universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Prototypical – A Board Game Development Framework“

verfasst von / submitted by

Vincente Andrew Campisi

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Abstract

Despite an iterative workflow and common characteristics, board game design differs from its video game counterpart in the tasks and frequency with which digital technologies are leveraged. We sought to develop a software application explicitly intended to support the board game design process and to identify why and how it might be used. While commercial solutions already exist, our efforts attempted an intuitive but novel approach by making use of proprietary concepts such as *component selectors*, *actions*, *checks*, *conditions* and *filters*. We created Prototypical, a board game development framework, to provide modelling, simulation and analysis features when prototyping and developing individual board game mechanics. Three examples were used to demonstrate its current capabilities and Monte-Carlo Tree Search AI agent performance, while a user study involving five participants and System Usability Scale surveys identified the need for user interface improvements. The ability to aggregate and display information is expected to help the designer test assumptions and make better-informed decisions. Our software demonstrates potential to achieve this and presents ways to inspect individual and evolving game state information.

Kurzfassung

Trotz einem sich wiederholenden Workflow und gemeinsamer Eigenschaften, unterscheiden sich Brettspieldesign und Videospieldesign durch die Aufgaben und Häufigkeit der Verwendung von digitalen Technologien. Es wurde versucht, eine Software, die den Brettspieldesignprozess unterstützt, zu entwickeln und zu eruieren, warum und wie sie verwendet werden könnte. Da kommerzielle Lösungen dafür schon existieren, war das Bestreben einen neuen aber intuitiven Ansatz, durch Nutzung von eigenen Konzepten wie *component selectors*, *actions*, *checks*, *conditions* und *filters*, zu bieten. Zur Modellierung, Simulation und Analyse bei der Entwicklung einzelner Brettspielmechaniken wurde Prototypical, ein Brettspielentwicklungsframework, erstellt. Anhand dreier Beispiele wurden aktuelle Features sowie die Performance des Monte-Carlo Tree Search Algorithmus demonstriert. Durch eine Nutzerstudie mit fünf Teilnehmern und die System Usability Scale Umfrage wurde die Notwendigkeit einiger User Interface Verbesserungen identifiziert. Die Strukturierung und Abbildung von Informationen über ein Brettspieldesign soll als Hilfestellung für den Entwickler dienen, um Hypothesen zu testen und fundierte Entscheidungen zu treffen. Unsere Software weist das Potenzial auf, dies zu fördern und liefert dementsprechende Möglichkeiten, sich entwickelnde Spielelemente über verschiedene Zeitfenster zu überprüfen.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1. Introduction

1.1. History and Motivation

With the increasing popularity of video games and the commercial success of the industry [45], the terms game design and game development are often used in the context of software. There is an abundance of choice when it comes to technologies meant to support the creation of digital games, ranging from game engines such as Unity [73] and Unreal Engine [74] to individual programming libraries focused on improving workflow. The game design process itself is naturally iterative, involving an initial and final version, the latter reached after navigating multiple intermediate versions through modifications.

The developer begins with an idea and initial design, then repeatedly performs playtesting and makes changes until a desirable result has been achieved. The process often includes human playtesters who participate in gameplay and provide valuable feedback about the user experience. This can be affected by a range of factors, such as the complexity of the game rules, number and types of game elements (e.g. components, characters and actions), the appearance and the interface through which it is played. While the video game industry has an estimated value of over 300 billion USD [1], other types of commercial games do exist and are worth considering.

Board games are a popular form of entertainment and commonly-explored subject of professional and academic efforts. They can be viewed as a physical counterpart to digital games because similar and/or equivalent concepts are observable in both mediums. Regardless of type, all games have several qualities in common and when comparing video games and board games, it is clear that the development process is equally as iterative. Time, resources and human assistance are required, while it is rare for a significant portion of this overhead to be offset by current technologies.

In contrast with the variety of available digital technologies to assist video game development, board game designers tend to take an analog approach. Digital solutions are used sparingly, such as to assist the formatting or production of physical components (e.g. art or publishing software) [11]. Another use might be to create a digital implementation of an existing physical game. With this discrepancy in mind, we attempted to create software with the explicit purpose of assisting the development of physical board games.

The scope was a single game mechanic at a time in the context of an iterative development approach. We assumed that changes in aspects of a game design can be used to estimate corresponding effects on the gameplay and user experience. We do not claim to have observed specific metrics or guidelines for interpretation, however. Using such software was expected to help the designer save time through automated playtesting and provide insights with comprehensive game state information. AI agents were seen as a way

to lower overhead normally associated with traditional playtesting. We also assume that a well-informed game designer is more likely to make superior design decisions throughout the development process. It is this perspective that Prototypical strives to enable.

We sought to create a software application to assist the board game designer during the iterative development process and analyzed why and how the software might be useful. We simulated the playtesting of game mechanics for several examples using an algorithm that is effective without significant domain knowledge. We then evaluated the utility of the obtained results.

Prototypical is the name of our GUI-based software application, which allows the user to model and simulate game mechanics through the input of information about game components, component state and interactions, success conditions and simulation parameters. The heuristic Monte Carlo tree search algorithm is used to handle decision making in the simulations. The user is given the ability to inspect game state minutiae and discover trends from which the effects on user experience might be inferred.

1.2. Synopsis

This section provides a brief description of each chapter and serves as an overview for the structure of this research. This paper does not include research from the field of Game Theory. Our focus is on physical, non-theoretical turn-based board games intended to be played by one or two individuals. All of the presented examples will cover the latter. There are countless applications relating to games and game development and it is impossible for the survey performed here to be exhaustive. The use of this software requires basic computer knowledge, the ability to interact with websites and web-based applications and an understanding of the concepts in 3.1.

Chapter one introduces the context and problem this research seeks to address, while noting a few limitations and assumptions for this research. The general relationship between video games and board games is mentioned and this synopsis provides an overview of the structure and focus of each chapter.

Chapter two covers existing work relating to this research. Existing literature in areas such as game properties, mechanics, balance and user experience is surveyed. Lastly, research using the Monte-Carlo Tree Search algorithm for decision making in games is covered.

Chapter three introduces fundamental concepts relating to games and artificial intelligence. A short description of Game Studies is provided. Game properties are described and the concept of user experience in the context of board games is introduced. The relationship between game properties and the resulting user experience is then argued and the claim that playtest data can be used to discover useful information is made. The necessity of artificial intelligence algorithms for decision making in the context of this research is explained. The Monte-Carlo Tree Search algorithm and UCT are introduced.

Chapter four thoroughly describes the Prototypical framework. Primary features are listed and implementation details provided. The implementation of Monte-Carlo Tree

Search is then explored. Hardware, programming languages and other implementation-specific details are also mentioned.

Chapter five introduces three use cases for the Prototypical software and the results from user surveys. A walk-through of implementation details for each is presented. The methods used for the user trials are recounted and the resulting user experiences are evaluated. An overall analysis including capabilities, successes and failures of the software is presented and discussed.

Chapter six contains final statements about this work and its contribution to the related areas of research. Improvements or new features are also described. These include new or enhanced AI algorithms, other supported game types and UI adjustments. Additional information regarding the deployment and future availability and use of the Prototypical application is provided.

2. Related Work

2.1. Literature Review

To discuss the game development process, fundamental characteristics inherent to games of all types must first be understood. [26] investigated details of games and contributed more than thirty characteristics by which board games may be differentiated from one another. An analytical framework to do so was presented by [17]. Comparisons attempted to reflect significant influences on generated player experience and focused on the four main traits of rules, randomness, representational backdrop and interaction. Efforts have been made to precisely measure luck across varying types of games, including modern commercial board games such as Race for the Galaxy and Seasons [31]. [35] provided an abstract analysis of games and presented a rules typology which applies to games of all types. Reasons for playtesting, as well as the link between game rules and the gameplay experience itself were described. Games were broken down into five major elements: components, procedures, environment, theme and interface. While a detailed implementation was lacking, the research served as a basis for linking gameplay mechanics and user experience.

The relationship between user experience and mechanics was further explored through an analysis of board game metadata, which included details about game mechanics, community rankings and user ratings from the BoardGameGeek website [59, 12]. Ten thousand board games for the forty year period of 1980-2020 were considered, with the efforts exploring trends in specific board game characteristics present in the industry (Empirical Analysis over Time), correlations between rankings and game properties (Correlation of Board Game Attributes) and interactions between specific mechanics for potential co-occurrences and mutual dependencies (Interactions of Board Game Mechanics). Positive correlations existed between complexity and number of mechanics, as well as between game rating and number of mechanics; however, most were moderate or negligible. The metadata may not have been representative of the entire community or accurate for older games.

Additional correlations between specific game metrics, the Information Theory concept of Relevant Information, identifiable design flaws and player experience have been suggested [58]. A method to approximate Relevant Information in a game design was presented, supporting the idea that a system implementing such functionality would prove useful when evaluating and/or balancing game mechanics.

[34] provided examples of common playtesting approaches for games and provided ways to recognize and address gameplay imbalances, which often survive to later stages of a game's development. AI playtesting through restricted play was utilized to develop a novel

playtesting software prototype which permitted the measurement of gameplay balance and rapid iteration. It was possible to guide the board game development process and improve both measurable and perceived gameplay balance using a digital tool. While our research is based on this premise, we do not focus exclusively on simultaneously-played two player perfect information games and our implementation is not tightly coupled to specific games.

Gameplay balance itself has been a popular research topic, which has included efforts to differentiate between implicitly and explicitly balanced games, as well as statically balanced and dynamically balanced games. Attempts to dynamically achieve game balance and reconcile gaps in player skill levels were at times successful, which led to a positive impact on a player's feeling of success [37]. It was hypothesized that multiple balanced designs for a given design are possible and an integrated (i.e. manual and automated) balancing process for different games was presented. When applied to a video game, it was demonstrated that automated balancing led to a greater number of satisfactory designs in less time (compared to manual playtesting), including previously unconsidered ones [8]. Such a system applied to board games could potentially assist in the design process, possibly discovering relationships between interconnected game parameters.

Similar efforts included the application of population-based meta-heuristics to map generation for the board game Terra Mystica [18], framed as a combinatorial optimization problem with two approaches: particle swarm optimization and steepest ascent hill climbing with a random restart. Resulting maps were checked for rules compliance as well as balance of initial and overall conditions for all players. It was concluded that manual playtesting feedback was necessary, despite successful generation of compliant maps. Similarly, Sentient Sketchbook was presented as a map sketch generation tool with an intuitive interface, real-time feedback and suggestions (generated by feasible-infeasible two-population paradigm and novelty search) which assisted game designers to create video game levels [41, 76].

Leveraging digital technology to support the game development experience is a mixed-initiative approach to a design experience, which is the topic of other efforts relating to game balancing. A case study on applying digital technologies to the board game development process was carried out and the extensibility of existing games was demonstrated by using digital technologies to conceptualize and manufacture modularized game components. These were used to generate new game layouts [48]. Differences in balancing digital and physical games has been described and techniques to utilize digital technology for the latter have been explored [3]. A resulting user study concluded that dynamic adjustments when reconciling player skill level differences led to a more positive overall player experience.

Other approaches included a time-saving Java-based programming framework to assist with digitally implementing multiplayer board games [40] and the FlexibleRules [28] framework. Results of the first showed that participating designers had a superior experience with the software compared to without it. The latter allowed the user to model board games and implement games digitally, such that game logic and implementation details were decoupled. It used a two-part structure consisting of a conceptual model of a

game design and a set of tools to implement it. Software capabilities were demonstrated using three games - Awele, Go and Himalaya - and characteristics such as entity types, rules complexity, user interaction and game variants were compared. The desired end result of both frameworks was a fully-playable digital implementation of a board game. Other mixed-initiative digital tools include Tenagra [64] and Ropossum [61], a software tool to assist in the design of 2D platforming levels and another to help create and test physics-based puzzle game levels. Procedural generation and verification was used in both of these solutions.

In considering a potential lack of research on measuring user experience in games, the Game Experience Questionnaire was applied as part of a mixed-method approach (i.e. using quantitative and qualitative methods) to a non-digital board game [5]. It was seen to be effective in measuring user experience, despite an original context of purely digital games. The results indicated a link between positive user experience and what can be considered good game design. Similar efforts tend to exist in the context of video games, such as Riot Games' increasing use of quantitative data from games to scientifically optimize a given design, with the goal of improving player experience [54].

The intersection of artificial intelligence and games has led to a significant body of research. Before the successes of AlphaGo in 2016 [2], Objective Monte-Carlo [16] incorporated move-selection and backpropagation improvements into the existing software Mango [44]. Monte Carlo tree search was originally proposed as a general game-playing artificial intelligence framework superior to hand-designed algorithms which make use of evaluation functions and domain knowledge [15]. The highly randomized and weakly-simulated approach was applied to turn-based, perfect information classic board games, modern turn-based, imperfect information board games with stochastic elements and real-time video games with complex and evolving state. The titles were Go, Settlers of Catan and Spring, respectively.

Similar efforts include the implementation of an MCTS agent for the full rule set of 7 Wonders, which performed better than a standard rule-based AI agent [56]. MCTS incorporating Upper Confidence Bound Applied to Trees (UCT) was implemented for the two player board game Mr.Jack [46]. Performance of MCTS agents with various improvements - including discretization and chance events - was compared for the game of Ticket to Ride [32].

UCT [13] was originally introduced to improve on vanilla Monte-Carlo move planning, which it achieved through selective sampling of actions, whereby a balance between testing the most-promising candidates and ignoring sub-optimal ones was achieved. In the context of deterministic games, it was concluded that UCT always converges faster to the lowest failure rate in all tested scenarios [36]. Additionally, knowledge-based decision-making with probabilities derived from domain knowledge was incorporated into move selection [13].

It was clearly demonstrated that performance of MCTS can be improved by using pseudo-random move selection (i.e. heavy playout) instead of uniform probability move selection during simulations. The motivation for such an approach referenced the strong tactical ability and weak global awareness of domain-knowledge dependent solutions, with

Monte-Carlo demonstrating the opposite. Other improvements to vanilla MCTS exist, such as mixmax backups and partial expansion, both of which were applied to the video game Super Mario Bros with satisfying performance results [33].

As the complexity and performance of game-playing AI agents grew, so did attempts to incorporate them in software and the game design process. Building on previous projects related to the application of artificial intelligence to the game of Go, [27] presented Fuego, an open-source software framework which provided functionality for developing full-information two-player board game agents using game-independent Monte Carlo Tree Search. Originally presented in [23, 24], new features of the Machinations framework [42] were introduced [25], all of which continued to assist the user in simulating gameplay mechanics of early prototypes to promote balanced game design. Machinations generates and utilizes diagrams to model the internal economy of game mechanics. Artificial players can simulate playtest behavior until specified end conditions are reached, and dynamic charts can display game information, such as resource values. The unreleased game SimWar was used to demonstrate that a subset of a design can be implemented using the framework, potentially helping the game designer during the prototyping stage of development. While the provided example was a digital game, all concepts would apply to board games. Further details regarding the Machinations tool will be covered in 2.2.

LUDOCORE, a "logical game engine" [63], served as a bridge between game design and conventional logic-based AI tools. It provided both higher-level syntax to describe games and achieved "a concise representation of a game's mechanics" [63], along with potential to playtest and simulate gameplay. Real-time graphics capabilities and input were leveraged in previous research [62] for the software BIPED, which produced both a playable prototype and a formal rules system from a single game description. Machine playtesting was found to be useful in combination with normal human playtesting and, collectively, both applications provided helpful feedback to board game designers during the early stage of development.

Using the game of Blackjack, [19] algorithmically generated heuristics which fit the profile of (or could easily be followed by) novice players. Some of the approaches were too resource intensive or could not easily be scaled up for games of higher complexity but these efforts have potential implications for measuring game balance by analyzing the performance of a profiled AI agent for a given game design.

[49] proposed seven different strategies complementary to playtest data in which AI and visualization techniques might help a game designer to extract knowledge from a game design. It was shown how using game traces can be used for tracking game state information, debugging and improving designer understanding of the dynamics of a given design. Thresholds for specific design constraints were determined using a logic-based branch-and-bound approach.

Among other examples of AI technology used to assist in balancing games, [43] studied the deck-building card game Dominion and showed that specific cards contribute to the perceived game balance, irrespective of player strategy. This was accomplished by comparing the competitive performance of three profiled AI agents. These findings should be applicable to other types of games and the area of automatic game design.

[38] designed and implemented a digital collectible card and board game hybrid, using an AI agent to play and progressively modify the game components. Such procedural balancing provided lackluster results but could potentially be applied elsewhere with more effectiveness, including non-digital games. [21] used an AI agent-based approach involving A* and MCTS to model 4 distinct playing styles of the game Ticket to Ride and analyze its mechanics, using data from eleven variants of the board game. Two game states not covered by the game rules were identified and it was demonstrated how small modifications to game entities could have drastic effects on preferred game strategies and gameplay itself. Such changes were verified to have a greater effect compared to simply altering the number of players. Another instance in which previously unknown corner cases were discovered involved exploring the potential of using AI agents - two custom gameplay strategies, A* and MCTS, respectively - to simulate gameplay in order to save time and resources during the board game design process. It was concluded that these efforts pointed toward the possibility of a future "robust system that aids modern board game designers" [20].

While not specifically focused on balancing games, events such as the General Game Playing AI competition [52] are proof that interest exists in these topics. Participants in the yearly event have been tasked with implementing artificial intelligence agents using the Video Game Definition Language (VGDL). The agents played video games without knowing the rule set beforehand. VGDL allows the modeling of game objects, their interactions and termination conditions, much like our own board game development framework.

2.2. Existing Software

We will now provide an overview of several software applications which assist or are otherwise involved in the development of board games. We believe this information to be comprehensive but not exhaustive. Some entries may be used with more than one objective in mind, but it can be helpful to group them according to those which support the game development process, assist playing digital implementations of games and those which do both.

The Machinations framework is similar to our approach and offers a web-based visual programming language to generate game simulation flow charts and view related information. It does not require programming knowledge and is Turing-complete, utilizing nine node types and two connection types for its economy-based flowchart perspective [42]. It allows specific game parameters to be monitored during random or Monte-Carlo-based simulations using generated charts (e.g. histogram). Such charts support plotting either raw values of node data or statistical operations. Currently supported operations are mean, median, minimum and maximum values. Either a random or Monte-Carlo-based simulation type can be selected.

In the context of prototyping, Machinations seeks to provide the designer with tools to simulate emergence in a given system and balance player experiences. As with our own framework, the end result should be an improved ability to spot problematic elements

of a game mechanic and make informed design decisions. We attempt to explore an approach not involving flowchart construction while still supporting similar functionality. Machinations is well-documented and provides both written and video tutorials, however, it still provides an initial impression of high complexity. We assume that some game designers might not require the extensive features provided by Machinations, especially those which target the design and prototyping of video games or projects designed in a corporate setting involving many participants.

Currently only available as a technical preview during active development, Boardgame Lab is a web application which allows the user to prototype and playtest tabletop games [10]. If all features are implemented as described, it will consist of three main focus areas - prototyping, playtesting and automation - and provide the user with similar functionality to Prototypical.

Iterary is a web application which provides basic prototyping tools and allows the user to model limited game mechanics and simulate results. Examples of components include card, die and table, all of which track textual information regarding state. Simulations do not utilize a specific AI algorithm. Game state can be shared in the form of a snapshot, accessible by a customized URL. Extensive log data is available. The appearance and feature set is rudimentary and, besides the most basic use cases, no useful insights regarding balance of the mechanics can be easily gained.

Tabletopia [72] is a virtual tabletop system with a suite of tools to assist designing, playing and publishing board games. It uses a graphical interface and no programming knowledge is required, but there are seemingly no tools related to simulations, gameplay balancing or rules enforcement. Tabletop Simulator [71] is comparable, but is rendered in a 3D environment. It lacks the direct publishing options of Tabletopia, but can be easily used for prototyping [57] and incorporated into the design process.

Vassal [75] is a free open-source tabletop game engine written in Java, popular for playing community-made digital implementations of physical turn-based card and board games. There is no modeling of game rules but there is extensive logging and scripting which allows the user to provide quality of life improvements such as automatic game setup and custom events. Live play through the internet or turn-based play by E-Mail is available, often paired with Voice over IP communication systems. ZunTzu [77] is also popular and provides networking and voice communication for online play. Both of these applications are intended to help the user play digital versions of existing games, but can potentially be used to some extent during the prototyping phase of game development.

Additionally, BoardGameGeek community members have discussed several other applications which assist the process of creating physical board game components during the early prototyping phase of development [9]. nanDECK [47] is software intended to speed up the process of designing and creating physical cards. Squib [65] is a domain-specific language in Ruby which has similar uses and even describes itself as “nanDECK done the Ruby way” [65]. It simplifies the process of preparing printable images from card descriptions. Neither of these applications allow the user to model gameplay information or run simulations.

3. Game and Artificial Intelligence Concepts

3.1. Game Concepts

Game Studies, also referred to as Ludology, can be defined as "a discipline that studies games in general" [29] and it is responsible for a wealth of research on the development and play of games. It provides a theoretical framework with which games of all types and their development process may be analyzed. While we defer requests for precise definitions of ludological terms to other authors, it is important to clarify first-hand our understanding of several concepts central to this research and make evident the resulting scope.

We begin with a general description of games. We hope to review terminology for relating these concepts with user experience. It is our expectation that a basic understanding of game design and its elements can be formed, making it possible to explore why and how software might be used in this context.

Games can be considered to have several equally-important elements such as mechanics, story, aesthetics and technology [60]. Mechanics are the rules of the game which provide structure and direction to participants, while story is the series of events and decisions that are underscored by the mechanics. Aesthetics and technology relate to how and in what ways a participant interacts with the game. These four elements form a product and resulting experience, created as an extension of a designer and conveyed to the participant. The product in this case is a physical or digital entity which, when engaged with according to the predefined rules and through the prescribed interface, dynamically generates a series of events and decisions. We refer to this entire process as gameplay.

The entity in the case of physical board games is the collection of physical pieces which serve to facilitate gameplay when used in conjunction with the game rules. Common physical objects include cards, dice and tokens and are referred to as components. A component may also be used interchangeably for abstract objects, however, such as a player area or token bag of arbitrary size and shape. Game participants themselves could be considered abstract game components. In the game of Chess, the black and white pieces, game board and even the individual fields which the pieces occupy can be considered components. The result of this classification is that components form the subject group of all action-related events and conditions which take place during gameplay and can be viewed as the subject or direct object of gameplay actions.

Game rules provide the set of conditions and permitted decisions from which gameplay events emerge. Ideally, all potential game states, including uncommon ones, are addressed. Rules are meant to guide player decision-making by imposing constraints where, if they

are too restrictive or too liberal, the subjective player experience might suffer. There is no single metric to apply here, as it is for the designer to determine the rules of the game so that they convey the desired experience.

