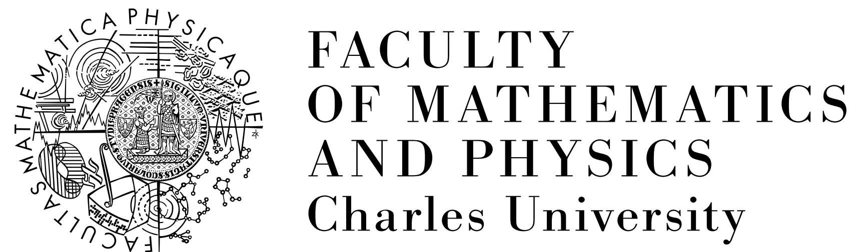

BACHELOR THESIS

Igor Minenko

Analysis and Development of Artificial Intelligence Based on Monte Carlo Tree Search for a Card Tabletop Game "7 Wonders"

Department of Software and Computer Science Education (201. • 32-KSVI)

Supervisor of the bachelor thesis: RNDr. Tomáš Holan, Ph.D.
Study programme: Computer Science (B0613A140006)

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my family for giving me the opportunity to do this work and their endless support. Also, many thanks to my supervisor, RNDr. Tomáš Holan, Ph.D., for his help and for the countless games played and the bugs he found during development. Special thanks to my friends who have been listening to me about this project throughout this time.

Title: Analysis and Development of Artificial Intelligence Based on Monte Carlo Tree Search for a Card Tabletop Game "7 Wonders"

Author: Igor Minenko

Department: Department of Software and Computer Science Education (201. • 32-KSVI)

Supervisor: RNDr. Tomáš Holan, Ph.D., Department of Software and Computer Science Education (201. • 32-KSVI)

Abstract: This thesis investigates the application of the Monte Carlo Tree Search (MCTS) algorithm to the board game 7 Wonders, with a focus on adapting it to the game's incomplete information and multi-player dynamics. We present a complete implementation of the game in Unity, integrated with an MCTS-based AI agent, and systematically analyze key algorithmic parameters—search depth, iteration count, and the UCB exploration constant—through simulations against Rule-Based and Random AI opponents under controlled conditions. The results demonstrate that, with appropriate modifications, MCTS is well-suited to address the unique challenges posed by 7 Wonders.

Keywords: Monte Carlo Tree Search, Card Tabletop Game, Artificial Intelligence, Games with Incomplete Information, Tree Search Algorithms

Název práce: Analýza a vývoj umělé inteligence založené na Monte Carlo Tree Search pro karetní deskovou hru "7 Divů světa"

Autor: Igor Minenko

Katedra: Katedra softwaru a výuky informatiky (201. • 32-KSVI)

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D., Katedra softwaru a výuky informatiky (201. • 32-KSVI)

Abstrakt: Tato práce zkoumá použití algoritmu Monte Carlo Tree Search (MCTS) ve stolní hře 7 Wonders, se zaměřením na jeho přizpůsobení neúplným informacím a vícehráčské dynamice této hry. Představujeme kompletní implementaci hry v Unity, integrovanou s AI agentem založeným na MCTS, a systematicky analyzujeme klíčové algoritmické parametry – hloubku prohledávání, počet iterací a explorační konstantu UCB – prostřednictvím simulací her proti pravidlovým a náhodným AI protivníkům za kontrolovaných podmínek. Výsledky ukazují, že s vhodnými úpravami je MCTS dobře použitelný pro řešení jedinečných výzev, které hra 7 Wonders představuje.

Klíčová slova: Monte Carlo Tree Search, Karetní desková hra, Umělá inteligence, Hry s neúplnou informací, Algoritmy vyhledávání stromů

Contents

Introduction **6**

1 Game Overview **7**
1.1 Short Game Description 7
1.2 Key elements 7

2 Game Analysis **11**

3 Related Work **13**

4 AI Agents **15**
4.1 Random AI 15
4.2 Rule-Based AI 15
4.3 Monte Carlo Tree Search 16
4.3.1 Multi-Armed Bandit Problem in MCTS 18

5 User Guide **21**
5.1 CLI Interface 21
5.2 Unity Interface 23

6 Programmer guide **32**
6.1 Implementation 32
6.2 Adding a New AI 33

7 Experiments **35**
7.1 Experimental Setup 35

Conclusion **44**

Bibliography **46**

List of Figures **48**

List of Tables **49**

List of Abbreviations **50**

A Attachments **51**
A.1 Technical Documentation 51
A.2 Python Script for Analysis 51
A.3 Unity .exe file 51
A.4 GameSimulationCLI folder 51
A.5 Core7W Library 51

Introduction

At the moment, Artificial Intelligence(AI) impresses with the speed of its development, and games are often involved in it. They have long been a reliable platform for testing AI agents due to their complexity, variability, different rules, and room for improvement. In games such as chess and GO, Artificial Intelligence testing is already considered classic, and AI does an excellent job in these games.

7 Wonders(7W), a card-based civilization-building game, offers particularly interesting challenges for AI implementation. As we will explore in the Game Analysis Chapter, the game's key challenges include hidden information such as unknown cards in opponents' hands, complex resource management and trading systems, multiplayer interactions with non-zero-sum outcomes and multiple victory paths.

These characteristics make traditional algorithms unsuitable for *7W*:

- **Minimax** fails primarily because it requires complete information about the game state, which *7 Wonders* doesn't provide due to hidden cards and simultaneous actions. Additionally, it's designed for two-player zero-sum games, while *7W* features multiple players with non-zero-sum outcomes.
- **Rule-based systems** struggle to adapt to the game's dynamic nature and unpredictable opponent behavior, often resulting in inflexible strategies.
- **Alternative approaches** like neural networks or evolutionary algorithms could potentially work, but would require extensive training data or computational resources, making them less practical for this implementation.

To address these challenges, this thesis focuses on designing, developing, and analyzing a Monte Carlo Tree Search (MCTS) implementation. MCTS is particularly well-suited for *7 Wonders* because it handles hidden information through statistical sampling and scales well to multiplayer games, also it balances exploration and exploitation naturally and adapts to changing game states without predefined rules.

1 Game Overview

1.1 Short Game Description

7 Wonders[1] is a card-drafting game in which each player leads an ancient civilization. At the start, each player receives a unique Wonder representing their civilization. The Wonder provides a starting resource and features several construction stages.

The game is played over three Ages - sets of rounds. At the beginning of each Age, a player gets 7 random cards specified for the current age.

Players use cards in one of three ways:

1. **Building a Structure:** Players pay the resource cost shown on the card to construct a building. There are different types of buildings, which is discussed in the Cards Subsection.

2. **Constructing a Wonder Stage:** A card may be used to build a stage of the Wonder if the player can pay the associated resource cost. Completing Wonder stages not only awards additional Victory Points(VP), but may also unlock special abilities.

3. **Discarding for Coins:** If a card does not fit into the player's strategy or if the required resources for other actions are unavailable, or the player just needs Coins, it can be discarded to gain 3 Coins.

After each round, the players pass the hand of cards to their neighbors clockwise(Age 1 and Age 3) and counterclockwise(Age 2).

Players collect resources through their buildings and can trade with their left and right neighbors to obtain missing resources. Moreover, after each Age, military conflicts are happened between neighboring players. The outcomes of these conflicts award VP to the winning side and penalize the losing side.

1.2 Key elements

Resources

In *7 Wonders*, managing resources is a fundamental aspect of gameplay. The game distinguishes between three main types of resources[2]:

- **Raw Resources**: These include Wood([3]), Stone(), Clay(), and Ore(). Raw resources are produced by specific buildings and are essential for the construction of many structures.

[1]The complete game rules and manual could be found in the official rulebook [1].

[2]The names of the following resources are not official, but we named them to simplify the development and explaining parts.

[3]All graphical assets in this chapter were taken from the official rulebook [1] and the reference sheet [2]

- **Manufactured Resources**: These comprise Glass( ), Papyrus( ), and Cloth( ). Manufactured resources are typically produced by specialized buildings and are often scarcer than raw resources.

- **Scientific Resources**: Cogwheel( ), Tablet( ) and Compass( ). These resources are significant for science and gaining a lot of VP through it.

Additionally, players manage Coins ( ), which act as the currency for resource trading, accumulate War Power( ) which is used to win the conflicts after each age, and the most important resource, Victory Points( ) - the indicator at the end of the game is who won.

The careful balance between accumulating resources and maintaining a coin reserve is crucial. Players must decide whether to build structures directly, to trade some resources for the next action, discard cards for Coins or even because a card is essential for the neighbor, that will get the hand after a turn.

Wonders

The game features seven Wonder boards(Figure 1.1), each corresponding to a famous ancient civilization.

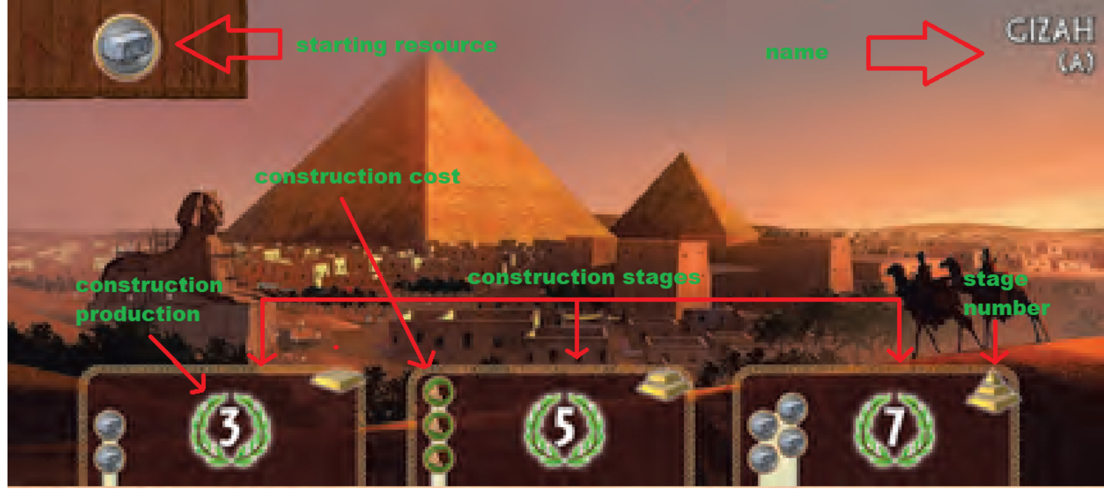

Figure 1.1 Describes Wonder board elements

Table 1.1 summarizes the Wonders along with the bonuses awarded at each construction stage.

