. How should a design goal be evaluated? The systems in this thesis considered goals for whether players can reach states in games and metrics on player behaviors, but this leaves open the question as to how to represent

the shared form of knowledge underlying these design goals. When and how should design goals change? Creative practitioners regularly change their goals for a design in response to audience reactions and learning about limitations of their chosen design space. Creative systems to date have largely overlooked the choice of what to design toward to begin with, leaving open the topic of how goals may evolve over time. This is particularly relevant when creative systems go beyond creating single artifacts to producing a corpus of results: a design goal may be deferred or limited for a current artifact to be later revisited in a future generative project. While these aspects of human creative practice are relatively mundane, automated creators are currently very limited in their abilities to represent and reason about the goals they pursue for creative products, in turn limiting the ways these products can be of value to their audiences.

Finally, the work in this thesis addresses a model of iterative game design practice. Game design, however, is not unified under a single creative practice. For example, game jam games often center on realizing a concept and aesthetic with relatively common choices of game mechanics. For game jams, the process of translating an aesthetic to mechanics is the main process for design [36, 37]. By contrast, "secret box" games focus on players experiencing an aesthetic with relatively few aesthetics and limited (if present) goals. For "secret box" games, the process of building an aesthetic that players enjoy is the main process for design [41]. These and many other types of game genres induce different human design practices, in turn leading to opportunities for research on different models for computational game creation. Contrasting the creative processes modeled by these systems can highlight commonalities among systems, identify new processes not addressed by existing work, and ultimately pave the way for systems that create new classes of artifacts through entirely novel creative processes. Ultimately, future extensions of automated iterative game design can further expand the ways we enable systems to create and provide shared methods for reuse across such systems.

Appendices

APPENDIX A

GAME GENERATION SYSTEM IMPLEMENTATION

The game generation system was implemented using Answer Set Programming (ASP) [8]—a form of declarative programming. Answer set solvers handle constraint satisfaction problems and ASP specifically allows for optimization among answers (valid combinations for the constraint solver). For this system I implemented the semantics for the domain definitions above and a planner in ASP. The constraint satisfaction problem then becomes finding a valid set of mechanics that meet design requirements such that the planner can meet playability requirements on given test game instances.

A.1 State Model

The state model defines ground predicates that will be used by the mechanic generation process to define transition model predicates and the planner to define state (Table A.1). For ASP all logical terms are statements ended with '`.`', conjunctions are specified using '`,`', and entailment is specified with '`:-`'. As syntactic sugar '`..`' indicates a range of values that are expanded into a set of individual facts: `op_range(player, xPos, -1..1).` becomes:

```
1 op_range(player, xPos, -1).
2 op_range(player, xPos, 0).
```

Entity(player)	`entity(player).`
Parameter(xPos)	`parameter(xPos).`
Has(player, xPos)	`has(player, xPos).`
AbsRange(player, xPos, [1,8])	`range(player, xPos, 1..8).`
RelRange(player, xPos, [-1,1])	`op_range(player, xPos, -1..1).`
Initial(xPos(player), 1)	`init((player, xPos, 1)).`

Table A.1: Examples of ASP code to implement domain entities.

`mech(M)`	Mechanic index `M`
`planop(Pop)`	Precondition or effect type `Pop` One of: `eq`, `neq`, `gt`, `lt`, `add`, or `set`)
`time_idx(T)`	Point in time `T`
`coord(C)`	Coordinate frame of reference `C` One of: `abs` or `rel`
`state(C, T, (E,P,V))`	A valid state element for the entity parameter value `(E, P, V)` at time `T` in coordinate frame `C`
`op(M, Pop, C, T, S)`	Precondition or effect `Pop` of mechanic `M` in coordinate frame `C` at time `T` defined by state `S`

Table A.2: Definitions for mechanics

```
op_range(player, xPos, 1).
```

The state model predicates all have direct translations into the ASP implementation as they simply define literals and facts that are used by systems to generate mechanics or check playability.

A.2 Mechanic Model

Mechanics are defined as a set of logical facts defining preconditions and effects that share an index (Table A.2). Any precondition or effect takes the general form: `op(M, Pop, C, T, S).` `M` defines a unique index for naming the mechanic to join together shared preconditions or effects. `Pop` defines the part of the mechanic being specified. Precondition may check for equality (`eq`), inequality (`neq`), a greater than (`gt`) or lesser than (`lt`) relationship. Effects simply alter state through relative addition (`add`) or setting a value (`set`). `C` defines the coordinate frame of reference to be absolute (`abs`) or relative (`rel`). `T` defines the time index. `state(C, T, (E,P,V))` is a predicate to define reusable chunks of game state, which define the value for an entity's parameter value (`(E,P,V)`) in a coordinate frame at a point in time. An individual `op(M, Pop, C, T, S)` predicate can define how a precondition should check game state, at which point in time, and relative to which coordinate frame. Alternatively, the predicate can define how an effect should update state at a point in time (where coordinate frame determines whether to alter a state

value or set a value). Mechanics are defined by one or more of these predicates sharing an index.

As an example mechanic, an RPG spell for $damage$ can test for player mana being greater than 0 and apply the effect of reducing the health of the enemy by one while also costing the player 1 mana:

```
op(1, gt, abs, 1, (player, mana, 0)).
op(1, add, rel, 1, (enemy, health, -1)).
op(1, add, rel, 1, (player, mana, -1)).
```

In this example `op(1, gt, rel, 1, (player, mana, 0))` checks for the game state of player mana at the time of action (time index 1) and compares this to the value of 0 (an absolute value). `op(1, add, rel, 1, (enemy, health, -1))` indicates the effect of taking enemy health at the time of action and adding (a relative change) the value of -1 to that health.

A.2.1 Mechanic Generation

The primary component of generating mechanics in ASP involves specifying a set of mechanic indices and allowing the choice of ground terms for the variables in those mechanics. To do so, we define the allowed ground values for each of the terms making up mechanics. We then define how to choose ground terms for the variables that make up a mechanic. The core component of mechanic generation is:

```
mech(1..nmech).
planop(set;add; eq;neq;lt;gt).
time_idx(1..time_max).

state(rel, T, (E,P,V)) :-
        coord(rel), time_idx(T), op_range(E,P,V).
state(abs, T, (E,P,V)) :-
```

```
8           coord(abs), time_idx(T), range(E,P,V).
9
10 0 { op(M, Pop, C, T, S) : state(C,T, S) } nop :-
11          mech(M), planop(Pop).
```

The first three lines define the elements of mechanics, taking as input the number of mechanics desired (`nmech`) and maximum game length (`time_max`). Note that '`;`' is used to enumerate sets of discrete facts while '`..`' enumerates ranges of integer values. The next two logical sentences derive all allowed state predicates for absolute or relative ranges of values (which are defined separately by the game domain). The state values are derived using entailment (`:-`) from conjunctions (`,`) of a coordinate frame, time, and relative or absolute range of allowed values. Note that ASP entailment is the reverse of standard logic syntax: derived literals appear on the left of the entailment symbol and the term being derived from appears on the the right.

The final sentence generates mechanics using ASP's syntax for a choice rule: a decision of how to choose ground values for variables. The outside braces and values define bounds on the count of predicates allowed between the left and right braces: here we allow between 0 and `nop` (an input maximum) of `op` predicates. The `:` within the braces indicates that `S` may be chosen to take any value defined by the state predicates—this ensures mechanics may only manipulate valid game states. The entailment makes choices for combinations of mechanics and their preconditions or effects—this leads to making choices for every mechanic for every type of precondition and effect. Together this final statement defines a mechanic as having between 0 and `nop` of each type of precondition or effect by choosing which valid game states the mechanic operates on. Those states are in turn derived from valid coordinate frames, points of time, and allowed values for a state (in an absolute or relative coordinate frame).