The observations, reactions and perspectives of a participant determined through gameplay are considered to be the user experience. Some examples of concepts relating to user experience include difficulty and fun. The first refers to user skill and how easily dexterity with the involved tasks may be achieved, whereas the second relates to a sense of satisfaction and the enjoyment obtained through participation in gameplay.

Balance is a concept which exists at the intersection of mechanics and user experience. It is both partially objective and subjective and can be described as the relationship between engaging player decisions and meaningful outcomes. Player agency in gameplay determines events and their effects to varying degrees, based on factors such as chance, player skill and the limitations or handicaps inherent to (or otherwise permitted by) the rule set. User experience can be negatively influenced if the effects of a player decision are unobservable or insignificant in succeeding game states, which is also possible if the perceived changes are not proportional to the resources, risk or effort required for the respective decision path. There are many more potential sources of discontent in user experience which we have not mentioned here. If poor balance is due to bias in player perspective, such as a lack of game rules understanding, it can be said in this case to be primarily subjective. If, however, it is due to the functional qualities of the game system, such as disadvantageous resource allotment or asymmetric effectiveness of otherwise equivalent actions, it can be said to be at least primarily objective.

We have mentioned several elements which relate to gameplay balance, but there are many more quantifiable aspects of games to be considered. These include component count, supported player count, action structure, chance, timing and other measurable aspects of game economy. As mentioned, it is possible for there to exist a direct or indirect relationship between these parts of a design and the resulting player experience. This research makes the assumption that analyzing quantifiable game data may potentially provide insights into the subjective user experience during gameplay. Furthermore, we propose that it may be possible for the designer to leverage such information during the development process.

Developing turn-based board games is an analog process, using software only selectively compared to other types (e.g. video games). In developing a board game, the designer firstly defines the components, mechanics and objectives. To explore the interactivity between these elements, they play the game with the help of volunteers using handmade physical components. Playtesting helps to discover gameplay anomalies and deficits through critical user feedback. It helps elicit improvements to be incorporated into the next iteration of the design. The process is continued until the desired result has been achieved.

However, playtesting physical board games is a challenge. Time constraints and geographical limitations notwithstanding, the circle of playtesters might never expand to more than just a few individuals, possibly also consisting exclusively of friends and personal acquaintances of the game designer. This can lead to biases in the design. Playtesting

can be described as "the process of trying out a game, examining what worked (and what didn't), and making the necessary adjustments to improve the players' experience." [22]. Gameplay at least initially occurs under direct visual supervision and/or participation of the designer and feedback is often direct. Co-creator of Pandemic Legacy and Restoration Games designer Rob Daviau recalls board game playtesting being compared to "inviting people over so they can tell you that your child is ugly." [22]. This dynamic differs strongly from impersonal or anonymous video game metadata and feedback forms which can exist entirely through online channels. While the internet might prove useful for Do It Yourself board game component creation (i.e. "print and play"), the final deliverable is a physical product intended to be experienced in person. There is the distinct possibility that playtesting in a digital environment does not map appropriately to the same user experiences in real life with the physical version. Additionally, creating physical game components requires effort, time and skill.

3.2. Artificial Intelligence Concepts

Drawing its name from the famous casino in Monaco, the Monte-Carlo method can be described as a class of algorithms which rely on random sampling of a given problem space to arrive at a best estimate of the optimal solution. There are many possible implementations and improvements for vanilla Monte-Carlo, but the problems for which it is used are often deterministic in nature and centered on two distinct categories: optimization and approximation.

In the context of board games, we find ourselves confronted mostly with the latter. More specifically, the Monte-Carlo method has become a popular choice when designing artificial intelligence agents for games. It has demonstrated both results and future potential when applied to the task of discovering optimal moves for a given player and turn. In this research, we concern ourselves with an implementation of Monte-Carlo called Monte-Carlo Tree Search. It is a search algorithm which uses heuristics and random simulations to approximate the ideal move from a given game state.

A tree consisting of one or more nodes at one or more levels is constructed, where each node represents a given or potential game state. Each edge between nodes represents a possible state-changing action and each level deeper in the tree represents a potential future game state. In the context of a two-player turn-based perfect information game, the root node represents the current game state. Each edge connection between two nodes represents a possible action that could be taken by the current player of the first node state which would result in the game state represented by the second node. The current player for a given state alternates at each level in the tree; therefore, the game state perspective at each level is specific to the current player and it changes accordingly. This is of interest when considering that each node also contains information regarding simulation counts and traversal history. As the algorithm executes, this information is propagated towards the root of the tree and updated at each relevant node. This influences how future traversals of the tree are performed.

MCTS consists of four phases: Selection, Expansion, Simulation and Backpropagation.

These phases execute sequentially and are repeated until either a specific duration of time has elapsed or a desired number of iterations has completed. It is referred to as an "anytime" algorithm, implying that a result is always available, regardless of execution time or iteration quantity. However, the quality of the result - i.e. how closely the algorithm approximates the optimal solution of the given problem space - generally improves over time. These characteristics make MCTS suitable for situations in which there is no hard limit for execution time or instances in which the availability of a result must be guaranteed at varying intervals. Pseudo-code can be seen in algorithm 1.

Data: Current state S_t, selection policy $P_{selection}$, rollout policy $P_{rollout}$, exploration policy $P_{exploration}$, playout policy $P_{playout}$, backpropagation policy $P_{backpropagation}$, best node policy P_{best}

Result: Node for estimated best action a, containing state S_{t+1}

$root \leftarrow S_t$;
while $completedIterations < maxIterations$ ***and*** $elapsedTime < timeLimit$ **do**
  $child \leftarrow$`selectNode(`$root$, $P_{selection}$`)`;
  `rollout(`$child$, $P_{rollout}$`)`
  $child_{child} \leftarrow$`selectNode(`$child$, $P_{exploration}$`)`;
  $result \leftarrow$`playout(`$child_{child}$, $P_{playout}$`)`;
  `backpropagate(`$result$, $P_{backpropagation}$`)`;
end
return `selectNode(`$root$, P_{best}`)`;

Algorithm 1: Monte-Carlo Tree Search Pseudo-code

A starting state S_t is the first input. It must make accessible all relevant contextual information necessary to structure and carry out simulated behavior. In the context of a turn-based board game, this would involve information regarding game rules, participating components and other implementation-specific information.

The remaining inputs represent the policies used at the various stages of the algorithm to determine the respective behavior. $P_{selection}$ defines how the tree is traversed and ultimately how a leaf node is chosen during the selection phase. $P_{rollout}$ describes how many child nodes - each corresponding to a potential move from the selected node - are to be created. $P_{exploration}$ refers to how one of these newly-added nodes are to be selected as the starting point for the playout phase and $P_{playout}$ determines how such playout simulations are conducted. $P_{backpropagation}$ describes how the results of the simulation are stored upwards throughout the tree and P_{best} determines how a node is chosen as the final result. The chosen node will always be an immediate child node of the root. While defined here individually, it is possible for two or more of the stated policies to implement the same strategy. Such would be the case if randomness were used in both the selection and playout phases, resulting in random leaf node selection and light playouts.

We see that the current state is cloned and stored as the root node of the tree. This also preserves the starting state and prevents the hypothetical states of MCTS from interfering. It is at this point that the main loop of the algorithm begins. The four phases of the algorithm execute until the computational budget has been reached. This can be specified as a number of iterations to be performed, a time limit to be reached, or both. The suitability of each may vary between theoretical and practical contexts.

The Selection phase is responsible for choosing the next node in the tree that is to be explored. It begins at the root node and navigates down the tree according to a given tree policy until a leaf node has been reached. This type of node does not have any child nodes and should not represent a terminal state (i.e. a state for which a terminating condition has been reached). The most basic selection policy chooses candidates which have not yet been considered and candidates with the greatest win potential (i.e. highest win rate). However, the best move may not always fit into either category, a phenomenon which has led to variations and policy improvements.

Those of Upper Confidence Bounds Applied to Trees are arguably the most popular in the context of board games. Instead of selecting the best candidate firstly according to visit count and secondly according to win rate, UCT attempts to balance the likelihood between exploration and exploitation. In this context, exploration refers to a tendency towards selecting nodes which have not yet been visited, and exploitation refers to those which have demonstrated the highest win rates. UCT will be covered in more detail at the end of this section.

Once a candidate has been selected, the Expansion phase begins. For each possible action, the resulting state is added as a child node to the current one. Nodes are initialized with metadata indicating a win rate and visit count of zero and are updated throughout the execution of the algorithm. One of these new leaf nodes is chosen according to a desired policy. For example, this might be done randomly.

A simulation is then started from the node during the Simulation phase. This is done according to the desired policy, which can be classified as either “light” or “heavy” playout. The first indicates that all moves during a simulation are chosen randomly and domain-specific knowledge is not necessary. Heavy playouts may make use of domain knowledge in evaluation functions or heuristics, making light playouts more suited for handling multiple game types and general game playing. An MCTS agent using light playouts would be easily applied to different game rule sets compared to heavy playouts because the latter would require adjustments to adhere to the unique rule sets. Moves are chosen until a result has been reached or a time limit has been exceeded, at which point, the final phase of MCTS begins.

Backpropagation is the last of the four phases and it serves to populate the results of the simulation upwards through the tree. After updating the win rate and visit count information of the current node, the scores and visit counts of the preceding nodes are also updated. It is important to consider the aforementioned alternating (per ply) current player during this phase. Depending on the game type and player count, a simulation result may need to be stored according to the perspective of the current player. For example, two-player zero sum games would require a win for the current node and

player to be stored as a loss for the other player in the previous and following nodes. Implementation-specific details to address this problem are provided in 4.2.2.

As these four phases execute, the tree expands while favoring either nodes with a higher win rate or nodes which have not yet been explored. This poses a problem, however, as there may be a discrepancy between the immediate and long-term value of a move. Value is relevant at both short-term and long-term gameplay horizons, however, these may not be equal at every point in time. Stated differently, a possible move may appear to have a low value for the current turn, but this does not necessarily imply that it will never lead to a highly advantageous state. This poses a dilemma, namely, how can the short- and long-term potential values of a given move be reconciled during run time? It is of course possible in some situations to fully solve the given problem to determine this. However, many games are highly complex and usually there are not enough computational resources for MCTS to do so. As mentioned earlier, Upper Confidence Bounds applied to Trees (UCT) attempts to address this in a relatively straightforward way.

One of many such improvements for the basic MCTS algorithm, UCT utilizes the perspective of a multi-armed bandit problem. This is a type of problem in which a single action may be taken among several, for which one or more potential payoffs are unknown. To understand UCT, it is worth first exploring characteristics of this problem type in more detail.

Multi-Armed bandit problems illustrate the dilemma of how to balance the exploitation of promising actions against the pursuit of those with hitherto unknown payoffs at the current decision point, while seeking the greatest overall long-term gain. The common analogy is that of a gambler presented with several slot machines of unknown payoffs, whose personal goal is clearly to maximize winnings in the given period of time.

As in this scenario, one or more actions are mutually-exclusive and there is a probability distribution for the payoff of available choices, where one or more are currently unknown. Any discrepancies may lead to more information regarding the payoffs of alternate (future) actions. The terms greedy and non-greedy are also relevant, in that the first prioritizes actions with high values and the second focuses instead on exploring to better estimate the average.

The payoff – or value – of an action is provided by equation 3.1 [67, 68]. At a decision point t, A_t is the selected action and R_t is the resulting reward. Therefore, if the chosen action is a, the value $q_\times(a)$ can be defined as the expected average value of its corresponding reward.

$$q_\times(a) = \mathbb{E}[R_t | A_t = a] \tag{3.1}$$

Since the true value of an action can be approached by repeating it and continuously averaging the resulting values, the estimated value of an action a at time t is given by equation 3.2 [55, 68]. The number of instances before time t in which action a was chosen is reflected by n. The respective reward at each time step is R_i. The cumulative nature of this estimated value means that it will converge faster and closer to the true value with

increased sampling frequency and more total samplings. This mirrors the performance profile of MCTS, where more iterations usually provide better results.

$$Q_t(a) = \frac{1}{n} \sum_{i=1}^{n} R_i \tag{3.2}$$

Now that the value of an action has been described, we are ready to consider the UCT equation 3.3. The UCT value can be described in three components, where W_n is the estimated value of the action corresponding to the node. Across many simulated playouts, this is the win ratio of the node and may vary by context. For zero-sum game simulations, it might be given by the number of wins plus draws minus losses, divided by total simulations. The second component is a constant value parameter C, relating to exploration. A common value for this is usually an approximation of $\sqrt{2}$, which is the theoretical value [36]. It is also possible to dynamically adjust this parameter during run-time. The last component is the square root of the given fraction, which is a ratio relating to visits of the parent node and visits of the current node. The natural log of parent visits is the numerator, while the number of visits to n is the denominator.

$$UCT_n = W_n + C \times \sqrt{\frac{lnV_p}{V_n}} \tag{3.3}$$

The first component W_n reflects the tendency toward exploitation of promising nodes, whereas the second C and third $\sqrt{\ln V_p / V_n}$ collectively reflect the tendency toward exploration of lesser-known ones. The UCT is repeatedly calculated and it improves over time.

4. Prototypical – A Board Game Development Framework

Prototypical provides a feature set intended to allow the user to implement and study one specific game mechanic at a time. Depending on the complexity of the target game, implementation of a complete game model may be possible. There exists no prescribed workflow or strategy when implementing mechanics in Prototypical and it is up to the designer to follow a modelling approach or set of guidelines. Similar to a programming language, Prototypical provides the user with a set of tools and a smaller set of enforced conventions, expecting the user to best apply these to the task at hand. While we make no attempt to suggest or otherwise outline a particular workflow, we will provide an overview of features and several use cases with example protocols.

There are often multiple ways to represent a given game entity or mechanical element. Current features of Prototypical can be broken down into the categories of modelling, simulation and analysis. It should be noted that words corresponding to entities from the Prototypical framework will be italicized and literal values will be in bold font.

4.1. Modelling

Modelling describes the process of representing game elements using the features and constraints of the Prototypical framework. While such entities may vary, Prototypical classifies all entities relating to a game design as one of either *component*, *action* or *condition*. Additional objects, such as *selectors*, are proprietary and specific to the experience of using the Prototypical framework. This is commonly due to software design choices. We will now describe each available type of *component* in detail.

4.1.1. Components

All *components* have a single *component type*, zero or more properties and zero or more child *components*. *Properties* can be used to reflect game-related information in the *component* state and they consist of one pair of a *property* key and related value. The set of *properties* or respective values may change over the course of gameplay simulation, depending on the configuration of the applied related *actions*. Child *components* can be assigned and/or removed, depending on the type of the parent and any *conditions* assigned to the *actions* which involve those in question. Said differently, the set of possible children for a *component* is determined by both its type and the set of *conditions* prerequisite to the execution of the *actions* configured to interact with the involved *components* during gameplay.

Component states collectively contribute to what constitutes the overall game state at any given time. Game state can be thought of as the aggregate of all *component* states at a specific step during simulation. *Group*, *field*, *card*, *deck* and *d6* are the five types which are currently supported and will now be described in this respective order.

The *group*, as the name implies, is for logically organizing any number or combination of *components*. It provides a single entity under which others are assigned as children. *Groups* may or may not correspond to a true organizational entity or phenomenon in the underlying game design and might exist solely to enable the implementation of a mechanic with respect to the details and constraints of the Prototypical framework. In other words, the user may choose this type to represent a small number of physical entities classified together out of convenience, such as tokens in a resource pool. This means it might not correspond to anything from the game design itself. If not directly mirroring the underlying design, a *group* would instead be used in a meta-informational sense to make implementation easier; for example, when tracking specific elements. Another example might be a *group* containing both players, making it easier to write and execute a *compound query* to analyze the sum of player scores across simulations. See 5.1.1 and 5.1.2 for examples.

Field components are intended to represent a discrete sub-area of the physical space in which a game takes place. They do not have inherent size or shape, although this can be represented through *properties* as necessary. As with *groups*, they may contain any number of children – including other *fields* – but such behavior can be guarded by *conditions* attached to any *actions* which involve an *add* or *remove* interaction (4.1.3) for which the given *field* is a receiver. A user may choose *field components* when representing areas of a game board or other logical units of two- or three dimensional space in which game pieces would normally be placed or otherwise arranged. For example, a Chess board could be modeled as a *group* containing 32 child *fields* with a *property* with name **color** and value **white**, and another 32 child *fields* with this set to **black**. Each *field* could then contain the respective algebraic notation coordinates (e.g. LOC:A4) as one or more additional *properties*.

Card components represent their physical counterparts which are usually two-sided rectangular pieces of thick paper or cardboard with printed images and/or text and game-related information. In the Prototypical framework, a *card* can be assigned as a child to another *component* of a different type, but it is not possible to hold child *components* itself. Any relationship of this type could be represented using *group*, *field* or *deck components* instead. For example, a *group* containing several *card* children could be used to represent a player tableau.

It is sometimes necessary to model game characteristics or states involving cards which are not inherent to the *card* component itself. This would be the case when one might be rotated 90 degrees to indicate a specific state. It is possible to use *properties* to manage this information, even if the states are closer to *component* metadata than actual entities. The user is free to discover and use any preferred approaches within the confines of the Prototypical framework.

Analogous to a stack of physical cards, the *deck* is the most appropriate way to group

multiple *cards* together. Behavior when drawing child *cards* from or placing them into or onto the *deck* can be set through any relevant *actions* and *conditions*, as well as the *selection strategy* of the relevant *component selectors*. While a *deck* functions similarly to the *group*, it is sensible to use both so that the language surrounding an implemented mechanic closely mirrors that of a physical design. In other words, using a *deck* to model what would normally be referred to as a deck in physical designs is simply meant to prevent confusion and the need for mental mapping between Prototypical framework components and their real-life counterparts.

The *d6* is intended to represent a cubical six-sided die, possibly the most common type found in tabletop games. The name is derived from the first letter of the word "die" and the face count of six. Following this same convention, d8, d10, d12 and d20 are other common types of dice found in tabletop games; however, these are not currently supported in the framework. It is of course possible to do this as part of future efforts. *D6 components* provide a random number value stored in the **value** *property* from **1** to **6**, inclusive.

4.1.2. Conditions

A *condition* reflects a *component* state at a given time interval through the evaluation of a set of one or more criteria. The application of a criterion yields a **true** or **false** result, which is combined to determine whether the given *condition* is currently met. It is possible to classify all *conditions* as either simple or complex, which refers to quantity and respective inter-relationships and logical structure. An example of the first might be **Card 1 is red**, which, when evaluated against the respective *component* state at the current time, would provide either **true** or **false**. The latter could range from three such *conditions* combined by a Boolean **AND** operator (where it is true only if all three are true) or could itself be a combination of complex *conditions* joined by varying Boolean operators. Prototypical does not enforce any limitations on such constructions but they may eventually become unwieldy to use.

Complex constructions are often used for *conditions* which trigger the end of a game, commonly known as victory conditions. A clear instance of this is the victory condition for the Pokemon Trading Card Game, which is met when any of the following are true: a player has drawn six prize cards, a player has no fielded Pokemon or a player must draw a card but does not have any remaining in the draw deck. The second and third sub-conditions can be further expanded, demonstrating the potentially recursive structure of composite arrangements.

All *conditions* have a corresponding type, provide a Boolean result when evaluated according to a provided state and optionally return information about which player will benefit once it is met. *Simple conditions* also contain information about which *component* holds the relevant state, in the form of a *component selector*. While there exists only one type of *simple condition*, complex *conditions* are represented by *and conditions*, *or conditions* and *none conditions* (or any combination thereof). The type is one of these four, while the name is a user-specified label which may or may not correspond to the underlying game design. Due to potentially multiple ways of expressing equivalent

Comparator	Symbol	Description
GT	$>$	greater than
GTE	$>=$	greater than or equal to
LT	$<$	less than
LTE	$<=$	less than or equal to
EQ	$=$	equal to
NE	$\neq$	not equal to

Table 4.1.: Comparators and corresponding symbols

conditional logic, the *conditions* of a game mechanic implemented in Prototypical may not exactly match those of the design itself. This is not a reason for concern, as long as the overall set of possible game states is mapped consistently between the model and the design.

A *simple condition* serves as a building block for complex *conditions* and it is incapable of containing subordinate *conditions* (i.e. *sub-conditions*). It contains a singular conditional statement, known as a *check*, which provides a Boolean result when assessed. A *check* might represent a statement such as **The count Property of the Discard Pile Deck component is equal to 120**. The result of the *check* directly determines whether the overall *condition* is met. If the *check* evaluates to **true**, the *simple condition* is met; likewise, if the conditional statement is **false**, the *simple check* is not met and it evaluates to **false**.

The conditional statement of a *check* consists of a value that is to be assessed using the given *comparator* against another reference value. This triple is also known in Prototypical as a *property filter*. The first value, *comparator* and second value are all mandatory but are non-exclusive. Accordingly, a *check* must have exactly one of each, but multiple *checks* may share similar (or even identical) constructions and are in such cases equally valid. The values correspond to the current and expected value of the same *component property*. The comparison makes use of one of the six supported *comparators* listed in table 4.1. Each corresponds to a relational operator. The Boolean result of a *property filter* therefore determines the Boolean result of the encompassing *check*.

The second type of supported *condition* is the *and condition*, which combines multiple *simple conditions* using a Boolean **AND** operation. The Boolean result of an *and condition* is therefore determined by the Boolean results of the respective *sub-conditions*, where **true** is only given if all subordinates evaluate to **true**. If any evaluate to **false**, the result of the encapsulating *and condition* will be **false**. There is no limit to the number of *sub-conditions* one *and condition* may store, nor are there rules regarding types or orderings. This means that a single *and condition* can store a mix of *condition types*, including *simple conditions* and *and conditions*.

Similar to the above but using a Boolean **OR** operation is the *or condition*. The result of an *or condition* is true if any of its *sub-conditions* evaluate to **true**. To illustrate, it would be possible to create an *or condition* with one hundred complex *sub-conditions* where, if the first 99 fail but the very last one passes, the overall result would evaluate

to **true**. Once again, there is no limit to the size or characteristics of the set of *sub-conditions* that can be configured. For the example above, this would also mean that the *true sub-condition* could exist at any index and the ordering of the others is irrelevant.

The last type is the *none condition*. This is the inverse of the *or condition* and **true** is returned only if all *sub-conditions* evaluate to **false**. Said differently, if one or more *sub-conditions* evaluate to **true**, the encompassing *none condition* will evaluate to **false**. As with the two previous types, there is no limitation on the quantity, types or structure of the stored *sub-conditions*.

As mentioned above, *conditions* compare a given *component* state against expected thresholds and can be either met or unmet at a given point in time. The *component* state is determined partly through the use of what is called a *component selector*. This is an entity which, when applied to the current game state, returns a set of *components* which fulfill the set of specified criteria. This comparison is performed by sequentially filtering the set of all *components* according to one or more conditional statements and including each entry for which all are met. Both a specific *component*, quantity and selection ordering process can be specified, among other information. We will now review the details which comprise a *component selector*.