Wonder	Stage 1	Stage 2	Stage 3
Babylon	+3	/ /	+7
Giza	+3	+5	+7
Alexandria	+3	/ / /	+7
Halicarnassus	+3	+ Discarded card	+7
Ephesus	+3	+9	+7
Olympia	+3	+ Free Build per Age	+7
Rhodes	+3	+2	+7

Table 1.1 Summary of Wonders and Their Stage Bonuses

Cards

The game deck is divided into three Ages, each representing a different phase of development in the civilization.

Each card, the example is on Figure 1.2, can have a cost in resources or could be free; an effect, such as resource production or military strength, and may have additional attributes such as chain construction.

Figure 1.2 Describes card elements

Cards are colored to indicate their primary function:

Brown	**Grey**	**Blue**	**Red**	**Green**	**Yellow**	**Purple**
Raw resources	Manufactured resources	Victory Points	War power	Science	Coins	Guilds

2 Game Analysis

Before we dive into the implementation, let's analyze *7 Wonders* in terms of complexity and the main features of games that are important to us.

Game Characteristics

Let's start the analysis with game characteristics that may cause problems for AI development.

Hidden Information

Although a fairly large amount of information remains visible to the player, such as built structures, opponents' resources and the statuses of the construction of the wonders, very important information is still hidden, such as the cards on other players' hands. Based on the knowledge of which cards are in the opponents' hands, it would be possible to make decisions related to the player's current move, for example, not to build the next part of the wonder now, but to build some kind of medium-importance card, because the next move the neighbor will hand over his cards, none of which are suitable for building to the player's current strategy and our player can decide to build the next part of the wonder on the next turn, and on the current turn build the structure suitable for the strategy and because of this feature, the game is a **hidden information** game.

Discreteness

7 Wonders is a fully **discrete** game. The main reason for it is that all actions players can perform are well-defined. The game progresses in rounds with structured turns where players make individual decisions in clear, discrete steps.

Stochastic Nature

There are elements of randomness in the game at the moment of dealing cards and at the moment of receiving unknown cards from an opponent, everything else in the game depends on the player and his strategic thinking, but this does not negate that the game is **stochastic**.

Simultaneous Moves

Another important feature of *7W* is that players make moves simultaneously within the main game cycle, meaning each player chooses the next action at different time intervals, but all players' actions are revealed at the same time. This adds an element of unpredictability to the game and puts it in the category of games with **simultaneous moves**.

Non-Zero-Sum Dynamics

Unlike chess, where when capturing a piece, depending on whether it is the correct move or not, one player loses something and the other gains, in *7 Wonders*,

the loss of one player does not necessarily lead to the gain of another (unless the war at the end of each Age belongs to this category) due to the variety of strategies, leading to a **non-zero-sum** game feature.

Non-Cooperative Play

The game is **non-cooperative**, as players independently build their civilizations, though there are elements of indirect interaction, such as resource trading with neighbors or war conflict.

Asymmetry

Each player starts the game with a unique wonder according to the rules, each such wonder gives a specific starting resource, just as the 2nd part of the building differs from each other. Many tactical decisions and even the strategy for the game are based on this. Because of this different start, *7W* falls into the category of **asymmetry**.

Decision Complexity

The game presents players with numerous choices each turn, creating a challenging decision space:

- **Card Actions**: For each card in hand, players typically have multiple options:
  - Construct the displayed building
  - Discard the card for coins
  - Use it to build a wonder stage (if applicable)
- **Resource Trading**: The ability to buy resources from neighbors further expands the range of possible actions.

This combination of multiple card options and potential resource trading creates a **high branching factor**, especially when accounting for the game's simultaneous moves and hidden information. These factors collectively make *7 Wonders* a complex environment for AI development, requiring approaches that can handle substantial uncertainty and combinatorial decision-making.

3 Related Work

Monte Carlo Tree Search has become a cornerstone of modern game AI, demonstrating remarkable success across both perfect and imperfect information games. The algorithm's flexibility and ability to handle large state spaces have made it particularly effective in domains where traditional search methods struggle. Seminal applications include:

- **Go:** The seminal work by Silver et al. [3] revolutionized computer Go by combining MCTS with deep neural networks in AlphaGo, ultimately defeating world champion Lee Sedol in 2016.
- **Real-Time Strategy Games:** Ontañón et al. [4] demonstrated MCTS's effectiveness in StarCraft through innovative techniques like adaptive macro-actions and opponent modeling, achieving human-competitive performance.
- **Card Games:** Cowling et al. [5] pioneered MCTS applications to Magic: The Gathering, developing novel determinization techniques that handle hidden information through multiple game state instances.

A key strength of MCTS lies in its ability to balance exploration and exploitation through the Upper Confidence Bound (UCB) formula, as comprehensively analyzed by Browne et al. [6]. Recent methodological advances have significantly expanded the algorithm's capabilities - particularly Keller's nested MCTS approach [7] for hierarchical decision-making and Baier and Winands' history-driven simulation strategies [8], which have proven effective in games with complex state transitions and partial observability.

Existing 7 Wonders Implementations

Several digital implementations of *7 Wonders* exist, providing different approaches to AI opponents:

- **Board Game Arena** (https://boardgamearena.com): Offers online multiplayer with rule-based AI opponents that follow predefined strategies.
- **Academic Implementations:** Pereira et al. [9] developed a framework specifically for AI research in *7 Wonders*, comparing different algorithmic approaches.

Academic Research on 7 Wonders AI

Recent academic work has explored various AI approaches for *7 Wonders*, but the most suitable for our work was introduced by Robilliard et al. [10], who established the foundation for MCTS in *7 Wonders*, addressing:

- **Progressive Widening:** To manage the game's expansive branching factor, the authors implement a progressive widening strategy that dynamically

adjusts the exploration of actions based on node visit counts. This technique effectively balances depth and breadth in the decision tree while maintaining computational feasibility.

- **Trading Mechanism Optimization:** The paper introduces depth using to resource trading between neighboring players.
- **Reward Function Formulation:** Through comparative analysis, the authors evaluate different reward structures, including:
  - Using an evaluation function
  - Using Monte-Carlo policy

  Their results indicate that using only the real score does not work.

Our Contributions

Our implementation makes several novel contributions:

- Our trading system incorporates advanced opponent modeling to predict resource availability
- Agents make decisions simultaneously using multi-threading
- Our work provides a visualization system in game engine Unity
- We don't use progressive widening as well as handle copying weak decisions for a reason, that will be discussed later 4.3.1
- We implemented more wider functionality of RbAI agent to compare with our MCTS agent

These advancements address key limitations in prior work while maintaining computational feasibility, as demonstrated in our experimental results (Chapter 7).

4 AI Agents

This section is devoted to the description of AI agents developed for this thesis, specifically for MCTS, we delved a little into the theory and provided the implementation details.

4.1 Random AI

Random AI is a baseline agent that makes decisions by randomly selecting an action from the available options. This agent serves as a naive benchmark for comparison against Rule-Based AI and MCTS.
In each game turn, the agent:

- Retrieves the list of available actions;
- Randomly selects one action;
- Returns that action to be executed;

4.2 Rule-Based AI

Rule-Based AI (RbAI) follows a predefined set of heuristic rules to determine its actions. Unlike Random AI, which makes arbitrary decisions, RbAI prioritizes specific game strategies based on human-designed logic.

The decision-making process of RbAI is governed by a hierarchical set of rules, where higher-priority rules take precedence over lower-priority ones. Below is the ordered list of rules that form the core of our Rule-Based AI implementation:

1. Play any card that gives two or more resources.
2. Play any card that provides a missing resource.
3. Play a military card if not the strongest in war.
4. Build a wonder stage with a random usable card.
5. Play the civilian card worth the most VP.
6. Play any science card.
7. If no other rule applies, play a random card.
8. Sell a random card.

It is worth noting that all these actions, except of selling of a random card, include the purchase of resources from opponents, if some action cannot be performed with their existing current goods.

4.3 Monte Carlo Tree Search

MCTS is a decision-making algorithm that is widely used in board games and other strategic applications. Unlike Rule-Based AI, which follows predefined rules, MCTS searches for the best move by simulating multiple possible future game states and evaluating their outcomes.

The MCTS algorithm works by repeating the following four steps:

1. **Selection** — The algorithm starts at the root node (current game state) and follows the most promising moves until it reaches a state that has not been fully explored.

2. **Expansion** — If the selected node is not a terminal state, one child is added to the search tree based on random taken action from the list of possible actions from the game state.

3. **Simulation** — A random simulation (also known as a "rollout") is performed from the newly added node to the end of the game. The result of this simulation is recorded.

4. **Backpropagation** — The result of the simulation is passed back up the tree, updating the values of the visited nodes to improve future decision-making.

