FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ákos Vermes

Artificial Intelligence for the Board Game Sagrada

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis: Adam Dingle, M.Sc.
Study programme: Computer Science (B0613A140006)

Prague 2024

I want to express my heartfelt appreciation to Adam Dingle for his incredible support and patience throughout this journey. His guidance was extremely helpful, especially during the times when the process took longer than expected. The tools he introduced for software and AI development were crucial to the success of this project and I have learned a lot from him.

Title: Artificial Intelligence for the Board Game Sagrada

Author: Ákos Vermes

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Adam Dingle, M.Sc., Department of Software and Computer Science Education (KSVI)

Abstract: The application of artificial intelligence to board games has become widespread in recent years inspired by achievements in chess and Go. This thesis presents a detailed exploration of artificial intelligence techniques in an implementation of the board game Sagrada. Sagrada presents challenges for artificial intelligence agents due to the complexity of the game, the high branching factor and the presence of hidden and stochastic information in the game. The primary focus of the thesis is on implementing and comparing the performance of minimax, Monte Carlo Tree Search (MCTS), and rules-based algorithms.

Keywords: artificial intelligence, board game, heuristic

Název práce: Umělá inteligence pro deskovou hru Sagrada

Autor: Ákos Vermes

Katedra: Katedra softwaru a výuky informatiky (KSVI)

Vedoucí bakalářské práce: Adam Dingle, M.Sc., Katedra softwaru a výuky informatiky (KSVI)

Abstrakt: Aplikace umělé inteligence na deskové hry se v posledních letech rozšířila, což bylo inspirováno úspěchy ve šachu a Go. Tato práce představuje podrobný průzkum technik umělé inteligence v implementaci deskové hry Sagrada. Sagrada představuje pro hráče řízené umělou inteligencí výzvy kvůli složitosti hry, vysokému větvení a přítomnosti skrytých a stochastických informací ve hře. Hlavním zaměřením práce je implementace a srovnání výkonu algoritmů minimax, Monte Carlo Tree Search (MCTS) a algoritmů založených na pravidlech.

Klíčová slova: umělá inteligence, desková hra, heuristika

Contents

Introduction **7**

1 Game Description **8**
- 1.1 Tool Cards . . . 8
- 1.2 Public Objective Cards . . . 10
- 1.3 Window Pattern Cards . . . 11

2 Game Analysis **13**
- 2.1 Branching Factor . . . 13
  - 2.1.1 Smarter players . . . 14

3 Project Structure **15**
- 3.1 OS support . . . 15
- 3.2 Language and libraries . . . 15
- 3.3 Architecture . . . 16
  - 3.3.1 Model . . . 16
  - 3.3.2 View . . . 18
  - 3.3.3 Controller . . . 19
- 3.4 Tournament Framework . . . 19

4 Building Blocks for AI Agents **22**
- 4.1 Board Evaluation . . . 22
  - 4.1.1 Board Evaluation in Sagrada . . . 22
- 4.2 Heuristic Filter . . . 24
- 4.3 Heuristic Sort . . . 25

5 Minimax Agent **27**
- 5.1 The Algorithm . . . 27
  - 5.1.1 Alpha-Beta Pruning . . . 28
  - 5.1.2 Handling Imperfect Information . . . 28
  - 5.1.3 Final Move Selection . . . 29
- 5.2 Heuristic Function . . . 30
- 5.3 Branching Factor . . . 32

6 Rules-Based Agent **34**
- 6.1 Only Die To Field Move Strategy . . . 34
- 6.2 All Moves Strategy . . . 35
- 6.3 Branching Factors . . . 35

7 Monte Carlo Tree Search Agent **37**
- 7.1 Selection . . . 38
  - 7.1.1 Upper Confidence Bound . . . 38
- 7.2 Expansion . . . 39
- 7.3 Simulation . . . 39
- 7.4 Backpropagation . . . 40
- 7.5 Handling Imperfect Information . . . 41

7.6 Information Set Monte Carlo 43

8 Tournament Results 44

8.1 MCTS 44

8.1.1 Playout Strategies 44

8.1.2 Determinizing World Count and Exploration Constant . . 45

8.1.3 Varying Configuration Comparison 49

8.2 Minimax 50

8.2.1 Hyperparameters 50

8.2.2 Search Depth and Determinizing World Count 51

8.2.3 Varying Configuration Comparison 52

8.3 Final Results 54

8.4 Deterministic vs Non-Deterministic Game Version Results 55

Conclusion 57

Future Work 58

Bibliography 59

List of Figures 60

List of Tables 61

List of Abbreviations 62

A Attachments 63

A.1 First Attachment 63

Introduction

In recent years, the application of artificial intelligence (AI) to traditional board games has reached unprecedented heights, showcasing remarkable advancements in strategic gameplay. Games such as Go, chess, and poker, each with their unique challenges, have served as testing grounds for AI algorithms. Go, known for its large branching factor and complex strategies, has been revolutionized by AI, notably with AlphaGo's victories [1] against world champions.

Sagrada is a dice-drafting board game [2] designed by Adrian Adamescu and Daryl Andrews and published in 2017. Each player constructs a stained-glass window using dice on a personal rectangular 4×5 board with restrictions on the types of dice that can be played on each space. Players gain points by completing public and secret objectives for dice placements, and the one with the most points after ten rounds is the winner. In case of a draw in total points achieved, the tie is broken by other criteria implying that there is always a winner in every game.

Unlike Go and chess, which are deterministic games with no hidden information, Sagrada includes both randomness and hidden information, adding layers of complexity to strategic decision-making. Additionally, Sagrada has a high branching factor, making its gameplay significantly complex. The motivation behind this study lies in solving the challenges of imperfect information and a high branching factor in developing a strong AI player for Sagrada.

Prior to the composition of this bachelor's thesis, a digital adaptation of the board game Sagrada had already been released by Dire Wolf Digital [3]. There is no public documentation available about the AI players of their implementation.

The primary aim of this study is to evaluate and compare the performance of different AI players in my implementation of the digital adaptation of Sagrada. My goal was to implement and analyze various AI strategies, including ones such as minimax, Monte Carlo Tree Search (MCTS), and rules-based agents. This research aims to examine the effectiveness of each approach by experimenting with different configurations of the agents.

Through statistical analysis and gameplay simulations, this thesis seeks to provide insights into the strengths and weaknesses of AI players, offering valuable implications for the development of intelligent gaming agents.

This thesis will concentrate on the two-player version of the Sagrada board game, although it's designed for 1 to 4 players.

1 Game Description

In this strategic puzzle game, players compete to create stunning stained glass windows using colorful dice and complete different patterns. Each player is tasked with carefully selecting and placing dice to fulfill specific pattern requirements while navigating constraints and maximizing their score.

For a comprehensive understanding of the game rules and mechanics, the reader should refer to the official rule book provided by the publisher. The rule book by FloodGate Games [4] serves as the authoritative source for setup instructions, gameplay rules and scoring criteria.

While the official rule book offers a comprehensive overview of Sagrada's gameplay mechanics, it does not delve into detailed descriptions of specific components or strategies. In the following sections, I aim to fill in the gaps left by the rule book, offering detailed descriptions and strategic insights into various aspects of Sagrada's gameplay.

Throughout the text, I am using the word "field" as a synonym for the official term "space".

1.1 Tool Cards

Tool Cards provide unique actions that players can take throughout the game to manipulate dice placements, modify their boards, or gain additional benefits. One important attribute of a Tool Card is whether it shifts turns or not. The ones that don't shift the turn allow the players to make either a passing move or a die-placing move after using the Tool Card. The following table illustrates the 12 different Tool Cards present in the game marking the ones that shift turns with a ✓and the ones that do not shift turns with a ✗:

Name (Shifts turns)	**Description**
1. Copper Foil Furnisher(✗)	Allows a player to move a die that is already placed on the board to another position ignoring the shade restriction of the destination space. The player must obey all other placement restrictions.
2. Eglomise Brush(✗)	Allows a player to move a die that is already placed on the board to another position ignoring the color restriction of the destination space. The player must obey all other placement restrictions.

3. Cork-backed Straightedge(✓)	Allows a player to place a die on the board to a space that is not adjacent to any other dice.
4. Flux Brush(✓)	Allows a player to re-roll a die from the Draft Pool. The die must be placed on the board if there is any space where the die could be placed with the re-rolled value. Otherwise, the die is returned to the Draft Pool.
5. Flux Remover(✓)	Allows a player to choose a die from the Draft Pool, swap it with a random die from the Dice Bag and choose the new die's value. The die must be placed on the board if there is any space where the die could be placed with the re-rolled value. Otherwise, the die is returned to the Draft Pool.
6. Glazing Hammer(✗)	Allows a player to re-roll all dice in the Draft Pool. This Tool Card may only be used on the second turn of a player before drafting.
7. Grinding Stone(✓)	Allows a player to flip a die from the Draft Pool to its opposite side (6 flips to 1, 5 flips to 2, 4 flips to 3). The player must place the flipped die on the board. The die can be flipped only if it is placeable on the board after the flipping.
8. Grozing Pliers(✓)	Allows a player to increase or decrease the value of a die from the Draft Pool by 1. 1 cannot be decreased to 6 and 6 cannot be increased to 1. The player must place the flipped die on the board. The die can be flipped only if it is placeable on the board after the flipping.
9. Lathekin(✗)	Allows the player to move exactly two dice that are already placed on the board to other positions obeying all placement restrictions.
10. Lens Cutter(✓)	Allows a player to swap a die from the Draft Pool with one from the Round Track. After swapping the dice, the new die has to be placed on the board.

11. Running Pliers(✗)	Allows a player to immediately draft a die after their first turn. This means that using this Tool Card allows the player to move twice in a row.
12. Tap Wheel(✗)	Allows a player to move one or two dice of the same color that are already placed on the board to a new position. The color of the dice must match the color of a die on the Round Track. The player must obey all the placement restrictions.

In the text below, the term "relocating tool cards" refers to Tool Cards that allow a player to move one or more dice that are already placed on the board to a new position. Namely, these are **the Copper Foil Furnisher**, **the Eglomise Brush**, **the Tap Wheel** and **the Lathekin** Tool Cards.

1.2 Public Objective Cards

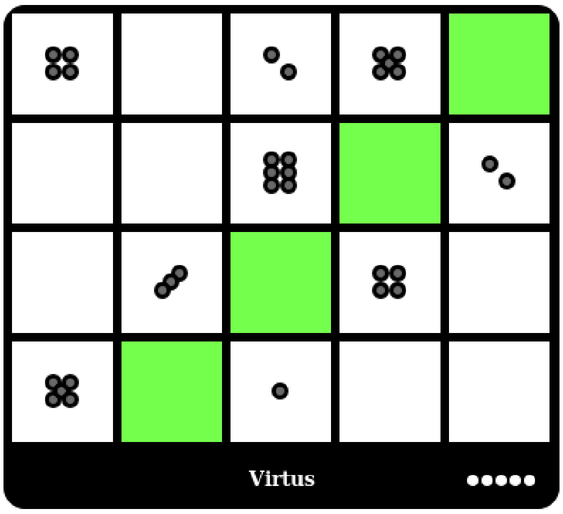
Public Objective Cards (PuOC) represent shared goals that all players strive to achieve throughout the game. These cards provide additional challenges and opportunities for players to earn points based on completing specific patterns. For each completed pattern, the player receives the satisfaction value that belongs to the Public Objective Card. There are 10 different Public Objective Cards:

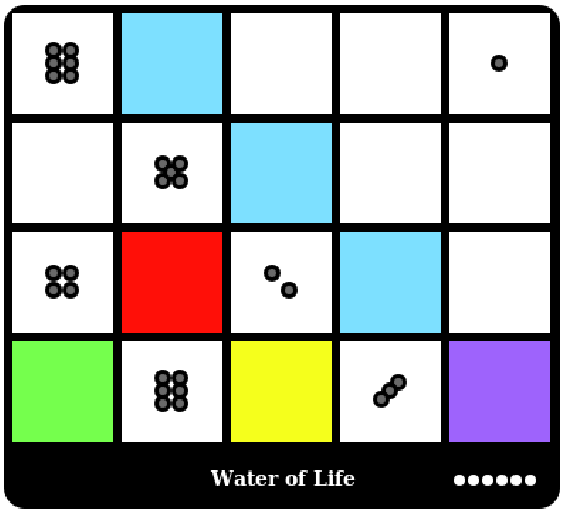
Name	**Description**
1. Color Diagonal	Awards one point for each die that has a diagonal neighbor of the same color, regardless of the specific colors involved.
2. Column Color Variety	The pattern is fulfilled when a given column contains dice whose colors are all different, ensuring that all spaces within the column are occupied by a die. Completing a pattern is worth 5 points.
3. Column Shade Variety	The pattern is fulfilled when a given column contains dice whose shades are all different, ensuring that all spaces within the column are occupied by a die. Completing a pattern is worth 4 points.