Component selectors contain a grouping of additional criteria, called a *filter*. The specified information is used to require values and characteristics of a *component* that could possibly be returned as part of a *component selector* result. If any provided details are not matched by the candidate result *component*, it is ignored and excluded from the result set. Each of the currently-supported characteristics will now be described.

If a single *component* is sought and its id is known in advance (i.e. before simulation execution), it is possible to specify this in the **id** field of the *filter*. No other *filter* or *component selector* information is required in such cases and will be ignored, as ids are unique per project and remain unchanged.

If known ahead of time, it is possible to specify the id of the *component* which maintains a parent relationship with the possible candidate *component(s)*. This is done using the **root** field of the *component selector filter* and can help to quickly narrow the possible candidates. For example, if an *action* were to involve drawing cards from a designated *deck component*, the respective *component selector* would only be concerned with those *components* which are children of the *deck*. Another example might be *actions* involving *components* belonging to a specific player, where those of other players could be completely ignored by the relevant *component selectors*. With this in mind, it might be useful to also think of **root** as the parent id.

Property filters have already been covered above, but a set may be optionally specified in a *filter* as *properties*. If any of the specified entries evaluate to **false**, the encompassing candidate *component* is excluded from the *component selector* result set. In other words, all *property filters* must evaluate to **true** in order for the *component* to which they belong to be included in the result set.

Expected types of candidate *components* can be specified using *types*. The default value is a set of all supported *component types*, but the user may specify any combination between this and a set containing a single type. It is useless to provide an empty set, as all

components have a mandatory value assigned upon creation and therefore the respective *component selector* would always only return an empty list.

The *selection strategy* of a *component selector* or *action selector* determines the ordering of the preliminary result set of *components* or *actions*. Once a list of result candidates has been compiled according to the above criteria, it is mutated according to a specified strategy before being returned. In effect, it enables the user to describe the priority with which otherwise equally-valid result entities should be used. This becomes relevant, for example, when *actions* require certain *components* to be prioritized over others when implementing a specific procedure (e.g. drawing a random card). Without the ability to specify how a result set should be ordered, it would not be possible to replicate such behavior, potentially leading to situations in which all *components* are selected in the same order for all *actions* across all simulations. This would not yield effective simulations.

The complexities of available *selection strategies* for *components* and *actions* are found in table 4.2. The *first* strategy has the lowest and completes in constant time, as no changes are made to the result list before it is returned. The *last* strategy reverses the ordering of this data and therefore is linearly dependent on the quantity of result candidates. This same complexity applies to the *random* strategy, which arbitrarily reorders the result data before returning it.

Selection Strategy	Complexity
first	$O(1)$
last	$O(n)$
random	$O(n)$

Table 4.2.: Complexity of available *selection strategies*, Big O notation

The default strategy is **FIRST**, which does not perform any ordering operations and simply returns the result set as it is. In cases where a single *component* is to be selected from a preliminary result set of size two or greater, this strategy will return the first element of the preliminary set (i.e. entry with the lowest index). The **FIRST** strategy is often useful when gameplay actions do not require specific ordering.

Opposite to this is the **LAST** strategy, which reverses the ordering of the result set. The entity at the first index becomes the last, the second becomes the second to last, and so on. Similarly, when a single *component* is to be selected, the entity located at the largest index of the preliminary result set will be returned.

RANDOM applies a randomly-driven reshuffling of the result set indexes. This could be used in conjunction with a *deck*, so that child *cards* are removed in a way which mimics a shuffled *deck*. If either of the **FIRST** or **LAST** strategies are used, there is potential for the same cards to always be returned. This could even happen across different simulations, making this functionality useful.

Quantity is used to set a hard limit for the size of the resulting set of *components*. This allows the user to expect an exact maximum number of matching entities. It is possible for the user to specify **1** if only a single *component* is expected to match. If the *quantity*

value is larger than or equal to the number of resulting *components*, the entire set will be used. If it is smaller, the result set will be truncated so that its size is equal to the *quantity* value. There are two ways to conceptualize this: items at larger indexes in the result set are dropped; or items are taken from the beginning until the size of the result set equals the *quantity* value and/or the result set does not contain any more values. These two descriptions may differ in implementation but, for all intents and purposes, the results are equivalent.

4.1.3. Actions

Actions drive forward the gameplay experience by means of discrete events and the resulting changes in game state. If game states are represented as nodes, *actions* can be conceptualized as the transitional edges between them. The starting state of an *action* in this case is the node before the edge, and the resulting state is the node after the edge.

Any subordinate *actions* or *action events* of which the *action* is comprised are applied successively along the edge until their collective result is reflected in the new state. An initial state (i.e. before it is applied) reflects no *sub-actions* or *action events*, while a resulting game state reflects all such state changes. With this in mind, the application of an *action* may result in several intermediate or preliminary states temporarily existing along this transitional edge; but these are not visible to the user. In considering a possibly nested hierarchy, an *action* is represented by an edge, which itself contains one or more *action events* and the potential for nested *actions*.

There are several types of *action* entities which enable the modelling of complex gameplay events and the respective changes in game state. Functionality and properties common to all *action* objects in Prototypical will be covered next, followed by a detailed description of each type.

Actions in Prototypical provide the ability to check if they may be carried out for a given game state, as well as attempt to do so. If one can be applied, it is equivalent to claiming that it is valid or that it can be performed. Whether or not an *action* may be applied to the current game state, however, depends on its type and the state itself. This varies and will be covered in detail for each. Assuming this prerequisite has been met, however, performing an *action* results in also performing any subordinates and invoking the respective *action events*. During this process, the last *component* involved is always taken into consideration. *Sub-actions* are simply nested *actions*, whereby they also adhere to the same characteristics. An *action event* is one of several types, all of which can be described as the elements of an *action* which collectively affect the changes in a game state. The types and structure of *action events* will be detailed shortly, but for now they can be considered the procedural steps by which an *action* transitions between the game states before and after it is performed. To summarize, an *action* may be one of several types but all provide information on whether they can be performed on a given starting state and what the resulting state would be after they have been fully applied.

Before reviewing the various types of *actions* available, we will first list the information which is present in all instances. Every *action* contains exactly one *component selector*, used at run-time to determine which *component(s)* may be a receiver of the action.

Receiver, just like in software programming, indicates the primary *component*, or subject, of the *action*. It is the *component* which carries out the *action*. For example, it would be the *deck* in an *action* where a *card* is added to a draw deck (e.g. **DrawDeck add Card1**). The *component selector* is used to differentiate between *actions* that are possible for each of the players during a simulation. Although potentially empty, *actions* contain the list of *sub-actions* which are to be performed during execution. The behavior by which they are carried out depends on the type of the containing *action*. Because some result in non-deterministic or random behavior, a *chance* value is stored for each *action*. This contains a number reflective of the upper limit of possible outcomes for the given *action*. For example, an *action* in which a *d6* is rolled would have a *chance* value of six (*6*). A default value is used if none is provided. For *actions* involving multiple *sub-actions*, the *last component* value makes available the *component* involved in the preceding *sub-action*. This is useful for *actions* containing *sub-actions* which all act on the same entity. For example, a **Discard to Discard Pile** *action* would need to have a *sub-action* to first remove a *card* from a player area and a second to place this same *card* in the discard area. To ensure that the same exact *component* is used throughout both steps, *last component* is used. Finally, all *actions* contain information regarding their type. We will now cover each of these in detail, using the following order: *and action*, *or action*, *ordered eager fold action*, *ordered breaking fold action* and *simple action*.

An *and action* is valid only if all of its *sub-actions* are valid. This type of all-or-nothing approach allows for it to depend on several areas of a game state. For example this might be used to model game rules such as "score one point and draw a card" and "roll a die, lose that number of resources and lose one point". The first example consists of two steps – "score one point" and "draw a card" – while the second uses three different steps: "roll a die", "lose that number of resources" and "lose one point". When an *and action* is performed, its *sub-actions* are performed in sequence and the information about the last-used *component* is updated at each interval.

The *or action* is considered valid if any of its *sub-actions* can be performed, making it less strict than the previous. When an *or action* is performed, the set of valid *sub-actions* for the current state is taken and each member is performed sequentially. It is therefore possible for the behavior of an *or action* to vary, depending on which of the *sub-actions* are valid. It is useful for modelling player decisions where only one of several possible choices may be selected. For example, if a game design allows the player to either roll a die or draw a card, this could be implemented in Prototypical as an *or action* named **Roll 1d6 or draw 1 Card**, where the first *sub-action* is an *action* to roll a *d6* and the second is an *action* to draw one *card*.

Despite the seemingly complex name, the *ordered eager fold action* is similar to the *or action*, whereby it is valid if any of its *sub-actions* are valid. It differs only slightly when it is performed, because an attempt will be made to sequentially execute every *sub-action*. This behavior can be summarized as performing as many *actions* as are possible. Instead of filtering before iterating, this *action* type will at least try to execute every *sub-action* in order. If an invalid entry is encountered, it is simply ignored and iteration continues normally.

For cases in which the user would want to stop immediately after encountering an invalid *sub-action*, the *ordered breaking fold action* may be used. The behavior of this can be summarized as performing until one fails, then immediately stopping. An *ordered breaking fold action* is considered valid if the first *sub-action* is valid.

As its name implies, the *simple action* is the most atomic *action* in Prototypical. The other types may contain nested entries, but entities at the deepest levels of such hierarchies will always be a *simple action*. The list of *sub-actions* for this type is therefore always empty, but there is additional information not found in other *action* types. A tuple consisting of a *condition* and an *action event* is used to store event information, which is used during a simulation to make changes to the game state. This information can also be described as a type of logical if/then statement, where the *condition* determines if an event should be applied to the game state, and the *action event* determines how the game state should be modified. *Conditions* have been previously covered in 4.1.2, so we will now describe *action events* in more detail.

There are three parts of which an *action event* is comprised. They collectively describe the type of event that is to occur, an initiating *component* and a targeted *component*. The latter two are represented using *component selectors* and are referred to as the *receivers selector* and *targets selector*, respectively. If we consider the receiver, event and target using the analogy of language, the receiver is the subject, the event is the verb and the target is the direct object. An *action event* describes a change in the relationship between the receiver and target by means of an event, the effects of which depend on its respective type. Although there are three different types of *action events*, all contain such information and are used to modify the state of involved *components* and, by extension, the overall game state.

The *receivers selector* is a *component selector* providing the set of *components* which serve as the subject of the event in question. Similar to *actions* as described above, a receiver here indicates that a resulting *component* is the subject or initiating entity of the event. An example of this would be a *deck* when removing a *card* from its set of children.

The *targets selector* is also a *component selector*, but one which instead provides the set of *components* to be acted upon. Target in this sense describes the *component* serving as the direct object of the event. In continuing the above example, the target would be the *card* itself that was removed from the *deck*.

Exactly how a game state is changed through an *action event* depends on both the type involved, as well as the acting *components* from the *receivers selector* and *targets selector*. This means that the same type of *action event* may cause different behavior when any of the involved parameters are different. See tables 4.3 and 4.4 for a breakdown of compatibility and behavior for combinations of currently-supported *components* for *add action events* and *remove action events*, respectively. The rows indicate the type of the *component* given by the *receivers selector*, and the columns indicate the *component* type given by the *targets selector*. The intersection of these is either empty to indicate an incompatible combination, + and the attributes which are to be incremented, **-** and the attributes to be decremented, or the phrase **update value** which indicates that the implementation-specific property would be updated according to the predefined behavior

of the relevant *component*. For example, *d6* assigns to a *property* a new random value between one and six, inclusive.

Despite such variation, general statements can be made about the original intent of each *action event* type. The *add action event* is meant to create a direct association with another *component*, often in the form of a parent-child relationship. Of the currently-supported *components* in Prototypical, all but *d6* behave this way. As mentioned, the latter instead assigns a new value to its **value** *property*. In all other cases, an *add action event* assigns the target as a child *component* of the receiver. For example, a player tableau can be described as a parent entity, where all associated items within are child entities. Unless already reflected in the initial game state, additional entities not currently belonging to the tableau may potentially be assigned as children through the execution of *add action events*.

Conversely, a *remove action event* is intended to undo such a relationship between the receiver and target *components*. While the same exceptions for the *d6* type must be considered, *remove action events* can otherwise be used to decouple *components* so that they can be associated with others. Gameplay is manifested in part through the changes in *component* relationships, which take form through *add action events* and *remove action events*.

The last of the currently implemented *action event* types, *update property action events* modify the state of only a single *component*. A *component property* is set to the specified value, and all relationships (or lack thereof) between *components*, as well as the *properties* of other *components*, remain unchanged. An example usage of an *update property action event* would be updating an arbitrary player score *property* upon the completion of a victory condition. *Deck components* make use of what can be described as an implicit *update property action event*, whereby the **count** *property* is automatically incremented upon each successful *add action event* and decremented upon each successful *remove action event*.

When using an *update property action event*, there are two ways in which the new value can be derived. The first, and simplest, is a *value expression*. This is used for static values which do not depend on any other element of game state and will always remain the same. For example, if an *update property action event* should always assign a value of **x** as seen in 5.1.3, a *value expression* would be appropriate. If, however, the new value is dynamic and it depends on some other state, a *property expression* can be used. Once configured with the name of a *component property* and a *component selector*, a *property expression* will retrieve the respective value from the result *component* at run time. This becomes useful when the effects of an *event* vary according to the preceding *event(s)*. An example use case for this might be adding the number of points from a *card* to a player score. A *property expression* in this case would use the name of the respective score property (e.g. **score**) and **null** for the *component selector*, assuming the target *component* in this case had been drawn immediately before.

	deck	*card*	*group*	*field*	*d6*
deck		+child,count			
card					
group	+child,count	+child,count	+child,count	+child,count	+child,count
field					
d6	update value	update value	update value	update value	update value

Table 4.3.: *add action event* compatibility and behavior. Row indicates receiver type, columns target

	deck	*card*	*group*	*field*	*d6*
deck		-child,count			
card					
group	-child,count	-child,count	-child,count	-child,count	-child,count
field					
d6	update value	update value	update value	update value	update value

Table 4.4.: *remove action event* compatibility and behavior. Row indicates receiver type, columns target

4.2. Simulation

After all entities and interactions have been modeled, it is possible to simulate gameplay. This is achieved by configuring and running *executions*, sometimes with the help of AI agents, to yield *execution results*. Requirements and details of these entities and concepts will now be covered.

4.2.1. Executions

An *execution* in Prototypical refers to a simulation of a modeled game design according to a set of parameters, collectively called an *execution context*. The *execution context* contains information such as the *component* which represents each player, all modeled entities of the game design, the simulation approach that is to be taken, and a *condition* which halts the *execution* when it is met. An *execution context* exists solely at the implementation level and is simply convenience terminology from a user perspective. Beginning at the starting game state and alternating for each player until the halting *condition* has been met, the chosen AI agent selects an appropriate *action* for each player and the current game state is updated with the result of its application. Intermediate states are saved and, along with other information, are used to construct an *execution result*. We will briefly cover in more detail the parameters within an *execution context*, followed by the

steps involved in running an *execution* and a listing of the currently-supported AI agents in Prototypical.

In order for an *execution* to be carried out, the user must first specify in the *execution context* the *component* corresponding to each player. For example, there might be two *groups* named **Player 1** and **Player 2**, where any *actions* and *components* relating to the first player would be associated with the **Player 1** *component*, with a similar setup for the **Player 2** *component*. This information is required because for each step in an *execution*, the next move must be chosen with respect to the currently-active player. The id of the first and second player *components* are used to instantiate an internal object in Prototypical called a *turn counter*. A *turn counter* is used during *executions* to manage the active and non-active players, but it is not visible to the user. It stores the id of each player, sets the first as the initial active player, and provides functions to both get the current active player, as well as swap the active and non-active players.

While perhaps an obvious inclusion, all modeled entities of the game design are also required in order to carry out an *execution*. This includes, but is not limited to, all *components*, *actions* and *conditions*. Internally, Prototypical manages entities using a respective service, such as *component service*, *action service* or *condition service*. This is again unseen by the user but it may be of interest to know that such service classes prevent accidental duplication of entities and any unexpected behavior. They are instantiated once per project, which is why entity ids are often only unique at the project-level.

Besides specifying the acting and modeled entities in an *execution*, the user must also provide some type of halting *condition* when creating an *execution context*. At each step of the *execution*, the *condition* is evaluated and, if met, no more *actions* are carried out. The current state is then evaluated and an *execution result* generated. The respective *condition* depends entirely on the motivations of the user and could be as simple as an arbitrary expression to always apply the same number of events; or, it could be complex and modeled after the victory condition of an existing board game. An example of the first might be a *condition* which halts the *execution* after a player takes a single *action*. An example of the second might be one which permits an *execution* to run until a player has drawn a specific number of cards, gained a set number of points or fulfilled one of several other *sub-conditions*.

The remaining information required for an *execution context* is the preferred AI agent. Prototypical currently supports a random agent and a UCT-based Monte-Carlo Tree Search agent. We will soon describe each of these in detail, but will first cover the general process by which an *execution* is carried out, then describe the structure of an *execution result*.

Once an *execution context* has been configured, it can be used to instantiate and carry out an *execution*. The relationship between these is one-to-many, as the same *execution context* may be used to create several *executions*. As described previously, the contexts are simply a grouping of parameters and can be thought of as a blueprint for an *execution*.

Carrying out an *execution* involves several steps which occur before, during and after a main simulation loop, which adheres to an execution timeout window. First, the initial game state is cloned so that it does not interfere with the modeled entities at the project

level. The *condition*, *turn counter* and all *components* and *actions* are grouped and copied to a *game* object separate from that of the initial state. A timestamp is then stored to mark the beginning of the *execution*. Continuing until either a terminal state has been reached or a timeout has been exceeded, the next move is performed and the resulting intermediate state is stored and used to evaluate the halting *condition*. At each iteration, the most recent game state is passed to the AI agent to determine the next move. If the AI is unable to suggest a next move, the *execution* is finished, as there are no possible *actions* remaining. If the timeout has been exceeded, the *execution* is also finished but with an **Error** status, as it was unable to complete in time. If, however, a move was made by the AI agent, the updated game state is returned and stored as both an intermediate state and the current game state. The halting *condition* is evaluated and the process continues unless it is met or the above timeout has been exceeded. The intermediate states are aggregated to provide a complete timeline of the evolution of the game state throughout the simulation.

Now that it is clear what must happen in order to create and carry out an execution, it is appropriate to discuss the related structure. *Executions* contain an *execution status* and an *execution result*. In Protoytpical, they are associated with an id unique to each *execution service*. Though overwhelmingly unlikely, this means it is possible for two *executions* to be assigned the same id if they belong to different Prototypical projects. For all intents and purposes, projects are treated independently in Prototypical, which prevents any conflicts from arising from this potentially duplicated data.

Execution status can be one of three values: **PENDING**, **DONE** or **ERROR**. The first of these is the initial value and is set during creation of the *execution* itself. It remains this value until either of the two other status have been obtained. **DONE** indicates that an *execution* has finished within the timeout value, which has a default timeout of two minutes. The final variant indicates that either the *execution* was unable to finish in the allotted time, or it has reached an exceptional state and can not be recovered. The exact reason for the failure is not specified here, but can be found in the second of the two fields of an *execution* (i.e. *execution result*). In summary, an *execution* which has not yet completed is expected to have a **PENDING** status, one that has successfully reached a valid terminal state within the configured timeout is expected to have a **DONE** status, and one that has failed exceptionally or exceeded the timeout is expected to have an **ERROR** status.

The *execution result* contains all information pertaining to a completed *execution* and is used to provide the analysis features described in 4.3. The id of the respective *execution* and its timestamps provide important information used to organize and sort the potentially multiple executions Prototypical might be managing at a given time. Ids are unique to each *execution result* on a per-project basis, similar to how *execution* ids are handled. The start time is a system timestamp in milliseconds, at a point shortly before the first turn is attempted by the AI agent. Similarly, the end timestamp is recorded immediately before the *execution result* itself is created and stored. As previously mentioned, the *execution result* may contain a message in order to provide additional details to the user. An example of this might be an error message indicating that the timeout has been

exceeded for an *execution* which finished with an error result.

In order to track the game state as it develops throughout an *execution*, two final pieces of information are stored in an *execution result*. The initial state is stored the same way as it is passed to the AI agent. To review, this is a bundle of information containing a *condition*, a *turn counter* and all modeled entities and respective state (e.g. *components* and *actions*). It follows that the information aggregated in the **initialState** variable of an *execution result* matches exactly the modeled game design at the moment before any *action* is applied. As the game state migrates away from this initial state during the course of an *execution*, changes are recorded through the generation of what Prototypical calls *frames*.

A *frame* is a grouping of two game states, where one state reflects the game before a specific *action* has been applied, and another state represents the game afterwards. In correlating these pre- and post-event states, Prototypical manages to track every change in state for the user. A list of *frames* allows the entire evolution of an *execution* to be reconstructed and evaluated. The post-event state stored in the *frame* at a given index in the list matches the pre-event state of the *frame* at the succeeding index. Comparisons to a linked list may be drawn. The initial, intermediate and terminal game states can all be encoded in the entries of a list of *frames*. The initial game state corresponds to the pre-event state of the *frame* at the first index, and the final game state corresponds to the post-event game state of the *frame* at the last index. With this understanding it is now possible to explore how intermediate states are generated.

4.2.2. AI Agents

We cover now in detail the two currently-supported AI agents in Prototypical: Random and Monte-Carlo Tree Search. For a general understanding on the latter, please refer to 3.2. Regardless of details specific to a given AI approach, all AI agents receive a complete snapshot of the current game state and return a similar snapshot which reflects the game state after a move has been chosen and applied. The state received as input includes information about the *execution* halting condition, both players (including active and non-active information from the *turn counter*) and all modeled *components* and *actions*. We will now describe how each AI approach operates within the Prototypical framework.

When the AI agent in an *execution context* is set to **Random**, an *action* from the set of possible *actions* for the game state and active player is chosen and performed. Given an indexed set of possible *actions* with size two or greater, however, an integer bounded by the size of the set is generated randomly and the *action* at the corresponding index is applied to the game state. If no moves are possible, none are performed. Winning player information is not required for top-level (or any) *actions*, as no domain-specific information is required when selecting moves randomly.