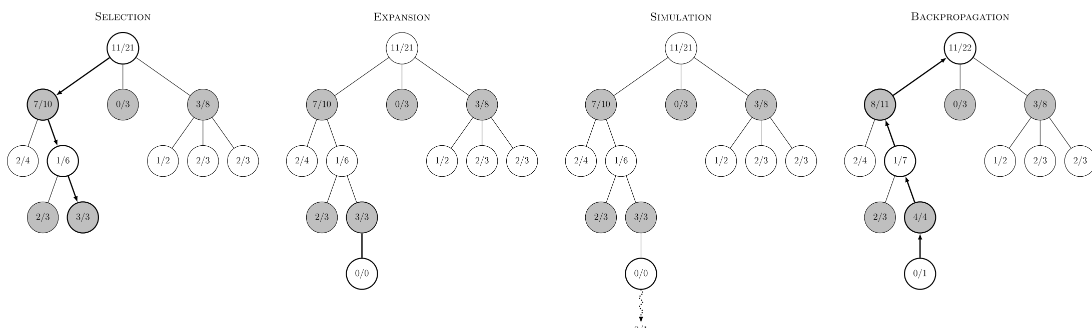

Figure 4.1 Four phases of Monte Carlo Tree Search (Selection, Expansion, Simulation, Backpropagation).
Source: Robert Moss, CC BY-SA 4.0, via Wikimedia Commons
`https://commons.wikimedia.org/wiki/File:MCTS-steps.svg`

MCTS continues running these four steps until a given time limit or iteration count is reached. The best move is then chosen based on the most visited or highest-value node.

Classical MCTS alternates between players' turns in its tree structure. For *7 Wonders* (where all players act simultaneously), we redefine nodes as a joint *game state* containing all players' chosen actions, where single expansion step processes all players' moves in parallel. Each node represents the game state after all players make their actions. It preserves the game's core mechanic while enabling MCTS search and solves the problem with simultaneous moves.

Algorithm 1 MCTS Main Loop

Require: Root node $root$, number of iterations N
Ensure: Best action a_{best}

```
1: for i ← 1 to N do
2:     node ← root
3:
4:     Selection Phase:
5:     while node is not leaf do
6:         Select child node using UCB:
7:         child ← argmax_c ( Q(c)/N(c) + C sqrt( ln N(node) / N(c) ) )
8:         node ← child
9:     end while
10:
11:    Expansion Phase:
12:    if node is not terminal then
13:        Expand all possible actions from node
14:        Select one child child randomly
15:        node ← child
16:    end if
17:
18:    Simulation Phase:
19:    Run simulation from node to terminal state
20:    Obtain simulation result Δ
21:
22:    Backpropagation Phase:
23:    while node ≠ root do
24:        Update statistics:
25:        N(node) ← N(node) + 1
26:        Q(node) ← Q(node) + Δ
27:        node ← parent node
28:    end while
29:
30: end for
31: return action with highest N(child) from root
```

At the beginning of each Age, players are dealt 7 hidden cards, with no knowledge of other players' hands. This hidden information aspect can be addressed using Perfect Information Monte Carlo (PIMC) [11].

Perfect Information Monte Carlo handles hidden information by sampling *determinizations*—hypothetical game states consistent with a player's observations. While effective in games like Dou Di Zhu, this approach suffers from two critical limitations:

- **Strategy Fusion:** Actions are evaluated independently across determinizations, leading to inconsistent utility estimates. For example, in *7 Wonders*, a card enabling long-term science strategies might be undervalued if sampled determinizations prioritize short-term resource gains.

- **Non-Locality:** In games like poker, PIMC may preserve impossible states (e.g., unplayed strong hands still being considered).

Relevance to 7 Wonders

In *7 Wonders*, these issues are partially mitigated due to card drafting's cyclical nature, which minimizes long-term strategy fusion risks and non-locality problem. Players openly track opponents' cards, reducing non-local state complexity by tracking the cards that have already been seen by our player, so the number of possible game states will decrease significantly and after n - 1 moves, where n is the number of players, our agent will know all the cards in the opponents' hands and will not have to do sampling. The example with three players on the Figure [4.2] explains how it works where Player C is our player and 0/7 means, that Player C does not have information about any of cards and 7/7 means the Player C knows every single card in a hand.

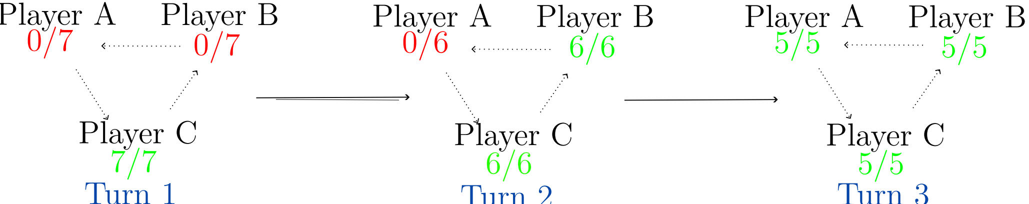

Figure 4.2 Cards transfering during the first 3 turns.

As we mentioned in 3, we introduce new technique of avoiding increase of branching factor by trading. If we do not apply any enhancements, it would led to huge amount of useless branches. So firstly, we don't add any of "only trading" branches to our tree. Secondly, all branches come from a list of all possible actions, that contains selling cards, building wonder and building structures without trading, but if player can not build a structure, our solution provides the most beneficial way how, what resources and from who to buy these resources. This allows us to add to the list of possible actions elements, that couldn't be performed without trading and instead of having more than one action with the same last actions (e.g. build structure or build wonder), but with different previous action such as buying different resources, we get only one action for the specific need, which solves the problem of increasing branching factor by useless actions.

4.3.1 Multi-Armed Bandit Problem in MCTS

Monte Carlo Tree Search relies on a balance between **exploration**, trying less-known actions to gather more information, and **exploitation**, choosing the best-known action to maximize rewards. This challenge is often framed as the *Multi-Armed Bandit* (MAB) problem, where an agent must decide which "arm" (action) to pull in order to achieve the best long-term gain. Imagine standing in front of four slot machines (Fig. 4.3), each with hidden reward probabilities:

- Machine 1 pays 50% of the time
- Machine 2 pays 70% of the time
- Machine 3 pays 35% of the time

- Machine 4 pays 45% of the time

The challenge: You don't know these probabilities in advance! To maximize your total rewards, you must:

- **Explore** different machines to learn their payouts
- **Exploit** the best-known machine once identified

This is the core trade-off in the Multi-Armed Bandit problem. In MCTS:

- Each "machine" represents a possible game move
- "Payout" is the estimated win probability from that move
- The UCB formula helps balance testing new moves vs choosing strong ones

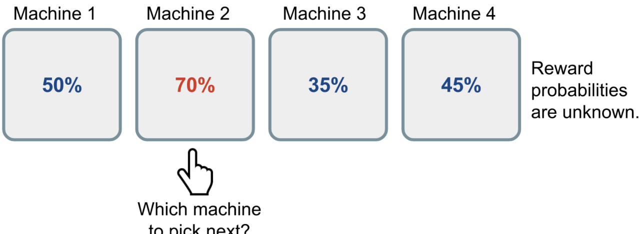

Figure 4.3 The exploration-exploitation dilemma: Choosing between machines with known (blue) and unknown (red) reward probabilities [12].

Upper Confidence Bound

To address this trade-off, MCTS uses a modified UCB formula that balances exploration and exploitation:

$$\text{UCB}(i) = \underbrace{\frac{w_i}{n_i}}_{\text{Exploitation}} + \underbrace{C\sqrt{\frac{\ln N}{n_i}}}_{\text{Exploration}}$$

- w_i: Cumulative reward of action i
- n_i: Visit count of action i
- N: Total visit count of parent node
- C: Exploration constant ($C > 0$)

In our implementation, we denote the exploration constant as C_{MCTS} (equivalent to `kUCT` in code) and optimize its value through empirical testing. The standard UCB1 formula uses $C = \sqrt{2}$, but we treat it as a tunable hyperparameter.

Impact on Decision-Making in 7 Wonders

In the game of *7 Wonders*, different decisions must be evaluated at different phases of the game:
- Early game: More exploration is encouraged to evaluate different strategies.
- Mid-game: A balance between exploration and exploitation is maintained to refine strategies.

- Late game: Exploitation is prioritized, as the agent has gathered sufficient knowledge about the effectiveness of different actions.

We tuned C_{MCTS} based on the decision making and the results of such a tuning are here[7.1]

On Handling Weak Moves

In many game AI implementations, such as chess engines, weak moves are often pruned to focus computational resources on promising lines of play. We initially attempted a similar approach in our *7 Wonders* AI by skipping card selling actions during best child evaluation - only considering them if no other moves were available.

However, this optimization proved problematic in late-game scenarios where selling cards can be strategically crucial. Consider this common situation:

- Player holds 3 cards, including one highly valuable (7 VP) card player cannot build
- Neighbor could potentially build this valuable card if passed
- The "optimized" evaluation would skip selling and choose to:
  - Build a worthless card (0 VP gain)
  - Pass the valuable card to the opponent

This results in a double disadvantage:

1. Wasted turn building a non-beneficial structure
2. Gifted opponent significant VP

We therefore maintain all possible moves in all our phases, recognizing that in *7W*, even ostensibly weak moves like selling cards can become strategically vital in specific contexts.

5 User Guide

The user documentation consists of two main parts covering different ways to run the program: via Command Line Interface (CLI) or through the Unity executable.

5.1 CLI Interface

The game simulation tool provides a flexible Command Line Interface that supports both interactive configuration and parameterized execution. This section documents all available usage modes and configuration options.