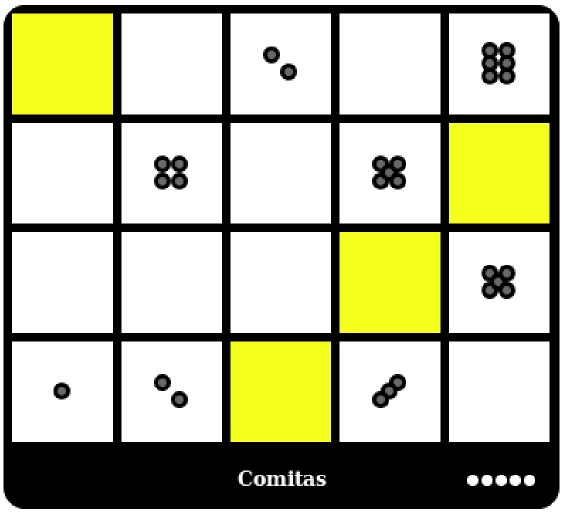
4. Ligh Shades	Awards two points for every pair of dice with a value of 1 and a value of 2. The position of the dice does not matter.
5. Medium Shades	Awards two points for every pair of dice with a value of 3 and a value of 4. The position of the dice does not matter.
6. Deep Shades	Awards two points for every pair of dice with a value of 5 and a value of 6. The position of the dice does not matter.
7. Row Color Variety	The pattern is fulfilled when a given row contains dice of different colors, ensuring that all spaces within the row are occupied by a die. Completing a pattern is worth 6 points.
8. Row Shade Variety	The pattern is fulfilled when a given row contains dice of different shades, ensuring that all spaces within the row are occupied by a die. Completing a pattern is worth 5 points.
9. Color Variety	Awards four points for every set of 1 die of each color. The position of the dice does not matter.
10. Shade Variety	Awards four points for every set of 1 die of each shade. The position of the dice does not matter.

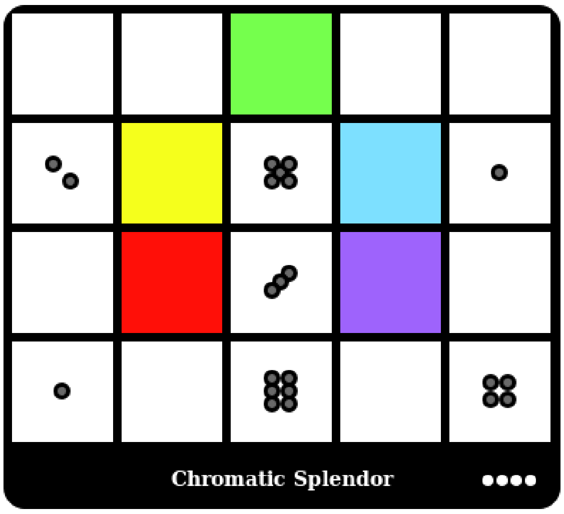
1.3 Window Pattern Cards

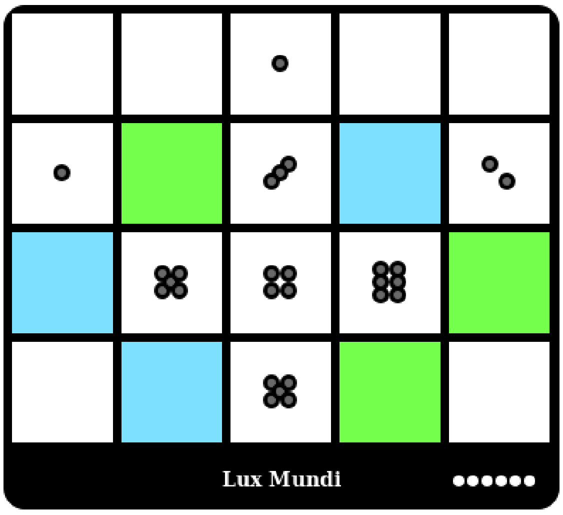
After drafting all the Tool Cards, Public Objective Cards and Private Objective Cards, the game continues with every player choosing a Window Pattern Card. These cards serve as the basic layout of each player's boards. Every space contains a shade restriction, a color restriction or no restriction at all. If a space contains a restriction, only dice with matching shades/colors can be placed on it. To every Window Pattern Card belongs a difficulty number which is given to the player in the form of Favor Tokens. The following 8 images illustrate examples of the 24 Window Pattern Cards in the game - the difficulty number is represented by the white circles in the lower right corner of the Window Pattern Cards:

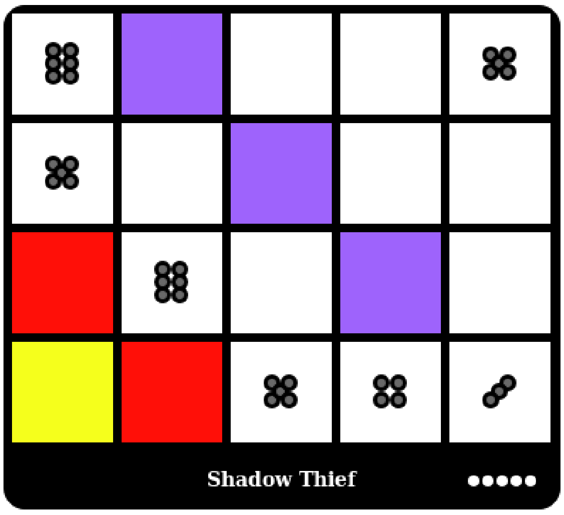

Virtus
●●●●●


Water of Life
●●●●●●


Comitas
●●●●●


Chromatic Splendor
●●●●


Lux Mundi
●●●●●●


Shadow Thief
●●●●●

2 Game Analysis

In this chapter, I will describe Sagrada from a game-theoretic perspective.

Asymmetric In asymmetric games, players may have different strategies and payoffs. Sagrada is asymmetric because each player has a different window pattern card.

Finite Game with Known Horizon A game that ends after a predetermined number of moves is considered to have a finite horizon. In the two-player variant of the game, the game ends after a fixed number of 20 moves by every player taking two turns in each of the 10 rounds.

Hidden Information The hidden information in the game is present in the form of Private Objective Cards.

Stochastic Information The stochastic information in the game is represented by future dice rolls.

Sequential Players make decisions one after another, allowing for a reaction to the preceding move.

2.1 Branching Factor

The **branching factor** is defined as the number of possible actions available to the agent at each decision point within a search tree. It is important because it greatly influences the efficiency of various search-based algorithms. A higher branching factor exponentially increases the number of possible paths within the search tree.

In Sagrada, the branching factor varies throughout the game, influenced by factors such as the stage of the game and the presence of different types of tool cards. Some tool cards can be used in only one way, so the presence of such a card increases the branching factor by one. In contrast, other tool cards introduce variability in branching factor, as their effects may differ based on the current state of the game or player choices. For example, relocating tool cards can dramatically increase the branching factor by introducing a huge amount of possibilities.

I will now introduce two basic AI agents: the Random agent, which chooses moves randomly, and the First agent, selecting the first possible move at each decision point. The first possible move of the player is determined by the order described in the `Game::possible_moves()` in Section 3.3.1. The following figure illustrates the average branching factor over 1000 games of the Random agent against the First agent across the rounds of the game.

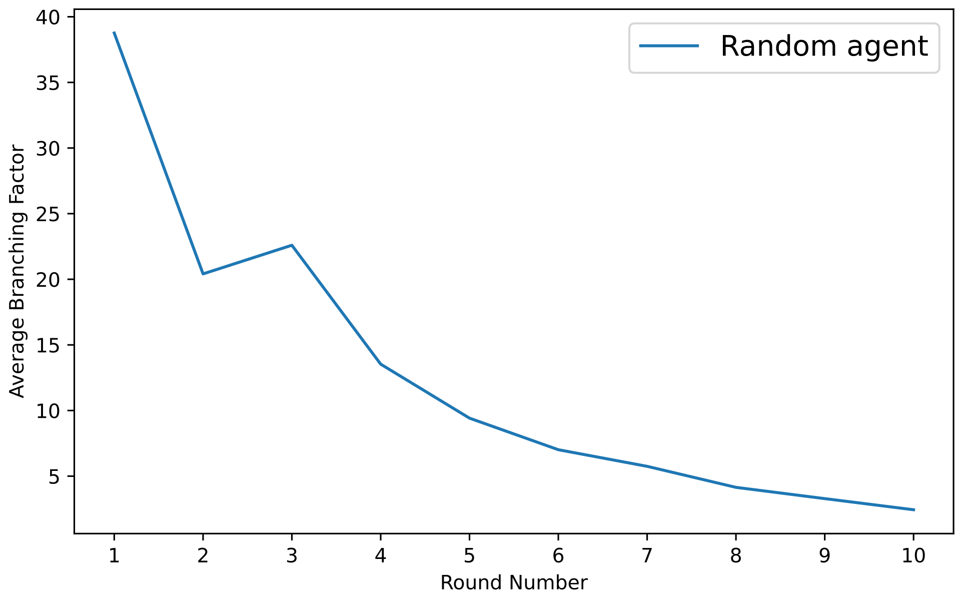


Figure 2.1 Branching factor across the rounds for the Random agent

2.1.1 Smarter players

While the Random agent provides a baseline understanding of the branching factor in the game, we will see that more strategic or skilled players will exhibit different branching factors. Smarter players tend to explore deeper and more selectively within the game tree, leading to a narrower but more focused set of potential moves.

3 Project Structure

I have created a custom implementation of Sagrada that allows a user to play interactively against 1-3 computer opponents using a graphical interface. Additionally, it can simulate games where multiple computer players (2-4) compete against each other, either interactively or through a command-line tournament. My implementation does not include the single player variant of the game in which a player doesn't have any opponents but tries to maximize the score with slightly different rules of the game.

3.1 OS support

I developed the game on Ubuntu, and it could potentially be ported to other operating systems such as Windows and other Linux distributions with relative ease, although I haven't attempted this. All the libraries used in this project are cross-platform, enhancing the potential for easy adaptation.

3.2 Language and libraries

For this project, the choice of the programming language primarily rested on the need for optimal performance and efficiency. I selected C++ as the programming language due to its reputation for producing highly performant code.

The following libraries are used in the project:

1. gtkmm - gtkmm [5] is the official C++ interface for the popular GUI library GTK. GTK is widely used for creating graphical user interfaces in applications that run on Linux, Windows, and macOS. I chose this library because it is easy to use, written in C++ and is cross-platform compatible.
2. nlohmann_json - an open-source library by Niels Lohmann [6] for parsing JSON objects. It is mainly used for storing game configurations and for unit testing
3. argparse - an open-source library used for defining the command-line interface [7]
4. GoogleTest - an open-source library used for unit testing [8]

3.3 Architecture

The architecture of the system follows the Model-View-Controller (MVC) design pattern, because of its clear separation of concerns and scalability. In the MVC architecture, the model represents the application's data and game logic, the view handles the presentation layer, and the Controller manages user inputs and coordinates the communication between the model and the view components.

This separation of concerns is important for many reasons. For instance, our system utilizes the model for different purposes, such as running games and simulations using the GUI and running tournaments between AI players using the CLI.

3.3.1 Model

The model component is the backbone of the application, responsible for managing data and game logic. Concretely in Sagrada, this means:

1. **Data management** - The model component in Sagrada holds the data about the game including the static data initialized once at the beginning of the game (the different cards, dice, etc.), the current state of the game that is changing (round number, the player whose turn it is to move, etc.) and other data that is needed to perform the operations it defines (the history of the game).

2. **Game logic** - The game logic in Sagrada is represented by the actions that users of the Model component perform during the gameplay. These actions include collecting the possible moves, evaluating the sent moves for correctness or evaluating the current state of the game.