The Monte-Carlo Tree search agent – **MCTS** – implements the principles outlined in 3.2, constructing and performing a best-effort search of the tree of potential game states. The root node of the tree corresponds to the input game state. Immediate child nodes of the root correspond to the game state after which a respective possible *action* has been performed. For example, if the set of possible *actions* consists of drawing one of three

cards, the child nodes below the current root node would exclusively represent the game state after having drawn the first, second or third of the three possible cards, respectively. All nodes in the tree contain references to any parent and children nodes, as well as all data required for the calculation of the UCT value. The latter includes wins, losses, draws and total visits. In order to build the tree and generate these child nodes, the four phases of the MCTS algorithm are performed. They are repeated a set number of iterations, after which the most-visited node is returned. The concept of visits will be clear after reviewing how the MCTS phases are implemented in this AI agent.

In the selection phase, the most-promising child node of the current root node is chosen according to largest UCT value. If there are no child nodes, the current root node is selected. In order to compute the UCT value of a node, its wins, losses, draws and visit count are considered. The visit count of the parent node is also necessary; otherwise **0** is used if no parent exists.

Kotlin code for the *computeUct* function can be found in A.1. Note that Xi corresponds to node losses subtracted from summed wins and draws, all of which is divided by the node visit count. A decimal approximation of the square root of two is used for the constant c. All input values to the function are converted to the **double** data type in order to prevent inaccuracies from **int** truncation. Also note that if there are no visits to the node, the maximum supported **double** value is used instead [4]. This default value is an approximation of infinity and ensures that unvisited nodes take precedence over previously-visited siblings. Alternatives to this approach exist, however, such as considering this value a "First-play Urgency" hyper-parameter in conjunction with a UCB1 algorithm variant [30].

After a node has been selected, the expansion phase begins. All possible moves for the given state with active player are collected in list form and handled sequentially. Each *action* is applied to a cloned instance of the game state and a node is instantiated with the result. All such nodes are then added to the tree as children of the node corresponding to the current state. It is important for the active player to be swapped in new nodes, so that the integrity of the alternating two player turn structure may be maintained.

In order to partially handle stochastic events, *actions* during this phase may be applied multiple times until either a child node has been added for each possible resulting state for the given action, or a timeout has been exceeded. The chance value of an **action** determines how many nodes are expected to be added in total. For example, a *d6 component* involved in an *action* with a chance value of **6** can be expected to result in six added child nodes during this phase in total. The AI agent compares the resulting game states after the *action* has been applied and disregards duplicate or equivalent states with consideration to the value. In other words, this phase will continue adding only distinct child nodes until either their quantity equals the chance value of the *action*, or the overall timeout of the process has been exceeded. The default timeout is five seconds and it is not configurable at this time.

Before the next phase of MCTS can begin, a node must be chosen as an exploration candidate. To do so, the MCTS agent refers to the set of child nodes of the current node. This set contains any newly-created nodes from the preceding operations of this phase. If

the set is not empty, the entity at the first index is chosen. Otherwise, if there are still no child nodes through this point in the expansion phase, the current node is taken instead. This would be the case if there were no possible actions to be made from the current node. In other words, no moves for the given state and active player are valid and the current node is what is called a terminal state or leaf node.

The selected exploration node then becomes the focus for the next MCTS phase: simulation. It is named this way because a standalone simulation of gameplay is initiated from this state, which continues until either a terminal state is reached, or an overall timeout has been exceeded. It is isolated in that it operates on a cloned game state and does not directly influence the original game state which was passed to the AI agent at the start. The purpose of these simulations is to explore the possible space of the given game design and review data potentially indicative of strategically preferable moves for the active player with the current game state.

There are three possible outcomes for a simulation, not considering error states. Either the terminal state indicates a win for the active player, a loss for the active player, or a draw. The first two are determined by consulting the winning player information contained in the halting *condition* of the *execution context*. If the *component* id matches that of the current player it is considered a win. If it does not, it is considered a loss. If the halting condition has not been met, the result is considered a draw. Bear in mind that wins for the active player and losses for the non-active player (and vice versa) are functionally equivalent in zero-sum games.

In order to carry out a single playout, the MCTS agent attempts to perform a valid move at random for the active and non-active players in an alternating fashion until the timeout or halting *condition* has been met or there are no possible moves to make. The *turn counter* is used at each iteration to construct the set of possible moves for the given state and active player. If the set is not empty, one *action* is performed randomly, the active player swapped and the halting *condition* checked. If the *condition* has been met, the game state is evaluated and a playout result returned. Otherwise, the next iteration is started. If, however, the set of possible moves was empty, the game design does not permit anything else to be done and the halting *condition* is now checked. As with the above, if the *condition* has been met, the game state is evaluated and the appropriate result returned.

It is for this reason, however, that winning player information must always be contained in the *actions* stored in any *execution contexts* which make use of the MCTS agent. Evaluation of a playout state is only possible when the agent is able to interpret the terminal state. The playout result is then prepared for backpropagation.

The purpose of the backpropagation phase is to update the tree with the result data from recently-performed simulations. Remember that every node stores information to track the quantity of visits, wins, draws and losses. This data is incremented or decremented according to the results. Beginning at the terminal node at which the MCTS simulation finished, the visit count is incremented by one and the draw count is incremented according to the result value. The win and loss count of the node is either incremented or decremented, depending on the ply at which backpropagation is currently

being handled.

For a constructed MCTS tree involving two players and a strictly alternating turn structure, a ply corresponds to one level in the tree hierarchy. Beginning at either the top or bottom of the tree, each ply upwards or downwards in the structure corresponds to the turn of a different player. If the terminal node represents the turn by the first player, the ply above corresponds to the second player. This pattern repeats in both directions through the tree and is tracked throughout the backpropagation phase by means of a Boolean variable called **isCurrentPlayerPly**. The variable is invisible and inaccessible to the user but is mentioned here for clarification purposes. The value of **isCurrentPlayerPly** at the terminal node is **false** because at that point, the hierarchy reflects the current player having completed a move and play having already been passed to the other participant. When recording simulation result wins or losses, this ply information is taken into account. If the result indicates a win and **isCurrentPlayerPly** is **true**, wins are incremented. If the ply value was instead **false**, the win result is recorded as a loss. Similar logic is used when recording loss results.

After simulation result data has been recorded to the current node, both the simulation result data and ply information is passed to the parent node, where the same backpropagation procedure is carried out. It is important to note, however, that the ply Boolean value is inverted upon being passed to the parent node backpropagation procedure. This must be done so that the wins and losses remain accurate from the perspective of the active player to which the respective ply corresponds. If done improperly, the MCTS agent will not perform as expected, as it will incorrectly estimate the value of outcomes originating from the decisive actions. This is explained from experience and was observed during the initial implementation of the AI agent.

Backpropagation continues until there are no parent nodes left to which the procedure can be relegated. It is at this point that this phase ends and, if iterations remain, the MCTS phases continue to loop. Any nodes at which simulation result data was recorded are said to have been visited, which is reflected by incrementing the value.

Once the predetermined iterations of the MCTS phases are finished executing, the AI agent is prepared to choose a final result. This will be one of the *actions* originating from the root node (representing the current game state of the *execution*), which leads to the child node with the highest visit count. Because of the MCTS approach used in this agent, more visits correspond to a higher potential payoff originating from the *action* in question. In selecting the node with the highest visit count, the AI agent is performing a best-guess attempt to steer the user (in this case, the first or second player *component* from the *execution*) towards strategically advantageous branches of the tree of possible decisions and the corresponding game moves. If no child nodes exist, nothing (**null**) is returned because there is no possible *action* which can be performed.

4.3. Analysis

Analysis is arguably the most important feature set of Prototypical and it affords insight into the inner workings of a game design through several methods of viewing and querying

simulation data. With the intent that designers will make better-informed decisions during the development process while using Prototypical, there exist several focuses when it comes to the analytical insights currently available. These are state inspection, chart generation, informational queries and compound informational queries. The latter two have been implemented as objects named *queries* and *compound queries*, respectively. State inspection uses *frames*, described in 4.2, to provide the user with the ability to view any aspect of game state at any step of a completed *execution*. "Step" here indicates an arbitrary point during the run time of an *execution*. If the user wishes to track a state element across several steps during an *execution*, this can be done by constructing a *query* and generating and visualizing the resulting chart data. For insights across several *executions*, however, a *compound query* can be used to link and plot the information referenced by simple *queries*. It is even possible to plot multiple *compound queries* on the same chart. This set of features, along with those from 4.1 and 4.2, is expected to provide the user with unique data to assist the game design process.

Both *query* and *compound query* objects outline a request for a set of *execution result* data. Similar to a database query, *query* objects of either type contain parameters which are used to retrieve and modify a set of data and return the result. Prototypical uses these to allow the user to track specific elements of game state across various *execution result* data sets.

The user constructs a *query* object by providing the relevant information, such as a *component property* name for *queries* or *query* id for *compound queries*. The *query* information is then applied to the relevant *execution result* data by an internal service. All *queries* exist at the project level. Details regarding how each *query* type is handled by the service will follow, but for now it is important to understand that a set of data points is generated through this process. Each data point is a pair of values, where the first value corresponds to the x-axis of a generated chart, and the second value corresponds to the y-axis. To display *queries*, Prototypical simply plots the values of any active *queries* to a color-coded line graph and scales the axes appropriately. Scaling occurs when the set of active *queries* concerns itself with a smaller or larger set of values compared to those which were displayed before. For example, an *execution* involving only five steps would involve scaling different than one involving fifty steps. Scaling and the resulting chart is adjusted dynamically, whenever the set of active *queries* is modified. Note that a query is considered active if the user has marked it in the interface by checking the respective box. We will now explain what exactly is meant by state inspection, what type of chart can be generated, and how the basic *query* is constructed. Then we will describe the structure and details of each *query* type and how several *queries* are combined into a *compound query*.

4.3.1. State Inspection

State inspection allows the user to view *component* state details for any step of a completed *execution*. It is possible to view the state either before or after the relevant *action* of the *frame*, labeled **Before** and **After**, respectively. These labels also include the active player name for that Frame. An example would be **Before Player 1 Action** and **After**

Player 1 Action, where the name of the active player for this *frame* is **Player 1**.

Both of these provide a complete list of *components* and the ability to individually view state information for each at the given step. The *component* list includes the *component* name and type. Upon inspecting an individual entry, this information, along with *component* id and any *properties* are listed. In effect, the user is able to focus on the the most-interesting areas of a game state at an arbitrary point of an *execution*. This information, however, is not editable and reflects state that no longer exists outside of the containing *execution result*.

Prototypical offers the ability to visualize the state information data. Currently, line graphs can be generated through the use of *queries* and *compound queries*; however, other chart types can easily be added in the future. Before covering details on how a chart is populated, it should be clear how to interpret the resulting visual data. All charts in Prototypical are dynamically generated and consist of an x-axis, y-axis and several data points connected by lines. The x-axis represents the steps of the relevant *execution(s)* and the y-axis represents the numerical values derived from one or more *query* and/or *compound query* objects. *Query* objects are used to track the numerical value of a specific *property* throughout the course of an *execution*. *Compound query* objects are meant to combine multiple *query* objects by means of a mathematical operator and track the calculated value across two or more *executions*.

More details on each type of *query* object will follow, but for now it is important to understand that Prototypical dynamically generates a chart in order to display the relevant data. The user selects which *execution* and *query*/*compound query* objects to plot by marking the respective elements as active. A chart displays data for multiple *queries* or any number of *compound queries*, given that all are applied to the same exact set of *execution result* data.

Because an *execution result* may contain lists of *frames* which vary in length, it is important that any applied *queries* or *compound queries* point to data sets similar in size. For example, unlimited *queries* may be applied to generate a chart, as a *query* may only ever refer to a single *execution*. It would be problematic for one *compound query* to refer to a data-set that is shorter than another, as there would be insufficient data points to combine. An example use case for generating a chart using normal *queries* might be to plot the score of each player for a single *execution*. An example utilizing *compound queries* might average these same *queries* across multiple *executions*. For clarification purposes, we use the term *query* to refer to simple query objects, *compound query* to refer to compound query objects, and query in lowercase to refer generally to either type.

4.3.2. Queries

The purpose of a *query* is to track a specific *component* property for one or more steps of an *execution*. The utility of Prototypical presents itself once multiple *queries* are plotted and the user is able to track more information than would otherwise be practical manually. Several parameters are required to construct one, which will now be listed.

Every *query* requires a name, which may contain any characters (alphanumerical or otherwise) of any length and must be unique per project. We suggest using a name

that is clearly indicative of the purpose and scope of the *query* itself. An example of an undesirable name would be **MyQuery**, whereas a useful name might be **Player 1 Score After**. It is immediately clear in the latter example that this *query* will plot the **score** *property* of the **Player 1** *component* at each step of an*execution* after the respective move has been made.

Because the scope of the *query* can vary, both a start and end index is needed. The *start index* value determines the first *frame* (i.e. step of the *execution*) that is to be plotted, where any preceding elements are simply ignored. Similarly, the *end index* value determines the last (i.e. largest) index of the data values which are to be plotted (exclusive). Both *start index* and *end index* values are optional and, if omitted by the user, will default to values such that all available data points are included. In other words, a *query* will always plot the entire set of *execution result* data, unless configured otherwise.

If a *query* should track a *component* property, it follows that a *component selector* and *property* name are required during construction. The **receiver** field holds the *component selector* which returns the *component* possessing the *property* to be observed. Please note that this *component selector* must always only return a single result. If no results are returned, the *query* lacks the necessary data; likewise, if two or more are returned, it would not be clear which *component* was originally desired. The *property* name must refer to an existing *property* for the resulting receiver *component* for the same reasons; however, *properties* are unique and the latter situation holds less relevance. In summary, both the *receiver* and *propertyName* fields are mandatory and must each refer to a single *component* and *property*, respectively.

The final field of a *query* is *query temporality*, an enumerated value which indicates whether the desired information should be observed before or after the *action* of the encapsulating *frame* was applied. Remembering that each *frame* contains both a **Before** and **After** game state, the *query* must be configured to focus on only one. This value may be useful when comparing the effects of an *action* on game state. The underlying enum, therefore, has only these two values, where the second is the default: **BEFORE** and **AFTER**.

Now that the structure of a *query* has been outlined, we will describe how they are used by the internal service to generate chart data. With both the *query* and relevant *execution* id at hand, the service selects the *frames* according to the start and end indexes. Iterating through this selection of *frames*, the game state is extracted according to the *query temporality* enum and the desired *component* is found. The *property* matching the *property name* is mapped to the respective *frame* index. This results in a map of integer and string values, where the integer is the *frame* index and the string value is the value of the property for the referenced *component* at this index and *query temporality*. As an example, suppose we have three *frames* and a *component* property which starts at zero and increases by one each *frame*. The resulting data would be a map of three entries, where entry one has the integer zero as its first element and **"0"** as the second value. The entire map in this case could be represented as follows: **{ 0: "0", 1: "1", 2: "2" }**. It should now be clear how *query* data can be generated and visualized.

4.3.3. Compound Queries

Compound queries differ from normal *queries*, as they are used to track property values across multiple *executions* using one of several mathematical operators. *Compound queries* use data from a normal *query* applied to several *execution results*. Given a set of data points for each *query* and *execution result*, the chosen operator determines how they are combined. The end result of a *compound query* is a single list of data pairs, similar to that of a normal *query* result set. We will first describe the required values when creating a *compound query*, then discuss how the internal service executes them and why they are useful.

Similar to regular *queries*, *compound queries* require a unique name. Internally they are also assigned a unique id so that they can be managed properly by an internal service. This service provides the same functionality as the one for *queries* and is likewise an implementation detail and not relevant to a user perspective.

The first values unique to *compound queries* are the *query* and *executions* fields. Since there is a direct one-to-one relationship to a *query*, its respective id must be known. This value must correspond to exactly one Query, considering project-level exclusivity. Similarly, a *compound query* applies to one or more *executions*, whose ids must be known. The relationship is one *compound query* to potentially many *executions*, which is reflected by this value being a list. If the list is empty, there are no values to plot and if list entries do not all correspond to existing and accessible *executions* (e.g. completed), result data can not be retrieved. It is important to ensure that these situations are avoided and that accurate, unambiguous id information is provided for both *query* and *execution* ids.

The last information necessary when creating a *compound query* is the mathematical operation that should be used to combine the sets of *query* result data into a single *compound query* result data set. The *compound query operation* is provided as an enum and may be one of four values: SUM, DIFFERENCE, PRODUCT, QUOTIENT and AVERAGE. When SUM is used, *query* results will be folded (i.e. added) at each corresponding index. The *compound query* result value at index zero corresponds to the sum of all *query* result values at this same index, and so on for all indexes. The DIFFERENCE operator successively subtracts values in a similar way. The PRODUCT operator multiplies these values and the QUOTIENT operator divides them. It would not be surprising for the SUM and PRODUCT operators to yield large results, and for the DIFFERENCE and QUOTIENT operators to yield comparatively small ones. The final operator is AVERAGE, which results in the mean of all respective values per index. This corresponds to the SUM operator result divided by the size (i.e. quantity of elements) of the data set.

It should already be at least partially clear how the internal service uses *compound queries* to obtain data to be plotted. For the sake of detail, however, we will explain this now. The process begins by retrieving the *executions* referenced in the *executions list* and mapping each to the result of performing the *query* referenced by *query id*. The result of this step is a list of *query* results. The entries of this list are then folded and the respective entry applied. Since the result of this procedure is a set of pairs containing two values, no special precautions or processing is required; each pair is mapped to a single

data point on the line chart, where the first element of the pair is the x-axis value and the second is the y-axis value.

Compound queries are useful because they provide the user with the ability to visualize data across many *executions* and potentially discover patterns and details that might otherwise go unnoticed. Using the previously described parameters, a *compound query* can be configured to focus on only the areas and data of interest. It is up to the user to utilize these tools to gain useful insights into a given game design.

5. Results

This chapter first covers three example uses of Prototypical, then presents the results of a five-participant user study. The hardware and software used for the implementation will be described, followed by general observations and assessments.

5.1. Examples

Three examples were implemented to demonstrate the capabilities of Prototypical. The first, Movement Points, covers a proposed card-based mechanic for determining player resources. The second focuses on the combat mechanic from a board game currently in development, Polis: Rise of the City State. The last example is the well-known game of Tic-Tac-Toe, presented here to demonstrate AI agent performance and potential support in Prototypical for complete rule sets.

5.1.1. Movement Points

This example is based on a proposed game mechanic which uses card values to determine player movement resources each turn. A deck of cards is used, where each has a **move** value with an initial value according to the distribution outlined in table 5.1. Each player draws the top card and may allocate the respective quantity of movement points during the current turn. Drawn cards are not added back into the draw deck and gameplay ends once the deck is empty. No actions during gameplay may otherwise modify the composition of the draw deck or any values of the contained cards. This is a strictly two-player action resource mechanic and the potential changes suit themselves for iteration because they could be implemented piecemeal and the updated design simulated before proceeding. If at any point the design avoids these pitfalls and the designer is satisfied, the process can be stopped because the potential changes are decoupled from one another and would not require a refactoring of the modeled components. This makes it a useful example to include and we will now explain how it was implemented in Prototypical then cover the observed data.

move Value	Quantity
0	6
1	6
2	6

Table 5.1.: Initial distribution of **move** values (Movement Points)

The complete process involved modeling the necessary entities, running simulations, making design modifications and iterating until a satisfactory result was achieved. This required thirty *executions* in total, where modifications were made after the first set of ten and after the second set of ten (i.e. after the tenth and twentieth *executions*). First, a new project named **Movement Points** was created. A *group* was created for each player, named **Player 1** and **Player 2**, respectively. Another *group* was created to represent the hand of each player, named **Player 1 Hand** and **Player 2 Hand**. The draw deck was modeled using a *deck component* named **Draw Deck**, to which all *card components* were added after they were created. Eighteen *cards* were used in this example (nine per player) and each was assigned an appropriate name with an integer index, such as **Movement Card 3** and **Movement Card 5**. An alternate naming scheme could have incorporated the actual movement value, such as **Movement Card 3 (M2)**. Each *card* was assigned a *property* named **move** with a value of either **0**, **1** or **2**. Six *cards* were assigned each of these values, dividing the eighteen total into three groupings. To clarify, six had a **move** value of **0**, six had a value **1** and the remaining six were given the value **2** (5.1). The last step to model the necessary *components* was to ensure that the *cards* were all added to the *deck* as child *components*.

Two primary *actions* were modeled in order to achieve the desired simulation behavior. These corresponded to one per player and each consisted of an *and action* which cleared the previous turn score, removed one draw deck child *component* randomly, added the same *component* to the player hand and updated the move score according to the associated **move** value. Note that it was not necessary to model this as a *component property* in advance, as it was assigned at run time and Prototypical always overwrites existing values or creates them if they do not yet exist. Before covering each of these four *sub-actions* individually, it is worth mentioning several remaining top-level characteristics. The name reflected the relevant player and intent, for example **Player 2 Draw 1 Card**. The receiver *component selector* was set to return the player *component* and the **chance** value was set to the default of **1**.

The four *sub-actions* of each *and action* were all *simple actions*, since the process could be broken down into: resetting the score, removing the card from the deck, adding the card to the player hand and updating the player move score. The first of these was named **Reset Player move score**. Since the *sub-actions* were player-specific, a *component selector* was used which returned the respective player *component*. An *always met condition* was paired with the *update property action event* because resetting the score was always possible with a basic property modification. The score value was stored as a *component property* on the player hand *component* (not the player *component*), so the appropriate *component selector* was used. This *property name* was set to **score** and a *value expression* was used to provide **0** as the new value.

The second *simple action* was named **Remove card from draw deck**. A *remove action event* was accompanied by a *condition* that was met whenever the draw deck was not empty. To validate this, a *simple condition* called **Draw Deck Not Empty** used a *property filter* which checked the **count** property using a *gt comparator* (4.1) against the value **0**. The selector was configured to use the draw deck id. When the *condition*

was met, the *remove action event* used this same *component selector* so that the draw deck was the receiver and that the target would be taken from it. The target itself was provided by a *component selector* which returned one *component* of type **card** using a **random** strategy where the draw deck was the root (i.e. the draw deck id was used for the root parameter). Using this *selection strategy* ensured that cards would be chosen randomly, modelling withdrawal from a shuffled set.

The **Add Card to hand from draw deck** *simple action* used an *always met condition* and *add action event* to achieve what the name implies. The details of the *card* (or any other *component*) would not have influenced the validity of the event, so an *always met* was used here. A *component selector* configured with the player hand **id** was used for the *receivers selector*, while the *target selector* was **null**. Since the *target selector* was not provided, the last *component* was used instead; and, in this case, it was the *component* used by the previous *simple action* **Remove card from draw deck**.

The final *simple action* updated the player move score using an *always met condition* and an *update property action event*, similar to the **Reset Player move score** *simple action*. Instead of using a *value expression*, however, a *property expression* for the **move** source property with a **null** receiver was used instead. The result was that the value of the **move** *component property* of the last used *component* was returned according to the then current game state.