A.2.2 Design Requirements

Design requirements come as hard constraints or soft optimization criteria. ASP provides syntax for forbidding logical terms from being allowed by using an empty entailment. Hard constraints are readily expressed using this syntax by specifying conjunctions of mechanic elements (`op` predicates) that are not desired. For example, to prevent a mechanic for have a precondition of both equality and inequality on the same state we use:

```
:- op(M, neq, C, T, S), op(M, eq, C, T, S).
```

which defines a conjunction of the same mechanic testing both inequality (`neq`) and equality (`eq`) and marks this as forbidden through the empty entailment (`:-`, where nothing is 'derived' on the left-hand side). The $Invalid$ proposition was used above to indicate this derivation.

Soft optimization criteria make use of ASP's optimization syntax. `#minimize` is used to define sets of predicates to minimize the count of (which may be weighted by values of the predicates). The example below derives a set of predicates expressing the use of types of preconditions my mechanics and then uses a `#minimize` statement to reduce the total number of any of these preconditions used by mechanics:

```
1  eq(M,C,T,S) :- op(M,eq,C,T,S).
2  neq(M,C,T,S) :- op(M,neq,C,T,S).
3  lt(M,C,T,S) :- op(M,lt,C,T,S).
4  gt(M,C,T,S) :- op(M,gt,C,T,S).
5
6  #minimize[
7          eq(_,_,_,_),
8          neq(_,_,_,_),
9          lt(_,_,_,_),
10         gt(_,_,_,_),
```

```
11          add(_,_,_,_) ].
```

where '_' is the ASP syntax indicating a variable in a predicate that will not be referenced for its value in the logical sentence. Other design requirements on mechanic structure can be expressed similarly. For example, cost-benefit balance can be described by deriving predicates that express the difference between costs and benefits and minimizing the absolute value of that difference.

A.3 Planner

Playability checking uses a planner to test whether a given set of mechanics can be used in a game instance to move from an initial state to a goal state without entering failure states. The planner tracks state through predicates for absolute world coordinates and coordinates relative to agents (those that may use mechanics). For clarity in exposition we focus on the base planning capabilities without considering multi-instance cases or agents with multiple goals.

The core components of state tracked by the planner are expressed with:

```
1 fluent(F) :- op(_,_,_,_,F).
2 fluent(F) :- init(F).
3 fluent(F) :- query(F).
4 fluent(F) :- fail(F).
5
6 holds(0,P) :- init(P).
7 sense(T, (E,P, V-Vplayer)) :-
8         holds(T, (E,P, V)),
9         holds(T, (player,P, Vplayer)).
```

The first four lines derive fluents that express changing game state from mechanics (`op`), initial state (`init`), goal state (`query`), or failure state (`fail`). Absolute game state is

tracked using `holds(T,F)`, where `T` is a time index and `F` is a state fluent. The absolute game state is initialized at time 0 from the initial state predicates (which are provided in a game domain). Relative game state is derived from the holds predicates by computing the difference in fluent specifications for state values between the player and any other entity.

With state tracking, the planning problem consists of choosing an action at each time step for the player to take such that the goal state is reached without entering failure states. State transitions track the occurrence of mechanics over time while enforcing conditions on how mechanics are used:

```
1 time(1..t).
2
3 1 { occ(T,A) : mech(A) } 1 :- time(T).
4
5 :- occ(T,A), eq(A,rel,Td,F), not sense(T-Td,F).
6 :- occ(T,A), eq(A,abs,Td,F), not holds(T-Td,F).
```

where `%` is the comment syntax in ASP. The first line defines the time predicate to track each time step possible in the game from an input parameter `t`. Line 3 uses ASP's choice syntax to decide which mechanic occurs at a time step (`occ(T,A)`) among the mechanics (`mech(A)`) for each time step (`time(T)`). The braces indicate exactly one mechanic (between 1 and 1) must be chosen and the entailment from time steps indicates a choice for each time step. The colon within the braces indicates the choice among the set of mechanics. This sentence expresses the element of choosing the actions in the plan—the remaining aspects of the planner use this choice to update state while performing checks to ensure the set of choices meet playability requirements.

The following two lines express constraints on when mechanic occurrences are possible (similar statements are used for the remaining constraints). The first term expresses a conjunction of a mechanic occurring at a time (`occ(T,A)`) where the mechanic has an relative (`rel`) equality constraint for a value (`F`) at a time difference (`Td`) and the

case that the player does not sense that relative value at the appropriate time difference (`sense(T-Td,F)`). This conjunction is forbidden through the empty entailment. The second conjunction is similar, only checking absolute coordinates (`abs`) and using the corresponding `holds` predicate instead.

The logic to express state update predicates is slightly more complex, making use of intermediate predicates to hold state update values:

```
1  add_action(T+Td-1,A, (E,P,V)) :-
2          occ(T,A), add(A, rel, Td, (E,P,V)).
3
4  add_value(T, (E,P,V)) :-
5          V = #sum[ add_action(T,M,(E,P,Vd)) : mech(M) :
6                  value(Vd) = Vd ],
7          time(T), entity(E), parameter(P).
8
9  holds(T, (E,P, V+Vd)) :-
10         holds(T-1, (E,P, V)),
11         add_value(T, (E,P,Vd)).
12
13 holds(T, (E,P, V)) :-
14         holds(T-1, (E,P, V)), time(T),
15         { add_value(T, (E,P,_)) } 0.
```

The first sentence derives when a state update should occur based on when a mechanic occurs and the effects of that mechanic (`add`).

The second line aggregates across the additions made by all mechanic effects at a time step using the ASP predicate for calculating a sum (`#sum`). The statement inside the square braces extracts the value change from every `add_action` across mechanics (lines 5-6). The `#sum` then sums up these values and assigns this value to a variable (`V`). The sum

is computed for every combination of times, entities, and parameters, expressed through the conjunction with these variables (line 7). Together, this statement sums up the updates made across mechanics for each time step to compute the single state update (`add_value`) to occur for a given entity and parameter combination at a time.

The third logical sentence (lines 9-11) applies the state update to absolute game state. The state at a given point in time is derived from the prior state value (`holds(T-1, (E,P,V))`) and the addition to be made to that state (`add_value(T, (E,P,Vd))`). Note that sensed state does not require a direct update as it is derived from absolute state.

The fourth sentence (lines 13-15) addresses the case where an entity parameter value has no value updates. In this case the current state is assigned to the same value as the previous state. The final part of the conjunction uses ASP's counting syntax (the braces) to find the state where there are no more than 0 predicates indicating to update an entity and parameter pair—i.e., there is no update to the entity parameter combination.

The only remaining aspect of planning is to ensure the choices of actions meet playability requirements. As a base requirement, no plans must ever leave the allowed range of absolute state values:

```
1 bad_holds :- holds(T, (E,P,V)), not range(E,P,V).
2 :- bad_holds.
```

The first line derives a proposition for the case where a state holds a value not allowed by any of the absolute state predicates. This proposition is forbidden, preventing any case of the state leaving allowed values. A similar approach enforces the playability requirements:

```
1 win :- query(F), holds(_,F).
2 :- not win.
3
4 failure :- fail(F), holds(_,F).
5 :- failure.
```

The first sentence checks for the goal state holding at a point in time and the second forbids the case that this does not occur. The second sentence checks for any failure state holding at a point in time and forbids this from occurring.