The main class in the model component is the `Game` class. It has the following structure:

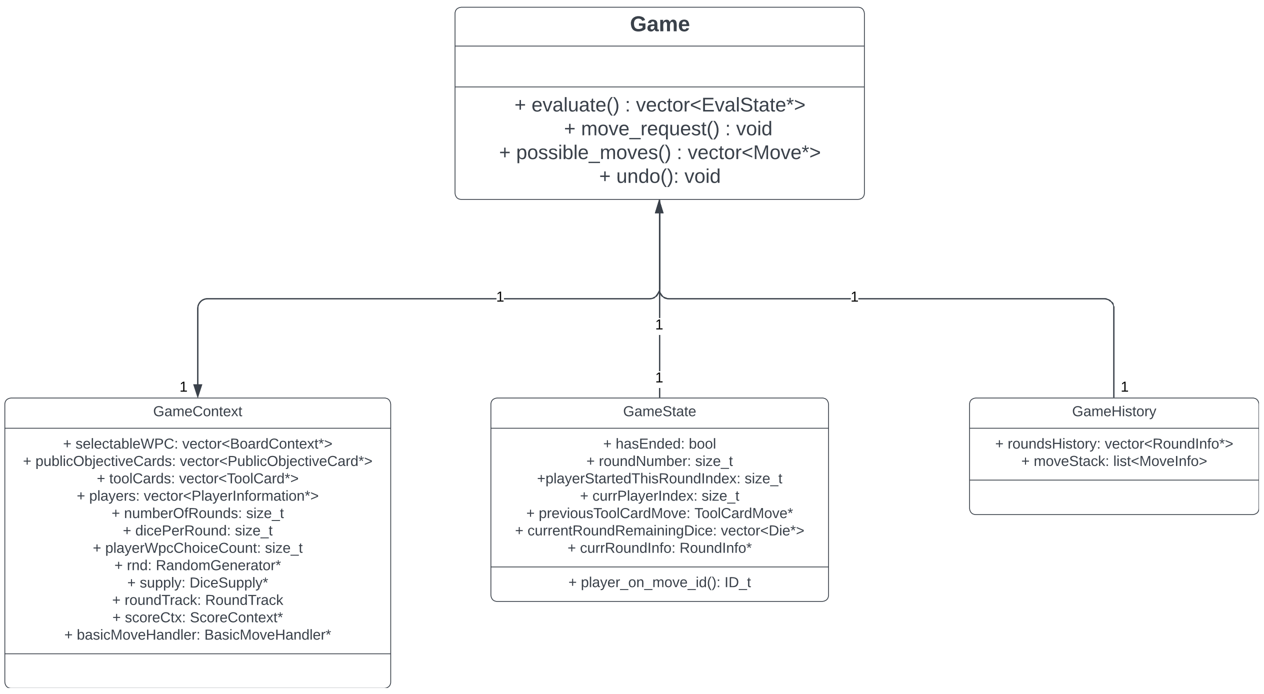

Game
+ evaluate() : vector<EvalState*>
+ move_request() : void
+ possible_moves() : vector<Move*>
+ undo(): void
1
1
1
1
1
1
GameContext
+ selectableWPC: vector<BoardContext*>
+ publicObjectiveCards: vector<PublicObjectiveCard*>
+ toolCards: vector<ToolCard*>
+ players: vector<PlayerInformation*>
+ numberOfRounds: size_t
+ dicePerRound: size_t
+ playerWpcChoiceCount: size_t
+ rnd: RandomGenerator*
+ supply: DiceSupply*
+ roundTrack: RoundTrack
+ scoreCtx: ScoreContext*
+ basicMoveHandler: BasicMoveHandler*
GameState
+ hasEnded: bool
+ roundNumber: size_t
+playerStartedThisRoundIndex: size_t
+ currPlayerIndex: size_t
+ previousToolCardMove: ToolCardMove*
+ currentRoundRemainingDice: vector<Die*>
+ currRoundInfo: RoundInfo*
+ player_on_move_id(): ID_t
GameHistory
+ roundsHistory: vector<RoundInfo*>
+ moveStack: list<MoveInfo>

Figure 3.1 Model structure UML diagram

Data representing the game state is separated into several classes:

`GameContext` This class represents the static data in the `Game` class. Its components are configured once and never changed afterward. The separation makes the API of the `Game` class smaller because creating these objects is the responsibility of the `GameContextBuilder` class. Also, it makes the `Game` class easily configurable by creating a `GameContext` instance and passing it to the constructor.

`GameState` Represents the dynamically changing data about the current state of the `Game` class. This class is independent to make the API of the `Game` class smaller and to make the system more scalable.

`GameHistory` Defining the Undo operation requires remembering the actions that lead to a given state of the class to be able to reproduce past states. This class holds the data required for fulfilling this functionality.

As illustrated in Figure 3.1 , the `Game` class has the following most important methods:

`vector<Move*> possible_moves()` Returns a vector of all possible moves that are available for the current player at the current state. These moves include Tool Card Moves, die-placing moves, and a Pass move, and appear in that order.

`void move_request(move_t move)` Receives a move as an argument and checks whether it is a correct move or not.

`void undo()` Performs an undo operation on the last move. This function is mainly used by AI agents such as the minimax agent to produce better performance than cloning the `Game` object every time a move is about to be evaluated.

`vector<EvalState*> evaluate()` Evaluates the current game state for each player and returns a vector of `EvalState` objects. This function is called at the end of the game for final evaluation and by AI agents.

Moves

In this chapter and the following chapters, I use the abbreviation **DTFM** for moves that place a die to the board.

The different moves in the game are represented by classes derived from the `Move` class. Many Tool Cards define their moves separately derived from ToolCardMove. There are two Tool Cards that consist of multiple submoves. These cards involve randomness: after the first submove a new random value is revealed, and the second submove places the new die onto a field if possible. Concretely, these are the mentioned two Tool Cards:

1. **Flux Remover** - The first submove chooses a die from the Draft Pool to be swapped and swaps it with a die from the Bag. The second submove chooses a value for the die from the Bag and places it onto a field if possible.
2. **Flux Brush** - The first submove chooses a die from the Draft Pool and re-rolls it. The second submove places the die onto a field.

3.3.2 View

In this section, we're exploring the view component. The GUI is structured into pages. Each page in the GUI is equipped with widgets to fulfill specific functions. The two most important pages and their base class are the following:

- `GamePlayingPage` - This is a base class for the two classes listed below since these two pages share the same functionalities except for some user interactions. It displays all the elements of the game such as the boards of the players or the current round's dice.
- `LocalGamePlayingPage` - This page is used when the user chooses to play against AI agents. This page has some specific functionalities other than its base class `GamePlayingPage`. These functionalities include selecting a die and placing it onto the player's board or using the Tool Cards.
- `SimulationGamePlayingPage` - This page is used for displaying simulations between AI agents. It is useful when during an experiment a game has strange results and we would like to check the steps each AI agent made leading to the odd result.

3.3.3 Controller

The controller component serves as an intermediary between the model and the view components. It primarily handles user input, processes requests, and provides appropriate responses. Concretely in Sagrada, the controllers manage the AI players and forward the human players' move requests in games with human players.

The `ControllerWithAIPlayer` class serves as a base class for the two concrete controllers. It stores a reference to the game that is being played and stores the AI players paired with the corresponding `PlayerInformation` object from the `Game` object.

In games with human players, an instance of the `LocalPlayerController` class is used. This class is derived from the `ControllerWithAIPlayer` class making it able to play games with both human and AI Players. Additionally, it stores the IDs of all human players providing easier handling for the view components.

Games that are played between only AI players use an instance of the `OnlyAIPlayerController` class. This class is used mainly in the Tournament framework.

3.4 Tournament Framework

The Tournament framework is independent from the GUI, which makes experimenting fast and easy. This way it is possible to automate running the different tournaments and processing their results by scripts. The `build/tournament` executable runs a tournament. It accepts the following arguments:

`-h` Displays help information.

`-v` Prints the results to the standard output game-by-game. This option is turned off by default.

`-s` The starting seed of the tournament i.e. the seed of game 1. The default value is 779.

`-n` The number of games in the tournament. There is no default value for this option and it is required to specify one.

`-d` Makes the games in the tournament deterministic i.e. all the information is available for all the agents including dice in the upcoming rounds and other players' Private Objective Cards. This option is turned off by default.

`-p` The agents that will play against each other in the tournament. This option accepts 2-4 parameters and has no default configuration but it is required to be defined.

`random` - An agent choosing a random move

`first` - An agent choosing the first possible move

`rules-based` - An agent with rules-based strategy.
Example config: `rules-based-strategy=only_dtfm`

`strategy` Choose a strategy for the player. The two possible options are `only_dtfm` and `all_moves`.

`minimax` - Minimax agent.
Example config: `minimax-depth=3,worlds=8,config_file=config.json`

`depth` Sets the depth for the minimax search.

`worlds` Sets the determinizing world count.

`config_file` Sets a config file with heuristic constants.

`mcts` - Monte Carlo Tree Search agent.
Example config: `mcts-it=120,expl=0.7,worlds=6,playout=random`

`it` Sets the iteration count for the algorithm.

`worlds` Sets the determinizing world count.

`expl` Sets the exploration constant for the algorithm.

`playout` Sets the playout strategy for the simulation step of the algorithm.

`-mode` The mode of the game. Allows the users to run tournaments with a custom configuration.

`-b` Runs all the permutations of players in a game using a given seed. This option is turned off by default.

`-csv` An option to save the results of the game into a directory of CSV files. The parameter is the name of the directory. This option is turned off by default.

Here is an example of how to run a tournament of 100 games using 800 as the starting seed between a minimax agent and a rules-based agent:

```
$ build/tournament -n 100 -p rules-based-strategy=only_dtfm \
minimax-depth=4,worlds=4 -s 800 -v
```

When the tournament ends, the framework prints statistics about the game and the players either to the standard output or to the specified CSV files. The statistics include the number of games won by each player, the average time it took for each player to make a move and a confidence interval for the win rate of the tournament-winning player.

A random seed is a number used to initialize a pseudorandom number generator. This means that changing this random seed for a game results in another game configuration and that way another gameplay. On the other hand, this also means that using the same seed multiple times produces the same output. The tournament framework uses the random seed received from the command-line and increases it by one after every game. This means that if the first game used the random seed 779, the second one will use 780 and so on.

There is an option to simulate games played between AI agents using the GUI. This comes in handy if some games produce an interesting outcome in the tournament and the user would like to check the choices of the AI agents one by one. The CLI is designed to make this task as user-friendly as possible. To simulate a game that was run in the tournament using the following command:

```
$ build/tournament -n 100 -p rules-based-strategy=only_dtfm \
minimax-depth=4,worlds=4 -s 800 -v
```

run the following command:

```
$ build/sagrada simulation -p rules-based-strategy=only_dtfm \
minimax-depth=4,worlds=4 -s 800 -v
```

4 Building Blocks for AI Agents

In this chapter and the following chapters, I refer to some fields as uncompletable. These are the fields where no die can be placed because of the neighboring dice and/or the field restrictions.

4.1 Board Evaluation

When implementing an AI agent for games, especially strategy-based ones like Sagrada, the ability to evaluate the game board accurately is paramount. This evaluation serves as the foundation upon which the agent makes its decisions, determining the quality of its moves.

4.1.1 Board Evaluation in Sagrada

Evaluating players' boards is one of the functionalities of the model component. The `EvalState` class is responsible for evaluating the state of a given player including a board evaluation and the points for the unused favor tokens. Each `EvalState` object contains the following components:

- The number of empty fields on the player's board
- The number of Private Objective Card points that the player has earned so far
- The number of unused favor tokens that the player holds
- A vector of PuocPatternState objects each representing the state of a concrete Public Objective Card
- The final score of the player that represents the number of points that the player would have if the game had ended before the evaluation

PuocPatternState

`PuocPatternState` objects represent the state of a board from the perspective of a given Public Objective Card. These objects are constructed by the `PublicObjectiveCard` classes during the board evaluation. They contain both rules-based information (currently earned points) and heuristic-based information (the heuristic state).

The heuristic-based pattern evaluation is divided into completable and uncompletable patterns. The uncompletable patterns are simply counted but for completable patterns, the number of dice that are missing to complete the given pattern is remembered as well. This makes the evaluation of given patterns stronger because agents can reward placing dice that bring patterns closer to completion.

The performance of an evaluation function is critical for agents using algorithms such as minimax, particularly in complex games such as Sagrada, where the branching factor is high. There is often a trade-off between the accuracy and efficiency of an evaluation function. One of the inaccuracies occurs when evaluating Public Objective Cards that contain either a row or a column matching for unique dice colors or shades. The reason for this behavior is that each row/column is evaluated independently. However, the neighboring rows/columns have an impact on each other.