Now that it is clear how the *components* and *actions* were modeled, we will explain the parameters used for the *executions*. The **Player 1** and **Player 2** *components* were set as the first and second players. Because this game mechanic related to resource management and the early phase of a game turn, the random AI agent was selected. The decision-making that the MCTS agent affords was unnecessary because there was only one possible action per player each turn when determining movement score. Because the *execution* was meant to continue until no *cards* were left in the draw deck, we created and assigned an appropriate halting *condition* called **Draw Deck Is Empty**. Using a *component selector* configured to return the draw deck *component*, it checked that the **count** *component property* was equal to zero by using the *eq comparator* (4.1) and value **0**. The winning player information was also not necessary, as a winning player was not meant to be decided during this phase of the related game.

Execution	Low	High	Median	Mean
1-10	1100	1167	1139	1139
11-20	711	777	732	735
21-30	682	755	737	729

Table 5.2.: *Execution* run times in milliseconds (Movement Points)

A set of ten *executions* was performed and a chart generated to display the average movement score of the players for this interval. Table 5.2 shows the lowest, highest, median and mean run times per *execution* for each set of ten, which were executed in parallel. Figure 5.1 shows both average player scores, where blue corresponds to the first

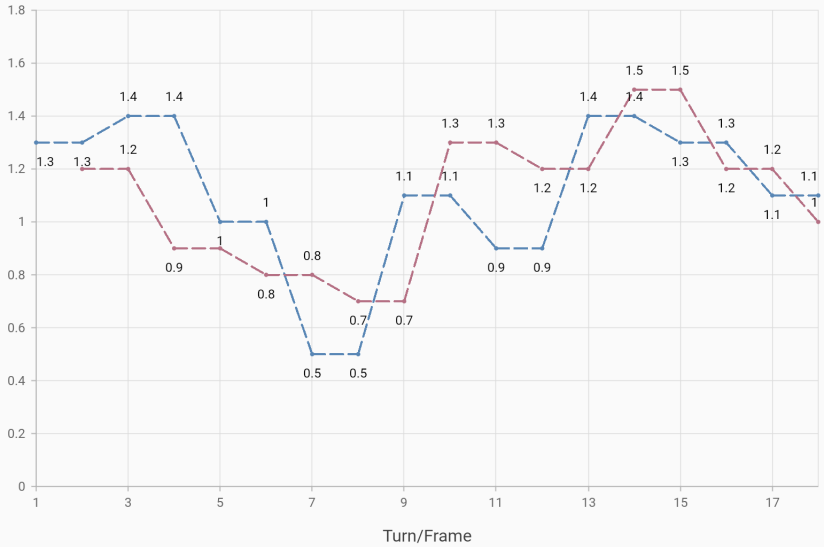

Figure 5.1.: Average score, *executions* 1-10. Player 1 in blue, player 2 in red (Movement Points)

player and red corresponds to the second. The x-axis shows the turn or *frame* number, where each actual turn has been represented by two *frames* and the y-axis corresponds to the average score value. Game state snapshots of both before and after a move by the active player of a given turn leads to duplicated data points, which can be seen by the plotted line segments with width two (i.e. both values are the same). Additionally, it is also possible for data points to be missing, as is the case for the second player during the first turn in the same figure (5.1). In this example, the second player has not yet taken a turn and no initial property value was set, so data for this interval is omitted. Note that all such details were considered for numerical calculations presented in this and following sections. Repeated data points may be safely ignored when viewing.

All observed values ranged from **0.5** to **1.5** (inclusive), which was deemed too low. Higher values were desired because low ones limit the quantity possible actions during gameplay, potentially resulting in limited or stagnant player experience. Modifications to the *card* values were made, such that six with a value of **0** were changed to have a value of **1**. Three additional *cards* were created with a value of **3** and added as child *components* to the draw deck. Ten more *executions* were then performed and the same *queries* and *compound queries* plotted, which can be seen in figure 5.2. The new range is between **1.3** and **2.5**, inclusive. While the shape of the plotted data is noticeably different from the previous, it is useful to consider figures 5.3 and 5.4.

The first (5.3) superimposes the observed first player average score values of the second set of iterations (11-20) onto those of the first set (1-10). The second (5.4) does the same for the second player. The values for the second set of *executions* (11-20) in both are consistently higher than those from the first set (*executions* 1-10) and there are more data points in the second set than in the first. As the simulations were performed until

the draw deck was empty, adding cards increased the number of turns/*frames* in the *execution* data.

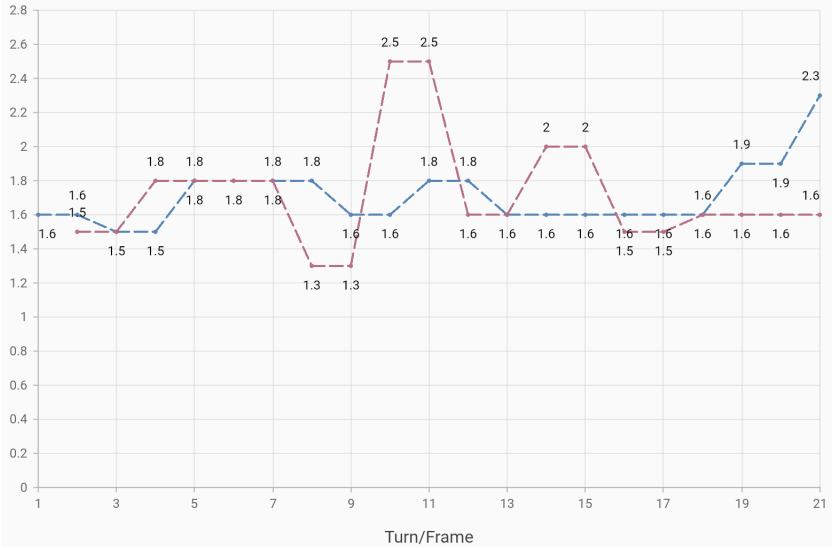

Figure 5.2.: Average score, *executions* 11-20. Player 1 in blue, player 2 in red (Movement Points)

In order to widen the range of observed values and create a tendency toward higher observed minimums, four *card* values were changed from **2** to **5**. The resulting chart can be seen in figure 5.5, which displayed satisfactory results. To further investigate the relationship between the original design, performed modifications and resulting simulation data, two additional charts can be referenced. Figures 5.6 and 5.7 compare the original simulation data of each player with the corresponding data from *executions* 21-30, once all changes had been implemented. The later data values are consistently higher than those of the initial set and there is a clear relationship between the desired trends in simulation data and the modifications made between *execution* sets. Table 5.3 shows the final distribution of *card* values and figure 5.8 visualizes the data as a combined bar graph, contrasting it with the initial distributions.

move Value	Quantity
1	12
2	2
3	3
5	4

Table 5.3.: Final distribution of **move** values (Movement Points)

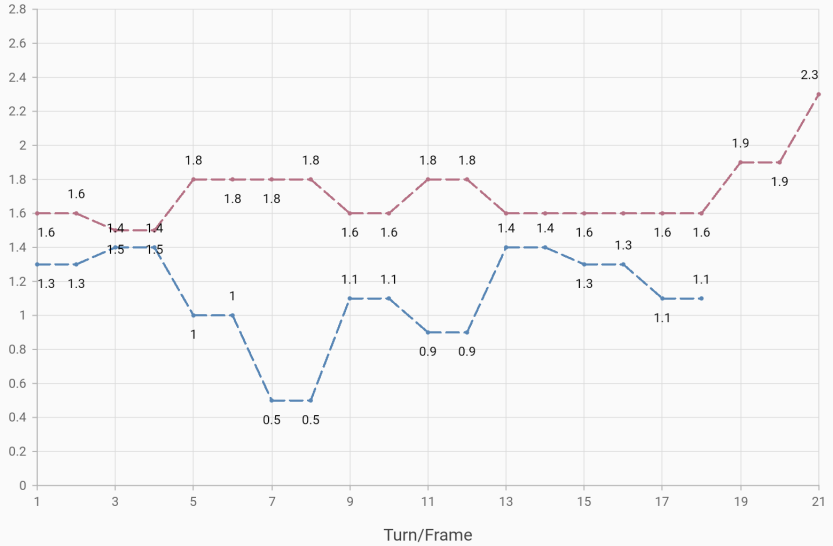

Figure 5.3.: Player 1 average score, *executions* 1-10 in blue, *executions* 11-20 in red (Movement Points)

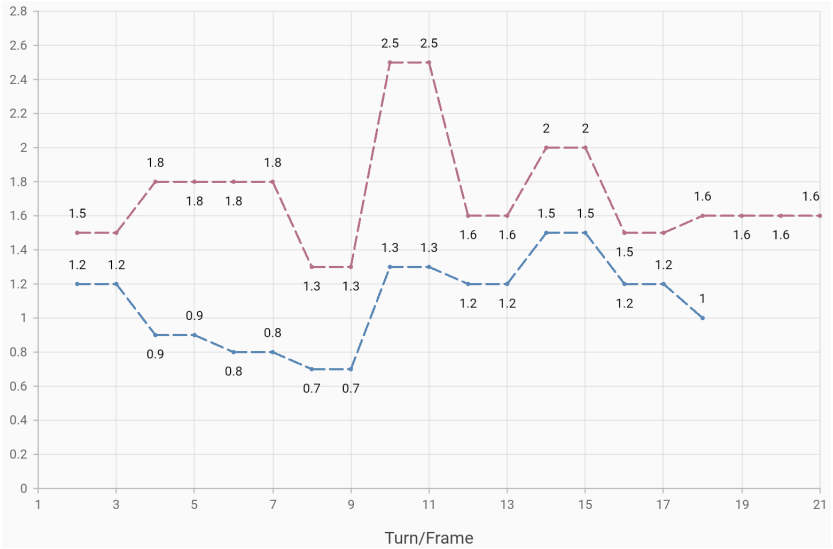

Figure 5.4.: Player 2 average score, *executions* 1-10 in blue, *executions* 11-20 in red (Movement Points)

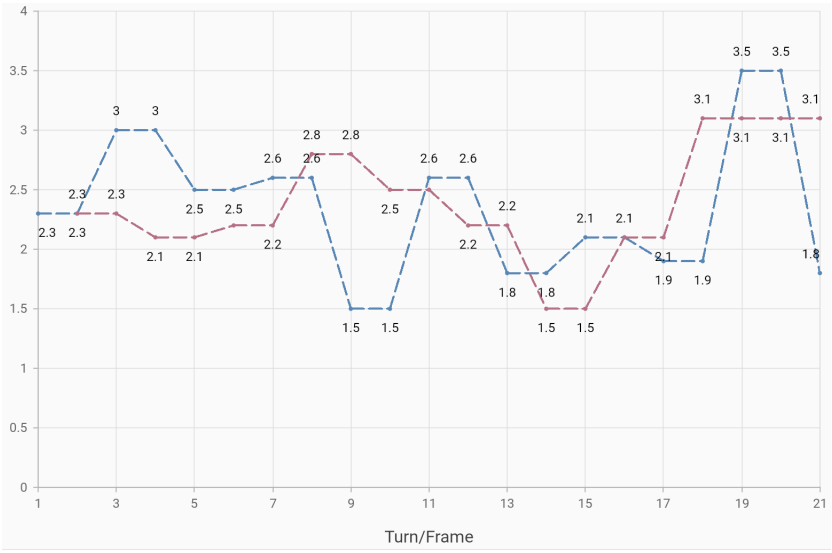

Figure 5.5.: Average score, *executions* 21-30. Player 1 in blue, player 2 in red (Movement Points)

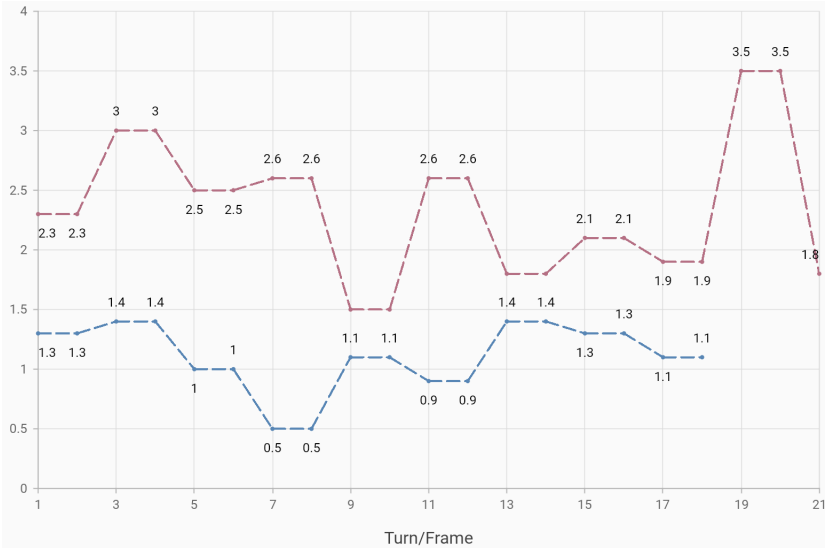

Figure 5.6.: Player 1 average score, *executions* 1-10 in blue, *executions* 21-30 in red (Movement Points)

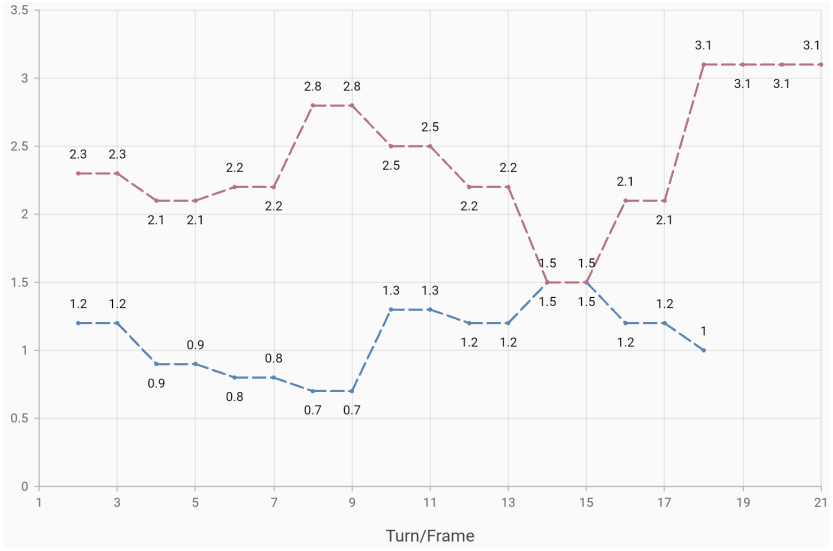

Figure 5.7.: Player 2 average score, *executions* 1-10 in blue, *executions* 21-30 in red (Movement Points)

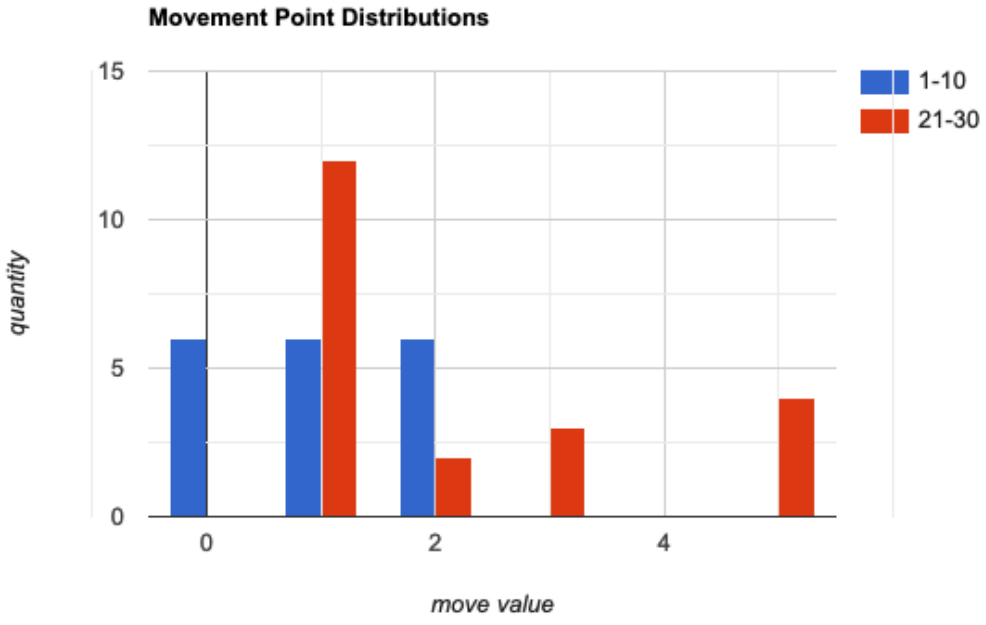

Figure 5.8.: Initial and final distributions of **move** values. *executions* 1-10 in blue, 21-30 in red (Movement Points)

To create the charts above, two *queries* and several *compound queries* were needed. The first was a basic *query* which recorded the **score** *component property* of the first or second player hand *component*, after the relevant turn has taken place. These were named **Player 1 Score After** and **Player 2 Score After** respectively. The receiver was a *component selector* which used the id of the corresponding player hand *component* and the property name was **score**. The *query temporality* was set to **AFTER_ACTION** and both the **start index** and **end index** was left as **null**, so that all *frames* from the *execution result* were considered.

Utilizing the two *queries* were *compound queries* for each player. Each referenced one third of the performed *executions*, which resulted in one *compound query* for *execution results* 1 to 10, another for 11 to 20 and a third for 21 to 30. This triple was created for each player, resulting in six *compound queries* in total. The names of the *compound queries* were chosen to indicate the corresponding player, relevant *component property* of the underlying *query* and the *executions* within its scope. An example of this is **P2-Score-Avg-1-10**, relating to the average score of the second player for the first ten *executions*. The **query id** parameter was set according to the related *query*, the AVERAGE operation was used and executionIds consisted of a list of the relevant *execution* ids.

5.1.2. Polis: Rise of the City State

This example involves a game currently in development called Polis: Rise of the City State [53]. It is a tableau-building game in which players research technologies, construct buildings and leverage military might to conquer territories in the hopes of founding the best empire, measured by victory points. Military power is represented using meeples, while technology and buildings take the form of cards. Regions exist in a shared location at the center of the game board and combat occurs when meeples belonging to both players have been placed in a shared region. Each player secretly picks a card from their hand, where the victory point value on the card becomes a combat modifier. The modifier is added to the quantity of friendly meeples in the region, resulting in a total score for each player. The scores are compared and the highest score deemed the winner. It is this mechanic that we chose to implement in Prototypical, with special focus on the composition of the technology and building cards deck and victory point values. We will now describe how this was accomplished.

This game mechanic was suitable for implementation in Prototypical for the same reasons presented in 5.1.1, as it is functionally similar. The designer wanted to simulate the combat system of the rules because it did not present itself as dynamic enough to produce interesting gameplay. It was explained that a sense of thrill for the user was desired, which might have been promoted by reducing the predictability of the combat outcomes. In other words, the possible results of a combat should not be overly predictable, regardless of outside influence (e.g. deck composition information). For this reason, the designer wanted to iteratively modify the composition of the card deck until the simulation data pointed toward more-varied outcomes. The range of the difference between player scores was permitted to increase between turns but the average difference across all simulations needed to remain relatively the same. Described differently, the

outcomes from one combat round to the next were expected to become more disparate, but neither player should have gained an inherent asymmetrical advantage due to the modifications.

The original design consisted of ninety-six cards in total, divided into victory point value categories of one to four. Thirty-eight cards had a value of **1**, twenty-six a value of **2**, twenty-three a value of **3** and nine had a value of **4**. While the game rules defined the possible range of meeple counts to be from five to fifteen (inclusive), the most-common range was expected to be nine through twelve. For this reason, we chose to model only the most-common meeple values.

To model the necessary entities, a *group* was first created for each player and the central board region, as well as an additional one for each possible quantity of meeples. The player *groups* were named **Player 1** and **Player 2** and the region was simply named **Region**, as there was only one. *Groups* were used for meeple quantities due to implementation-specific reasons which will be covered later; however, all included information in the name indicated the respective value (e.g. **9 Meeples**). The property name was used to store the corresponding value of either **9**, **10**, **11** or **12**.

A *deck component* named **Cards** was created, along with 96 *card components* which were then added to it as children. The **vp** property of each was set according to the distribution outlined in 5.9 (provided by the designer).

Combat			
Value		**Copies**	**Deck %**
1	17	38	39.58
2	10	26	27.08
3	9	23	23.96
4	4	9	9.38
		96	

Figure 5.9.: Original Card VP Distribution (Polis)

A *condition* was created to check whether the deck was empty. This was not strictly necessary, as the game rules dictated that a maximum of four combat rounds may occur per game (i.e. one per round for four rounds), which could never fully exhaust a deck of ninety-six. Another *condition* was created to be met after four rounds had been played, corresponding to the second player having taken four top-level *actions*. To do so, its *component selector* was configured with the region *component* id to check that the property **round** was greater than or equal to four. For this a *gte comparator* (4.1) checked against the value **4**. The **round** property was updated after each second player turn. A *simple action* called **Record End of Round** used a second player-linked *component selector* with a chance value of **1**, an *always met condition* and an *update property action event*. The *action event component selector* was configured with the region *component* id. The

property name was set to **round** and the value was given by a *value expression* with a value set to **1** and *add operator* (4.1). This ensured that it would be incremented each round.

Player turns consisted of a single *and action* per player and had six *simple actions* as *sub-actions*. Most of these were used for copying property values to be tracked by *queries* and were not directly related to the game mechanics. The first *sub-action* cleared the player score. The second copied the meeples value from a randomly-selected *group* of meeples to a player **meeples** *component property*. The third was responsible for removing a random *card* from the *deck*, while the fourth copied the associated **vp** property to a player combat score property. The final two *actions* summed the meeple and combat values to create a combined, total score.

Aside from round-related upkeep, both player *actions* were identical *and actions* as described above. The player score was cleared with the **Player _ Clear Score** *simple action*, where _ corresponds to the player number (i.e. **1** or **2**). An *update property action event* with an *always met condition* was carried out with the region as a receiver, targeting the **Player _ Score** *component property*. A *value expression* with value **0** was used. The **acceptedReceiverComponentsSelector** field for this *action* – and all others to be discussed here – used the respective player id.

To simulate placing an arbitrary number of meeples onto the region board, the **Player _ meeples** *simple action* updated the **Player _ meeples** property of the region *component*. This was accomplished using a *property expression* configured for the source property **meeples** and a *component selector* with the **random** *selection strategy*. This source property corresponded to that of the meeple *groups* and the **Random Meeple Group** filtered using the region *component* id, accordingly. This method to randomly choose one set of meeples each turn was previously mentioned and was a simpler approach in Prototypical than modelling individual meeple *components*.