A.4 Domain Example

Defining domains for mechanic generation consists of specifying the entities, parameters, and allowed absolute and relative ranges for the game. The simplified platformer domain used to generate the $lift$ and $ride$ mechanics above can be defined using:

```
1  entity(player;enemy; b1;b2;b3;b4;b5;b6;b7;b8;b9;b10).
2
3  parameter(x; y).
4
5  has(player,x;y).
6  range(player,x,1..8).
7  range(player,y,1..5).
8
9  has(enemy,x;y).
10 range(enemy,x,1..8).
11 range(enemy,y,1..5).
12
13 has(b1;b2;b3;b4;b5;b6;b7;b8;b9;b10, x;y).
14 range(b1;b2;b3;b4;b5;b6;b7;b8;b9;b10, x,1..8).
15 range(b1;b2;b3;b4;b5;b6;b7;b8;b9;b10, y,1..5).
16
17 op_range(player,x,-2..2).
18 op_range(player,y,-2..2).
19
20 op_range(enemy,x,-2..2).
21 op_range(enemy,y,-2..2).
```

The first statement creates entities for the player, enemy, and blocks making up the ground and second statement defines the spatial coordinate parameters. The player is allowed allowed to occupy positions in the grid from (1,1) to (8,5) in Cartesian coordinates (lines 5-7); similarly for the enemy (lines 9-11) and all of the blocks (lines 13-15). Mechanics are allowed to move the player by 2 units in either direction (lines 17-18); the same for the enemy (lines 20-21). By not defining allowed ranges for changes to block position the

mechanic generation can only alter the player or enemy position with mechanics.

The level instance can then be created by initializing the player, enemy, and block positions:

```
1  init( (player,x,1) ).
2  init( (player,y,2) ).
3
4  init( (enemy,x,4) ).
5  init( (enemy,y,2) ).
6
7  query( (player,x,8) ).
8  query( (player,y,5) ).
9
10 init( (b1;b2;b3;b4;b5;b6;b7;b8, y,1) ).
11
12 init( (b1, x,1) ).
13 init( (b2, x,2) ).
14 init( (b3, x,3) ).
15 init( (b4, x,4) ).
16 init( (b5, x,5) ).
17 init( (b6, x,6) ).
18 init( (b7, x,7) ).
19 init( (b8, x,8) ).
```

The player is initialized to the position (1,2) (lines 1-2) and the enemy to (4,2) (lines 4-5) with the player goal being to reach the position (8,5) (lines 7-8). All blocks are arranged along the same y position and given x positions to create a solid ground across the level.

We next add gravity as an engine constraint:

```
1  op(0, add, rel, 1, (player,y,-1)).
```

```
2 occ(T,0) :- time(T).
```

Where the first statement defines the mechanic of gravity moving the player down a single unit on the y axis and the second statement derives that mechanic (index 0) at each time step. Note this code defines gravity for the player—the same statements can be added for any other entity of interest.

Finally, we define failure by the player occupying the same location as the enemy:

```
1 loc(T,E,(X,Y)) :- holds(T,(E,x,X)), holds(T,(E,y,Y)).
2 dead(T) :- loc(T,player,Location),
3          loc(T,E,Location), E!=player.
4 failure :- dead(_).
```

The first statement derives a location predicate from the conjunction of an entities x and y coordinates. The second statement defines the dead predicate as occurring when the location of the player and another entity is the same. The final statement derives failure from being dead at any time. This case illustrates more complex failure checks by using the same failure predicate checked in the planner when validating basic failure cases (e.g., the player reaching a specific location in the world).

APPENDIX B

CARDONOMICON CARDS

Table B.1: Cards used in *Cardonomicon* experiments.

Card Name	**Health**	**Cost**	**Attack**
Stonetusk Boar	1	1	1
Dire Wolf Alpha	2	2	1
Defias Ringleader	1	2	2
Kobold Geomancer	2	2	2
Ironfur Grizzly	2	2	1
SI Agent	1	2	2
Fen Creeper	7	5	3
Southsea Captain	4	4	5
Abusive Sergeant	1	1	1
Angry Chicken	1	1	1
Blood Imp	1	1	1
Super OP	2	3	1
Aldor Peacekeeper	3	3	3
Arcane Golem	2	3	4
Dalaran Mage	4	3	1
Dark Cultist	4	3	2
Scarlet Crusader	1	3	3
Ancient Brewmaster	6	4	3
Chillwind Yeti	5	4	4
Boulderfist Ogre	7	6	6

APPENDIX C

LEARNED DESIGN HYPOTHESES

In chapter 6 I presented a number of predictive models learned by the system about how card parameters influence game outcomes. Below are a set of more complex models the system learned that account for interactions between the game length and action metrics as well as the influence of agent parameters on action metrics. These models illustrate the potential to acquire more complex knowledge through iteratively considering a set of alternative design hypotheses. Note that I provided the choice of parameters to consider in the model, thus these are not yet fully automated learned models.

C.1 Game Length

The game length model learned in chapter 5 only accounted for card parameters, ignoring any influence of agent strength on the length of games. Skill-based design metrics are intended to help understand when a design does (not) respond to differences in play skill, thus I had the system model a model of how player skill influences game length along with card parameters. Adding features for agent strength relative to the baseline weak

Game length vs card and agent parameters	
feature	coefficient
attack = 4	**0.97**
attack = 7	**0.94**
health = 4	1.00
health = 7	1.00
cost = 4	**1.04**
cost = 7	**1.04**
p1 moderate	**1.05**
p2 strong	1.01

Table C.1: Effect of card and agent parameters on game length. Bold values indicate significance ($p < 0.05$)

Attack frequency vs card and agent parameters	
feature	coefficient
player = moderate	1.03
player = strong	1.03
attack = 4	**0.77**
attack = 7	**0.82**
health = 4	**1.56**
health = 7	**2.23**
cost = 4	**0.64**
cost = 7	**0.28**
player = moderate X attack = 4	1.09
player = moderate X attack = 7	0.87
player = strong X attack = 4	1.08
player = strong X attack = 7	0.86
player = moderate X health = 4	1.04
player = moderate X health = 7	0.98
player = strong X health = 4	0.93
player = strong X health = 7	0.95
player = moderate X cost = 4	0.91
player = moderate X cost = 7	0.87
player = strong X cost = 4	0.94
player = strong X cost = 7	1.00

Table C.2: Effect of card and agent parameters on card attack rates. Bold values indicate significance ($p < 0.05$).

agent showed the strength of the first agent to very modestly increase game length, with the strength of the second agent only having marginal significance ($p < 0.1$) (Table C.1). As before, greater attack parameter values predicted reduced game length while greater cost parameter values predicted increased game length. Thus, the system learned how to account for the effects of agent skill on game length, finding these to be comparatively weak effects against the main effects of card parameters.

C.2 Card Attack Rates

Agent strength may influence more than game length, potentially altering the rate at which agents choose to play or attack with cards. To test this case the system learned a model that included the strength of the focal player in the game and interactions of this player with the

Play frequency vs card and agent parameters	
feature	coefficient
player = moderate	1.05
player = strong	1.05
attack = 4	0.95
attack = 7	0.99
health = 4	1.05
health = 7	1.02
cost = 4	**0.82**
cost = 7	**0.48**
player = moderate X attack = 4	0.96
player = moderate X attack = 7	0.94
player = strong X attack = 4	0.95
player = strong X attack = 7	0.91
player = moderate X health = 4	0.97
player = moderate X health = 7	0.95
player = strong X health = 4	1.01
player = strong X health = 7	0.99
player = moderate X cost = 4	0.99
player = moderate X cost = 7	1.05
player = strong X cost = 4	1.04
player = strong X cost = 7	0.96

Table C.3: Effect of card and agent parameters on card play rates. Bold values indicate significance ($p < 0.05$).

card parameters (Table C.2). As in the predictive model for card parameters' influence on rates of playing the "Stonetusk Boar" card, the card cost, health, and attack values had the same effects. Player strength, however, had no significant effects. Thus, the system learned that, as in the base game, the design variants did not find scenarios where agent strength had a significant effect on agent action choices in the game.

C.3 Card Play Rates

As with card attack rates, the system also learned a model of the effect player strength had an on card play rates (Table C.3). The model accounted for the interaction of player strength with card parameters in the Poisson regression model. Only card cost appeared as a significant effect, and this effect was that greater costs reduced card play frequency.