I will demonstrate this phenomenon with an example. Suppose that one of the Public Objective Cards is the Row Shade Variety. This means that the player is trying to place dice so that each row will contain dice with unique shades. The following figure shows an example state:

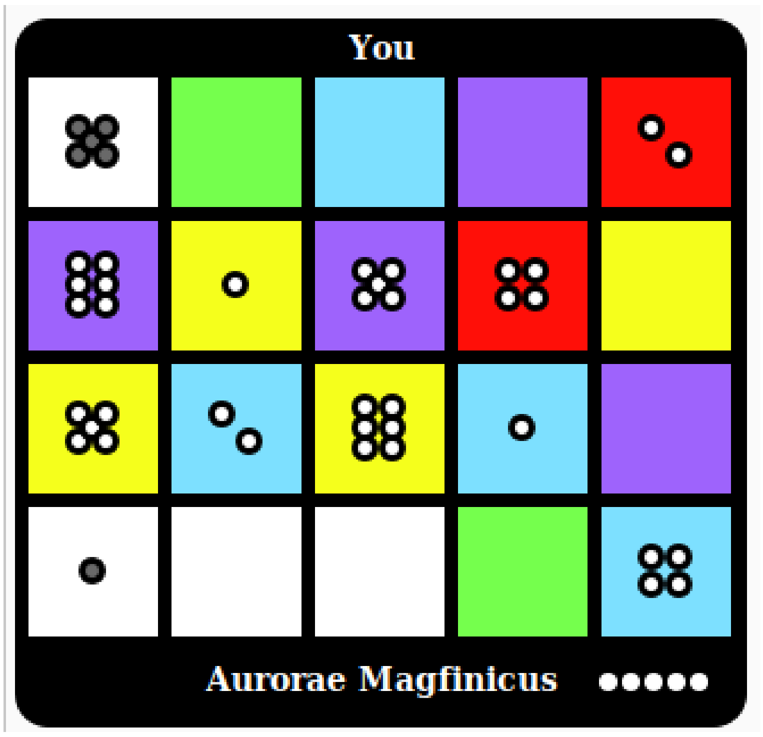


Figure 4.1 Incorrect evaluation board position example

Notice that the second row could be completed with a 2 or a 3, and the third row could be completed with a 3 or a 4. Since the first row contains a red die with the value of 2 at the rightmost column, a die having a value of 2 cannot be placed on the neighboring fields so the only value left for completing the second row is 3. The same principle applies to the third row and the blue 4 die in the fourth row. This means that both rows can be completed only with a 3 which is not possible at the same time because of the rules of the game. However, the board evaluation processes the rows independently so it will report that both of these rows can be finished at the same time. The reason for not implementing it perfectly is connected to performance. This phenomenon occurs rarely and avoiding it would require looking for a concrete placement of dice onto each empty field. I decided that implementing this feature would not be worth the performance overhead.

4.2 Heuristic Filter

Heuristic filters play a crucial role in AI algorithms, particularly in scenarios where exhaustive search or perfect evaluation is impractical due to computational constraints. Heuristic filters help narrow down the search space by quickly discarding unpromising choices, allowing the AI algorithm to focus its computational resources on more promising alternatives. This technique is called "forward pruning" in the popular AI textbook [9] .

In Sagrada, one common pitfall is inadvertently limiting future placement options by positioning dice in a way that conflicts with the game's color or shade restrictions. Recognizing this, a heuristic filter can be employed to identify and avoid such "bad moves", both in regular die-to-field placements and when using Tool Cards. When considering potential moves for placing a die on the board, the heuristic filter evaluates each option to determine if it would place a die adjacent to a field that shares the same color or shade restriction as the placed die. If such a placement is found, it's flagged as undesirable and filtered out. By proactively avoiding these conflicting placements, the AI ensures that it maintains flexibility and maximizes its future placement opportunities.

The heuristic filter is implemented in the `HeuristicFilter.h` header file that is part of the AI code of the project. The main function is `filter_moves`, which receives all the moves to be filtered and separates them into the `bestMoves` and the `backupMoves` containers using the filtering policy defined above. The "backup moves" are used when there are no "good moves" to work with.

The following figure illustrates the impact of the heuristic filter presented by the two AI agents that use it:

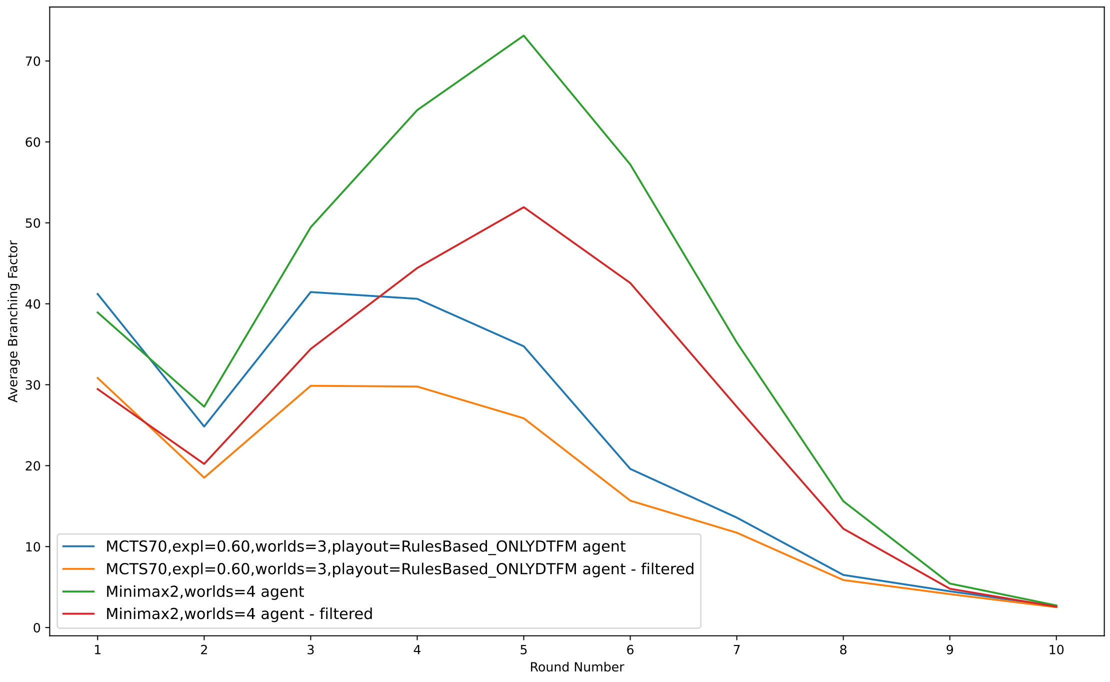


Figure 4.2 The impact of the heuristic filter on the branching factor of two different AI players

4.3 Heuristic Sort

A component that sorts possible moves heuristically is useful for AI agents, since it allows them to consider the most promising moves first in any situation. A heuristic sort will be called many times, so it should be efficient.

In the previous section, I have already talked about filtering the least promising moves. This section provides a deeper understanding of how the remaining moves are sorted. Since the function is called a lot of times, it cannot use any heuristic-based sorting technique that would be connected to the Public Objective Card patterns because it would make the sorting process slow. There is one component in the scoring structure that is easy to use as the basis of the sorting algorithm, namely the Private Objective Card's color.

The sorting algorithm decides between two moves according to the following criteria:

1. If neither of the moves is a die-placing move, a random move is prioritized.
2. If one of the moves is a die-placing move and the other one is not, the die-placing move is prioritized.

3. If one of the dice matches the color of the Private Objective Card and the other does not, the die matching the Private Objective Card color is prioritized.

4. If one move places a die to a field that has a restriction and the other one does not, the one placed to a field with a restriction is prioritized.

5. Finally, the die with the higher value is prioritized.

The heuristic sort is implemented by the comparator functor `MoveHeuristicCMP` that is part of the AI code. It is designed to be used with the `std::sort` algorithm from the standard library as the comparator function.

5 Minimax Agent

In the world of artificial intelligence and game strategies, the minimax algorithm is a key method for making decisions when competing against others. The main idea of minimax is to predict and counter the opponent's moves while trying to maximize its own advantage. At its core, the minimax algorithm operates on the principle of recursively exploring the game tree, considering all possible moves by both players up to a certain depth or until a terminal state is reached. Each node in the tree represents a game state, and the algorithm assigns a value to each node based on the potential outcome of the game from that state. The key idea driving minimax is the assumption that the opponent will make moves that are optimal for them, aiming to minimize the agent's potential gain.

While minimax is powerful, it is also a heuristic-based approach. It doesn't guarantee finding the optimal solution but rather approximates it by searching through the game tree. Despite its effectiveness, minimax does have limitations, primarily related to the vast branching factor of certain game trees, which can lead to exponential growth in computation as the depth increases.

While minimax excels in deterministic games like chess, its application becomes more complex in games with imperfect information, where players have limited or incomplete knowledge about the game state or their opponents' actions. In non-deterministic games, traditional minimax strategies struggle due to the inability to accurately assess the state space. Unlike deterministic games where the entire game state is known, non-deterministic games introduce uncertainty, making it difficult for the agent to construct a comprehensive game tree.

In this chapter, I delve into the implementation of minimax specifically adapted for Sagrada. I discuss how I approached challenges such as high branching factors and imperfect information within the context of Sagrada's gameplay mechanics. By detailing my approach to integrating minimax into the game's decision-making process while addressing these factors, I aim to provide valuable insights into the development of AI agents for complex board games.

5.1 The Algorithm

At its core, the minimax algorithm operates on the premise of minimizing the possible loss for a worst-case scenario while simultaneously maximizing potential gains. It's particularly effective in two-player, zero-sum games, where each player aims to outmaneuver the other.

The game's possible future states are represented as a game tree, with each node denoting a possible move by a player. In Sagrada, a game with a high branching factor, this tree can rapidly expand, making exhaustive search impractical. Each terminal node or leaf of the game tree is assigned a heuristic value, reflecting the desirability of that game state for the player. Starting from the terminal nodes, these values are propagated upwards through the tree. Generally, at each level, the algorithm alternates between maximizing and minimizing the values, simulating the moves of both players. For a comprehensive overview of the minimax algorithm, see [9].

This works for games where the players make moves one after the other. Note that in Sagrada choosing the minimizing and the maximizing player for the next layer is not that straightforward because the players do not strictly alternate moves. Also, the implementation of some Tool Cards consists of multiple submoves as described in Section 3.3.1 making it harder to determine whose turn it is to move in any given game state.

To explore the state nodes of the game tree, the minimax agent uses the preprocessed possible moves using the heuristic filter already described in Section 4.2 . This is one of the ways I am dealing with the high branching factor.

Thanks to the implementation of the undo operation of the `Game` class, trying out moves becomes fast since instead of cloning the `Game` object for each move, the agent simply makes a move and then, at the right time, undoes it.

5.1.1 Alpha-Beta Pruning

While the minimax algorithm is powerful, its effectiveness can be hampered by the sheer size of the game tree, especially in games such as Sagrada with a high branching factor. Alpha-beta pruning is a technique used to reduce the number of nodes evaluated by the minimax algorithm. It works by cutting off branches of the game tree that are guaranteed to be unfruitful. In essence, it prunes away portions of the search tree that need not be explored further, significantly reducing the computational overhead. A detailed explanation of alpha-beta pruning is available in section 5.3 of the book [9].

In my implementation, after the moves are filtered according to the description in the previous section, the remaining moves are sorted using the heuristic sort comparator described in Section 4.3 . When the moves are selected in the sorted order, they are processed one by one. Processing the most promising moves first makes the alpha-beta pruning especially effective.

5.1.2 Handling Imperfect Information

A minimax agent for Sagrada has to deal with handling stochastic information, particularly regarding the dice available in future rounds. One option for handling this situation easily would be simply not allowing the minimax player to explore moves beyond the current round. This technique is easy to implement but I chose to experiment with a more complex solution.

Another way of handling imperfect information is to create different worlds in which all the information is deterministic and visible for all the players. This approach creates different worlds by choosing a concrete random value for every hidden and non-deterministic piece of information in the game. When the worlds are created, the algorithm is run in each of them. After the algorithm has finished in every world, the sub-results are combined to make a final move. This technique is described in the AI textbook [9]. The authors call it “averaging over clairvoyance” because it assumes that all the players will have access to every piece of information in the game.

Selecting a fixed number of deterministic worlds to create is non-trivial. For this reason, the number of worlds created is given by a parameter to the minimax player making it possible to experiment with different world counts.