Basic Usage

The simulation can be started with two distinct modes:

- **Interactive mode**: Launched by simply executing the program without arguments:

  ```
  dotnet run
  ```

- **Parameterized mode**: Allows specifying agents and their configurations through command line arguments:

  ```
  dotnet run --ai=<agent-specifications>
  ```

Interactive Mode

When launched without arguments, the program enters an interactive configuration mode:

1. The system presents a menu of available AI types:
   - Random
   - MCTS
   - RuleBased
2. For MCTS agents, users can configure:
   - Iterations count
   - Time limit per move in seconds
   - Search depth
3. Random and RuleBased agents require no configuration
4. The process continues until the user selects "Finish configuration"

Parameterized Mode

The CLI accepts agent specifications in the following format:

```
--ai=<agent1-spec>;<agent2-spec>;...;<agentN-spec>
```

Agent Specification Formats

- **Simplified form** (for Random and RuleBased):

  ```
  random
  rulebased
  ```

- **Full form** (for MCTS with parameters):

  ```
  mcts:iterations,time_limit,max_depth
  ```

- **Examples**:

  ```
  --ai=random;mcts:100,10,4;rulebased
  --ai=mcts:500,10,20;mcts:200,10,20;rulebased
  ```

Configuration Parameters

Agent Type	Parameter	Default	Description
Random	-	-	No configuration needed
RuleBased	-	-	No configuration needed
MCTS	Iterations	100	Number of MCTS iterations
MCTS	Time limit	10s	Maximum time per move
MCTS	Max depth	100	Tree search depth limit

Table 5.1 Agent Configuration Parameters

Runtime Options

After agent configuration, the simulation asks for Number Of Games parameter, which specifies how many complete games to simulate (default: 1000)

Output and Results

The simulation provides real-time progress updates and summarizes results upon completion:

- Progress is displayed as number of completed games
- Final statistics are saved in the specified output directory
- Console output confirms successful completion or reports errors

Listing 5.1 Example successful execution output

```
=== Game Simulation CLI ===

=== Final Configuration ===
Configured agents:
Agent 1: MCTS - Iterations=500, TimeLimit=5s, MaxDepth=6
Agent 2: Random - Simple strategy (no parameters)
Agent 3: Random - Simple strategy (no parameters)

Starting Game Simulation...

Number of games to simulate (default 1000): 1000
0
1
.
.
998
999
Simulation ended.
```

All errors are reported with descriptive messages to help diagnose configuration problems.

5.2 Unity Interface

Upon launching the .exe[1] file, the game displays the main menu (Figure 5.1) containing two buttons:

- **Play** - Starts a new game session
- **Quit** - Exits the application

Figure 5.1 Main menu interface with play and quit options

After clicking the Play button, the game settings window appears (Figure 5.2). Table 5.2 explains each interactive component.

[1]See attachments - A.3

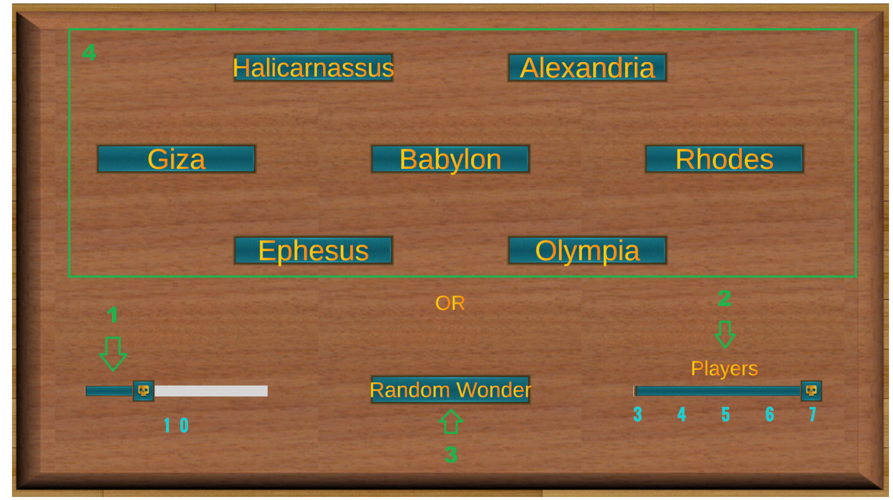

Figure 5.2 Game configuration window with settings options

#	Component	Description
1	Time Slider	Adjusts the time limit (in seconds) for MCTS agents' turn computation
2	Player Count Slider	Sets the total number of players (including human player)
3	Random Wonder	Button to assign a random wonder to all players
4	Wonder Buttons	Buttons to select specific wonder (click for details in Figure 5.3)

Table 5.2 Game Settings Components

Figure 5.3 Detailed wonder information panel

After configuring the settings, the game session begins. Figure 5.4 shows an

example session with the Alexandria wonder selected and 4 total players. The interface components are explained in Table 5.3.

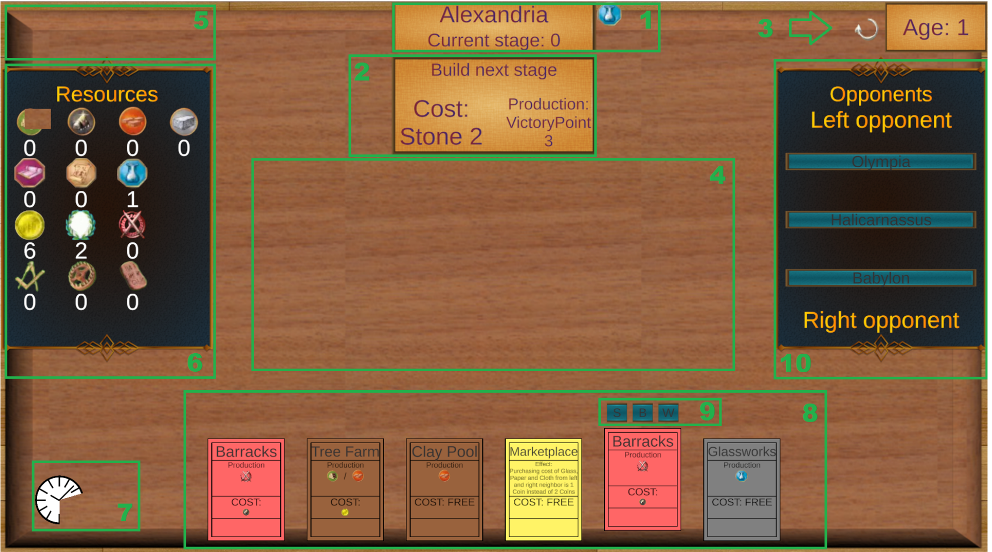

Figure 5.4 Main game interface during play session

#	Component	Description
1	Wonder Info Panel	Displays current wonder name, stage, and starting resource
2	Next Stage Panel	Shows requirements and production for the next stage
3	Age Panel	Displays current age and the direction of card rotation
4	Built Card Area	Shows cards built by the human player, organized by type
5	Notifications Area	Displays game notifications
6	Resource Panel	Displays all of the human player's resources
7	Time Panel	Indicates remaining time for AI decision-making
8	Hand Cards Panel	Shows all cards in the human player's hand
9	Action Buttons	When hovering over a card, three action buttons appear above (S - sell card, B - build card, W - build wonder stage)
10	Opponent Buttons Panel	Contains opponent buttons; clicking them displays Figure 5.5

Table 5.3 Game Interface Components

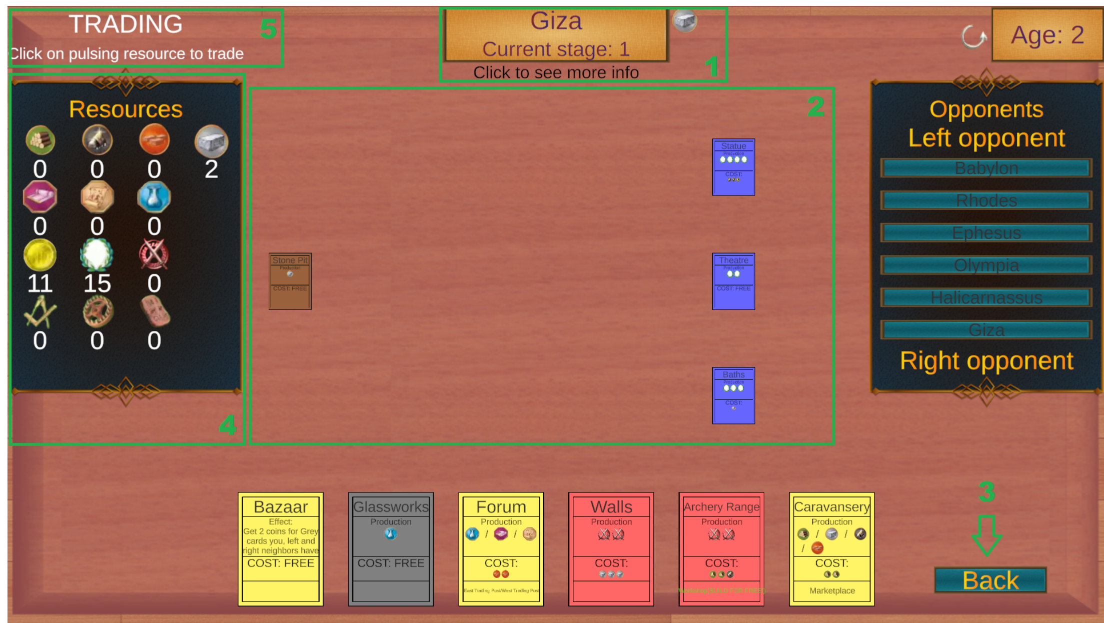

Figure 5.5 Opponent view interface

#	Component	Description
1	Wonder Info Panel	Displays AI player's wonder name, stage, and starting resource
2	Built Card Area	Shows cards built by the AI player, organized by type
3	Back Button	Returns to the human player's view
4	Resource Panel	Displays all of the AI player's resources (available trading resources pulse)
5	Trading Hint Area	For neighboring AI players (left or right of human player), displays a "Trading" note indicating possible trades

Table 5.4 Opponent Interface Components

When a human player hovers over the "B" button to build a card, missing resources turn red with the required amount (Figure 5.6). If no resources are red, the player can build the selected card.