Attack option frequency vs card and agent parameters	
feature	coefficient
player = moderate	1.05
player = strong	1.05
attack = 4	**0.95**
attack = 7	**0.99**
health = 4	**1.05**
health = 7	**1.02**
cost = 4	**0.82**
cost = 7	**0.48**
player = moderate X attack = 4	0.96
player = moderate X attack = 7	0.94
player = strong X attack = 4	0.95
player = strong X attack = 7	**0.91**
player = moderate X health = 4	0.97
player = moderate X health = 7	0.95
player = strong X health = 4	1.01
player = strong X health = 7	0.99
player = moderate X cost = 4	0.99
player = moderate X cost = 7	1.05
player = strong X cost = 4	1.04
player = strong X cost = 7	0.96

Table C.4: Effect of card and agent parameters on card attack option rates. Bold values indicate significance ($p < 0.05$).

Thus, as with card attack rates, card parameters do not interact with agent strength to alter choices of cards to play. This further reinforces the notion that *Cardonomicon* does not effectively differentiate agent skill levels.

C.4 Card Attack Options

Learning a model of the relationship between agent strength interacted with card parameters produced similar outcomes between the attack option and attack action cases (Table C.4). Agent parameters had no significant effect, and only one significant interaction effect with card attack. All baseline card parameters had a significant effect. As before, this learned predictive model suggests *Cardonomicon* has a shallow strategic space where card features dominate.

Play option frequency vs card and agent parameters	
feature	coefficient
player = moderate	0.94
player = strong	0.89
attack = 4	0.93
attack = 7	**0.86**
health = 4	0.93
health = 7	**0.86**
cost = 4	**1.33**
cost = 7	**1.10**
player = moderate X attack = 4	1.00
player = moderate X attack = 7	1.09
player = strong X attack = 4	1.03
player = strong X attack = 7	1.08
player = moderate X health = 4	1.03
player = moderate X health = 7	**1.15**
player = strong X health = 4	1.10
player = strong X health = 7	**1.18**
player = moderate X cost = 4	1.12
player = moderate X cost = 7	0.99
player = strong X cost = 4	**1.14**
player = strong X cost = 7	0.96

Table C.5: Effect of card and agent parameters on card play option rates. Bold values indicate significance ($p < 0.05$).

C.5 Card Play Options

The system also learned a model for how agent strength interacted with card parameters to influence card play options (Table C.5). Agent strength interacted significantly with high health settings, suggesting stronger agents are better able to take advantage of additional card health. Overall, however, the lack of other significant results suggests agent strength does not strongly interact with card parameters, as seen with the card play rates earlier.

Together the models learned in this section demonstrate how an automated system can evaluate the potential effectiveness of a design to differentiate among agents of differing skill. The models learned here show little or no significant effect of agent strength on card use, with a modest influence on game length. Enabling a system to search for and discon-

firm these relationships can ultimately improve techniques for procedural generation by learning which aspects of a design space to ignore for more efficient sampling of potential designs. The models here further reinforce the capability of learned models to detect design flaws by identifying when design goals (such as different agent choices based on skill) are not met.

REFERENCES

[1] N. Abe and H. Mamitsuka, "Query learning strategies using boosting and bagging," in International Conference on Machine Learning, 1998.

[2] E. Adams and J. Dormans, Game Mechanics: Advanced Game Design. New Riders, 2012.

[3] T. Adams, Dwarf Fortress, Game [Windows, Mac, Linux], Bay 12 Games, 2006.

[4] A.I. Design, Rogue, Game [Amiga, Amstrad CPC, Atari 8-bit, Atari ST, Commodore 64, DOS, Mac, TOPS-20, TRS-80 CoCo, Unix, ZX Spectrum], Epyx, 1980.

[5] E. Andersen, S. Gulwani, and Z. Popović, "A trace-based framework for analyzing and synthesizing educational progressions," in ACM SIGCHI Conference on Human Factors in Computing Systems, 2013.

[6] E. Andersen, Y.-E. Liu, E. Apter, F. Boucher-Genesse, and Z. Popović, "Gameplay analysis through state projection," in 5th International Conference on the Foundations of Digital Games, 2010.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," Machine learning, vol. 47, no. 2-3, pp. 235–256, 2002.

[8] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.

[9] O. Bartheye and É. Jacopin, "A real-time PDDL-based planning component for video games," in 5th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2009.

[10] A. W. Bauer, S. Cooper, and Z. Popović, "Automated redesign of local playspace properties," in 8th International Conference on the Foundations of Digital Games, 2013.

[11] A. W. Bauer and Z. Popović, "Rrt-based game level analysis, visualization, and visual refinement," in 8th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2012.

[12] A. Benbassat and M. Sipper, "Evomcts: Enhancing MCTS-based players through genetic programming," in IEEE Conference on Computational Intelligence in Games, 2013.

[13] S. Björk and J. Holopainen, Patterns in Game Design. Cengage Learning, 2005.

[14] Blizzard North, Diablo II, Game [Windows, Mac], Blizzard Entertainment, 2000.

[15] I Bogost, Persuasive Games: The Expressive Power of Videogames. The MIT Press, 2007.

[16] I. Bogost, Unit Operations: An Approach to Videogame Criticism. MIT Press, 2006.

[17] ——, How to Do Things with Videogames. University of Minnesota Press, 2011.

[18] E. Brochu, "Interactive bayesian optimization: Learning user preferences for graphics and animation," PhD thesis, University of British Columbia, 2010.

[19] C. Browne, "Deductive search for logic puzzles," in IEEE Conference on Computational Intelligence in Games, 2013.

[20] C. Browne and F. Maire, "Evolutionary game design," IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, pp. 1–16, 2010.

[21] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, pp. 1–43, 2012.

[22] E. Butler, E. Andersen, A. M. Smith, S. Gulwani, and Z. Popović, "Automatic game progression design through analysis of solution features," in ACM SIGCHI Conference on Human Factors in Computing Systems, 2015.

[23] E. Butler, E. Andersen, A. M. Smith, S. Gulwani, and Z. Popović, "Automatic game progression design through analysis of solution features," in ACM SIGCHI Conference on Human Factors in Computing, 2015.

[24] E. Butler, A. M. Smith, Y.-E. Liu, and Z. Popović, "A mixed-initiative tool for designing level progressions in games," in ACM Symposium on User Interface Software and Technology, 2013.

[25] T. Cadwell, "Counterplay and teamplay in multiplayer game design," in Game Developers Conference, 2013.

[26] A. Canossa and G. Smith, "Towards a procedural evaluation technique: Metrics for level design," in 10th International Conference on the Foundations of Digital Games, 2015.

[27] A. Cenkner, V. Bulitko, and M. Spetch, "A generative computational model for human hide and seek behavior," in 7th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[28] K. Chaloner and I. Verdinelli, "Bayesian experimental design: A review," Statistical Science, vol. 10 (3), pp. 273–304, 1995.

[29] Y.-H. Chang, R. T. Maheswaran, T. Levinboim, and V. Rajan, "Learning and evaluating human-like NPC behaviors in dynamic games," in 7th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[30] G. Chaslot, J.-T. Saito, J. W. H. M. Uiterwijk, B. Bouzy, and H. J. van den Herik, "Monte-Carlo strategies for computer Go," in 18th Belgian-Dutch Conference on Artificial Intelligence, 2006.

[31] S. Colton, M. Cook, R. Hepworth, and A. Pease, "On acid drops and teardrops: Observer issues in computational creativity," in 7th AISB Symposium on Computing and Philosophy, 2014.

[32] S. Colton, A Pease, and J Charnley, "Computational creativity theory: The FACE and IDEA descriptive models," in 2nd International Conference on Computational Creativity, 2011.

[33] K. Compton, J. C. Osborn, and M. Mateas, "Generative methods," in 4th Workshop on Procedural Content Generation in Games, 2013.

[34] D. Cook. (Jul. 2007). The chemistry of game design, UBM Tech.