5.1.3 Final Move Selection

The final move that is chosen by the minimax agent is the one that has shown the best behavior across all the determinized worlds. To achieve this, For every possible move, the agent computes its heuristic value in every world. Then, the heuristic values of the same moves are combined by adding them together. After combining the sub-results, the minimax agent chooses the move with the highest combined heuristic values.

Algorithm 1 Minimax algorithm

```
function MAKE-NEXT-MOVE( )
    $combinedWorldResults \leftarrow []$
    for $i \leftarrow 1$ to $worldCount$ do
        $world \leftarrow game.$ CLONE-WITH-RANDOM-FUTURE( )
        $worldResults \leftarrow []$
        MINIMAX-ALGORITHM($world$, $depth$, $\infty$, $-\infty$, TRUE, $worldResults$)
        update $combinedWorldResults$ with $worldResults$
    end for
    return the move with the highest combined heuristic value
end function

function MINIMAX-ALGORITHM($game$, $depth$, $\alpha$, $\beta$, $maxPlayer$, $moveCont$)
    if $depth = 0$ or $game.has\_ended$ then
        return NULL, HEURISTIC($game$)
    end if

    $allMoves \leftarrow game$.POSSIBLE-MOVES( )
    $bestMoves \leftarrow$ FILTER-MOVES($game$, $allMoves$)
    SORT($bestMoves$, MoveHeuristicCMP)
    $currBestHeuristics \leftarrow$ if $maxPlayer$: $-\infty$, else: $\infty$
    for $move$ in $bestMoves$ do
        $game$.MOVE-REQUEST(move)
        $nextLayerPlayer \leftarrow$ GET-NEXT-LAYER-PLAYER($game$)
        $heuristicVal \leftarrow$ MINIMAX-ALGORITHM($game$, $depth - 1$, $\alpha$, $\beta$, $nextLayerPlayer$, NULL)
        if $moveCont$ not NULL then
            $moveCont$.ADD($move$, $heuristicVal$)
        end if
        $game$.UNDO-LAST-MOVE( )
        update $bestMove$,$currBestHeuristics$ if $heuristicVal$ is the new best
        update $\alpha$ and $\beta$ if needed
        if $\beta \le \alpha$ then
            break
        end if
    end for
    return $bestMove$, $currBestPlayerVal$
end function
```

5.2 Heuristic Function

In the context of the minimax algorithm, the heuristic function plays a pivotal role in guiding the agent's decision-making process within the search tree. The quality of the heuristic function directly impacts the efficiency and effectiveness of the minimax algorithm.

The `EvalState` object described in Section 4.1.1 serves as a basis to calculate the heuristic value of a player's state. `HEURISTIC_MIN` and `HEURISTIC_MAX` are the smallest and the greatest values defined for the heuristic value of a game state. My heuristic function in Sagrada is divided into two parts: evaluating games that have already ended and all the other game states. I will begin by introducing the simpler case, evaluating a game that has already ended. In this scenario, the heuristic function returns `HEURISTIC_MIN` if the agent lost, `HEURISTIC_MAX` if the agent won.

Computing the heuristic value of a given state when the game has not ended yet is more sophisticated. The `HeuristicConstants` structure holds constants used in the heuristic function defining weights for different components. The values of these weighted constants were derived experimentally. This experiment is described in Section 8.2.1.

The heuristic value of a state is computed from 4 components: rewards for the already achieved points, rewards for the completable Public Objective Card patterns, penalties for uncompletable Public Objective Card patterns and finally penalties for fields where the player won't be able to place a die without using a relocating Tool Card. Motivating the agent to continue completing patterns closer to completion is important. For this reason, calculating the reward for completable Public Objective Card patterns is designed to follow this idea prioritizing the ones with higher satisfaction values. The other 3 components are simply multiplied by the weight gained from the `HeuristicConstants` object. The following pseudocode illustrates the evaluation function:

Algorithm 2 Minimax heuristic

```
function GET-HEURISTIC-VALUE-FOR-STATE($evalState$)
    $puocUncompletablePoints \leftarrow 0$
    $weightedPuocCompletable \leftarrow 0$
    for puocState in evalState.puocStates do
        $sv \leftarrow puocState.satisfactionValue$
        $puocUncompletablePoints \leftarrow$ puocUncompletablePoints +
            $puocState.uncompletablePatternCount * sv$
        for pattern in puocState.completablePatterns do
            $diceCompleted \leftarrow$ number of dice completed in $pattern$
            $weightedDiceToComplete \leftarrow diceCompleted^{puocCompletableWeight}$
            $weightedPuocCompletable \leftarrow$
                $weightedPuocCompletable + sv * weightedDiceToComplete$
        end for
    end for
    $weightedCompletedPoints \leftarrow$
        $evalState.completedPoints * completedPointsWeight$
    $weightedUncompletablePoints \leftarrow$
        $puocUncompletablePoints * puocUncompletablePointsWeight$
    $weightedUncompletableFields \leftarrow$
        $evalState.uncompletableFieldCount * uncompletableFieldCountWeight$
    return $weightedCompletedPoints + weightedPuocCompletable - weightedUncompletablePoints - weightedUncompletableFields$
end function
```

The weight constants `completedPointsWeight`, `puocUncompletablePointsWeight` `uncompletableFieldCountWeight` and `puocCompletableWeight` are the constants from the `HeuristicConstants` structure.

5.3 Branching Factor

As discussed in Section 2.1.1, smarter players exhibit markedly different branching factors compared to random agents. In the case of minimax agents, the branching factor tends to be significantly higher because the heuristic function guides the player towards more promising moves. This results in choosing moves that increase the branching factor in the following turns because it rarely chooses moves that produce uncompletable fields and fields that are completable with only a small amount of different dice. The following figure visualizes the average branching factor over 1000 games between a minimax and a random agent:

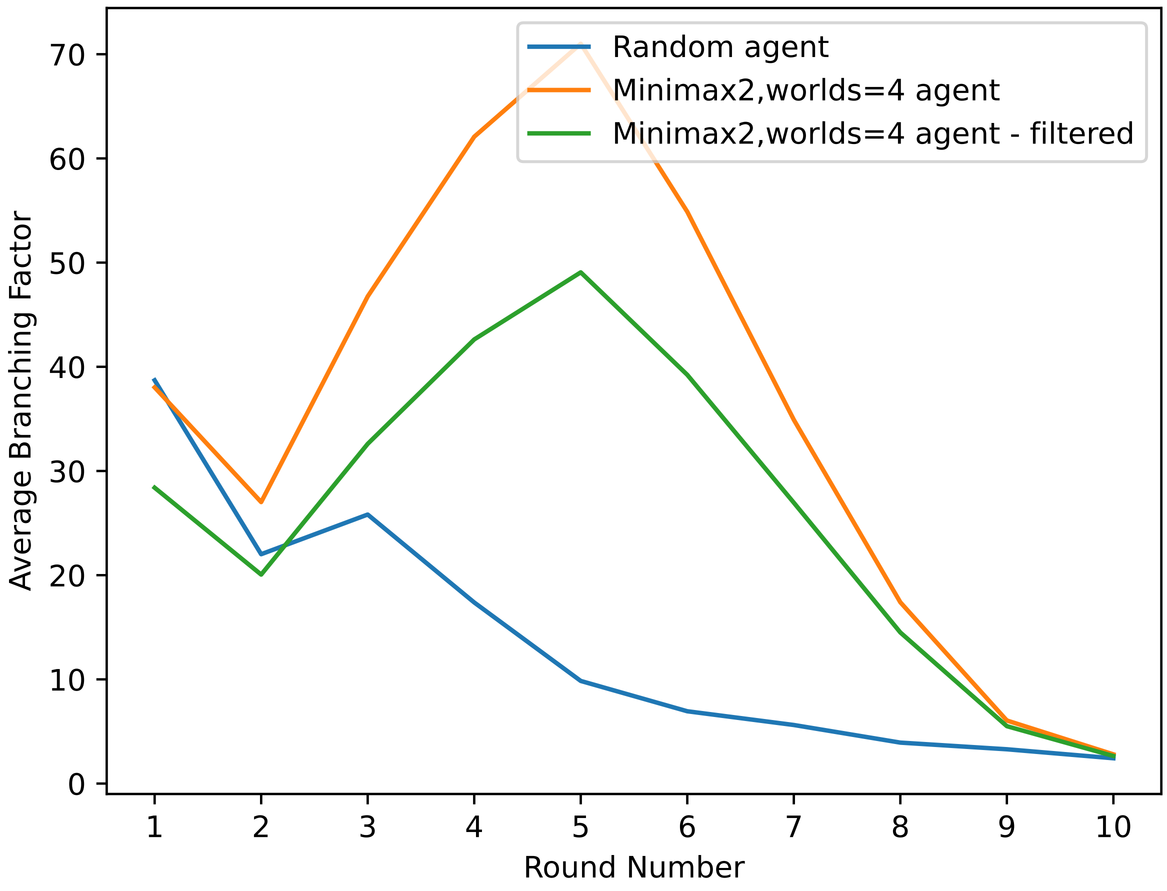


Figure 5.1 Branching factor comparison between the minimax agent and the random agent

6 Rules-Based Agent

Rules-based AI agents in board games operate on predefined sets of rules and strategies programmed by developers. These agents analyze the game state, evaluate available moves, and select the best course of action based on a predetermined set of rules.

My rules-based agents assess the potential moves within the current state but refrain from constructing a comprehensive tree of possible game states by exploring multiple move sequences across various depths. This implies that the rules-based agents don't have to deal with imperfect information since all the information is available at the current state and they don't look beyond the current move and they evaluate only their boards as described in the following sections.

6.1 Only Die To Field Move Strategy

As the name suggests, this strategy of the rules-based agent makes only moves that place a die onto a field i.e. this agent doesn't use any Tool Cards. This agent is intended to perform well because not using any Tool Card moves decreases the average branching factor. Agents using this strategy not only don't have to consider using Tool Card moves but they save time by not even generating the Tool Card moves.

The rules-based agents don't use the heuristic filter and the heuristic sort functionalities. The algorithm begins with saving information about the current state. After this, all the moves are tried out and the new states are compared with the initial one. The moves are evaluated using the following criteria:

1. **Minimize the count of newly appearing fields that cannot be filled without using a relocating tool card** - Increasing the count of these fields potentially leads to increasing uncompletable Public Objective Card patterns and empty fields at the end of the game. For this reason, if the algorithm encounters a move that increases this number compared to the current minimum, the move is immediately skipped.

2. **Public Objective Card state** - The agent evaluates the state of the game from the perspective of the Public Objective Cards with a heuristic function that is very similar to the one described in Section 5.2.

3. **Color of the die** - If the color of the die matches the color of the agent's Private Objective Card, the state value returned by step 2 is increased by the weighted value of the die.

The algorithm updates the current best move if the processed move has a higher Public Objective Card state value or the Public Objective Card state value is slightly worse than the current best but the total points achieved are higher than the current best plus a threshold.

6.2 All Moves Strategy

The algorithm begins with separating all the possible moves into three categories: relocating tool card moves, tool card moves that contain randomness and all the other moves including die-placing moves and other tool card moves. First, the relocating moves are processed. The main criterion followed is to choose the move that makes the number of uncompletable fields as small as possible. If there is at least one such move, it will be used. Second, using the algorithm described in the previous section, the agent evaluates the die-placing moves and all the tool card moves that contain placing a die onto a field as a submove. If a die-placing move selected by the Only Die To Field Move Strategy has a positive Public Objective Card state value, it will be chosen as the main move. Otherwise, if there are any moves containing randomness, a random one of these moves will be chosen. Finally, if none of the previous actions led to a final move, the very first possible move will be chosen.