The next *sub-actions* involved drawing the card and recording the relevant values. The removal of one card each turn by the player used a *remove action event* and the *deck not empty condition*. The event used the deck id and the *random card selector*. The latter functioned similarly to that of meeple *groups*, selecting from *card* child *components* of the *deck* instead. An important difference, however, is that the *card* was fully removed from the set of child *components* and discarded because it was not added to another *component*. This was possible because the victory point value of the *card* held significance only when it was drawn. It was immediately recorded to the **Player _ combat card** property in the next *simple action* using the **vp** source property. The *receivers selector* here was left **null** because the previously drawn *card* contained the relevant property value. The meeple value was then recorded with **Player _ Score** and **Player _ meeples property name** and **source property** values for the region. The *add operator* ensured that the scores were summed throughout all four rounds. The overall player combat score was then recorded with the **Player _ Record Score Combat** *simple action* using the **Player _ Score** and **Player _ Combat card property name** and **source property** values. The *add operator* (4.1) was again used for the same reason. The *executions* were configured with **Player 1** and **Player 2** *components* as first and second players and the

described **Played Round 4** *simple condition.*

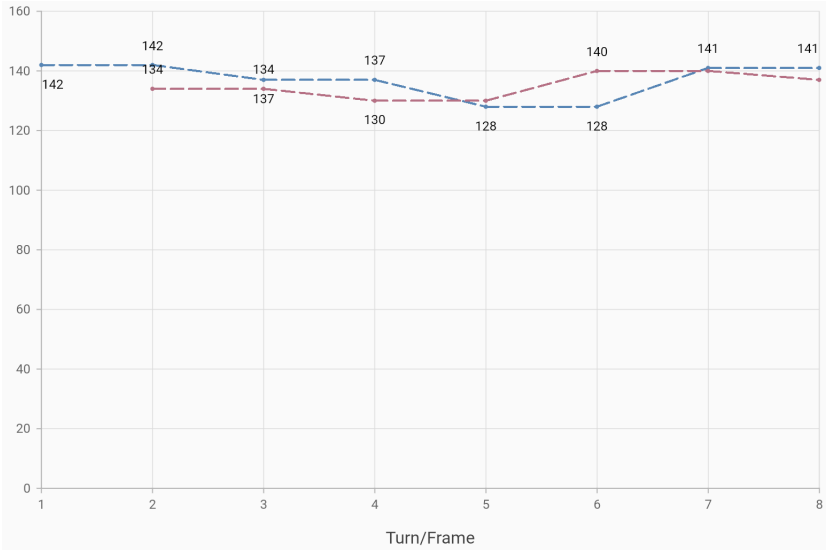

Figure 5.10.: Score sum, *executions* 1-10. Player 1 in blue, player 2 in red (Polis)

The **random** AI agent was chosen and ten *executions* were performed as a starting point. Figure 5.10 shows the resulting data from both players, which was bound to an overall range from **128** to **142**, inclusive. The data itself was derived by summing the score of the respective player at each corresponding interval (i.e. turn/*frame*) across the ten *executions* belonging to the set.

Fifteen **vp** values were then changed from **1** to **0** to lower the possible or expected scores. After running ten more *executions*, figures 5.11 and 5.12 were generated to compare the original and new data for the first and second player, respectively. While 5.11 did not clearly indicate that the individual summed scores of the first player were now lower, the total sum for the original data was **548**, whereas it was **540** for the second. This – along with the resulting average of the summed scores of **137** for the first *execution* set (1-10) and **135** for the second (11-20) – remained consistent with the expected results when considering the implemented changes at this stage of iteration. Figure 5.12 further supported this, as all data points for the second player in the second iteration set (11-20) were indeed lower than those at the respective intervals in the first set (1-10).

After ten more simulations, the changes were considered too excessive and were partially reverted by changing ten card values from **0** back to **1**. Additionally, thirteen values were changed from **2** to **3** to increase the frequency of higher sums. Figures 5.13 and 5.14 showed that the values observed in the next *execution* set (21-30) were consistently higher than the previous ones, with a single data point of the first player as an exception. The second player observed a new higher upper bound of **144** (5.14).

Ten further *executions* led to results which centered too closely on mid-range values, so eight card values were changed from **3** to **4** in an attempt to add variation. Figures 5.15 and 5.15 showed the data from the newest set of ten *executions*. Both players showed

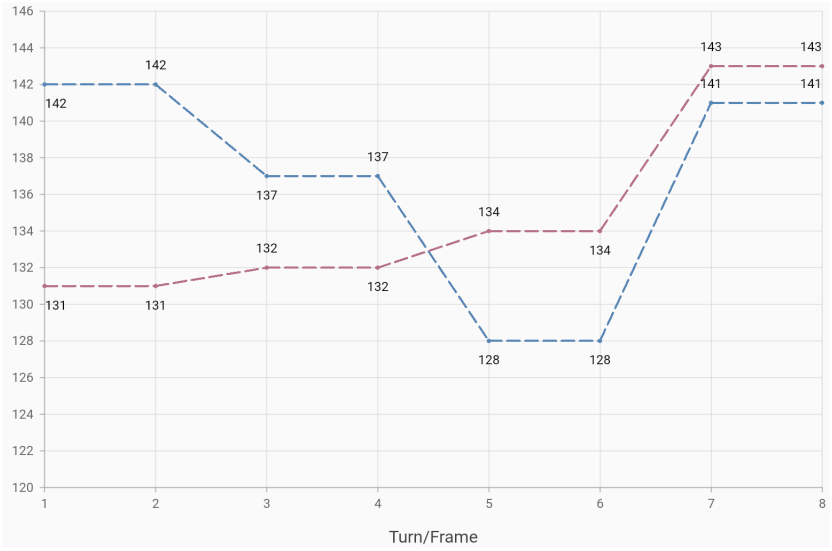

Figure 5.11.: Player 1 score sum. *Executions* 1-10 in blue, *executions* 11-20 in red (Polis)

values which could be considered dynamic and not concentrated among the middle range.

A higher upper bound was once again desired, so four values were changed from **4** to **5**. The collective result of this, and all previous changes, was most evident in figure 5.17, which compared all first player values up to this point and showed that a highest upper bound of **154** was observed by the first player. The final changes involved modifying two values from **4** to **7** and the resulting values did not have a noticeable effect on the simulation data for either player in the generated charts.

However, figure 5.18 provides insight into not only the final change set, but also offers a comprehensive view of how the raw score data evolved throughout the iterations. Each of the six groups in the table and box plot are labeled and color-coded with respect to a single set of iterations performed. Groups one through six correspond to iteration sets **1-10**, **11-20**, **21-30**, **31-40**, **41-50** and **51-60**, respectively. Each group consists of eighty values because every *execution* set involves ten performed *executions* with four played rounds (5.1.2). Table 5.4 lists the minimum, maximum, median and mean run times for each of the six *execution* sets in milliseconds. The difference in first and second player scores summed across all *frames* for *executions* 1 through 10 and 51-60 was **35** and **16**, respectively. The corresponding averages were **8.75** and **4**.

Comparing the first and sixth groups provides a clear impression that both the range and distribution of scores changed. The minimum and maximum observed scores of any group were **9** and **18**, where both values were seen in the sixth group (*executions* 51-60) and neither in the first (*executions* 1-10). The median value remained the same between first and last groups, while the mean value slightly increased from **12.3875** to **12.5125**. The standard deviation, however, increased by over twenty-five percent, from **1.4539** to **1.8211**.

Six *queries* were created in total, corresponding to three per player. All tracked

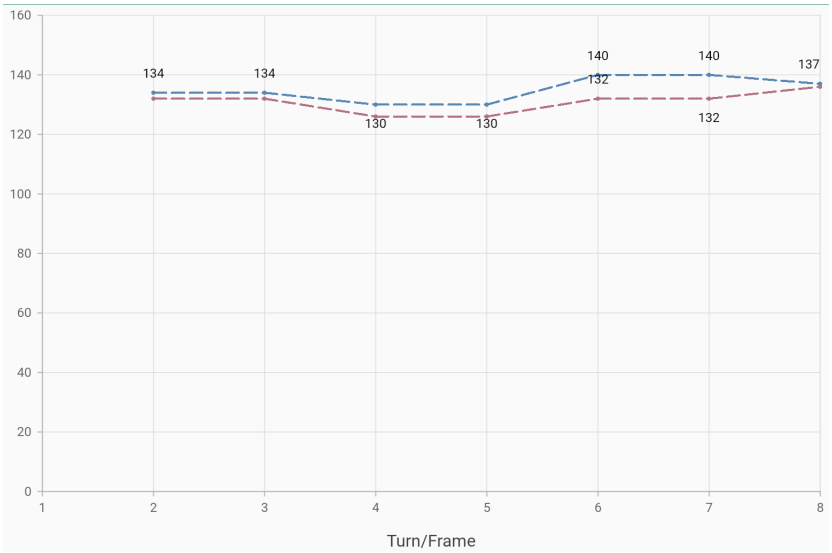

Figure 5.12.: Player 2 score sum. *Executions* 1-10 in blue, *executions* 11-20 in red (Polis)

properties were stored in the region *component*, so an appropriate *component selector* with this id was used. They were configured with **AFTER_ACTION** *query temporality* and **null start index** and **end index** values. The tracked properties were **Player _ meeples**, **Player _ combat card** and **Player _ Score**.

A *compound query* was created for each player per set of ten *executions* to track the summed total combat score at each interval. Names were reflective of this (e.g. **P2 Score Sum 21-30**). The **query id** corresponded to the respective **Player _ Score** property and the operation was **sum**. The linked **execution ids** corresponded to the *executions* of the relevant set (e.g. 10-20).

Figure 5.19 shows an empty chart and *execution* details. The *queries* described above can be seen on the left hand side of the screen in the **Queries** panel, while *compound queries* are in the **Compound Queries** panel on the right. It is possible to mix and match *queries* and *compound queries* on the same generated chart, as seen in figure 5.20. In this example, the individual scores for the first and second players in *execution* **0** are in blue and red, respectively. Orange corresponds to the sum of player 1 scores for *executions* **1-10** and yellow to that of player 2 across the same interval.

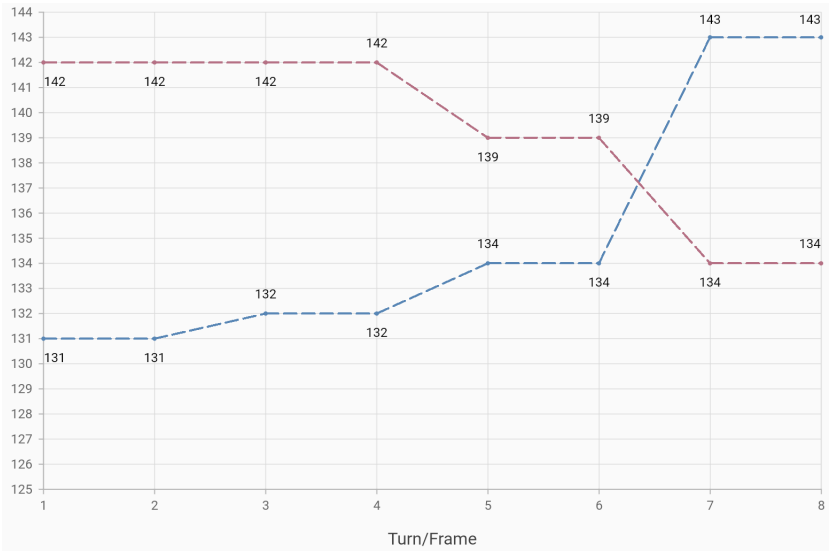

Figure 5.13.: Player 1 score sum. *Executions* 11-20 in blue, *executions* 21-30 in red (Polis)

Execution	Low	High	Median	Mean
1-10	506	563	544.5	540.6
11-20	360	437	417.5	411
21-30	421	443	430.5	431
31-40	358	398	377	374.6
41-50	391	420	406.5	406.9
51-60	222	281	250.5	250

Table 5.4.: *execution* run times in milliseconds (Polis)

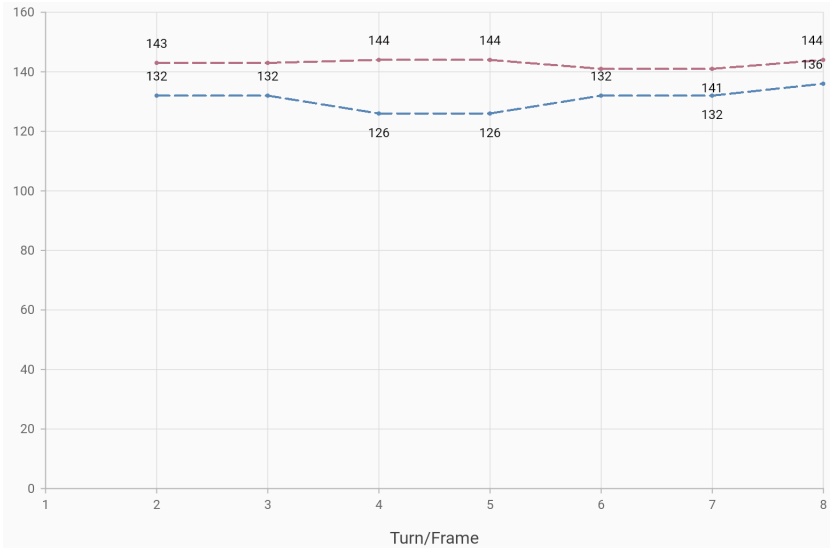

Figure 5.14.: Player 2 score sum. *Executions* 11-20 in blue, *executions* 21-30 in red (Polis)

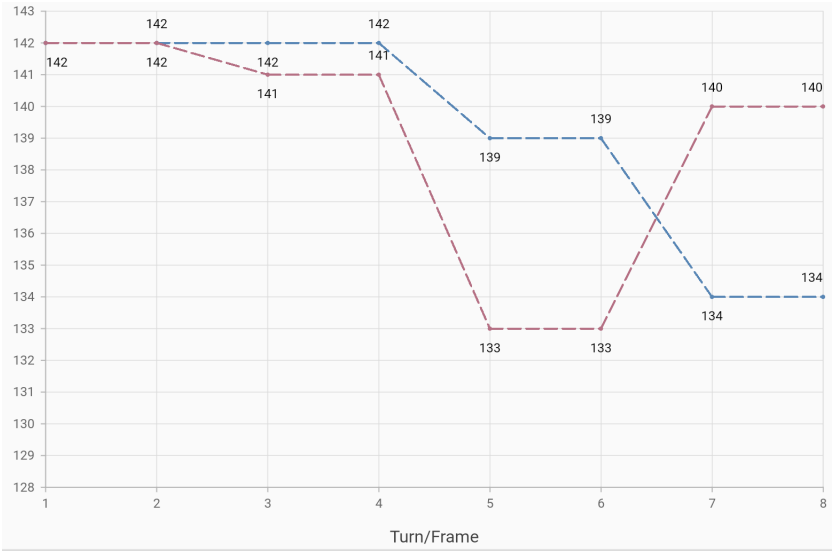

Figure 5.15.: Player 1 score sum. *Executions* 21-30 in blue, *executions* 31-40 in red (Polis)

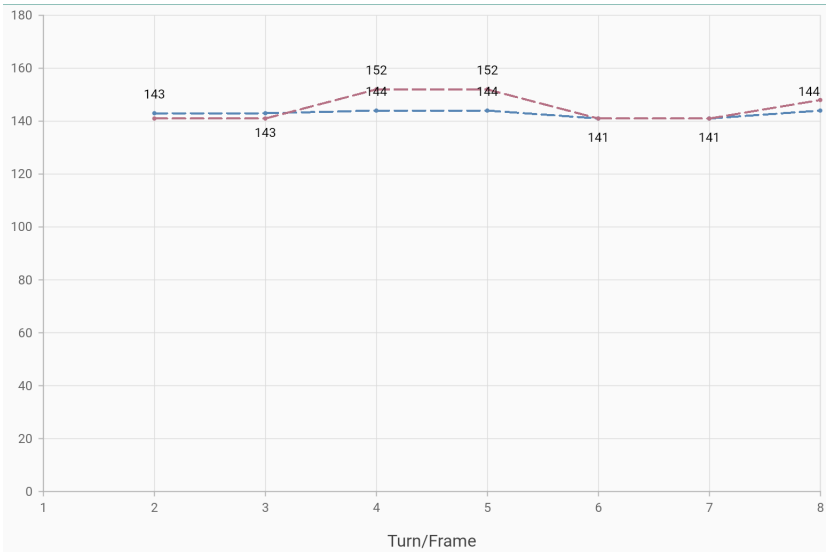

Figure 5.16.: Player 2 score sum. *Executions* 21-30 in blue, *executions* 31-40 in red (Polis)

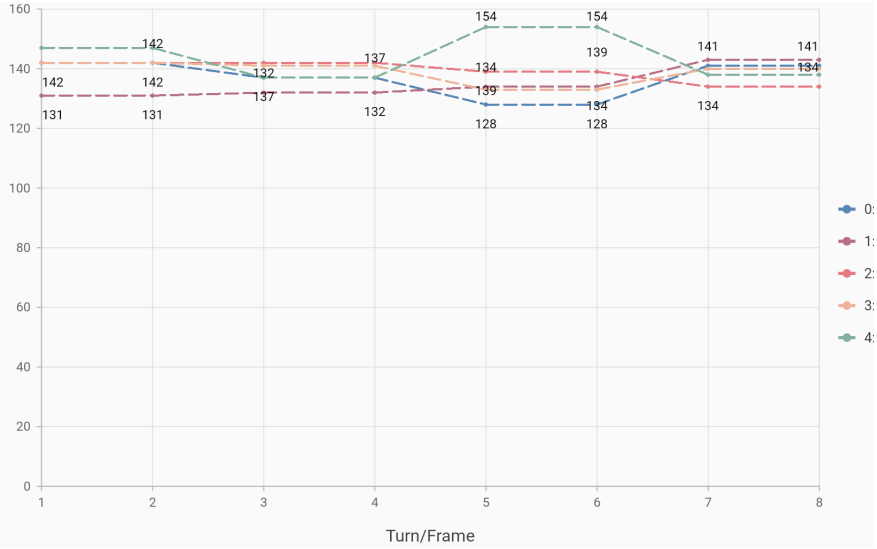

Figure 5.17.: Player 1 score sum. *Executions* 1-10 in blue, 11-20 in red, 21-30 in orange, 31-40 in yellow, 41-50 in green (Polis)

Data Summary								
Groups	N	Min	Q_1	Median	Q_3	Max	Mean	SD
Group 1	80	10	11	12	13	16	12.3875	1.4539
Group 2	80	9	11	12	13	16	12.1375	1.5731
Group 3	80	10	12	13	14	16	12.825	1.5159
Group 4	80	9	12	13	14	16	12.9875	1.7099
Group 5	80	9	12	13	14.25	17	13.0875	1.7945
Group 6	80	9	11	12	14	18	12.5125	1.8211

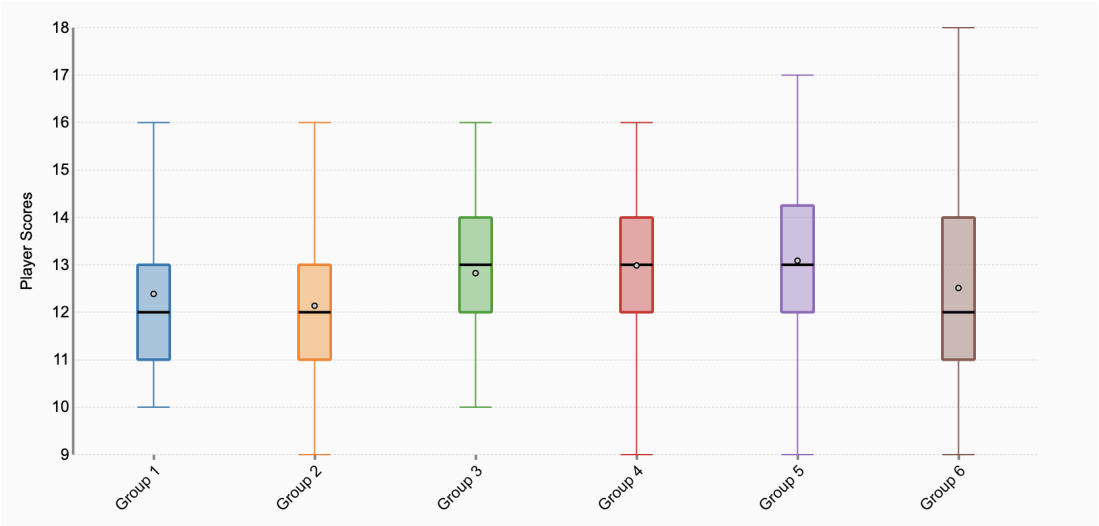

Figure 5.18.: All player scores. Blue group 1 is *executions* 1-10, orange group 2 is 11-20, green group 3 is 21-30, red group 4 is 31-40, purple group 5 is 41-50 and brown group 6 is 51-60 (Polis)

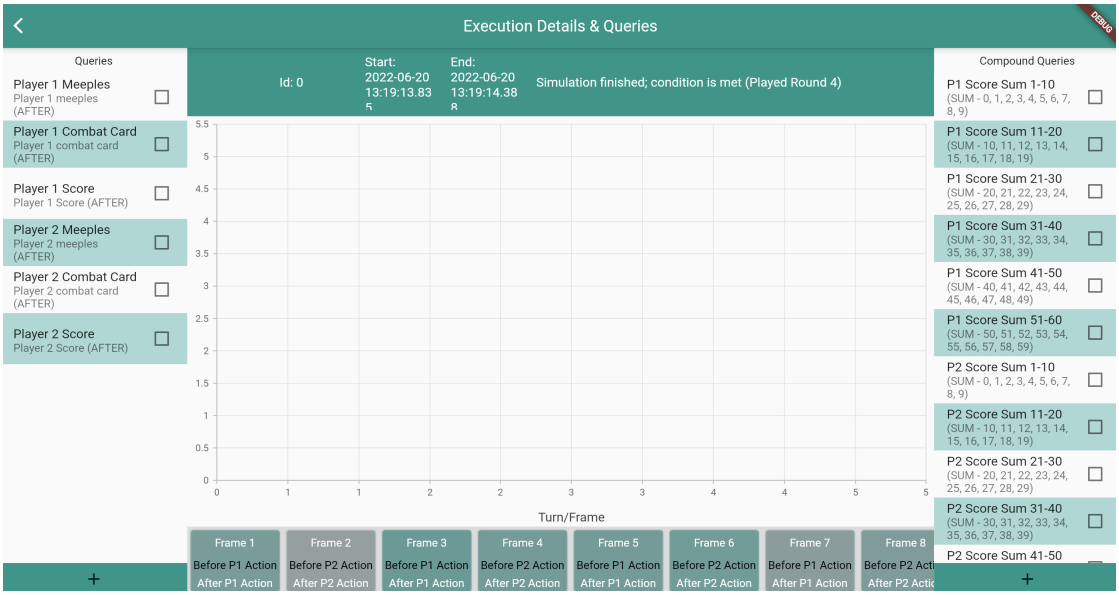

Figure 5.19.: Empty chart with created *queries* on the left and *compound queries* on the right (Polis)

Figure 5.20.: Chart involving both *queries* and *compound queries* (Polis)

5.1.3. Tic-Tac-Toe

Tic-Tac-Toe was implemented in Prototypical for several reasons, among which are the characteristics of the game itself and the simplicity of gameplay and strategies. It has been the subject of a vast body of research. Due in part to a low branching factor, it is considered a solved game and all possible game states can be easily mapped out with an identifiable optimal strategy. It contains no hidden or stochastic information and is a zero-sum game. To review, in zero-sum games a win for one player is equivalent to a loss for the opposing player. The inverse also holds true. It is therefore suitable for both general implementation in Prototypical, as well as for the use of the MCTS AI agent. For the sake of completeness, we will also point out the two-player turn-based structure of its gameplay, low component count, few possible actions, and a typically short game duration.