[35] M. Cook and S. Colton, "A Rogue Dream: Automatically generating meaningful content for games," in Experimental AI in Games Workshop, 2014.

[36] M. Cook, S. Colton, and J. Gow, "Automating game design in three dimensions," in AISB Symposium on AI and Games, 2014.

[37] ——, "The ANGELINA videogame design system, part I," IEEE Trans. Computational Intelligence and AI in Games, vol. PP, p. XX, 2016.

[38] M. Cook, S. Colton, and A. Pease, "Aesthetic considerations for automated platformer design.," in 8th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2012.

[39] M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic Miner: Reflection-driven game mechanic discovery and level design," in EvoGAMES, 2013.

[40] M. Cook, J. Gow, and S. Colton, "Danesh: Helping bridge the gap between procedural generators and their output," in 7th Workshop on Procedural Content Generation, 2016.

[41] M. Cook and G. Smith, "Formalizing non-formalism: Breaking the rules of automated game design," in 10th International Conference on the Foundations of Digital Games, 2015.

[42] S. Cooper, A. Treuille, J. Barbero, A. Leaver-Fay, K. Tuite, F. Khatib, A. C. Snyder, M. Beenen, D. Salesin, D. Baker, Z. Popović, and F. Players, "The challenge of designing scientific discovery games," in 5th International Conference on the Foundations of Digital Games, 2010.

[43] V. Corporation, Dota 2, Game [Windows, Mac, Linux], Valve Corporation, 2013.

[44] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," IEEE Trans. Computational Intelligence and AI in Games, vol. 4, pp. 120–143, 2012.

[45] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble determinization in Monte Carlo Tree Search for the imperfect information card game Magic: The Gathering," IEEE Trans. Computational Intelligence and AI in Games, vol. 4, pp. 241–257, 2012.

[46] E. Cox, E. Schkufza, R. Madsen, and M. R. Genesereth, "Factoring general games using propositional automata," in IJCAI Workshop on General Intelligence in Game-Playing Agents, 2009.

[47] M. Csikszentmihalyi, Creativity. Harper Collins, 1996.

[48] ——, "Handbook of creativity," in, R. J. Sternberg, Ed. Cambridge Univ Press, 1999, ch. Implications of a Systems Perspective for the Study of Creativity, pp. 313–337.

[49] S. Dahlskog and J. Togelius, "Procedural content generation using patterns as objectives," in Applications of Evolutionary Computation, Springer Berlin Heidelberg, 2014.

[50] I. Dart and M. J. Nelson, "Smart terrain causality chains for adventure-game puzzle generation," in IEEE Conference on Computational Intelligence and Games, 2012.

[51] I. M. Dart, G. De Rossi, and J. Togelius, "SpeedRock: Procedural rocks through grammar and evolution," in Workshop on Procedural Content Generation in Games, 2011.

[52] S. Das, A. Zook, and M. O. Riedl, "Examining game world topology personalization," in ACM SIGCHI Conference on Human Factors in Computing Systems, 2015.

[53] N. Davis, B. Li, B. O'Neill, M. Riedl, and M. Nitsche, "Distributed creative cognition in digital filmmaking," in 8th ACM Conference on Creativity and Cognition, 2011.

[54] S. Devlin, A. A., N. Sephton, C. P., and R. J., "Combining gameplay data with Monte Carlo Tree Search to emulate human play," in 12th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2016.

[55] J. Dormans, "Machinations: Elemental feedback structures for game design," in GAMEON-NA, 2009.

[56] ——, "Adventures in level design: Generating missions and spaces for action adventure games," in 1st Workshop on Procedural Content Generation in Games, 2010.

[57] ——, "Simulating mechanics to study emergence in games," in 1st Workshop on Artificial Intelligence in the Game Design Process, 2011.

[58] dotGEARS, Flappy bird, Game [Android, iOS], dotGEARS, 2013.

[59] A. Drachen, A. Canossa, and G. N. Yannakakis, "Player modeling using self-organization in Tomb Raider: Underworld," in IEEE Conference on Computational Intelligence and Games, 2009.

[60] D. Duvenaud, J. R. Lloyd, R. Grosse, J. B. Tenenbaum, and Z. Ghahramani, "Structure discovery in nonparametric regression through compositional kernel search," in International Conference on Machine Learning, 2013.

[61] G. Elias, R. Garfield, and K. Gutschera, Characteristics of Games. MIT Press, 2012.

[62] A. E. Elo, The rating of chessplayers, past and present. Arco, 1978.

[63] R. Evans and E. Short, "Versu - a simulationist storytelling system," IEEE Trans. Computational Intelligence and AI in Games, vol. 6, no. 2, pp. 113–130, 2014.

[64] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," Artificial Intelligence, vol. 2, pp. 189–208, 1972.

[65] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius, "A card game description language," in Applications of Evolutionary Computation, Springer, 2013, pp. 254–263.

[66] ——, "Towards the automatic generation of card games through grammar-guided genetic programming," in 8th International Conference on Foundations of Digital Games, 2013.

[67] T. Fristoe, J. Denner, M. MacLaurin, M. Mateas, and N. Wardrip-Fruin, "Say it with systems: Expanding Kodu's expressive power through gender-inclusive mechanics," in 6th International Conference on Foundations of Digital Games, 2011.

[68] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius, "Investigating MCTS modifications in general video game playing," in IEEE Conference on Computational Intelligence and Games, 2015.

[69] T. Fullerton, C. Swain, and S. Hoffman, Game Design Workshop. Morgan Kaufmann, 2008.

[70] R. Games, League of legends, Game [Windows, Mac], Riot Games, 2009.

[71] R. Garfield, Magic: The Gathering, Game [Physical], Wizards of the Coast, 1993.

[72] P. Gervás, "Dynamic inspiring sets for sustained novelty in poetry generation," in 2nd International Conference on Computational Creativity, 2011, pp. 111–116.

[73] M. Ghallab, D. Nau, and P. Traverso, Automated Planning: Theory & Practice. Elsevier, 2004.

[74] M. E. Glickman, "Parameter estimation in large dynamic paired comparison experiments," Applied Statistics, pp. 377–394, 1999.

[75] ——, "Dynamic paired comparison models with stochastic variances," Applied Statistics, vol. 28, no. 6, pp. 673–689, 2001.

[76] A. K. Goel and S. Rugaber, "Interactive meta-reasoning: Towards a CAD-like environment for designing game-playing agents," in Computational Creativity Research: Towards Creative Machines, Springer, 2015, pp. 347–370.

[77] P. Gorniak and I. Davis, "SquadSmart: Hierarchical planning and coordinated plan execution for squads of characters," in 3rd AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2007.

[78] R. B. Grosse, R. Salakhutdinov, W. Freeman, and J. Tenenbaum, "Exploiting compositionality to explore a large space of model structures," in Uncertainty in Artificial Intelligence, 2012.

[79] M. Günther, S. Schiffel, and M. Thielscher, "Factoring general games," in IJCAI Workshop on General Game Playing, 2009.

[80] M. Guzdial and M. O. Riedl, "Toward game level generation from gameplay videos," in 6th Workshop on Procedural Content Generation in Games, 2015.

[81] M. Guzdial, N. Sturtevant, and B. Li, "Deep static and dynamic level analysis: A study on Infinite Mario," in Experimental AI in Games Workshop 3, 2016.

[82] G. Gygax and D. Arneson, Dungeons & Dragons, Game [Physical], TSR and Wizards of the Coast, 1974.

[83] K. Hartsook, A. Zook, S. Das, and M. Riedl, "Toward supporting storytellers with procedurally generated game worlds," in IEEE Conference on Computational Intelligence in Games, 2011.

[84] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning, 2nd. Springer, 2009.

[85] Hello Games, Spore, Game [PlayStation 4, Windows], Hello Games, 2016.