6.3 Branching Factors

To demonstrate the difference between the branching factor of the two strategies that highly influences the average time it takes to make a move, I ran 1000 games between two agents using the two different strategies. The following plot visualizes the average branching factor over 1000 games:

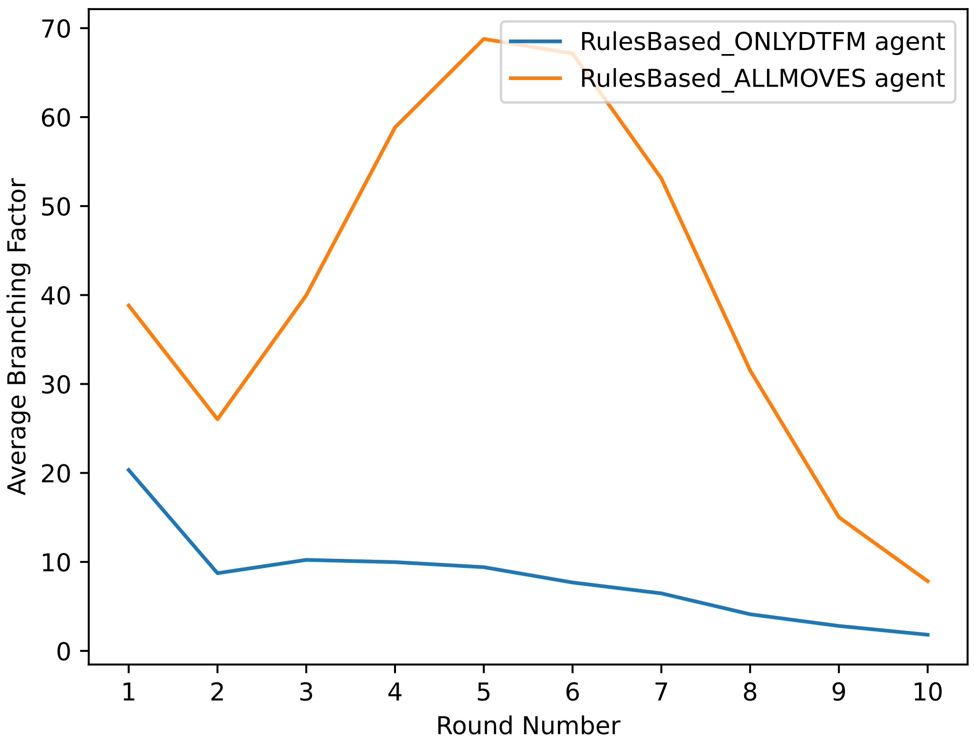


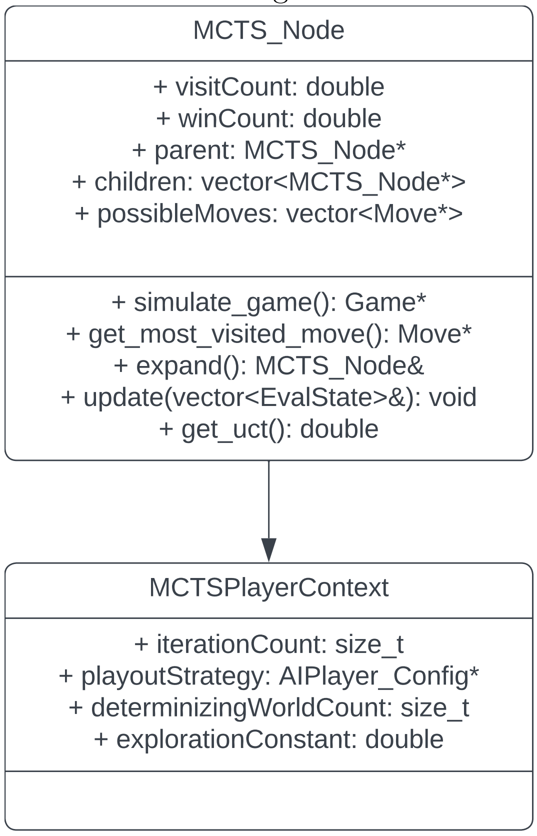
Figure 6.1 Branching factor across the rounds for the two rules-based strategies

7 Monte Carlo Tree Search Agent

The Monte Carlo Tree Search (MCTS) algorithm is a powerful method that has shown great results in games such as chess and Go. The algorithm works by building a decision tree and using simulations with different strategies to explore the possible outcomes from the current state. It iteratively selects and expands the most promising options based on previous simulation results.

The algorithm is divided into four phases: selection, expansion, simulation, and backpropagation. In the selection phase, the algorithm traverses the tree from the root to a leaf node, employing a selection strategy, often based on the Upper Confidence Bound (UCB) algorithm, to choose the most promising child nodes. Second, the expansion phase adds one or more child nodes to the selected node, representing possible future states of the game. Once a leaf node is reached, playouts from the current state are performed until a terminal state is reached. The outcomes of these simulations provide feedback about the value of each child node. In the backpropagation phase, these results are used for updating the rewards of each node along the path traversed during selection. For a deeper understanding of the algorithm, I recommend reading the paper [10] that specifically describes the application of the MCTS in board games.

In this section, our focus shifts to the application of Monte Carlo Tree Search in Sagrada. The nodes have the following structure:


MCTS_Node
+ visitCount: double
+ winCount: double
+ parent: MCTS_Node*
+ children: vector<MCTS_Node*>
+ possibleMoves: vector<Move*>
+ simulate_game(): Game*
+ get_most_visited_move(): Move*
+ expand(): MCTS_Node&
+ update(vector<EvalState>&): void
+ get_uct(): double
MCTSPlayerContext
+ iterationCount: size_t
+ playoutStrategy: AIPlayer_Config*
+ determinizingWorldCount: size_t
+ explorationConstant: double

7.1 Selection

The selection phase in the MCTS algorithm plays a crucial role in efficiently navigating the search tree towards the promising regions while balancing exploration and exploitation. One frequently used algorithm for the selection is the Upper Confidence Bound for Trees (UCT) approach being a promising solution for the exploration-exploitation dilemma.

7.1.1 Upper Confidence Bound

The UCT formula balances the exploitation of known good actions and the exploration of less explored ones. It calculates the value of each possible action by considering both its average reward and the uncertainty associated with it.

The UCT formula is expressed as:

$$UCT = \bar{X}_i + C \cdot \sqrt{\frac{\ln N}{n_i}}$$

where:

- $\bar{X}_i$ is the average reward of action i
- N is the total number of times the current node has been visited.
- n_i is the number of times action i has been selected.
- C is the exploration constant balancing exploration and exploitation.

Finding an exploration constant that produces the best results possible is not a trivial task. A higher value increases the weight of the exploration term in the formula. This leads to more exploratory behavior, as actions that have been sampled less frequently will have a higher UCT value, making them more likely to be chosen. On the other hand, a lower value of the exploration constant diminishes the influence of the exploration term, thus favoring exploitation. This behavior pushes the agent more towards the moves that produced favorable results in the past. As described in [11], a common choice of the exploration factor is $\sqrt{2}$ but in my implementation, I chose to make the exploration constant a configurable parameter so it was possible to experiment with different constants and compare the results.

The following pseudocode illustrates my implementation of the selection phase:

Algorithm 3 MCTS Selection

```
1: function GET-UCT($node$)
2:     $exploitationFactor \leftarrow node.winCount/(node.visitCount \times 5)$
3:     $explorationFactor \leftarrow C \times \sqrt{\frac{\log(node.parent.visitCount)}{node.visitCount}}$
4:     return $exploitationFactor + explorationFactor$
5: end function
6:
7: function SELECT($root$)
8:     $currNode \leftarrow root$
9:     while IS-FULLY-EXPANDED($currNode$) and not IS-LEAF($currNode$) do
10:        $currNode \leftarrow currNode$'s child with the highest GET-UCT value
11:    end while
12:    return $currNode$
13: end function
```

The constant 5 in the first line of the GET-UCT() function is present to balance the bonuses awarded for reaching specific thresholds. This award system is described in Section 7.4. The constant is selected to be 5 because in each backpropagation step, the node's `winCount` field increases by a maximum value of 5.

7.2 Expansion

After the selection phase, the expansion step involves adding new nodes to the search tree to explore new parts of the game state space. If a leaf node is not a terminal state and has unexplored actions, the expansion step adds one or more child nodes to the tree corresponding to these actions. Each child node represents a possible action from the current game state.

In my MCTS agent for Sagrada, the moves that lead to new game states represented by the nodes are filtered and sorted using the heuristic filter and the heuristic sort comparator described in Sections 4.2 and 4.3 . For the same reason as in the case of any other agents, these methods are applied to reduce the branching factor and to eliminate the weak moves.

7.3 Simulation

The simulation phase, often referred to as a playout, is a crucial step in the MCTS algorithm where the search tree is traversed from a leaf node to a terminal state through a series of simulated gameplays. The primary objective of the simulation phase is to estimate the potential outcome of a game from a given leaf node by conducting a series of simulated plays until a terminal state is reached. These simulated plays help in evaluating the quality of the current game state and guide the selection of promising actions.

In the simulation phase of the MCTS algorithm, the strategy employed must strike a balance between speed and strength. In my implementation, the playout strategy of the MCTS player is a configuration parameter that can be changed for different games. This allows experimenting with different strategies to perfect the balance and be as strong as possible. Each of the configurable strategies is an AI agent itself such as the Random agent, the Minimax agent or the Rules-based agent.

7.4 Backpropagation

The backpropagation step is responsible for updating the statistics of nodes in the search tree based on the outcome of simulated playouts. Backpropagation involves updating the statistics of all nodes along the path from the leaf node to the root node. The most common technique in backpropagation is to propagate the outcome of the playout (e.g., a win or loss) back up the tree and update the win count and visit count of each node accordingly.

I decided to use a more sophisticated reward system. The visit count of each node is always incremented by one in the backpropagation phase regardless of the result of the game. The win count of each node is incremented by 1 if the player wins the game in the simulated playout. Additional bonuses are awarded based on the player's score differential compared to their opponent. Specifically, the algorithm awards extra points for score differentials of 5, 10, 15, and 20 points. This means that if the player's score is 5 points higher than their opponent's, one extra win count is added to the node and it works the same way for the other point thresholds well.

Algorithm 4 MCTS Backpropagation

```
function UPDATE($node, evalStates$)
    if agent that made the move leading to $node$ won the simulation then
        UPDATE-WIN-COUNT($node$, $evalStates$)
    end if
    $node.visitCount \leftarrow node.visitCount + 1$
    if parent for $node$ exists then
        UPDATE($node.parent$, $evalStates$)
    end if
end function

function UPDATE-WIN-COUNT($node, evalStates$)
    $node.winCount \leftarrow node.winCount + 1$
    $winnerTotalPoints \leftarrow evalStates[0].totalPoints$
    $secondTotalPoints \leftarrow evalStates[1].totalPoints$
    for $pointThreshold$ in $[5, 10, 15, 20]$ do
        if $winnerTotalPoints - pointThreshold >= secondTotalPoints$ then
            $node.winCount \leftarrow node.winCount + 1$
        end if
    end for
end function
```

7.5 Handling Imperfect Information

Like the minimax agent, the MCTS agent handles imperfect information by running the algorithm in a set of determinized "worlds". This technique goes by various names in the literature. It is called "averaging over clairvoyance" in Russell and Norvig [9], and in the context of MCTS other authors [12] call it "Perfect Information Monte Carlo (PIMC)".

The authors suggest that PIMC search will work well in games that exhibit certain characteristics based on their findings from game analyses. These characteristics include:

High Leaf Correlation - Games where the payoffs in terminal nodes (end states) of the game tree are highly correlated, tend to be favorable for PIMC. In these scenarios, the outcome of one decision path often gives a good indication of the outcomes of other, similar paths. In Sagrada, completing patterns in most cases requires multiple correctly placed dice. On the other hand, the patterns are mostly independent especially when talking about patterns that belong to the same Public Objective Cards. For this reason, in the last moves of the game that represent the leaf nodes, the leaf correlation is high because in the last turns of the game players have a low branching factor and cannot influence the outcome of the game in a major way.

Bias - Games with a high bias, meaning the game inherently favors one player or another in many of its states, also suit PIMC well. In these games, large homogeneous sections of the game tree can be expected, which PIMC can exploit to improve its play efficiency and effectiveness. Concretely in Sagrada, there seems to be no high bias. The players aim to complete Public Objective Card patterns that are shared goals of all players. The color of the Private Objective Cards are selected randomly and there is no advantage to having a concrete color since the dice are selected randomly in each round and that way any color could be beneficial. Taking turns is balanced as well since the players alternate the starting player of the rounds and in each round, the turns are designed to be as fair as possible.

To test if the initial player of the tournament had any advantage or disadvantage, I ran 2 tournaments of 10,000 games between the First and the Random agents. In tournament 1, the First agent played as player 1 and in tournament 2, the Random agent played as player 1. The following table illustrates the results of this experiment:

Tournament number	**First win count**
1	6695
2	6693

Table 7.1 Results of the First and the Random agents in an experiment exploring potential player advantages

According to this experiment, the standard deviation of the win counts of the First agents through the three tournaments is 0.014% which is negligible meaning that there is no advantage to being the initial or the subsequent player in a tournament in general.