Tic-Tac-Toe is presented here as a proof of concept and demonstration that Prototypical may potentially support the implementation of complete rule-sets. This is opposed to the stated scope of only a singular mechanic at a time. Additionally, no iterative changes were made to the design itself. This game also holds value in demonstrating the performance of the MCTS agent, as sensible gameplay moves can be easily inferred and poor strategic choices quickly identified. All necessary elements have been modeled in Prototypical and several *executions* were performed using MCTS agents. The MCTS agent played the game against itself. We estimated performance by running *executions* and periodically changing the MCTS iteration count parameter. The observed gameplay choices for both agents adhered to all game rules and had the appearance of a natural human-like strategy. We will now explain the implementation of Tic-Tac-Toe in Prototypical, then we will present the collected simulation data and review AI agent performance.

Tic-Tac-Toe was implemented in a Prototypical project of the same name. To model the playing area, a *component* of type *group* with the name **Board (3x3)** was created. The nine potential areas which may be marked by either player throughout the course of gameplay each were modeled using a *field*. These were named using an integer value between zero and eight (inclusive) and corresponded to a specific location on the 3x3 grid. Location numbers increased from left to right and top to bottom, as seen in table 5.1.3. Examples of this naming scheme include **Field 3** and **Field 7**, as the number corresponds to a grid location that the *field* is supposed to represent. Additionally, all such *fields* had a *component* property called **value**, instantiated as a **-** character. This property was changed when either player had marked the *field* with either an **x** or **o** and was updated accordingly. The original value of **-** indicated that it is empty. The property was set to **x** when the *field* had been marked by the first player and **o** when it was marked by the second player. More details on these *actions* will follow. The nine empty *field components* were assigned to the Board *component* as children and the **count** property of the board was implicitly updated, while the modelling of each player was accomplished using a *group* for each. The first was named **Player X** and the second **Player O**.

The possible game actions for each player were then modeled. Eighteen *actions* were created in total, which corresponded to a count of one per potential game action per player. If we consider that there were at most nine spaces on the board from which to

field 0	*field* 1	*field* 2
field 3	*field* 4	*field* 5
field 6	*field* 7	*field* 8

Table 5.5.: Grid layout of 3x3 Tic-Tac-Toe board

choose at any given point during the game, it follows that each player would have had a maximum of nine possible moves to be modeled. Each *action* was named according to which *field* it targeted and which character (**x** or **o**) would be assigned to it. Some examples included **Mark Field 0 with x** and **Mark Field 0 with o**, as well as **Mark Field 3 with x** and **Mark Field 3 with o**. All *actions* were of type *simple action* and therefore did not contain any *sub-actions*. Each used a *component selector* which targeted the player respective of the symbol that was to be marked. This meant that two *component selectors* in total were necessary for the project - one for each player. Each used a quantity parameter of **1**, a **first** *selection strategy* and a *filter* which contained the *component* id of the respective player. The quantity was set here because an *action* was always unique to one player (i.e. either *x* or *o*). The *selection strategy* was set to the default and was irrelevant, because of the *component* id used in the *filter*. The root id remained the default of **null**, because neither player *component* was a child *component*. The accepted *component types* and list of *filter properties* also stayed at the default values to include all types and an empty list, respectively.

In order for each of the *actions* to function as expected, it was necessary to determine during a simulation whether a space was empty or had been previously filled. Therefore, a *simple condition* was created for each of the *fields* and named accordingly. For example, the **Field 0 is empty** *condition* corresponded to the *field* at grid space 0 and was met if it has not yet been assigned a value from either player. It was configured with a *check*, where the *property filter* had a name of **value**, an *eq comparator* and a **-** value. If the **value** property was still set to the initial value (**-**), the *condition* was met, as this implied that the *field* remained untouched by either player and must be empty.

Action events were the last modelling necessary for the *actions* of this example. One was created for each *field*, where both the receiver and target *component selectors* were configured to return the respective *field*. The property name was set to **value** and a *value expression* was used to provide the new value (**x** or **o**), depending on the player with which the encompassing *action* was associated.

The id of the previously mentioned first and second player *components* were set accordingly in the *execution context*, and a halting *condition* was created. For this, an *or condition* was created and named **Either player 3-in-a-row**. As the name implies, this *condition* was met if either player had filled three adjacent *fields*, either horizontally, vertically or diagonally. The sub-conditions list contained two additional *or conditions*, **x3-in-a-row** and **o3-in-a-row**. These were met if the corresponding player had achieved

any of the winning patterns, for which there were eight distinct possibilities. For each acceptable combination, the **winningPlayer** parameter was assigned a *component selector* to return the respective player *component*; if this *condition* was met, it indicated a win for that player. Each of the eight *sub-conditions* were modeled using an *and condition* containing three *simple conditions* and was met when all three *sub-conditions* were met. Each such *simple condition* corresponded to a specific *field* and player and was named accordingly (e.g. as **5 Filled x**). To clarify, the halting *condition* contained an *or condition* for each player to check for one of eight winning patterns. Each winning pattern was represented using an *and condition*, which checked whether the three respective *fields* were filled through three *simple conditions*. Each of the *sub-conditions* checked to see if the relevant *field* was filled using a *check* whose *property filter* referenced the **value** parameter using the *eq comparator* and a player-dependent value of either **x** or **o**. See figures 5.21 and 5.22 for another representation of this.

- x3-in-a-row [OrCondition] [winningPlayer first]
  - topRowFilledX [AndCondition] [winningPlayer first]
    - 0 Filled X [SimpleCondition]
    - 1 Filled X [SimpleCondition]
    - 2 Filled X [SimpleCondition]
  - midRowFilledX [AndCondition] [winningPlayer first]
    - 3 Filled X [SimpleCondition]
    - 4 Filled X [SimpleCondition]
    - 5 Filled X [SimpleCondition]
  - bottomRowFilledX [AndCondition] [winningPlayer first]
    - 6 Filled X [SimpleCondition]
    - 7 Filled X [SimpleCondition]
    - 8 Filled X [SimpleCondition]
  - leftColFilledX [AndCondition] [winningPlayer first]
    - 0 Filled X [SimpleCondition]
    - 3 Filled X [SimpleCondition]
    - 6 Filled X [SimpleCondition]
  - midColFilledX [AndCondition] [winningPlayer first]
    - 1 Filled X [SimpleCondition]
    - 4 Filled X [SimpleCondition]
    - 7 Filled X [SimpleCondition]
  - rightColFilledX [AndCondition] [winningPlayer first]
    - 2 Filled X [SimpleCondition]
    - 5 Filled X [SimpleCondition]
    - 8 Filled X [SimpleCondition]
  - diagDownFilledX [AndCondition] [winningPlayer first]
    - 0 Filled X [SimpleCondition]
    - 4 Filled X [SimpleCondition]
    - 8 Filled X [SimpleCondition]
  - diagUpFilledX [AndCondition] [winningPlayer first]
    - 2 Filled X [SimpleCondition]
    - 4 Filled X [SimpleCondition]
    - 6 Filled X [SimpleCondition]

Figure 5.21.: **x3InARow** *condition* with *sub-Condition*

- o3-in-a-row [OrCondition] [winningPlayer second]
  - topRowFilledO [AndCondition] [winningPlayer second]
    - 0 Filled O [SimpleCondition]
    - 1 Filled O [SimpleCondition]
    - 2 Filled O [SimpleCondition]
  - midRowFilledO [AndCondition] [winningPlayer second]
    - 3 Filled O [SimpleCondition]
    - 4 Filled O [SimpleCondition]
    - 5 Filled O [SimpleCondition]
  - bottomRowFilledO [AndCondition] [winningPlayer second]
    - 6 Filled O [SimpleCondition]
    - 7 Filled O [SimpleCondition]
    - 8 Filled O [SimpleCondition]
  - leftColFilledO [AndCondition] [winningPlayer second]
    - 0 Filled O [SimpleCondition]
    - 3 Filled O [SimpleCondition]
    - 6 Filled O [SimpleCondition]
  - midColFilledO [AndCondition] [winningPlayer second]
    - 1 Filled O [SimpleCondition]
    - 4 Filled O [SimpleCondition]
    - 7 Filled O [SimpleCondition]
  - rightColFilledO [AndCondition] [winningPlayer second]
    - 2 Filled O [SimpleCondition]
    - 5 Filled O [SimpleCondition]
    - 8 Filled O [SimpleCondition]
  - diagDownFilledO [AndCondition] [winningPlayer second]
    - 0 Filled O [SimpleCondition]
    - 4 Filled O [SimpleCondition]
    - 8 Filled O [SimpleCondition]
  - diagUpFilledO [AndCondition] [winningPlayer second]
    - 2 Filled O [SimpleCondition]
    - 4 Filled O [SimpleCondition]
    - 6 Filled O [SimpleCondition]

Figure 5.22.: **o3InARow** *condition* with *sub-Condition*

To benchmark MCTS AI agent performance, *executions* were carried out and the result data aggregated. This can be seen in 5.1.3. Sixty *executions* were run in total, divided

Playouts	Draw	P1 Win	P2 Win	*executions*	Time (s)	Per *execution* (s)
50	20	0	0	20	150.07	7.50
500	20	0	0	20	266.73	13.34
1000	20	0	0	20	515.13	25.76

Table 5.6.: Tic-Tac-Toe win rates and execution time benchmarks

across sets of varying parameters. Both the number of MCTS iterations - referred to interchangeably as playouts or playout count - and quantity of *executions* per data set varied. The winning player (or draw) of each was recorded and summed across *execution* sets based on playout parameter count. There were three different values used: **50**, **500** and **1000**, where twenty *executions* were performed in total for each. Twenty *executions* which made use of **50** playouts were broken into two sets of ten sequential *executions*. Those using **500** playouts were divided into one set of ten and two sets of five. For the **1000** playouts set, two sets of ten were again used. The start and end timestamps for each sequential set were recorded and the respective mean time per playout set was calculated (see last column of table 5.1.3). All *executions* ended with a draw result, where neither AI agent won against the other.

The end game states from several *executions* were observed at random and no anomalies or rules errors were observed. The playout values (**50**, **500** and **1000**) were chosen because a direct relationship between higher playout values and superior performance was observed. This, however, was at the expensive of time and computing resources, which can be seen in 5.1.3. The shortest *execution* times were observed for **50** playouts, while the longest were for **1000**. Lower values than these produced unsatisfactory decision-making results, while higher values took significantly longer to finish. We believe the chosen values to be a good indication of what can be expected from normal use cases. If the number of playouts and corresponding wait time for a result is too high, the user experience will suffer. Likewise, if the results are generated quickly but without sensible strategic decision-making, Prototypical will not be as usable (or useful) in real game design context.

5.2. User Trials

A small study was conducted to measure the user experience of the current Prototypical web application interface. The goal was to present and explain the features of the software and estimate its utility through participant interviews. Another purpose was to solicit feedback and considerations for future work and improvements.

The target demographic was individuals of any age who identified as a board game designer or player. The latter in this case means that the individual willingly participated in a board game with others, whether regularly or only occasionally. Regularly implies a frequency of at least once per month, while occasionally indicates at least once per year.

Sessions took place individually over the course of two weeks. Each began with a short explanation about the intended use of Prototypical, followed by a tour of the

example from 5.1.1. The user was then instructed to explore various features and ask questions. It was then requested that the example was modified and one or more changes implemented in order to arrive at what the participant would consider a more-balanced or interesting design. Some users created additional *cards*, while others only modified *component property* values. All participants regularly asked questions and consistently required additional instruction.

To qualitatively measure these experiences, users were asked to fill out the System Usability Scale (SUS) questionnaire. The SUS is a Likert scale with ten statements, meant to gauge subjectively the usability of a given system. Answers range from scores one to five (inclusive), where one indicates that the participant strongly disagrees with the given statement. A score of five indicates that the user strongly agrees with the statement. The survey was presented electronically and without a time limit. All were completed immediately following the demo and each took fewer than fifteen minutes. The results were aggregated and translated into scores out of 100 [66].

The statements and raw user response data can be seen in table 5.8. Participant identities were mapped to letters A through E and all users answered all survey questions. The final two columns are the unconverted sum and average response value for the respective survey question. Similarly, table 5.2 provides the total survey score from each user, as well as the minimum, maximum, median and mean for the entire participant data set (all converted to 100-based range). The median total score was 62.5, where the overall range was between 50 and 75, inclusive. The mean score was 61.5, which indicates that the usability of the current Prototypical interface is "poor" (see 5.23). No further user studies are planned until after the relevant considerations in 6.1 and 6.2 have been made.

Score A	Score B	Score C	Score D	Score E	Minimum	Maximum	Median	Mean
65	55	62.5	50	75	50	61.5	62.5	61.5

Table 5.7.: System Usability Scale results from five participants (converted)

Score	Rating
> 80.3	A / Excellent
68 – 80.3	B / Good
68	C / Okay
51 – 68	D / Poor
< 51	F / Awful

Figure 5.23.: System Usability Scale Rubric

Statement	A	B	C	D	E	**Sum**	**Avg**
I think that I would like to use this system frequently.	4	3	3	5	4	**19**	**3.8**
I found the system unnecessarily complex.	3	2	1	4	1	**11**	**2.2**
I thought the system was easy to use.	4	3	3	2	4	**16**	**3.2**
I think that I would need the support of a technical person to be able to use this system.	2	4	3	3	4	**16**	**3.2**
I found the various functions in this system were well integrated.	5	4	5	5	5	**24**	**4.8**
I thought there was too much inconsistency in this system.	2	2	1	1	1	**7**	**1.4**
I would imagine that most people would learn to use this system very quickly.	4	3	2	2	3	**14**	**2.8**
I found the system very cumbersome to use.	3	2	4	4	2	**15**	**3.0**
I felt very confident using the system.	3	2	3	2	4	**14**	**2.8**
I needed to learn a lot of things before I could get going with this system.	4	3	2	4	2	**15**	**3.0**

Table 5.8.: System Usability Scale statements and participant responses (unconverted)

5.3. Hardware and Software

Implementation and experiments were done on an Apple MacBook Pro 2019 laptop with Intel i7 2.8 GHz Quad-Core processor, 16 GB of 2133 MHz DDR3 RAM, Intel Iris Plus Graphics 655 and MacOS 11.4. The IntelliJ (2020.1.1-2021.1.1) IDE with relevant plugins was used for both development and testing, with Git and GitHub for version control. Dependency management was handled by Gradle (5.6-7.1.1) and the web application was run in the Chrome (83.0-93.0) browser.

The back-end module was implemented in Kotlin (1.3-1.5.21) with a Java 1.8 compile target. The front-end module used the Flutter SDK beta channel until Flutter Web was officially released; after which SDK version 2.2.3 stable was used.

The SyncFusion [69] library handled all chart-related functionality in Prototypical. For this text, however, tables were created with assistance from Tables Generator [70], box plot diagrams were made using Box-and-Whisker Plot Maker [14] and bar graphs were done with or Bar Graph Maker [7] or code adapted from [6].

5.4. Discussion

Prototypical was successful in modelling the game concepts necessary to simulate playtesting of a limited scope. It demonstrated potential for board game designers if incorporated into the game design process and provided early playtest data without the need for human participants. This was accomplished using AI agents without significant domain knowledge and, after running simulations, the framework visualized subsets of the simulation data and displayed changes in game state parameters and potential trends. Both individual mechanics and a full simple rule set for two-player games were implemented, while a user study demonstrated that the current user interface is inadequate for practical use. We hold the opinion that improvements would make Prototypical a viable board game development tool for general public use. As such, it would offer a unique approach to modelling game mechanics without flowcharts like other existing software [42]. While hindered by the UI, current features of Prototypical do support the defined goals of modeling, simulating and analyzing game mechanics, as evidenced by the Movement Points, Polis: Rise of a City State and Tic-Tac-Toe examples.

Table 5.9 outlines the number of objects created when modeling the mechanic(s) of each example. Movement Points required the fewest and was the simplest mechanic of the three to implement. The second example required the most *components* but the fewest *conditions*, needing only one which was always met and another for tracking the current round number. It is unsurprising that Tic-tac-toe used the most *conditions*, as it contained the entire rule set of the game and *conditions* were necessary for every potential winning board state per player (e.g. *fields* **0**, **1** and **2** containing the same player value).

While implementing the examples may have been possible using fewer *components*, these quantities can be expected to represent typical usage of the software and should be considered from a usability perspective. Currently, all objects must be modeled individually, as no duplication features exist. As a result, all objects had to be created

	component	*action*	*condition*	Total
Movement Points	23	10	3	36
Polis: Rise of the City State	104	15	2	121
Tic-tac-toe	12	18	77	107

Table 5.9.: Object counts required to implement each example

and configured individually which required significant time. We see this as detrimental to overall usability and would expect improvements such as spreadsheet import and duplication to decrease the time required to model games and increase the usefulness of Prototypical as a whole.

From a positive perspective, most *executions* were run considerably fast. Tables 5.2 and 5.4 show the lowest, highest, median and mean run times per *execution* for each set of ten performed in the first two examples. Surprisingly, measurements were consistently higher for the Movement Points example than for Polis, which does not support a linearly increasing relationship between component count and run time. While this type relationship is expected to exist at a minimum, since more objects in memory requires more resources to manage, we believe the observed values are obfuscated by the resource overhead of the framework itself. This could be due to either the design and implementation choices or even those of the Kotlin programming language or Java Virtual Machine.

The resource requirements for the MCTS AI agent are evident from testing performed for Tic-Tac-Toe, which varied from roughly 2576 to 7500 milliseconds per individual *execution* (table 5.1.3). While longer than the run times of the other two examples, these benchmarks are still significantly faster than it would take two human players to complete games against one another. As *executions* are performed using the user interface asynchronously, we expect Prototypical to scale well for instances where run time far exceeds these ranges.

Support for asynchronous *executions* is achieved by performing each one on a separate thread in parallel, where the total time to execute all from a given set is close to that of its maximum recorded value. In other words, the total time to execute a set can never be less than that of any of its individual values and is normally expected to be slightly higher than its highest value. The consequences of this environment are again positive, implying that users can potentially perform many sets of simulations in a short period of time. When considering the amount of time required to carry out a manual playtest – disregarding any organizational and social overhead such as scheduling – the potential advantages of fast run times becomes clear. While we do not offer a process by which the value of simulated and manual play-tests can be compared or contrasted, we find it reasonable to assume that the time savings offered by the former would enable the designer to iterate quickly during the early stages of prototyping and development. This is an important characteristic for an iterative workflow and would avoid disrupting concentration due to long wait times.

The Movement Points example made use of several different *component* types and permitted design changes to be made iteratively. A complex AI agent was unnecessary, so the random AI agent chose moves accordingly. This did not require the user to specify winning player information using domain knowledge, only requiring input on which *components* corresponded to each player and what the respective top-level choices were. The performed simulations produced *execution result* data, which allowed the user to inspect game state before or after any move had been made. The ability to review this type of data after gameplay had already finished would not be easily achieved through standard playtesting. Other possible solutions include video, audio and manually-written gameplay event protocols, but all of these methods would have high technical overhead and/or become intrusive for participants and interrupt gameplay. We therefore consider these features of Prototypical to be a superior approach.

This first example was particularly suitable for Prototypical because it involved only cards and had low rules complexity. It was easy to implement gameplay-altering design changes, but this could have been attributable to the simplicity of the mechanic itself. Changing small quantities of values caused noticeable effects on the observed game states, possibly due to the low component count involved (eighteen cards). The changes themselves focused on updating one property type, which was also the case for the resulting simulation data. This made it easy to view both the values of the **move** property for each *frame* of an *execution result* data set, but also across multiple sets. The average of this property was displayed on a line graph and could have also been superimposed on other data, such as the *deck* **count** property.

We were successful in our attempt to use Prototypical to iteratively guide the development of this mechanic and effectively made use of the data visualizations to better understand gameplay effects from each modification set. Figures 5.3 and 5.4 show how consistently the first design modifications increased the minimum observed values, as all data points for the second *execution* set (11-20) are above those from the first set (1-10). This also holds true when comparing the first and last sets of visualized simulation data, as seen in figures 5.6 and 5.7. Figure 5.8 becomes relevant, as the bars for *executions* 21-30 are of a larger quantity and are distributed across a wider interval of higher values on the x-axis. These details support our claim that the design modifications succeeded in the original objectives of increasing the observed range and minimum values. It is also possible to visually confirm using figure 5.5 that the collective changes were symmetrical and did not provide either player a significant inherent advantage or disadvantage. While not expected to be exactly the same in shape or length, the line data for both players have a similar range and frequency of slope changes.

With this in mind, observing the characteristics of the visualized properties made it clear that design changes had taken effect but it did not help explain to what extent. It also did not provide enough information to estimate appropriate future changes. The end result of this example was that Prototypical could assist the designer to make observations and hypotheses about the current state of a game design but could not fully guide the development process alone. For this, a clearer characterization of the direct relationship between objective simulation data and the subjective user experience would be required.

The Polis: Rise of the City State example was of particular interest as a mechanic from a board game in active development. It was proposed directly by the designer and was unintentionally similar to the Movement Points example. Comparatively, however, a greater number of *components* were needed to model all of the necessary concepts. The manner in which meeple placement was handled demonstrated the flexibility of the framework and how a user has freedom when defining the details with which Prototypical constructs a given design. Some modeled entities had no design equivalent and existed as helpers for simulating a unique sub-mechanic. Likewise, this revealed how useful it might be to support additional physical and abstract component types, such as meeples and random number generators. With these available, meeple placement could have been modeled using an *action* to add a quantity of meeple components defined by the number generator, where meeple counts would be found by filtering for children of the respective type.

A modest number of simulations were taken into consideration when analyzing results, but there is no strong indication that this adequately sampled the game space. It is possible that the results and assumptions in this (and the previous) example contained biases and did not reflect the normal outcomes of the game. In other words, we do not have enough information to determine whether the observed results are typical or atypical of the given designs. To do so, we would require either mathematical analysis or real playtest data and evaluation for improved estimates.