[86] R. Herbrich, T. Minka, and T. Graepel, "Trueskill(tm): A bayesian skill rating system," in Advances in Neural Information Processing Systems 20, MIT Press, 2007, pp. 569–576.

[87] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila, "Hierarchical plan representations for encoding strategic game AI," in

1st AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2005.

[88] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, "Evolving personas for player decision modeling," in 2014 IEEE Conference on Computational Intelligence and Games, 2014.

[89] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, "Generative agents for player decision modeling in games," in 9th International Conference on the Foundations of Digital Games, 2014.

[90] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, "Personas versus clones for player decision modeling," in 13th International Conference on Entertainment Computing, 2014.

[91] ——, "Monte-Carlo Tree Search for persona based player modelling," in First Workshop on Player Modeling, 2015.

[92] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in the Mario AI framework," in 9th International Conference on the Foundations of Digital Games, 2014.

[93] R. Hunicke and V. Chapman, "AI for dynamic difficulty adjustment in games," in AAAI Workshop on Challenges in Game Artificial Intelligence, 2004.

[94] Interactive Data Visualization, Inc. (2002). Speedtree.

[95] A. Isaksen, D. Gopstein, and A. Nealen, "Exploring game space using survival analysis," in 10th International Conference on the Foundations of Digital Games, 2015.

[96] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Discovering unique game variants," in ICCC Workshop on Computational Creativity and Games, 2015.

[97] K. Isbister and N. Schaffer, Game Usability: Advancing the Player Experience. Morgan Kaufmann, 2008.

[98] E. J. Jacobsen, R. Greve, and J. Togelius, "Monte Mario: Platforming with MCTS," in Genetic and Evolutionary Computation, 2014.

[99] A. Jaffe, "Understanding game balance with quantitative methods," PhD thesis, University of Washington, 2013.

[100] A. Jaffe, A. Miller, E. Andersen, Y.-E. Liu, A. Karlin, and Z. Popović, "Evaluating competitive game balance with restricted play," in

8th Conference on Artificial Intelligence and Interactive Digital Entertainment, 2012.

[101] R. Jain, A. Isaksen, C. Holmgård, and J. Togelius, "Autoencoders for level generation, repair, and recognition," in ICCC Workshop on Computational Creativity and Games, 2016.

[102] M. R. Johnson, Ultima Ratio Regum, Game [Windows], 2016.

[103] J. Jones, C. Parnin, A. Sinharoy, S. Rugaber, and A. K. Goel, "Adapting game-playing agents to game requirements," in 5th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2009.

[104] K. M. Kapp, The Gamification of Learning and Instruction. John Wiley & Sons, 2012.

[105] C. Kemp and J. B. Tenenbaum, "The discovery of structural form," Proceedings of the National Academy of Sciences, vol. 105, no. 31, pp. 10 687–10 692, 2008.

[106] A. Khalifa, D. P'erEz-li'ebana, S. M. Lucas, and J. Togelius, "General video game level generation," in GECCO, 2016.

[107] P. Klint and R. van Rozen, "Micro-machinations: A dsl for game economies," in Software Language Engineering, Springer, 2013.

[108] J. L. Kolodner, "Understanding creativity: A case-based approach," in 1st European Workshop on Case-Based Reasoning, Springer, 1994.

[109] Konami, Frogger, Game [Arcade], Sega, 1981.

[110] R. Koster, A Theory of Fun in Game Design, 2nd. Paraglyph press, 2013.

[111] A. Kozbelt, R. A. Beghetto, and M. A. Runco, "The cambridge handbook of creativity," in, J. C. Kaufman and R. J. Sternberg, Eds. Cambridge University Press, 2010, ch. Theories of Creativity, pp. 20–47.

[112] J. E. Laird and J. C. Duchi, "Creating human-like synthetic characters with multiple skill levels: A case study using the Soar Quakebot," in AAAI 2000 Fall Symposium on Simulating Human Agents, 2000.

[113] F. Lantz. (2016). The depth project.

[114] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a generic method of evaluating game levels," in 8th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2013.

[115] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," IEEE Trans. Computational Intelligence and AI in Games, vol. 6, pp. 78–89, 2013.

[116] Y.-E. Liu, E. Andersen, R. Snider, S. Cooper, and Z. Popović, "Feature-based projections for effective playtrace analysis," in 6th International Conference on Foundations of Digital Games, 2011.

[117] Y.-E. Liu, T. Mandel, E. Brunskill, and Z. Popović, "Towards automatic experimentation of educational knowledge," in ACM SIGCHI Conference on Human Factors in Computing Systems, 2014.

[118] ——, "Trading off scientific knowledge and user learning with multi-armed bandits," in Educational Data Mining, 2014.

[119] J. R. Lloyd, D. Duvenaud, R. Grosse, J. B. Tenenbaum, and Z. Ghahramani, ""automatic construction and Natural-Language description of nonparametric regression models"," in Association for the Advancement of Artificial Intelligence, 2014.

[120] J. D. Lomas, "Optimizing motivation and learning with large-scale game design experiments," PhD thesis, Carnegie Mellon University, 2014.

[121] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specification," Stanford University, Tech. Rep., 2008.

[122] T. Lubart, "How can computers be partners in the creative process?" International Journal of Human-Computer Studies, vol. 63, no. 4, pp. 365–369, 2005.

[123] T. Mahlmann, "Modelling and generating strategy games mechanics," PhD thesis, IT University of Copenhagen, 2012.

[124] T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Modelling and evaluation of complex scenarios with the strategy game description language," in IEEE Conference on Computational Intelligence and Games, IEEE, 2011, pp. 174–181.

[125] J. Mariño, W. Reis, and L. Lelis, "An empirical evaluation of evaluation metrics of procedurally generated Mario levels," in

11th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2015.

[126] C. Martens, "Ceptre: A language for modeling generative interactive systems," in 11th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2015.

[127] C. Martens, A. Summerville, M. Mateas, J. Osborn, S. Harmon, N. Wardrip-Fruin, and A. Jhala, "Proceduralist readings, procedurally," in Experimental AI in Games Workshop 3, 2016.

[128] M. Mateas and A. Stern, "Architecture, authorial idioms and early observations of the interactive drama Façade," Carnegie Mellon University, Tech. Rep., 2002.

[129] M. Mateas and N. Wardrip-Fruin, "Defining operational logics," in DiGRA, 2009.

[130] Maxis, Spore, Game [Windows, Mac, iOS], Electronic Arts, 2008.

[131] J. McCoy, M. Treanor, B. Samuel, A Reed, M. Mateas, and N. Wardrip-Fruin, "Social story worlds with Comme il Faut," IEEE Trans. Computational Intelligence and AI in Games, vol. 6, no. 2, pp. 97–112, 2014.

[132] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL - the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep., 1998.

[133] B. Medler, "Play with data — an exploration of play analytics and its effect on player expereinces," PhD thesis, Georgia Institute of Technology, 2012.

[134] Mojang, Minecraft, Game [Windows, Mac, Linux, Android, iOS, Xbox 360, Xbox One, PlayStation 3, PlayStation 4, PlayStation Vita, Raspberry Pi, Wii U], Mojang and Microsoft Studios, 2011.

[135] Monolith, F.e.a.r. Game [Windows, Xbox 360, PlayStation 3], Sierra Entertainment, 2005.

[136] R. G. Morris, S. H. Burton, P. M. Bodily, and D. Ventura, "Soup over bean of pure joy: Culinary ruminations of an artificial chef," in 3rd International Conference on Computational Creativity, 2012, p. 119.

[137] Mossmouth, Spelunky, Game [Windows, Xbox 360, PlayStation 3, PlayStation 4, PlayStation Vita, Chrome OS], Mossmouth, 2008.

[138] Namco, Pac-Man, Game [Arcade], Namco, 1980.

[139] M. J. Nelson, "Game metrics without players: Strategies for understanding game artifacts," in 1st Workshop on Artificial Intelligence in the Game Design Process, 2011.