Disambiguation Factor - Games where the players' uncertainty about the game state reduces quickly as the game progresses (high disambiguation) are also suitable for PIMC. This factor is significant in games where player actions or game progress naturally narrows down the possible game states or outcomes, allowing PIMC to make more accurate predictions as the game advances. In Sagrada, when a new round begins, 5 additional dice are revealed. Counting with the fact that in the first round, there are 35 unrevealed dice left, I don't consider this reduction of uncertainty quick.

According to these characteristics, Sagrada may not be an ideal domain for PIMC search.

7.6 Information Set Monte Carlo

There are other ways of dealing with imperfect information in the Monte Carlo Tree Search. One of these techniques is called Information Set Monte Carlo [13]. Information Set Monte Carlo Tree Search (ISMCTS) is an extension of the standard Monte Carlo Tree Search specifically adapted for games with imperfect information. Unlike traditional MCTS, which operates on actual game states, ISMCTS operates on information sets. An information set in game theory is a set of possible game states indistinguishable to the player due to the rules of the game masking some information.

Traditional MCTS builds a search tree where each node represents a possible state of the game, and edges represent moves. ISMCTS, however, constructs a tree where nodes represent information sets rather than specific states. In games with elements of chance, such as dice selections and rolls in Sagrada, each possible outcome of the dice creates a branching path from a node. The other important part of the imperfect information in Sagrada is the Private Objective Card color of the opponent player.

I think an implementation of this technique could be a strong extension of the MCTS agent in Sagrada. Using weighted sampling could have some advantages over PIMC for example in handling the Private Objective Card color of the opponent player. It is known that there are 5 different colors of this card meaning that the opponent's color is one of the four remaining ones. Using weighted sampling, the number of samples could be equally divided into four groups each one having a different Private Objective Card color assigned to the opponent. This technique cannot be used in PIMC because the colors of the opponent cards are chosen randomly which may influence the evaluation of the given worlds in the wrong direction.

8 Tournament Results

In this chapter, I will show how the agents described in the previous sections perform using different configurations in tournament experiments. To ensure fairness when comparing AI agents, each agent needs to have the same amount of resources. When comparing AI agents in computer games in general, this is usually achieved by setting a time limit for average move time. Especially for Sagrada, counting on that each game consists of 20 turns, I decided to set this time limit to 300ms.

All the experiments present in this section were run on a computer with an Ubuntu distribution. It has an AMD Ryzen 7 5800H CPU with 8 cores and 16 threads and 16GB of RAM. I am using an optimized build of the tournament executable that is achieved by adding the `-buildtype=debugoptimized` flag to the Meson build command.

For a better understanding of the performance of given agents in the tournaments, I am using the Wilson confidence interval [14] for the win rate of the players. I chose a confidence level of 98%. These confidence intervals provide a range where the true win rate is likely to fall with 98% of certainty. The "true win rate" refers to the actual probability of winning that would be observed if a theoretically infinite number of games were played.

8.1 MCTS

The MCTS agent has 4 configurable parameters that are grouped in the `MCTSPlayerContext` structure described in the introduction to Chapter 7. In this section, I will try to find the strongest possible configurations through experiments against different agents.

8.1.1 Playout Strategies

Choosing the strongest playout strategy is about finding the perfect balance between speed and strength. In this section, I will talk about the results of four different playout strategies. These are other AI agents that can play games on their own. Namely, these are the First agent, both strategies of the rules-based agent and the minimax agent with a search depth of 1.

First, I determined the number of iterations for each playout strategy that fits the time limit by manually experimenting until the desired results were achieved. The following table displays the results of these experiments:

Playout strategy	Iteration count
First	5400
Minimax	230
Rules-based only DTFM	820
Rules-based all moves	300

Table 8.1 Iteration count that fits the time limit for the chosen playout strategies

8.1.2 Determinizing World Count and Exploration Constant

According to the results from the previous section, the different playout strategies have varying usable iterations to fit the time limit capacity. This means that dividing the total number of iterations among the determinizing worlds and choosing a strong matching exploration constant is expected to differ among the playout agents.

The branching factor plays an important role in this process because according to the selection phase described in Section 7.1, the first iterations of the algorithm simply explore all the possible moves from the current state. This means that if the iteration count is set too low, the exploration constant is irrelevant. For this reason, I ran an experiment to get the average branching factor for the 4 playout strategies each run in 100 games against the Minimax agent with a depth of 3. The following figure illustrates the results:

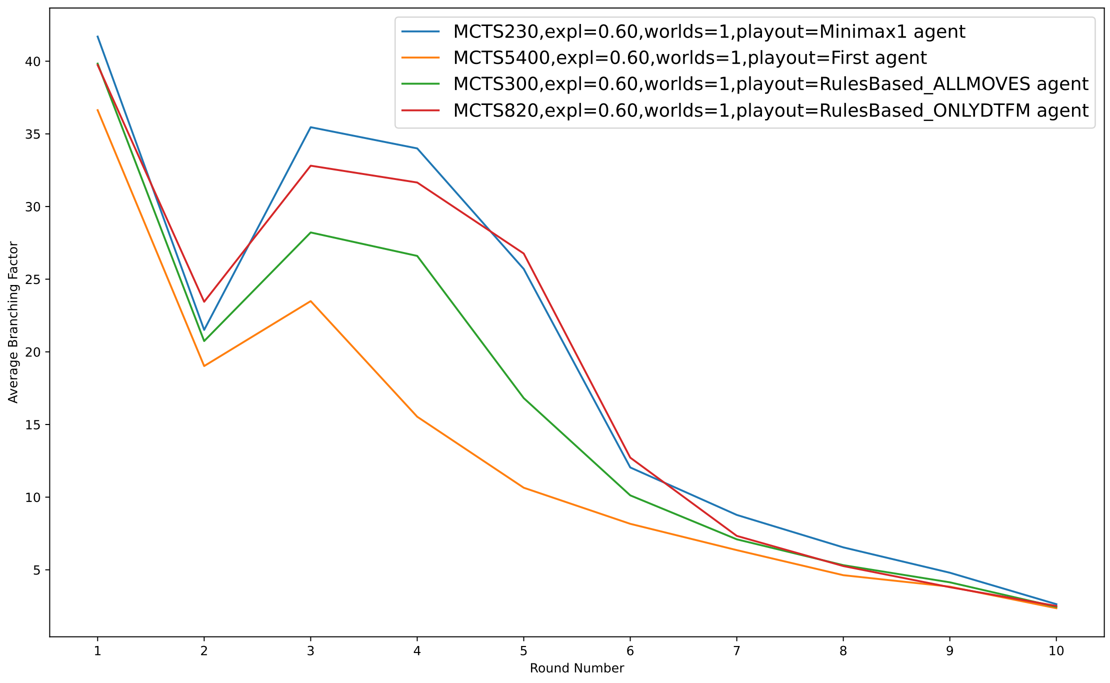


Figure 8.1 Average branching factor for the MCTS agent's all playout strategies

The figure shows that the average branching factor for the MCTS agents is the highest in round 1 with a value slightly above 40. So the algorithm's selection strategy could make a significant difference, I choose to configure a minimum of 80 iteration count for the agents.

As already mentioned in Section 7.1.1, an often chosen value for the exploration constant is $\sqrt{2}$. This experiment consisted of multiple iterations. First, I chose to experiment with all values in the range 0.4 to 2.6, with increments of 0.2. I ran separate experiments for the four playout strategies. In case of the rules-based only DTFM agent, the best win rates were produced using exploration constants close to the upper bound of the original range of values. This led to another iteration of exploration constant experiments where I used the values $[2.8, 3.0, 3.2]$. The following figures illustrate the highest win count for the different world counts displaying the exploration constant used. Every configuration used the same amount of iterations. The available iterations were evenly distributed amongst the worlds. Each tournament contained 500 games and every exploration constant, iteration count-determinizing world count pair has a corresponding tournament. All the tournaments were run against the Minimax agent with a search depth of 3 and determinizing world count of 1.

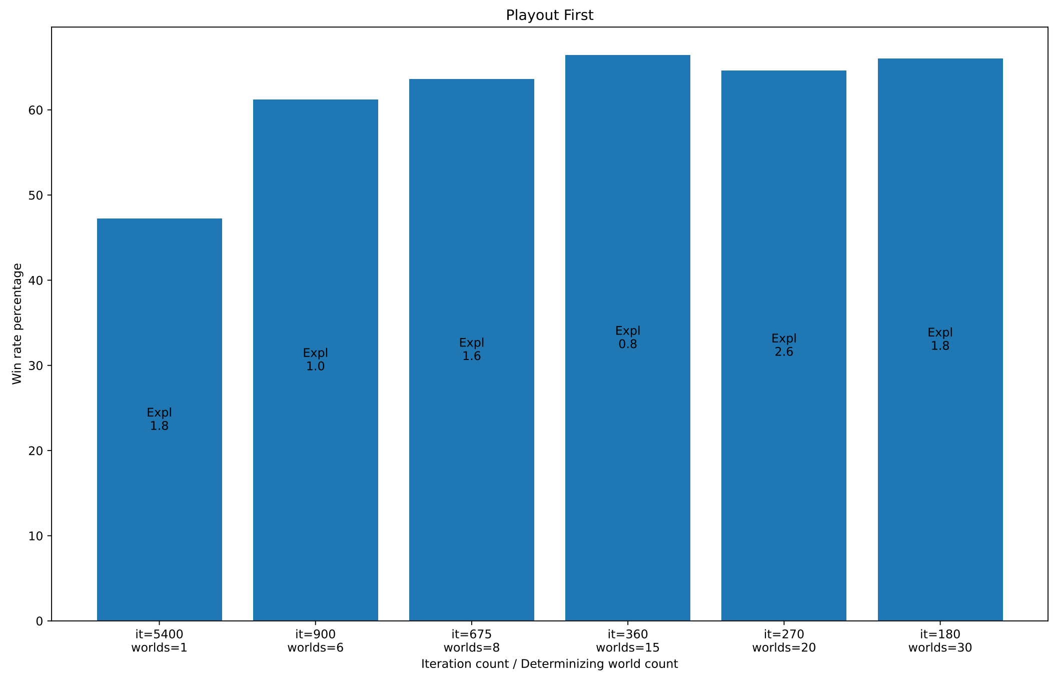


Figure 8.2 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the First playout