Figure 5.6 Not enough resources example

After each age, military conflicts occur between neighboring players. Each number represents a player's military strength. After all players complete their actions, the war window appears (Figure 5.7) with a "battle" animation for the first conflict. Clicking the "NEXT FIGHT" button initiates the animation for the next pair of players. If no animation occurs, the result is a draw.

Figure 5.7 War conflict resolution interface

Players can build cards for free using chain construction (Figure 5.8). As stated in the official rules (see [1].): "Some structures in Age II and Age III have, to the right of their resource cost, the name of a structure from a previous age. If the player has built the named structure during a previous age, they can build the current structure for free, without fulfilling the resource cost."

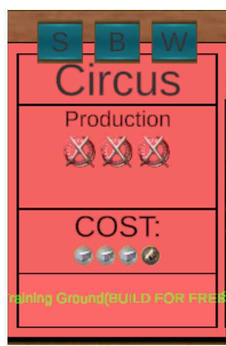

Figure 5.8 Free construction example

Two wonders in the game have special abilities when their second stage is built: Halicarnassus (Figure 5.9) and Babylon (Figure 5.10). The Halicarnassus ability allows the player to build any card from the discard pile for free immediately after completing the second stage. The Babylon ability grants an additional scientific symbol of the player's choice at the end of the game.

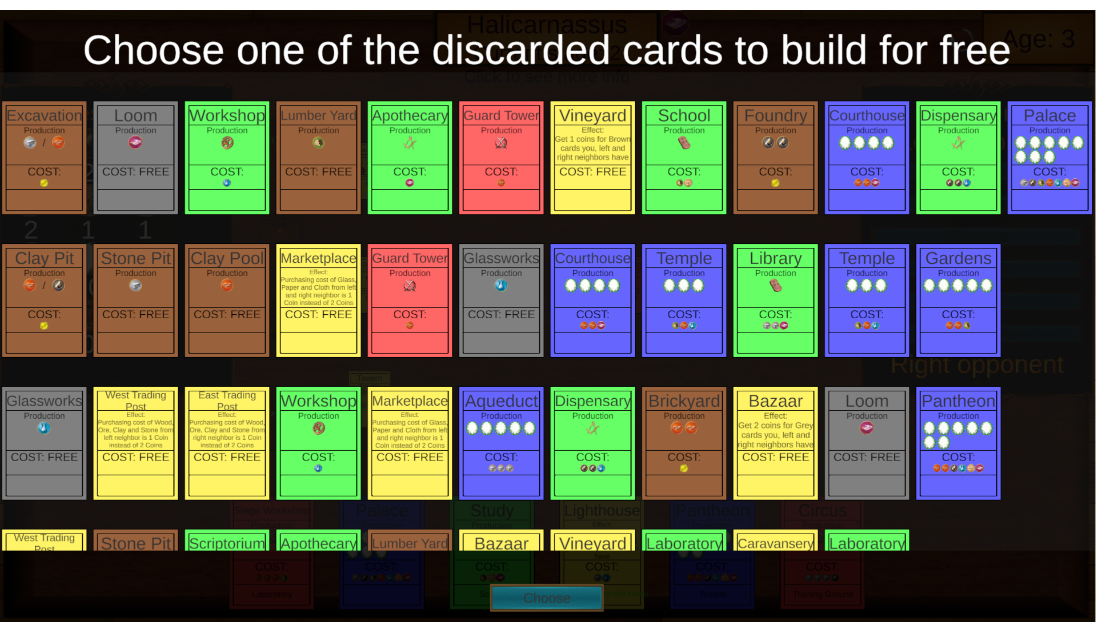

Figure 5.9 Halicarnassus special ability interface

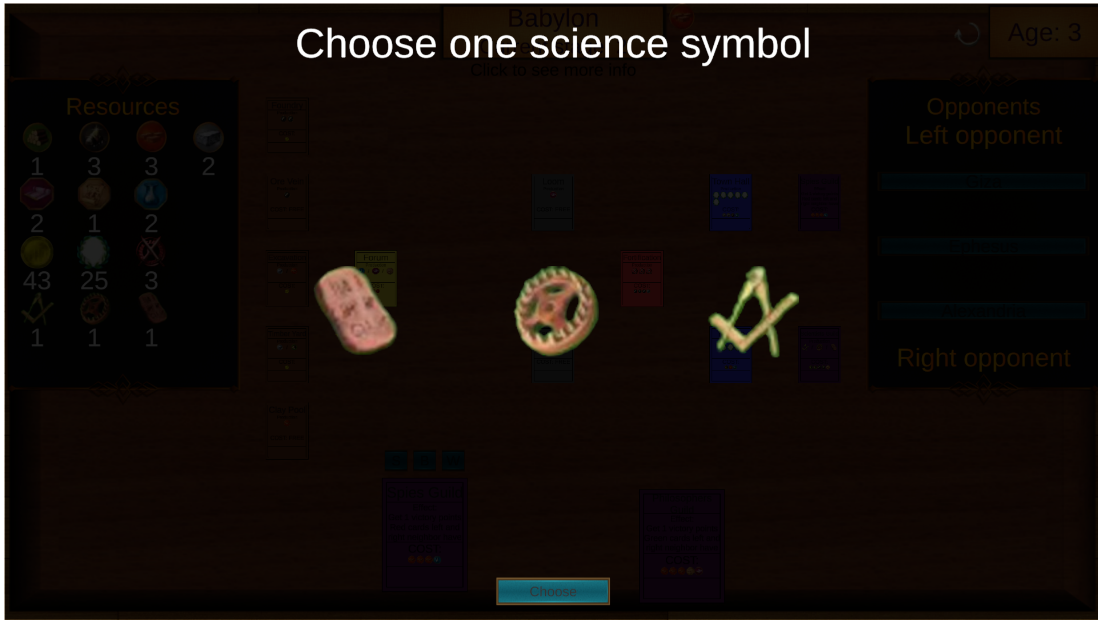

Figure 5.10 Babylon special ability interface

After the final military conflict of the third age, the results are calculated and displayed (Figure 5.11). The player can either return to see the final game state by clicking the "Back" button or proceed directly to the Rankings (Figure 5.12) to view global results, where all previous best games are saved and stored.

Figure 5.11 Final results screen

RANKINGS

PLACE	NAME	WONDER	OVERALL SCORE	WAR POWER	BUILD STRUCTURES	BUILD WONDER PART	COINS	DATE
1	Napoleon Bonaparte	Ephesus	56	2	17	1	3	1821-05-05
2	Alexander the Great	Rhodes	52	1	16	1	6	1323-06-10
3	Genghis Khan	Halicarnassus	48	1	14	1	0	1227-08-18
4	Julius Caesar	Halicarnassus	48	1	15	1	7	1044-03-15
5	You	Olympia	48	9	17	1	26	2025-04-16
6	You	Rhodes	48	9	15	1	25	2025-04-16
7	Cleopatra VII	Alexandria	46	2	14	2	18	1030-08-12
8	Abraham Lincoln	Giza	44	2	15	1	10	1865-04-15
9	Hannibal Barca	Giza	44	2	14	2	7	1183-01-015
10	Saladin	Babylon	43	1	14	1	13	1193-03-04

MAIN MENU

Figure 5.12 Global rankings screen

Clicking the Main Menu button returns the player to the starting scene (Figure 5.1), ready to begin a new session.

6 Programmer guide

This chapter presents the key elements of the AI system implementation for the game *7 Wonders* with a primary focus on the Monte Carlo Tree Search algorithm. It provides an overview of the AI architecture, core decision logic, and practical instructions for extending the framework with new AI agents. For more detailed technical documentation check A.1.

6.1 Implementation

The primary entry point for any AI-driven decision-making lies in the `AITurnAsync` method defined in `TurnCore.cs`:

```
public static async Task AITurnAsync(Player opponent,
    List<ActionTurn> availableActions,
    CancellationToken token,
    Determinization determinization = null)
```

This method delegates the decision-making to a specific AI agent, which is dynamically selected using a factory pattern (`AIFactory`). The agents implement a common interface:

```
public interface IAgent {
    Task<ActionTurn> MakeDecisionAsync(Player currentPlayer,
        List<Player> players,
        List<ActionTurn> availableActions,
        CancellationToken token);
}
```

MCTS Core Logic

The `MCTS` class is the heart of the AI system for advanced decision-making. It encapsulates the full MCTS process: selection, expansion, simulation, and backpropagation. The method `RunMCTSAsync` is responsible for executing the MCTS cycle with a given number of iterations or time constraints.

```
public async Task<Node> RunMCTSAsync(
    GameInfoSet rootInformationSet,
    int maxIterations,
    float maxTime,
    CancellationToken token)
```

Each simulation is performed on a `Determinization` of the current game state, ensuring stochastic aspects are handled correctly. The selection policy uses UCB to balance exploration and exploitation:

```
double exploration = kUCT *
    Math.Sqrt(Math.Log(totalVisits) / childNode.Visits);
```

Simulations proceed using the actual game logic with opponents taking actions via recursive calls to `AITurnAsync`.

Key Components

- **Node.cs** – Represents a node in the MCTS tree and tracks visits and rewards.
- **Determinization.cs** – Encapsulates a fully specified game state, allowing simulations to proceed deterministically.
- **GameInfoSet.cs** – Encodes the partial knowledge of a player, supporting imperfect information scenarios.
- **MCTSAI.cs** – Adapts MCTS logic into an `IAgent` implementation, making it pluggable.

6.2 Adding a New AI

Adding a new AI agent to the system is a straightforward process, thanks to the use of polymorphism and factory instantiation.