[140] M. J. Nelson and M. Mateas, "Recombinable game mechanics for automated design support," in 4th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2008.

[141] M. J. Nelson and A. M. Smith, "ASP with applications to mazes and levels," in Procedural Content Generation in Games, Springer, 2015.

[142] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, "Towards generating arcade game rules with VGDL," in 2015 IEEE Conference on Computational Intelligence and Games, 2015.

[143] T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, "General video game evaluation using relative algorithm performance profiles," in Applications of Evolutionary Computation, Springer, 2015.

[144] Nintendo R&D4, Super Mario Bros. Game [Nintendo Entertainment System], Nintendo, 1985.

[145] A. Normoyle, J. Drake, M. Likhachev, and A. Safonova, "Game-based data capture for player metrics.," in AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2012.

[146] B. O'Neill and M. Riedl, "Supporting human creative story authoring with a synthetic audience," in 7th ACM Conference on Creativity and cognition, ACM, 2009, pp. 399–400.

[147] J. Orkin, "Agent architecture considerations for real-time planning in games," in 1st AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2005.

[148] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis, "Imitating human playing styles in super mario bros.," Entertainment Computing, vol. 4, no. 2, pp. 93–104, 2013.

[149] J. Orwant, "Eggg: Automated programming for game generation," IBM Systems Journal, vol. 39, no. 3.4, pp. 782–794, 2000.

[150] J. Osborn, A. Grow, and M. Mateas, "Modular computational critics for games," in 9th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2013.

[151] J. C. Osborn and M. Mateas, "A game-independent play trace dissimilarity metric," in 9th International Conference on the Foundations of Digital Games, 2014.

[152] J. C. Osborn, B. Samuel, J. A. McCoy, and M. Mateas, "Evaluating play trace (dis)similarity metrics," in 10th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2014.

[153] D. Perez, Spyridon, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 general video game playing competition," IEEE Trans. Computational Intelligence and AI in Games, 2015.

[154] D. Perez, J. Togelius, S. Samothrakis, P. Rolhfshagen, and S. M. Lucas, "Automated map generation for the physical travelling salesman problem," IEEE Trans. Computational Intelligence and AI in Games, vol. 18, pp. 708–720, 2013.

[155] R Pérez y Pérez and M. Sharples, "Mexica: A computer model of a cognitive account of creative writing," Journal of Experimental and Theoretical Artificial Intelligence, vol. 13, no. 2, pp. 119–139, 2001.

[156] D. Pérez-Liébana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, "Analyzing the robustness of general video game playing agents," in IEEE Conference on Computational Intelligence and Games, 2016.

[157] E. J. Powley, S. Gaudl, S. Colton, M. J. Nelson, R. Saunders, and M. Cook, "Automated tweaking of levels for casual creation of mobile games," in ICCC Workshop on Computational Creativity and Games, 2016.

[158] A. N. Rafferty, M. Zaharia, and T. L. Griffiths, "Optimally designing games for cognitive science research," in 34th Annual Conference of the Cognitive Science Society, 2012, pp. 280–287.

[159] C. E. Rasmussen and C. K. Williams, Gaussian Processes for Machine Learning. MIT press Cambridge, MA, 2006, vol. 1.

[160] H. W. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," Policy Sciences, vol. 4, no. 2, pp. 155–169, 1973.

[161] S. M. S. Ronald A Finke Thomas B Ward, Creative Cognition: Theory, Research, and Applications. MIT Press, 1992.

[162] R. van Rozen and J. Dormans, "Adapting game mechanics with micro-machinations," in 9th International Conference on the Foundations of Digital Games, 2014.

[163] M. A. Runco, Creativity: Theories and Themes. Elsevier, 2014.

[164] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd. Prentice Hall, 2009.

[165] J. Ryan, M. Mateas, and N. Wardrip-Fruin, "A simple method for evolving large character social networks," in 5th Workshop on Social Believability in Games, 2016.

[166] J. O. Ryan, A. Summerville, M. Mateas, and N. Wardrip-Fruin, "Toward characters who observe, tell, misremember, and lie," in Experimental AI in Games Workshop 2, 2015.

[167] K. Salen and E. Zimmerman, Rules of Play: Game Design Fundamentals. Cambridge Mass.: MIT Press, 2003, ISBN: 9780262240451.

[168] ——, The Game Design Reader: A Rules of Play Anthology. Cambridge Mass.: MIT Press, 2006, ISBN: 9780262195362.

[169] T. Schaul, "A video game description language for model-based or interactive learning," in IEEE Conference on Computational Intelligence in Games, 2013.

[170] A. I. Schein and L. H. Ungar, "Active learning for logistic regression: An evaluation," Machine Learning, vol. 68, no. 3, pp. 235–265, 2007.

[171] J. Schell, The Art of Game Design: A Book of Lenses. Elsevier/Morgan Kaufmann, 2008, ISBN: 9780123694966.

[172] K. Schenk, A. Lari, M. Church, E. Graves, J. Duncan, R. Miller, N. Desai, R. Zhao, D. Szafron, M. Carbonaro, and J. Schaeffer, "Scriptease II: Platform independent story creation using high-level patterns," in 9th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2013.

[173] S. Schiffel, "Symmetry detection in general game playing," in AAAI Conf. on Artificial Intelligence, 2010.

[174] D. A. Schön, The Reflective Practitioner: How Professionals Think in Action. Basic Books, 1983.

[175] I. Schreiber. (). Game balance concepts.

[176] M. Seif El-Nasr, A. Drachen, and A. Canossa, Eds., Game Analytics. Springer London, 2013.

[177] B. Settles, Active Learning. Morgan & Claypool Publishers, 2012, pp. 1–114.

[178] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in Procedural Content Generation in Games, Springer, 2015.

[179] N. Shaker, G. Smith, and G. N. Yannakakis, "Evaluating content generators," in Procedural Content Generation in Games, Springer, 2015.

[180] N. Shaker, J. Togelius, and M. J. Nelson, Procedural Content Generation in Games. Springer, 2015.

[181] N. Shaker, J. Togelius, and G. N. Yannakakis, "The experience-driven perspective," in Procedural Content Generation in Games, Springer, 2015.

[182] N. Shaker, G. N. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in 6th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2010.

[183] M. Sharples, How We Write: Writing as Creative Design. Psychology Press, 1999.

[184] B. Shneiderman, "Creativity support tools: Accelerating discovery and innovation," Communications of the ACM, 2007.

[185] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484–489, 2016.

[186] H. A. Simon, The Sciences of the Artificial, 3rd. MIT Press, 1996.

[187] K. Siu, A. Zook, and M. O. Riedl, "Collaboration versus competition: Design and evaluation of mechanics for games with a purpose," in 9th International Conference on the Foundations of Digital Games, 2014.

[188] K. Siu, E. Butler, and A. Zook, "A programming model for boss encounters in 2d action games," in Experimental AI in Games Workshop 3, 2016.

[189] K. Siu and M. O. Riedl, "Reward systems in human computation games," in CHI PLAY, ACM, 2016.

[190] A. M. Smith, "Mechanizing exploratory game design," PhD thesis, University of California, Santa Cruz, Dec. 2012.

[191] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović, "A case study of expressively constrainable level design automation tools for a puzzle game," in 7th International Conference on the Foundations of Digital Games, 2012, pp. 156–163.

[192] A. M. Smith, E. Butler, and Z. Popović, "Quantifying over play: Constraining undesirable solutions in puzzle design," in 8th International Conference on the Foundations of Digital Games, 2013.

[193] A. M. Smith and M. Mateas, "Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games," in IEEE Conference on Computational Intelligence and Games, 2010.

[194] ——, "Knowledge-level creativity in game design," in 2nd International Conference on Computational Creativity, 2011.

[195] A. M. Smith, M. J. Nelson, and M. Mateas, "Computational support for play testing game sketches," in 5th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2009.