What is clear in the second example, however, is that the chosen *component* modifications did not significantly increase the difference, or average thereof, between observed scores summed across multiple *executions*. As presented earlier, this difference in first and second player scores summed across all *frames* for *executions* 1 through 10 and 51-60 was 35 and 16, respectively. The corresponding averages were 8.75 and 4. The last ten *executions* (51-60) actually lowered this difference, potentially dispelling concerns about the implemented changes having given an inherent advantage to one of the players. Since we would expect these values to fluctuate in any case, this observation does not have clear consequences for our evaluation but we believe it is worth mentioning, regardless. This fulfills our initial condition that no modifications may introduce a large discrepancy in opposing scores.

The generated charts show that it is possible to unify multiple game state properties and associate them with a single visualization. The charts were used to display the total score of each player at each *frame* and used *compound queries* to do this for multiple *executions*. However, the combat score itself was comprised of the meeple count and card victory point value, demonstrating that it is possible to simultaneously track multiple elements of game state.

While not generated by Prototypical itself, figure 5.18 is valuable in analyzing the potential effects of the implemented changes on gameplay. We consciously made changes to increase the overall range and variability of the player scores, which is reflected by comparing several values from the first and last *execution* sets. Group 1 had a range from **10** to **16** (inclusive) which increased in size to eventually span from **9** to **18** in Group 6. It is possible (and likely) that the simulation results from this quantity of *executions* to

only partially sample the possible value space; however, we expect the results from ten simulations to provide an adequate estimate in order to make design decisions pertaining to the next iteration of development.

What is most pertinent about 5.18 is both the notable increase in standard deviation, as well as the visual differences in plotted data between the first and last groups. The standard deviation for Group 1 was **1.4539** and was **1.8211** for Group 6, which corresponds to an increase of twenty-five percent. We interpret this as an indication that the design changes implemented over the course of the six *execution* sets led to player score values more-evenly distributed and across a larger range.

The potential gameplay or user experience implications of this include reducing gameplay predictability and increasing the sense of fairness of the design. If a player can easily predict gameplay events and outcomes, the user experience would likely be negative. We expect the inverse of this to be true, namely, that these observed changes would positively influence the sense of enjoyment, fairness and challenge for the player experience. We expect similar insights to be valuable during the board game design process, which further supports the idea that Prototypical could be used to this end.

The Tic-Tac-Toe example proved that Prototypical can potentially be used to implement a full rule set and simulate an entire game using AI agents. The implementation did not require any workarounds for peculiarities of the framework itself. Gameplay was easy to follow during analysis, likely because of the simplicity of the game and the small board size. It was not a useful demonstration for how Prototypical might be involved in the development process but this is because the original aim was to demonstrate the modelling capabilities of the framework and the AI agent performance. Another appealing aspect of this game design was the ease with which the actions of the AI could be evaluated. Tic-Tac-Toe is easy to learn and it has a clear optimal strategy. While we did not use optimal moves as a benchmark, the random sampling of performed moves did not reveal obviously strategically sub-par choices which would indicate poor performance. Additionally, the results from table 5.1.3 indicate consistent decision-making behavior. Inconsistent win rates would indicate unpredictability from one or both opposing agents, indicative of an incorrect implementation [4]. We did not observe performance differences at these levels in the benchmarks but the execution time required increased linearly with the configured number of MCTS iterations. This was expected, as higher decision-making performance requires more iterations per selected move, resulting in longer computing time.

The user study and accompanying SUS surveys revealed both successes and failures of the Prototypical user interface. The time required to explain the premise and functionality of Prototypical, which was done before completion of the survey, was acceptable. This is supported by the responses to survey question two, which were generally low (i.e. positive sentiment). While participants asked questions throughout the demonstration, this could be interpreted in several ways. Without more information it is equally likely that the questions were asked out of curiosity rooted in comprehension or that they resulted from a lack thereof. Based on the collated average scores and an unexceptional median score of **3.0** (unconverted), we assume it is the former.

The results, however, indicated without doubt severe deficits in the UI. With only five participants, the user study was small in scale. We do not believe that increasing the sample size would have affected this result, due to the strong consistency of the responses. At best mixed, tending toward negative scores, it is unlikely that additional responses would have created a discrepancy or bias toward significantly different results.

We stated that no further user surveys would be pursued until additional features have been implemented because we believe the existing results to be reflective of the current feature set and UI. We expect that implementing features and improvements related to usability would have a strong positive influence on future survey scores. The same is also likely if additional documentation or an interactive tutorial were made available. It should be noted that we emphasize the utility of the core features of Prototypical and hold the opinion that it should not be overshadowed by the inadequacies of the UI. It may be worth exploring efforts to make Prototypical available as a programming library if improving the UI becomes untenable.

Other areas of concern are the ineffective handling of games with imperfect information and/or stochastic events. While the current implementation using chance nodes handles select situations well, it is clear that either a user will eventually encounter difficulties implementing other games or the simulation data (and AI performance) will not accurately reflect the modeled designs. The inability to comprehensively handle imperfect information limits the potential of Prototypical. We were unable to address these concerns within the scope and constraints of this work, but we believe future efforts could potentially rectify them. It may be possible to accomplish most of the work within a new AI agent, but other core framework changes would likely be required.

Prototypical may be of particular interest to researchers or other individuals who require a general platform for testing game-playing AI agents. Although not clear from the user interface, the framework was designed with extensibility in mind. It would be easy to implement new AI agents and provide a generic platform for modelling game rules. There is a potential use for general platforms which interface AI game-playing agents and provide rules modelling and enforcement, such as those utilized in the General Game Playing AI competition [52]. This saves the researcher time and the effort of implementing boilerplate functionality.

Applications of Monte-Carlo Tree Search appear to focus on one specific domain and rule set. Prototypical finds itself among other software such as Machinations.io [42] and Board Game Lab [10] in offering a generic MCTS AI agent in the context of games. These peers underscore its importance and the potential advantages when used to further the principles and techniques of game design.

We acknowledge potential in our approach and suggest that future research might explore fixing the UI-related issues or extracting the core functionality of Prototypical to a different interface altogether. As suggested, this could take the form of a programming library or API.

6. Conclusion and Future Work

6.1. Conclusion

This research aimed to create a software application to assist the board game designer in the context of an iterative development process. It attempted to investigate why such a tool might be useful and in what ways the assistance might take form. Our efforts yielded Prototypical, a board game development framework which has a graphical user interface and supports modelling, simulation and analysis of game mechanics. The included examples demonstrated all three of these features, as well as inadequacies of the user interface. However, it is clear that Prototypical may be useful to a game designer by providing new ways to view game state information, such as inspection of individual properties and the tracking and charting of one or more of these throughout the course of simulated gameplay. We also recognized potential for Prototypical to replace human playtesters, saving resources in the early stages of a mechanic's development.

In order to be modeled in the framework, a game mechanic must be turn-based and involve two players. Perfect information games and those with minimal stochastic events are where Prototypical excels, as hidden information is not supported and the use of chance nodes to handle non-deterministic actions is not optimal. The Movement Points and Polis: Rise of the City State examples showed how a simple card-based resource mechanic could be modified across several iterations. It was clear from the generated charts from before and after the implemented changes how the movement or score properties evolved throughout the performed simulations. It was immediately clear when the range of observed values increased or decreased and when the values varied at specific points in simulations. This comparative data and additional information is potentially helpful to the designer when making decisions. With the assumption that more information promotes improved decision-making, the utility of Prototypical becomes clear.

The Tic-Tac-Toe example further demonstrated the flexibility of the Prototypical framework, as well as its robustness and ability to support complete rule sets. Although the original scope remains a single mechanic at a time, evidence of other potential use cases remains beneficial to this research. Another potential use would see Prototypical become a general platform for testing AI agents, which was not considered until after its development. The AI and game rules are decoupled, which would allow other work to focus on applying AI agents to the game mechanics by avoiding excessive overhead. This example also demonstrated the performance of the UCT-based MCTS AI agent, which was found to be sufficient for the objectives of this research. Tests showed behavioral consistency, while random inspection of moves yielded only strategically-sound choices and the resulting AI agent decisions collectively resembled the skill of a human player.

While similar to several peers in overall functionality (see 2.2), Prototypical takes a novel non-flowchart approach using proprietary concepts such as *component selectors*, *and actions*, *checks*, *conditions* and *filters*. Although applying these concepts in practice appeared mostly intuitive to the study participants, the overall complexity of the software was higher than desired. This is evident in the critical aspects of the user interface and low System Usability Scores. Many usability features are absent from the UI, such as tool-tips and shortcuts to create many *components*, *actions* or *executions* at once. We would rate the Prototypical UI lower than its peers in this regard, yet on equal footing when it comes to its modelling and simulation approach and potential.

We are confident that future efforts have the potential to fix the aforementioned issues and implement additional features, making Prototypical a valuable tool for developing board game mechanics. Increased player counts, additional game types, improved UI workflows and stability improvements are all areas which come to mind. In the current state, the software controls a modest set of features and demonstrates potential to be naturally integrated into the board game design process. It is our hope that others agree and are willing to drive its evolution alongside the ever-changing landscape of board game development.

6.2. Future Work

The current AI algorithm is the core which drives simulation in Prototypical and could be extended to enhance both the simulation quality and coverage. The first obvious area of improvement is player count. For example, [50] point out the increasing popularity of MCTS in the context of multi-player games with more than two players and propose enhancements. Supporting even only four players would allow Prototypical to simulate many more game designs, as thousands of already available board games are intended for this number of participants [12].

The MCTS AI agent is convincing when applied to zero-sum games without hidden player information and, at most, inadequately so for others. Improving handling of stochastic events would improve overall simulation quality and the variety of supported designs by correctly considering hidden information, additional actions and more complex strategies. This could be achieved through additional AI agents, such as those following an Open-Loop MCTS approach as described in [51] for the context of video games. Improved AI would also enable Prototypical to support new game types.

The technical limitations of Prototypical make it most suitable for a small subset of game types. As covered in 3.1, many other types of games exist, which include simultaneous-play, social deduction and storytelling. While it is not within the scope of this work to provide approaches for supporting these genres, it is easy to expect Prototypical to both eventually be compatible with and useful in some of these contexts. For example, AI agents could be implemented to mimic specific human-like strategies during deductive actions or to have specific reactions to keywords in properties for components of a narrative-driven design. The previous work of [32] would be useful as a starting point for such work. Additional component types could of course be implemented in order to support actions specific to

any genre.

As the most immediate point of interaction between a user and the Prototypical framework, the user interface and respective aesthetics mold the user experience and present several areas of potential improvement. Additions to the user interface in order to improve the aesthetics or workflow and allow new usability features could be made. The results from user trials in 5.2 and respective interpretation in 6.1 further support this view.

The overall theme and visual style, including such aspects as button shapes, font styles, and color palette are perhaps the easiest and fastest to update. A simple and functional appearance was the goal during this research and there was no particular emphasis on aesthetic appeal or cohesion of the above stylistic considerations. Improving any of these would likely result in a different, possibly enhanced, user experience. Without venturing too far into the area of human-computer interaction, the user experience might also benefit from a resizing, reordering or restructuring of the current interface components. Some examples might be enlarging text for visibility on mobile devices, using wider drop-down menu buttons or adding help icons and tool tip popups to remind the user of specific features or elements. We suggest the field of user interface design as a source for principles to follow and incorporate, if further efforts in this area are made.

The first of such features could be the ability to input modelling data through spreadsheet import. It is possible to extend Prototypical with a parser for CSV-based (or proprietary format) data containing the necessary information to create multiple *components* at once. Since many game designs incorporate significant quantities of unique game pieces, it would likely be easier in many cases to use this import feature instead of manually creating each one through the existing interface.

In a similar vein, the ability to duplicate an existing *component* or *action* by use of a button might also prove useful. Due to implementation details, the underlying id and name of the object itself must vary; however, a dialog could allow the user to immediately set the required information during the duplication process.

Both UI areas would benefit from improved user input validation. There are several situations in which a user is able to provide incompatible input throughout the stages of *component*, *action* or *execution* creation. This may lead to ambiguous error messages and confuse the user as to how to proceed. A list of reserved words (e.g. count, id) could be maintained and shown to the user as a text tool-tip popup, along with a descriptive error message whenever the user disregards this.

These are only a few suggestions on how Prototypical might be extended to further achieve its original design goals. The software currently controls a small set of features but demonstrates potential to comprehensively assist the designer when it comes to inspecting simulated playtest data and searching for trends in evolving game state. We consider well-informed designers as having an inherent advantage when in designing enjoyable games and we believe that Prototypical can play a part in the process. At the very least, this software serves as inspiration for what a modern board game design system might provide and we invite the board game design community to explore and grow with it.

Bibliography

[1] Gaming: The next super platform. https://www.accenture.com/us-en/insights/software-platforms/gaming-the-next-super-platform. Accessed: 2022-02-28.

[2] The google deepmind challenge match. https://deepmind.com/alphago-korea. Accessed: 2021-10-03.

[3] David Altimira, Jenny Clarke, Gun Lee, Mark Billinghurst, Christoph Bartneck, et al. Enhancing player engagement through game balancing in digitally augmented physical games. *International Journal of Human-Computer Studies*, 103:35–47, 2017.

[4] Monte carlo tree search for tic-tac-toe game in java. https://www.baeldung.com/java-monte-carlo-tree-search. Accessed: 2022-02-04.

[5] Jonathan Barbara. Measuring user experience in board games. *International Journal of Gaming and Computer-Mediated Simulations (IJGCMS)*, 6(1):64–79, 2014.

[6] Basic bar chart with text as x axis labels. https://tex.stackexchange.com/a/8584/5645. Accessed: 2022-06-05.

[7] Bar graph maker. https://www.rapidtables.com/tools/bar-graph.html. Accessed: 2022-06-22.

[8] Marlene Beyer, Aleksandr Agureikin, Alexander Anokhin, Christoph Laenger, Felix Nolte, Jonas Winterberg, Marcel Renka, Martin Rieger, Nicolas Pflanzl, Mike Preuss, et al. An integrated process for game balancing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[9] Early prototype software. https://boardgamegeek.com/thread/1982913/early-prototype-software. Accessed: 2021-10-02.

[10] Boardgame lab. https://boardgamelab.app/. Accessed: 2021-10-02.

[11] Tools & resources. https://boardgamedesignlab.com/tools-resources/#art. Accessed: 2022-06-03.

[12] Boardgamegeek. https://boardgamegeek.com/. Accessed: 2021-10-02.

[13] Bruno Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. *Information Sciences*, 175(4):247–257, 2005.

[14] Box-and-whisker plot maker. https://goodcalculators.com/box-plot-maker/. Accessed: 2022-06-05.

[15] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. *AIIDE*, 8:216–217, 2008.

[16] Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[17] David Chircop. An experiential comparative tool for board games. *Replay. The Polish Journal of Game Studies*, 3(1):11–28, 2016.

[18] Luiz Jonatã Pires de Araújo, Alexandr Grichshenko, Rodrigo Lankaites Pinheiro, Rommel D Saraiva, and Susanna Gimaeva. Map generation and balance in the terra mystica board game using particle swarm and local search. In *International Conference on Swarm Intelligence*, pages 163–175. Springer, 2020.

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

[20] Fernando de Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. Ai as evaluator: Search driven playtesting of modern board games. In *AAAI Workshops*, 2017.

[21] Fernando de Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. Ai-based playtesting of contemporary board games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–10, 2017.

[22] Scythe, pandemic legacy and yogi creators on the unending, essential process of playtesting board games to perfection. `https://www.dicebreaker.com/topics/playtesting/feature/playtesting-board-games-scythe-pandemic`. Accessed: 2021-09-10.

[23] Joris Dormans. Visualizing game dynamics and emergent gameplay. In *Proceedings of the Meaningful Play conference*, 2008.

[24] Joris Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, volume 20, pages 33–40, 2009.

[25] Joris Dormans. Simulating mechanics to study emergence in games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, 2011.

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

[27] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[28] Fulvio Frapolli, Amos Brocco, Apostolos Malatras, and Béat Hirsbrunner. Decoupling aspects in board game modeling. In *Interdisciplinary Advancements in Gaming, Simulations and Virtual Environments: Emerging Trends*, pages 78–96. IGI Global, 2012.

[29] Gonzalo Frasca. Simulation versus narrative: Introduction to ludology. In *The video game theory reader*, pages 243–258. Routledge, 2013.

[30] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[31] Daniel E Gilbert and Martin T Wells. Ludometrics: luck, and how to measure it. *Journal of Quantitative Analysis in Sports*, 15(3):225–237, 2019.

[32] Carina Huchler. An mcts agent for ticket to ride. *Master's esis. Maastricht University*, 2015.

[33] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 293–300, 2014.

[34] Alexander Benjamin Jaffe. *Understanding game balance with quantitative methods*. PhD thesis, University of Washington, 2013.

[35] Aki Järvinen. Making and breaking games: a typology of rules. In *DiGRA Conference*, 2003.

[36] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[37] Eva Kraaijenbrink, Frank van Gils, Quan Cheng, Robert van Herk, and Elise van den Hoven. Balancing skills to optimize fun in interactive board games. In *IFIP Conference on Human-Computer Interaction*, pages 301–313. Springer, 2009.

[38] Jonas Krucher. Algorithmically balancing a collectible card game. *Bachelor's esis. ETH Zurich*, 2015.

[39] Thesis template - faculty of computer science, university of vienna. `https://www.overleaf.com/latex/templates/thesis-template-faculty-of-computer-science-university-of-vienna/whyzmtqggxzz`.

[40] Wei-Po Lee, Li-Jen Liu, and Jeng-An Chiou. A component-based framework for rapidly developing online board games. *International Journal of Computers and Applications*, 33(4):293–302, 2011.

[41] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: computer-assisted game level authoring. In *FDG*. ACM, 2013.

[42] Machinations.io. `https://machinations.io/`. Accessed: 2021-10-02.

[43] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Evolving card sets towards balancing dominion. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[44] Mango. `https://project.dke.maastrichtuniversity.nl/games/go4go/mango/index.htm`. Accessed: 2021-10-03.

[45] Videogames are a bigger industry than movies and north american sports combined, thanks to the pandemic. `https://www.marketwatch.com/story/videogames-are-a-bigger-industry-than-sports-and-movies-combined-thanks-to-the-pandemic-11608654990`, 2020. Accessed: 2022-06-03.

[46] Ahmad Mazyad, Fabien Teytaud, and Cyril Fonlupt. Monte-carlo tree search for the "mr jack" board game. *International Journal on Soft Computing, Artificial Intelligence and Applications (IJSCAI)*, 2015.

[47] nandeck. `http://www.nandeck.com/`. Accessed: 2021-10-02.

[48] Taro Narahara. Exploring board game design using digital technologies. In *ACM SIGGRAPH 2014 Studio*, pages 1–1. Association for Computing Machinery, 2014.

[49] Mark J Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[50] J Pim AM Nijssen and Mark HM Winands. Enhancements for multi-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 238–249. Springer, 2010.

[51] Diego Perez Liebana, Jens Dieskau, Martin Hunermund, Sanaz Mostaghim, and Simon Lucas. Open loop search for general video game playing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 337–344, 2015.

[52] Diego Perez-Liebana, Jialin Liu, and SM Lucas. General video game ai as a tool for game design. In *Tutorial at IEEE Conference on Computational Intelligence and Games (CIG)*, 2017.

[53] Rise of the city state | undead design lab. `https://www.undeaddesignlab.com/rise-of-the-city-state`.

[54] Riot games launches player dynamics to help improve multiplayer experiences. `https://venturebeat.com/2020/03/11/riot-games-launches-player-dynamics-to-help-improve-multiplayer-experiences/`. Accessed: 2021-10-02.

[55] Steve Roberts. Multi-armed bandits: Part 1. `https://towardsdatascience.com/multi-armed-bandits-part-1-b8d33ab80697`. Accessed: 2021-10-24.

[56] Denis Robilliard, Cyril Fonlupt, and Fabien Teytaud. Monte-carlo tree search for the game of "7 wonders". In *Workshop on Computer Games*, pages 64–77. Springer, 2014.

[57] How to make a tabletop simulator demo of your board game. `https://brandonthegamedev.com/how-to-make-a-tabletop-simulator-demo-of-your-board-game/`. Accessed: 2021-10-02.

[58] Christoph Salge and Tobias Mahlmann. Relevant information as a formalised approach to evaluate game mechanics. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 281–288. IEEE, 2010.

[59] Dilini Samarasinghe, Michael Barlow, Erandi Lakshika, Timothy Lynar, Nour Moustafa, Thomas Townsend, and Benjamin Turnbull. A data driven review of board game design and interactions of their mechanics. *IEEE Access*, 9:114051–114069, 2021.

[60] Jesse Schell. *Tenth Anniversary: The Art of Game Design: A Book of Lenses*. AK Peters/CRC Press, 2019.

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

[62] Adam Smith, Mark Nelson, and Michael Mateas. Prototyping games with biped. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, 2009.

[63] Adam M Smith, Mark J Nelson, and Michael Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98. IEEE, 2010.

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

[65] Squib. `https://squib.rocks/`. Accessed: 2021-10-02.

[66] Sus calculator. `https://uiuxtrend.com/sus-calculator/`. Accessed: 2022-02-16.

[67] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition. edition, 2018.

[68] Sutton barto summary chap 02 - multi-armed bandits. `https://lcalem.github.io/blog/2018/09/22/sutton-chap02-bandits`. Accessed: 2021-10-24.

[69] Buildbeautiful mobile & web apps with flutter widgets | syncfusion. `https://www.syncfusion.com/`. Accessed: 2022-06-03.

[70] Tables generator. `https://www.tablesgenerator.com/`. Accessed: 2022-01-21.

[71] Tabletop simulator. `https://www.tabletopsimulator.com/`. Accessed: 2021-10-02.

[72] Tabletopia. `https://tabletopia.com/`. Accessed: 2021-10-02.

[73] Unity real-time development platform | 3d, 2d vr & ar engine. `https://unity.com/`. Accessed: 2022-02-28.

[74] The most powerful real-time 3d creation tool - unreal engine. `https://www.unrealengine.com/en-US/`. Accessed: 2022-02-28.

[75] Vassal. `https://vassalengine.org/`. Accessed: 2021-10-02.

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

[77] Zuntzu. `https://www.zuntzu.com/`. Accessed: 2021-10-02.

A. Appendix

```
fun computeUct(
  parentVisits: Int,
  wins: Int,
  draws: Int,
  losses: Int,
  visits: Int
): Double {
  val v: Double = if (visits != 0) visits.toDouble() else return Double.
    MAX_VALUE
  val w: Double = wins.toDouble()
  val d: Double = draws.toDouble()
  val l: Double = losses.toDouble()
  val xi: Double = (w - l + d) / v

  return xi + c * sqrt(ln(parentVisits.toDouble()) / v)
}
```

Listing A.1: computeUct function