Step-by-Step Integration

1. **Implement the `IAgent` Interface**

   Create a new class implementing `IAgent`. For instance, a greedy AI[1] might look like this:

   ```
   public class GreedyAI : IAgent {
       public async Task<ActionTurn> MakeDecisionAsync(
           Player currentPlayer,
           List<Player> players,
           List<ActionTurn> availableActions,
           CancellationToken token) {

           return availableActions.OrderByDescending(
           a => a.EstimatedValue).First();
       }
   }
   ```

2. **Add an Enum Entry to `AIType`**

   Include your new AI type in the `AIType` enum:

   ```
   public enum AIType {
       Random,
       MCTS,
       RuleBased,
       Greedy
   }
   ```

[1] We don't have an implementation for ActionTurn.EstimatedValue, so we will use it just for an example.

3. **Extend the Factory**

   Update the `AIFactory` to support your new type:

   ```
   public static IAgent CreateAgent(AIConfig config = null) {
       return config.AgentType switch {
           AIType.Random => new RandomAI(),
           AIType.MCTS => new MCTSAI(config),
           AIType.RuleBased => new RuleBasedAI(),
           AIType.Greedy => new GreedyAI(),
           _ => throw new ArgumentException(
           $"Unknown AI: {config}"
           )
       };
   }
   ```

4. **Configure the Agent**

   Ensure that your game's AI configuration system (`AIConfig`) as well as `GameSimulation.sln` allows selecting the new AI type.

7 Experiments

This chapter presents the experimental evaluation of the artificial intelligence agents developed in this thesis, with a primary focus on assessing the performance of the Monte Carlo Tree Search agent. The goal is to quantitatively measure the strength of the MCTS agent compared to baseline strategies, namely a Random agent and a Rule-Based agent, within the context of the game *7 Wonders*.

Some experiments are designed to produce statistically significant results by simulating many games under controlled, diverse conditions. This applies to those experiments where we will analyze one specific parameter and only one MCTS out of all players will be tested, such as depth or C_{MCTS}, because such a setup ensures that the evaluation of the performance difference between different games will depend only on these different parameters. To enable the experimental setup with Wonder combinations and card dealing variations, set `isExperimentalSetup = true` in `Program.cs` file of GameSimulationCLI folder A.4 before compiling.

All visualizations presented in this chapter - including win rate progression curves and score distribution plots - were generated programmatically from the experimental data, ensuring reproducibility of our findings. The script for analysis is included in the Attachments[See Python script for analysis].

7.1 Experimental Setup

AI Agent Parameters

The different AI agents were configured as follows:

- **Random Agent:** Selects actions uniformly at random from the set of legal moves available at each decision point. No parameters are required.
- **Rule-Based Agent:** Implements the heuristic strategy described in Section 4.2. No parameters are required.
- **Monte Carlo Tree Search (MCTS) Agent:** Implements the MCTS algorithm detailed in Section 4.3. Key parameters were set as follows:
  - **Time for one turn**
  - **Number of iterations for one turn**
  - **Exploration Constant ($C_{\mathbf{MCTS}}$)**
  - **Simulation depth**

Performance Metrics

The primary metrics used to evaluate agent performance are:

- **Win Rate:** The percentage of games won by an agent within a given experimental setup.
- **Average Score:** The mean final score achieved by an agent across all games in a setup.

Preface

Firstly, good to be noted, we employ a dynamic iteration scaling system based on the current Age of the game. The number of MCTS iterations increases by 1.5x for Age II and 2x for Age III compared to the base value used in Age I, because in later Ages, the remaining game depth decreases significantly, causing simulations to complete much faster. It means if we name our agent as MCTS-100, this agent has 100 iterations for the first Age, 150 for the second and 200 for the third, but we will use this notation for simplicity and one common standard.

Secondly, in original rules if some of the players have the same number of VP, the Coins are considered to be main aspects to identify the winner, but we count a win for our agent only if it has more Victory Points, than others, because in our evaluating function we use VP amount only.

We chose exactly 3 agents for the experiments, because in this way each player can influence everyone through trading, military conflicts and the transfer of cards. For example, if there were more than 3 players in the game, then there would be pairs of players who could not trade with each other, just as at the end of each Age they would not have military conflicts with each other. Also, in such a setup, it does not matter in which order the players are placed, unless there is a slight difference in the direction of the transfer of cards, because if we designate the players as A, B and C, and "put" the players at the table in the order from left to right A, B, C, then the arrangement of B, A and C will differ in that player A passes the cards in the first age to player B, and in the arrangement A, B, C to player C. We suggest making sure that this does not affect our implementation as well as look how our implemented agents play against one Random agent with random seed data in Table 7.1.

Good to be noted, that parameter Number Of Games, that we stated in subsection 5.1, does not effect experiments since it will always start 1050 games because of the nature of the setup.

Configuration	Agent	Win Rate (%)	Avg. VP
Group 1: MCTS-100 as Player 1			
MCTS-100 vs MCTS-50 vs Random	MCTS-100	67.2 ± 2.9	39.0
	MCTS-50	31.9 ± 2.9	33.7
MCTS-100 vs Random vs MCTS-50	MCTS-100	66.5 ± 3.0	38.4
	MCTS-50	32.1 ± 3.0	34.0
Group 2: MCTS-50 as Player 1			
MCTS-50 vs MCTS-100 vs Random	MCTS-100	62.5 ± 3.0	38.3
	MCTS-50	36.5 ± 3.0	34.5
MCTS-50 vs Random vs MCTS-100	MCTS-100	65.7 ± 2.9	38.6
	MCTS-50	33.6 ± 2.9	34.1
Group 3: Random as Player 1			
Random vs MCTS-100 vs MCTS-50	MCTS-100	67.3 ± 2.9	38.8
	MCTS-50	31.8 ± 2.9	33.7
Random vs MCTS-50 vs MCTS-100	MCTS-100	65.0 ± 3.0	37.8
	MCTS-50	34.0 ± 2.9	34.1
Overall Performance			
Average	MCTS-100	65.0 ± 3.0	37.8
	MCTS-50	34.0 ± 2.9	34.1

Table 7.1 Position impact analysis with N=1000 games per configuration, Random's efficiency is not reported here due its low results, where MCTS-100(MCTS-50) means MCTS agent with 100(50 resp.) iterations per turn

We can state two important things based on these results:

- Our MCTS agents totally outperformed Random agent
- The seating order does not effect the results and because of that we will not test all possible seating order combinations in all the following experiments for simplicity.

Setup 1: MCTS vs. Rule-Based Agents

Objective

The objective of this setup is to compare the performance of the developed MCTS agent against two instances of the Rule-Based agent in a 3-player game setting and to analyze influence of key parameters such as depth, C_{MCTS} for UCB formula and given iterations per turn.

Methodology

- **Player Configuration:** The three players were configured as:

  - Player 1: MCTS Agent
  - Player 2: Rule-Based Agent
  - Player 3: Rule-Based Agent

- **Variable Factors:** To ensure results are not specific to a single game setup, variability was introduced through:
  - **Wonder Assignment:** A set of 21 distinct combinations of 3 Wonders was used. These combinations were selected to ensure each base game Wonder was included in multiple matchups against different opposing Wonders. Each combination assigns one Wonder to each of the three players.
  - **Card Dealing Sequence:** For each of the 21 Wonder combinations, 50 different games were simulated. Each simulation used a unique random seed for shuffling the Age decks and dealing the initial hands, ensuring diverse card distributions and drafting scenarios.

- **Number of Games:** A total of 21 (Wonder Sets) $\times$ 50 (Card Seeds) = 1050 games were simulated for this experiment.

- **Reproducing Experiments:** To run identical experiments:
  - Use the following command with the specified parameters:

```
dotnet run --ai=mcts:{iterations},{time},{depth};
            rulebased;
            rulebased
```

  - To modify the C_{MCTS} in MCTS, adjust the `kUCT` parameter in: `Core\Data\GameData.cs` (provided in Core7W folder A.5)

UCB Exploration Constant

We evaluated different UCB constants ranging from 0.1 to 0.9. MCTS with 250 iterations was chosen. Results showed minimal variation in performance:

C_{MCTS}	Win Rate (%)	Avg. Score
0.1	82.4 ± 2.3	46.1
0.2	80.4 ± 2.4	46.2
0.3	81.5 ± 2.3	46.1
0.4	83.4 ± 2.3	46.1
0.5	80.4 ± 2.4	46.0
0.6	78.7 ± 2.5	45.8
0.7	80.4 ± 2.4	45.9
0.8	80.5 ± 2.4	45.8
0.9	80.4 ± 2.4	46.0

Table 7.2 MCTS Performance based on C_{MCTS}(mean value ± 95% confidence interval)

Also, we implemented an age-dependent exploration parameter adjustment:

$$C_{\text{MCTS}}(\text{Age}) = \begin{cases} C_0 & \text{for Age I} \\ \frac{C_0}{2} & \text{for Age II} \\ \frac{C_0}{4} & \text{for Age III} \end{cases}$$

where C_0 should be relatively large due to the subsequent division. But this adjustment shows almost no statistically no difference and because of that we chose for all subsequent experiments $\mathbf{C_{MCTS} = 0.4}$ as a parameter configuration.

Simulation Depth Analysis

We tested limiting simulation depth for MCTS agent with 250 iterations per turn:

Simulation Depth	Win Rate (%)	Avg. VP
4	45.0 ± 3.0	39.2
8	68.9 ± 2.8	43.5
12	78.7 ± 2.5	45.5
16	82.2 ± 2.3	45.9

Table 7.3 Impact of number of given iterations(mean value ± 95% confidence interval)

Complete simulations to the terminal state provided significantly better performance than partial simulations. This aligns with *7 Wonders'* property where final scoring combinations make early game state evaluation unreliable. All subsequent experiments use **full-depth** simulations.