[196] ——, "Ludocore: A logical game engine for modeling videogames," in IEEE Conference on Computational Intelligence and Games, 2010.

[197] A. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 187–200, 2011.

[198] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in 1st Workshop on Procedural Content Generation in Games, ACM, 2010, p. 4.

[199] S. Snodgrass and S. Ontañón, "A hierarchical approach to generating maps using markov chains," in 10th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2014.

[200] ——, "Experiments in map generation using markov chains," in 9th International Conference on Foundations of Digital Games, 2014.

[201] ——, "A hierarchical MdMC approach to 2D video game map generation," in 11th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2015.

[202] N. Sorenson, P. Pasquier, and S. DiPaola, "A generic approach to challenge modeling for the procedural creation of video game levels," IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 229–244, 2011.

[203] F. Southey, G. Xiao, R. C. Holte, M. Tommelen, and J. Buchanan, "Semi-automated gameplay analysis by machine learning," in 1st AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2005.

[204] Square, Final Fantasy, Game [Nintendo Entertainment System], Square, 1987.

[205] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," in International Conference on Machine Learning, 2010.

[206] N. Sturtevant, "An argument for large-scale breadth-first search for game design and content generation via a case study of fling!" In 2nd Workshop on Artificial Intelligence in the Game Design Process, 2013.

[207] A. Summerville, M. Guzdial, M. Mateas, and M. Riedl, "Learning player tailored content from observation: Platformer level generation from video traces using LSTMs," in Experimental AI in Games Workshop 3, 2016.

[208] A. Summerville and M. Mateas, "Sampling Hyrule: Multi-technique probabilistic level generation for action role playing games," in Experimental AI in Games Workshop 2, 2015.

[209] A. Summerville, B. Morteza, M. Mateas, and A. Jhala, "The learning of Zelda: Data-driven learning of level topology," in 10th International Conference on Foundations of Digital Games, 2015.

[210] A. Summerville, S. Philip, and M. Mateas, "MCMCTS PCG 4 SMB: Monte Carlo Tree Search to guide platformer level generation," in Experimental AI in Games Workshop 2, 2015.

[211] S. Swink, Game Feel: A Game Designer's Guide to Virtual Sensation. Morgan Kaufmann, 2009.

[212] Taito, Space Invaders, Game [Arcade], Taito, 1978.

[213] S. Tekofsky, P. Spronck, A. Plaat, J. van den Herik, and J. Broersen, "Play style: Showing your age," in IEEE Conference on Computational Intelligence and Games, 2013.

[214] J. B. Tenenbaum, C. Kemp, T. L. Griffiths, and N. D. Goodman, "How to grow a mind: Statistics, structure, and abstraction," Science, vol. 331, no. 6022, pp. 1279–1285, 2011.

[215] M. Thielscher, "A general game description language for incomplete information games.," in AAAI, vol. 10, 2010, pp. 994–999.

[216] C. Thompson. (Aug. 2007). Halo 3: How microsoft labs invented a new science of play.

[217] C. Thurau, K. Kersting, and C. Bauckhage, "Convex non-negative matrix factorization in the wild," in 9th IEEE Conference on Data Mining, 2009.

[218] J. Togelius, M. J. Nelson, and A. Liapis, "Characteristics of generatable games," in 5th Workshop on Procedural Content Generation in Games, 2014.

[219] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in IEEE Symposium on Computational Intelligence and Games, 2008.

[220] J. Togelius and N. Shaker, "The search-based approach," in Procedural Content Generation in Games, Springer, 2015.

[221] J. Togelius, N. Shaker, and G. N. Yannakakis, "Active player modelling," in 9th International Conference on the Foundations of Digital Games, 2014.

[222] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 172–186, 2011.

[223] E. Tomai and R. Flores, "Adapting in-game agent behavior by observation of players using learning behavior trees," in 9th International Conference on the Foundations of Digital Games, 2014.

[224] E. Tomai, R. Salazar, and R. Flores, "Mimicking humanlike movement in open world games with path-relative recursive splines," in 9th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2013.

[225] M. Treanor, "Investigating procedural expression and interpretation in videogames," PhD thesis, University of California, Santa Cruz, 2013.

[226] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "The micro-rhetorics of Game-O-Matic," in 7th International Conference on the Foundations of Digital Games, 2012.

[227] J. Tremblay, A. Borodovski, and C. Verbrugge, "I can jump! exploring search algorithms for simulating platformer players," in Experimental AI in Games Workshop, 2014.

[228] J. Tremblay, P. A. Torres, and C. Verbrugge, "An algorithmic approach to analyzing combat and stealth games," in IEEE Conference on Computational Intelligence and Games, 2014.

[229] P. Ulam, J. Jones, and A. K. Goel, "Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents," in 4th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2008.

[230] Valve Corporation, Half-Life 2, Game [Windows, Xbox, Xbox 360, PlayStation 3, Mac, Linux, Android], Valve Corporation, 2004.

[231] N. Van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber, "Robust player imitation using multiobjective evolution," in IEEE Congress on Evolutionary Computation, 2009.

[232] D. Ventura, "A reductio ad absurdum experiment in sufficiency for evaluating (computational) creative systems," in 5th International Joint Workshop on Computational Creativity, 2008, pp. 11–19.

[233] D. B. Wallace and H. E. Gruber, Creative People at Work. Oxford University Press, 1989.

[234] G. Wallner and S. Kriglstein, "Visualization-based analysis of gameplay data – a review of literature," Entertainment Computing, vol. 4, no. 3, pp. 143 –155, 2013.

[235] G. Wallner, "Sequential analysis of player behavior," in CHI PLAY, ACM, 2015, pp. 349–358.

[236] C. D. Ward and P. I. Cowling, "Monte Carlo search applied to card selection in Magic: The Gathering," in IEEE Conference on Computational Intelligence and Games, 2009.

[237] T. B. Ward, S. M. Smith, and R. A. Finke, "Handbook of creativity," in, R. J. Sternberg, Ed. Cambridge Univ Press, 1999, ch. Creative Cognition, pp. 189–212.

[238] N. Wardrip-Fruin, Expressive Processing. The MIT Press, 2009.

[239] G. A. Wiggins, "Searching for computational creativity," New Generation Computing, vol. 24, no. 3, pp. 209–222, 2006.

[240] G. N. Yannakakis, M. Maragoudakis, and J. Hallam, "Preference learning for cognitive modeling: A case study on entertainment preferences," IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, vol. 39, no. 6, pp. 1165–1175, 2009.

[241] G. Yannakakis and J. Togelius, "Experience-driven procedural content generation," IEEE Transactions on Affective Computing, vol. 2, no. 3, pp. 147–161, 2011.

[242] J. Young and N. Hawes, "Learning micro-management skills in RTS games by imitating experts," in 10th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2014.

[243] D. Yu, Spelunky. Boss Fight Books, 2016.

[244] H. Yu and T. Trawick, "Personalized procedural content generation to minimize frustration and boredom based on ranking algorithm," in 7th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[245] G. Zoeller, "Game analytics," in, M. Seif El-Nasr, A. Drachen, and A. Canossa, Eds. Springer London, 2013, ch. Game Development Telemetry in Production, pp. 111–135.

VITA

Alexander Zook is a Senior Data Scientist at Blizzard Entertainment. He has studied applications of artificial intelligence to game design during his Ph.D. and worked in the games industry since 2010. As an intern at Bioware Austin and Blizzard Entertainment he developed models to segment types of player behaviors and use this to inform game design and development decisions for *Star Wars: The Old Republic* and the *World of Warcraft*. As a Senior Data Scientist at Blizzard Entertainment he has pioneered applications of recommender systems to game content, developed innovative matchmaking algorithms, and helped design ranked gameplay systems. He co-organized the Experimental AI in Games workshop series to foster ongoing innovation in applications of AI to games and started the AI Game Jam to encourage the development of new applications of AI to games.