MCTS Iterations Analysis

MCTS Iterations Analysis shows how increasing number of iterations influences the results and, as we can see, more iterations we give to the agent - more games it wins:

Iterations	Win Rate (%)	Avg. Score
100	71.6 ± 2.7	43.6
250	81.9 ± 2.3	46.2
500	88.4 ± 1.9	47.8

Table 7.4 Impact of number of given iterations(mean value ± 95% confidence interval)

Performance Analysis

The experimental results reveal several key insights about the MCTS agent's performance characteristics:

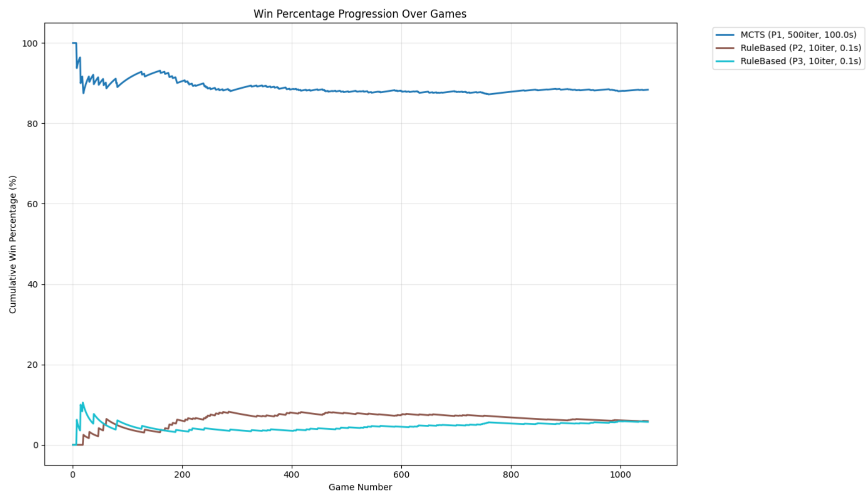

Figure 7.1 Cumulative win percentage progression over 1050 games, showing MCTS agent (500 iterations) versus two Rule-Based agents.

As shown in Figure 7.1, the MCTS agent demonstrates consistent performance improvement over time. This learning curve suggests that:

- Two RbAI lose much more often, then win, against MCTS and play, in general, with the same performance
- After 300 games, the win rate stabilizes around 88%, indicating convergence

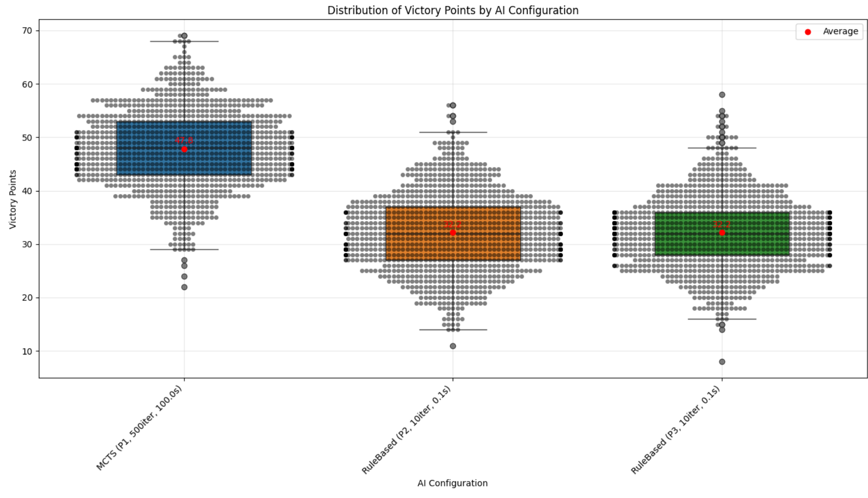

Figure 7.2 Distribution of victory points across different AI configurations. The box plots show median (line), interquartile range (box), and full range (whiskers) of scores.

Figure 7.2 presents the score distribution analysis, revealing that:

- The MCTS agent achieves higher median scores (47.8 VP) compared to Rule-Based agents (38.2 VP)
- The number of points for MSTS has never dropped below 20 and exceeded 60 several times, while none of the RB agents has ever scored more than 60 points and finished the game with less than 20 points several times.

Wonder-Specific Performance

The agent's performance varies significantly depending on the assigned Wonder:

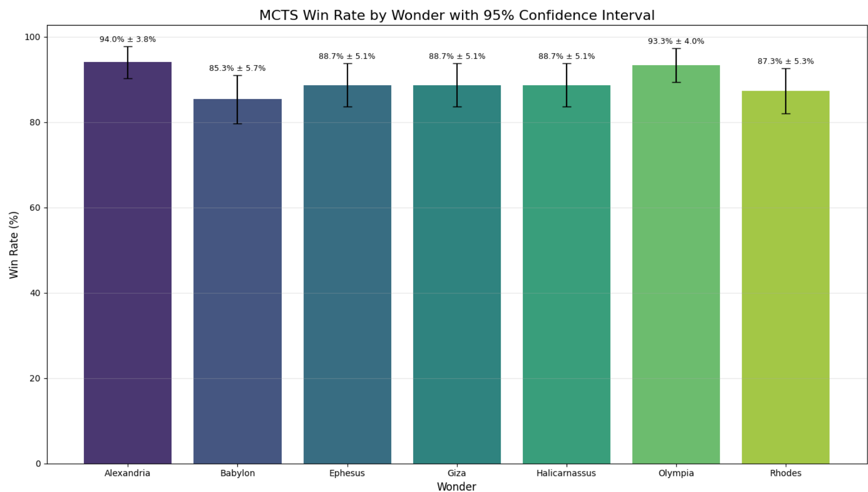

Figure 7.3 MCTS win rates by Wonder type with 95% confidence intervals. Results based on 150 games per Wonder configuration.

As demonstrated in Figure 7.3, the agent performs exceptionally well with Olympia (93.3% ± 4.0%) and Alexandria (94.0% ± 3.8%), while showing relatively weaker (though still dominant) performance with Babylon (85.3% ± 5.7%).

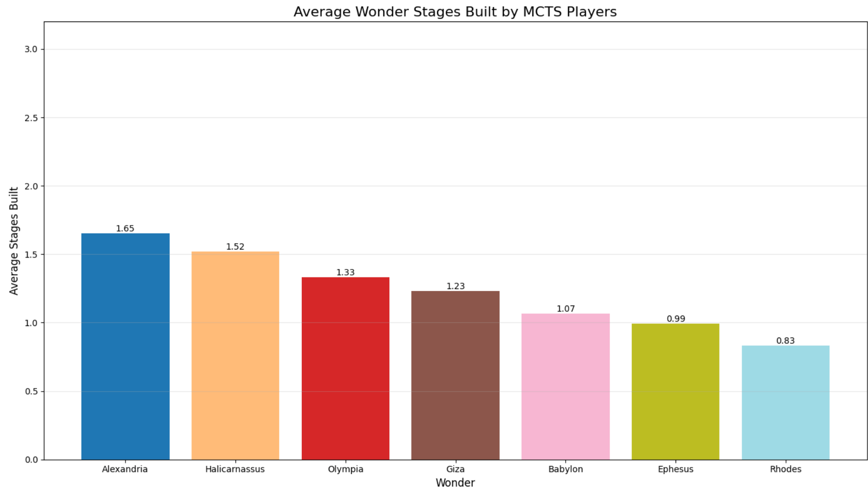

Figure 7.4 Average number of Wonder stages completed per game by Wonder type.

Figure 7.4 complements this analysis by showing construction patterns. Notably:

- Alexandria Wonder has the highest completion rate (1.65 stages/game)
- Rhodes has the lowest completion (0.83 stages/game)
- This correlates with win rates, suggesting Wonder completion is a key success factor

Strategic Card Selection

The frequency analysis of winning card combinations reveals the agent's strategic preferences:

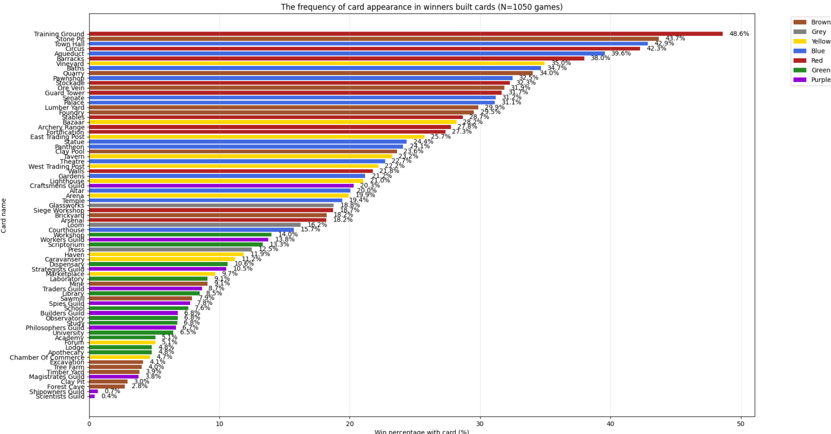

Figure 7.5 Frequency of card appearances in winning builds (N=1050 games).

Figure 7.5 highlights several strategic tendencies:

- Strong preference for scientific and military cards
- Guilds rarely turn out to be built by the winners
- The cards that are more numerous in the deck are more often found lined up by the players
- Cards that produce mixed resources appear less frequently than cards with single resources

Setup 2: MCTS vs. MCTS vs. RbAI

This setup aims to evaluate the MCTS agent's performance when playing against another MCTS agent with different parameters and a RbAI agent. We choose to not test it under close game data with specific seed because we want to see how our MCTS agents will succeed in random conditions, as it would be in a real game.

Methodology

- **Player Configuration:** P1=MCTS, P2=MCTS, P3=RbAI
- **Number of Games:** A total of 1000 random games were simulated per setup for this experiment.
- **Reproducing Experiments:** To run identical experiments: Use the following command with the specified parameters:

```
dotnet run --ai=mcts:{iterations},{time},{depth};
          mcts:{iterations},{time},{depth};
          rulebased
```

P1 vs **P2**	MCTS-100	MCTS-250	MCTS-500	MCTS-1000
MCTS-100	-	28.6% ± 2.8	23.8% ± 2.6	22.0% ± 2.6
MCTS-250	58.8% ± 3.1	-	35.8% ± 3.0	35.4% ± 3.0
MCTS-500	67.7% ± 2.9	55.5% ± 3.1	-	40.8% ± 3.0
MCTS-1000	71.2% ± 2.8	59.1% ± 3.0	53.2% ± 3.1	-

Table 7.5 Win rate comparison between MCTS agents, no RbAI is mentioned due its weakness (N=1000 games per matchup and mean value ± 95% confidence interval)

From results in Table 7.5 we can notice, if MCTS plays against an agent with a relatively close number of iterations, then the game becomes more equal.

Conclusion

This thesis presented an exploration of Monte Carlo Tree Search applied to the complex card-drafting game *7 Wonders*. Below we summarize the key contributions from each chapter and outline promising directions for future research.

In Introduction, we discussed the development of AI at the moment and presented the main problems for creating AI for the game *7 Wonders*, revealed why artificial intelligence based on MCTS can be a good solution for the chosen board game.

In Game Overview and Game Analysis, we figured out the basic rules of the game, the necessary elements to understand what our AI will be based on, and also took a deeper look at the problems of the game and calculated the game complexity.

In Related Work, we reviewed the existing theoretical materials for the implementation of MCTS in games, as specifically for *7W*, and also gave examples of the technical implementation of some similar projects.

In AI Agents, we described the 3 types of agents that we have developed and in particular discussed in more detail about MCTS, the problems that the algorithm faces in this particular game, as well as some improvements.

In User and Programmer Guides, we explained how to use the presented solutions such as Unity and CLI, as well as how to develop and add a new agent to an existing implementation.

And in Experiments, we conducted numerous experiments to understand how the developed agent behaves in various conditions, with all kinds of parameters and against different opponents, and also analyzed the results.

Future Work

This thesis demonstrates the effectiveness of MCTS for *7 Wonders*, but several promising extensions could further enhance the AI's capabilities. One key direction is the integration of neural networks and machine learning techniques to augment the MCTS framework. A neural network could serve as a learned evaluation function, replacing the default random rollouts with more informed simulations. This hybrid approach—inspired by AlphaGo's[3] success—would train on game states and outcomes to predict win probabilities, reducing the computational burden of deep rollouts while improving decision quality in mid-to-late game phases. The network could be trained via supervised learning on human gameplay data or through self-play reinforcement learning, with its effectiveness benchmarked against the current rollout strategy in terms of win rate and computational efficiency.

Another avenue is exploring partial observability handling through alternative MCTS paradigms. The current implementation uses determinization (PIMC) to sample possible game states, but this could be extended or replaced with multi-observer MCTS (MO-MCTS[13]), where each observer maintains a separate belief state about hidden information (e.g., opponents' hands). Comparing this to the existing single-observer approach would reveal whether explicit belief tracking improves performance in *7 Wonders'* drafting mechanics, particularly in early-

game scenarios where hidden information dominates. Metrics would include win rate against rule-based agents and our MCTS implementation, robustness to opponent strategies, and memory overhead.

Bibliography

1. Bauza, Antoine. *7 Wonders Rulebook* [https://tesera.ru/images/items/8722/7-Wonders-Rulebook-EN.pdf]. Repos Production, 2010. Accessed: 2025-05-05.
2. Chang, Henry. *7 Wonders Quick Reference: Card List & Numbers v0.5* [https://boardgamegeek.com/filepage/92676/7-wonders-quick-reference-card-list-and-numbers]. 2013. Fan-made reference sheet uploaded as user hinghenry. Accessed: 2025-05-06.
3. Silver, David; Huang, Aja; Maddison, Chris J, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016, vol. 529, no. 7587, pp. 484–489.
4. Ontañón, Santiago; Synnaeve, Gabriel; Uriarte, Alberto, et al. A survey of real-time strategy game AI research and competition in StarCraft. In: *IEEE Transactions on Computational Intelligence and AI in Games*. 2013, vol. 5, pp. 293–311. No. 4.
5. Cowling, Peter I; Powley, Edward J; Whitehouse, Daniel. An investigation of MCTS and its enhancements for combat scenarios in Magic: The Gathering. In: *IEEE CIG*. 2012, pp. 1–8.
6. Browne, Cameron B; Powley, Edward; Whitehouse, Daniel, et al. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, vol. 4, no. 1, pp. 1–43.
7. Keller, Thomas; Eyerich, Patrick. MCTS based on nested entropy search. In: *International Conference on Automated Planning and Scheduling*. AAAI, 2012, pp. 114–121.
8. Baier, Hendrik; Winands, Mark H. M. Simulation-based approaches for partially observable Markov decision processes. *IEEE Transactions on Computational Intelligence and AI in Games*. 2017, vol. 9, no. 4, pp. 389–403.
9. Pereira, Lucas et al. An implementation of the 7 Wonders board game for AI-based players. *SBGames*. 2020.
10. Robilliard, Denis; Fonlupt, Cyril; Teytaud, Fabien. Monte-Carlo Tree Search for the Game of *7 Wonders*. In: *European Conference on Artificial Intelligence (ECAI)*. Prague, Czech Republic: Springer, 2014, pp. 64–77. Available from doi: 10.1007/978-3-319-14923-3_5.
11. Long, Jeffrey; Sturtevant, Nathan; Buro, Michael; Furtak, Timothy. Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2010, vol. 24, pp. 134–140. Available from doi: 10.1609/aaai.v24i1.7562.
12. Weng, Lilian. The Multi-Armed Bandit Problem and Its Solutions. *lilianweng.github.io*. 2018. Available also from: https://lilianweng.github.io/posts/2018-01-23-multi-armed-bandit/.

13. LANCTOT, Marc; WAUGH, Kevin; ZINKEVICH, Martin; BOWLING, Michael. Monte Carlo sampling for regret minimization in extensive games. *Advances in neural information processing systems*. 2009, vol. 22.

List of Figures

1.1 Describes Wonder board elements 8
1.2 Describes card elements 9

4.1 Four phases of Monte Carlo Tree Search (Selection, Expansion, Simulation, Backpropagation). *Source:* Robert Moss, CC BY-SA 4.0, via Wikimedia Commons `https://commons.wikimedia.org/wiki/File:MCTS-steps.svg` 16
4.2 Cards transfering during the first 3 turns. 18
4.3 The exploration-exploitation dilemma: Choosing between machines with known (blue) and unknown (red) reward probabilities [12]. . 19

5.1 Main menu interface with play and quit options 23
5.2 Game configuration window with settings options 24
5.3 Detailed wonder information panel 24
5.4 Main game interface during play session 25
5.5 Opponent view interface 27
5.6 Not enough resources example 28
5.7 War conflict resolution interface 28
5.8 Free construction example 29
5.9 Halicarnassus special ability interface 29
5.10 Babylon special ability interface 30
5.11 Final results screen 30
5.12 Global rankings screen 31

7.1 Cumulative win percentage progression over 1050 games, showing MCTS agent (500 iterations) versus two Rule-Based agents. . . . 40
7.2 Distribution of victory points across different AI configurations. The box plots show median (line), interquartile range (box), and full range (whiskers) of scores. 40
7.3 MCTS win rates by Wonder type with 95% confidence intervals. Results based on 150 games per Wonder configuration. 41
7.4 Average number of Wonder stages completed per game by Wonder type. 41
7.5 Frequency of card appearances in winning builds (N=1050 games). 42

List of Tables

1.1 Summary of Wonders and Their Stage Bonuses 9

5.1 Agent Configuration Parameters 22
5.2 Game Settings Components 24
5.3 Game Interface Components 26
5.4 Opponent Interface Components 27

7.1 Position impact analysis with N=1000 games per configuration, Random's efficiency is not reported here due its low results, where MCTS-100(MCTS-50) means MCTS agent with 100(50 resp.) iterations per turn 37
7.2 MCTS Performance based on C_{MCTS}(mean value ± 95% confidence interval) 38
7.3 Impact of number of given iterations(mean value ± 95% confidence interval) 39
7.4 Impact of number of given iterations(mean value ± 95% confidence interval) 39
7.5 Win rate comparison between MCTS agents, no RbAI is mentioned due its weakness (N=1000 games per matchup and mean value ± 95% confidence interval) 43

List of Abbreviations

MCTS - Monte-Carlo Tree Search
7W - 7 Wonders
VP - Victory Points
AI - Artificial Intelligence
C# - C Sharp
RbAI - Rule-Based Artificial Intelligence
UCB - Upper Confidence Bound
PIMC - Perfect Information Monte Carlo
MAB - Multi-Armed Bandit
CLI - Command Line Interface

A Attachments

A.1 Technical Documentation

Technical documentation:

`\Attachments\Technical Documentation.txt`

A.2 Python Script for Analysis

Script for a result analysis could be found as:

`\Attachments\analyzingThesisScript.py`

A.3 Unity .exe file

Unity build file is available as:

`\Attachments\7Wonders\build\7Wonders.exe`

A.4 GameSimulationCLI folder

Contains the main simulation configuration and is available as:

`\Attachments\GameSimulationCLI`

A.5 Core7W Library

Core game logic and AI implementations and is available as:

`\Attachments\Core7W`