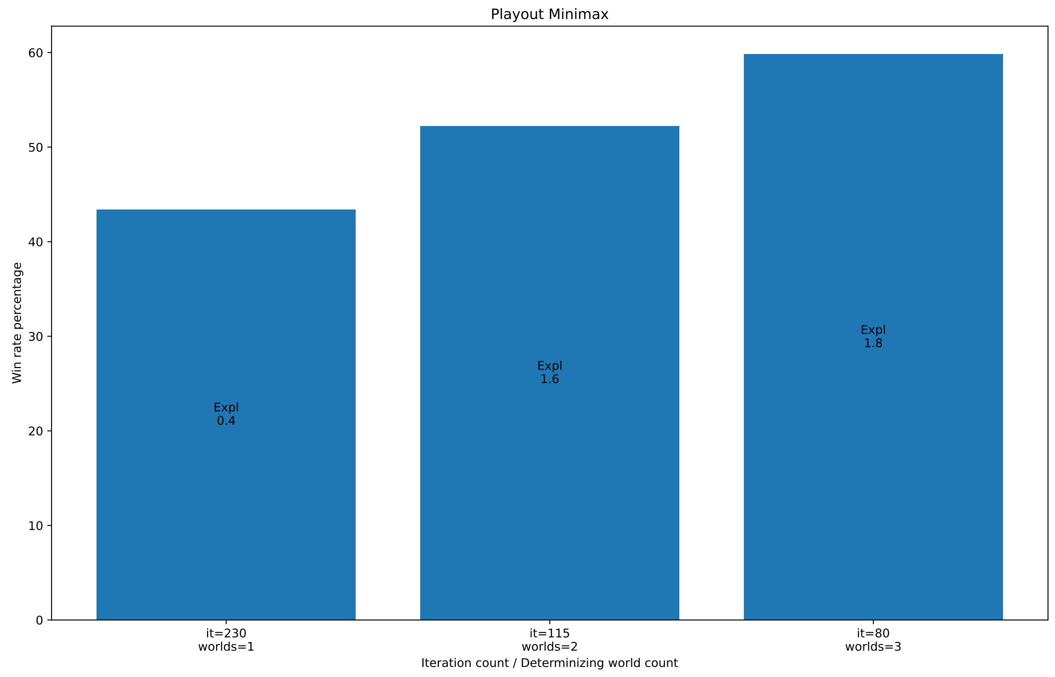


Figure 8.3 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Minimax playout

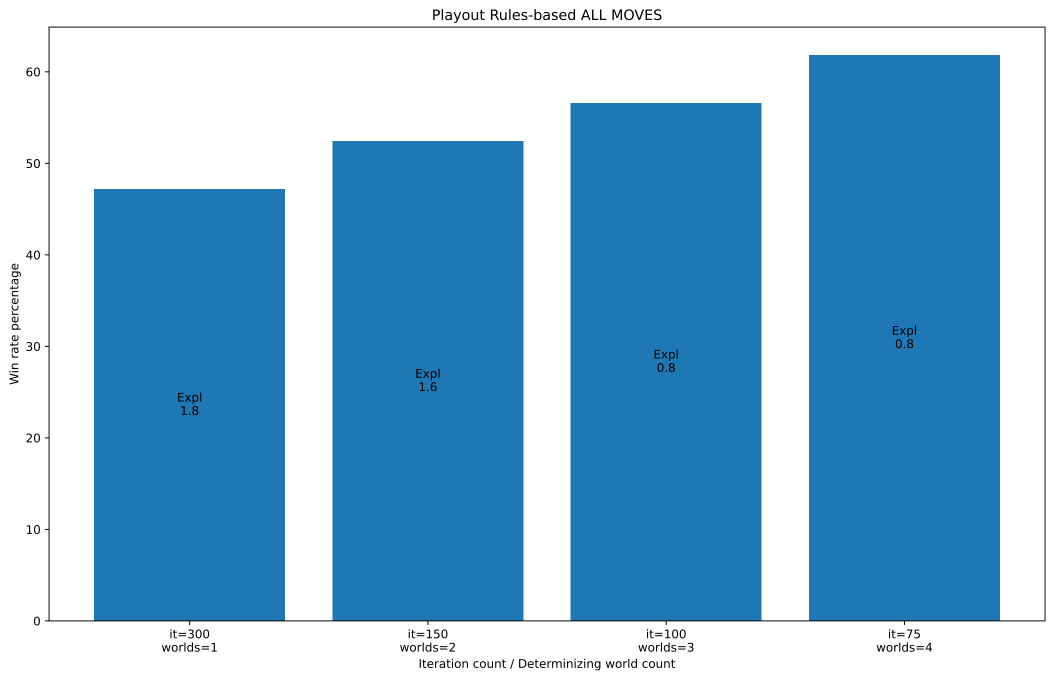


Figure 8.4 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Rules-based ALL MOVES playout

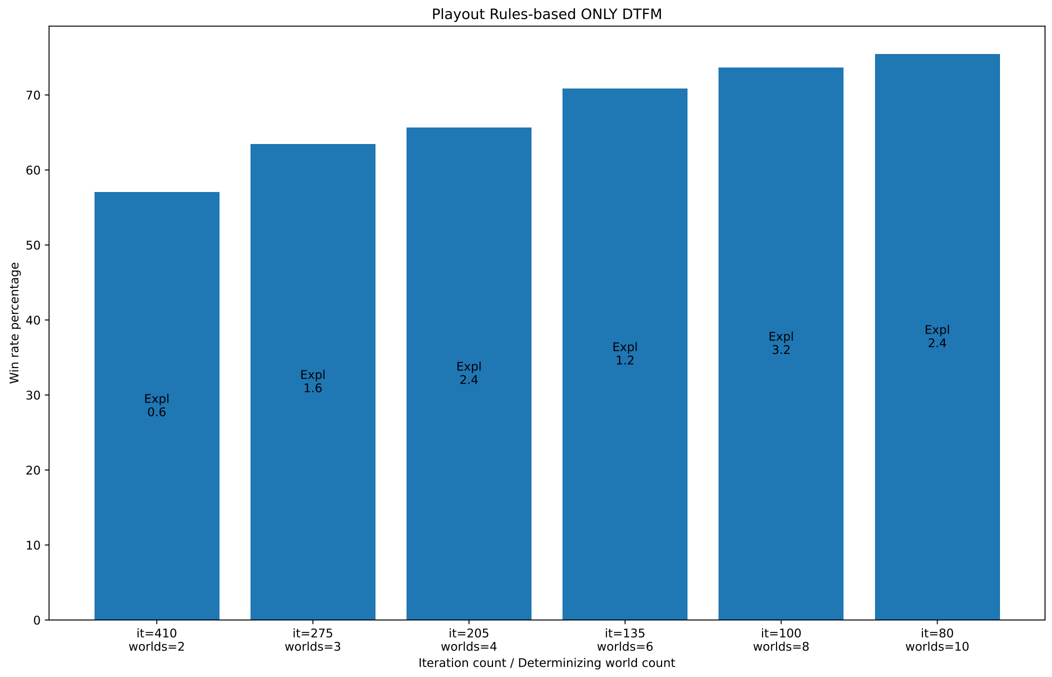


Figure 8.5 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Rules-based ONLY DTFM playout

According to this experiment, the strongest configuration for the MCTS agent is the one with the Rules-based only DTFM playout strategy using 80 iterations in 10 determinized worlds with an exploration constant of 2.4.

8.1.3 Varying Configuration Comparison

In this section, I will compare the performance of the MCTS player with increasing the iteration count. In this experiment, I used a fixed exploration constant of 2.4 and playout strategy rules-based only DTFM. The idea is to check whether increasing the iteration count increases the win rate of the player. I am using the rules-based all moves agent as a baseline opponent in this experiment and determinizing world counts of $[3, 6, 8, 10]$. Each tournament consists of 488 games.

The following figure illustrates the results of this experiment:

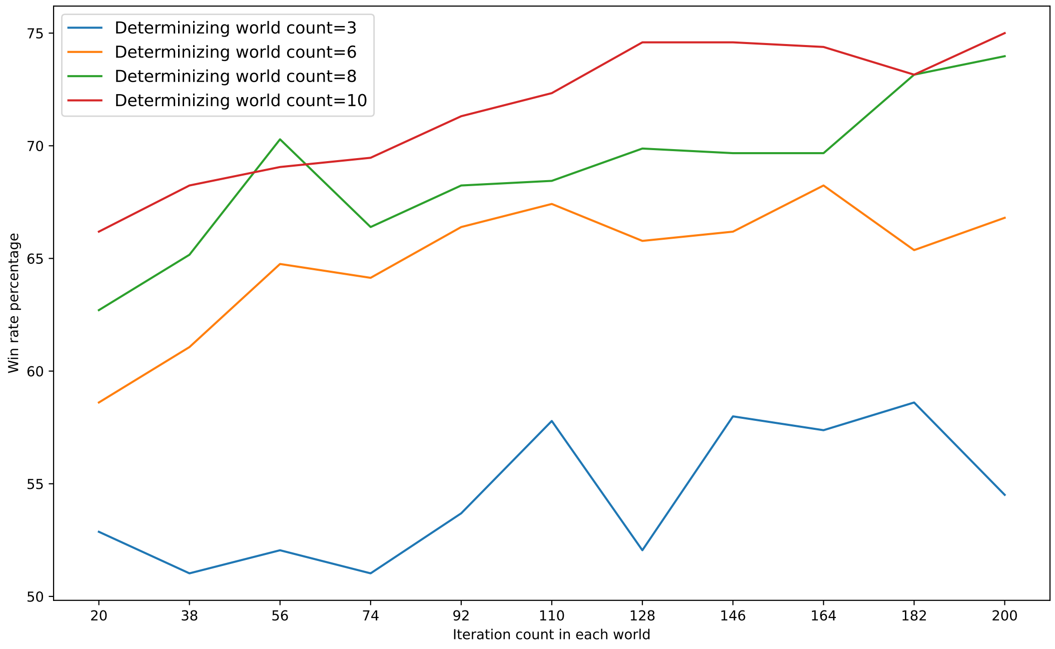


Figure 8.6 Performance measurement with varying iteration count for the MCTS player

The expected result was to have a monotonous increase in the win rate with the increasing amount of iterations available. These expectations are fulfilled because except for some cases, the win rate continually increases. Another observation of this experiment is that the win rate of an agent is highly influenced by the determinizing world count.

8.2 Minimax

8.2.1 Hyperparameters

Choosing the constants that represent the weights of the component weights in the `HeuristicConstants` structure is not a trivial task. After trying out different constants manually, I decided to automate this task by creating random configurations, running some tournaments and evaluating the results. The two main attributes that could be maximized during this experiment are average points and win count. I chose to maximize average points, which seems like a more robust metric.

To obtain the current configuration, I ran an experiment of 10,000 tournaments between the rules-based player and the minimax player. Each of these tournaments contained 112 games. Before a tournament started, I created a random configuration by choosing a random value for each component. The following table illustrates the ranges that defined the lower and the upper bound for the components:

Weight constant name	**Weight constant range**
OpponentInfluencingFactor	0.01 - 10
PuocCompletablePower	0.01 - 10
MinusPointsPerUncompletableField	10 - 1,000
CompletedPointsWeight	1 - 500
MinusPointsPerUncompletablePuocPoints	1 - 100

Table 8.2 Ranges for the possible values of the randomly chosen heuristic constant weights

When the tournaments finished, I chose the configuration that produced the highest average score for the minimax player. This experiment produced the following weight constants:

Weight constant name	**Weight constant value**
OpponentInfluencingFactor	9.84
PuocCompletablePower	3.11
MinusPointsPerUncompletableField	373
CompletedPointsWeight	416
MinusPointsPerUncompletablePuocPoints	7

Table 8.3 Currently used heuristic weight constants

Please notice that running this experiment with the current build will produce a different outcome since when I ran this experiment, I was using a previous build and the old configuration for the minimax agent. The results of this experiment can be found in the `tournament_results/heuristic_constant_experiments` directory.

8.2.2 Search Depth and Determinizing World Count

There are two other configurable parameters left after choosing the globally used heuristic weight constants as described in the previous section. The goal of this section is to find the perfect balance between search depth and determinizing world count producing the strongest possible configuration.

The following table illustrates the determinizing world count for search depths fitting the time limit:

Search depth	**Determinizing world count**
1	2000
2	200
3	20
4	2

Table 8.4 Minimax configurations fitting the time limit

I will now compare these minimax configurations in an experiment. This experiment employs a round-robin format meaning that every player plays against every other player. First, a fixed number of 320 games were run in each tournament. To achieve the best results possible, additional tournaments of 160 games were run until the lower bound of 50% of the win rate confidence interval was reached for any of the players. The upper bound for the number of games played between two concrete AI players was set to 2240. To eliminate any type of advantage, the tournaments were run with the `-b` option specified meaning that each seed is used twice, once with player1 as the initial player, and once with player2 as the initial player. This method helps to avoid any advantage connected with being the initial or the subsequent player. In the future, I will refer to this experiment format as **Sagrada Round-Robin**.

The following abbreviations for agent names are used in the following tables:

Mini1,2000 - `minimax-depth=1,worlds=2000`

Mini2,200 - `minimax-depth=2,worlds=200`

Mini3,20 - `minimax-depth=3,worlds=20`

Mini4,2 - `minimax-depth=4,worlds=2`

The following table displays a win rate confidence interval and total games played between all the minimax agents. The table has on the field in the i-th row and j-th column the confidence interval of the j-th column's agent against the i-th row's agent.

	Mini1,2000	**Mini2,200**	**Mini3,20**	**Mini4,2**
Mini1,2000		320 50.7%-63.5%	480 52.0%-62.4%	2240 48.8%-53.7%
Mini2,200	320 36.5%-49.3%		2240 48.3%-53.3%	2240 47.5%-52.5%
Mini3,20	480 37.6%-48.0%	2240 46.7%-51.7%		2240 45.5%-50.4%
Mini4,2	2240 46.3%-51.2%	2240 47.5%-52.5%	2240 49.6%-54.5%	

Table 8.5 Minimax agent configuration comparison

From the 6 tournaments run between the minimax agents, 4 of the tournaments continued to the maximum number of games chosen for this experiment. This means that there is not a huge difference between the different configurations performance-wise. On the other hand, the strongest configuration was the one with a search depth of 3.

8.2.3 Varying Configuration Comparison

In the following experiments, I illustrate how the varying parameters affect the win rate of the minimax agent. I chose the rules-based all moves agent as a baseline opponent for the experiments. Note that the purpose of these experiments was to evaluate different configurations, therefore the agents didn't have a time limit to make a move.

First, the following figure illustrates the win rate of the minimax agent with fixed search depts of 1, 2 and 3 with varying determinizing world counts. The experiment contained 6 tournaments for every search depth representing the 6 different world counts tested and each tournament contained 500 games.

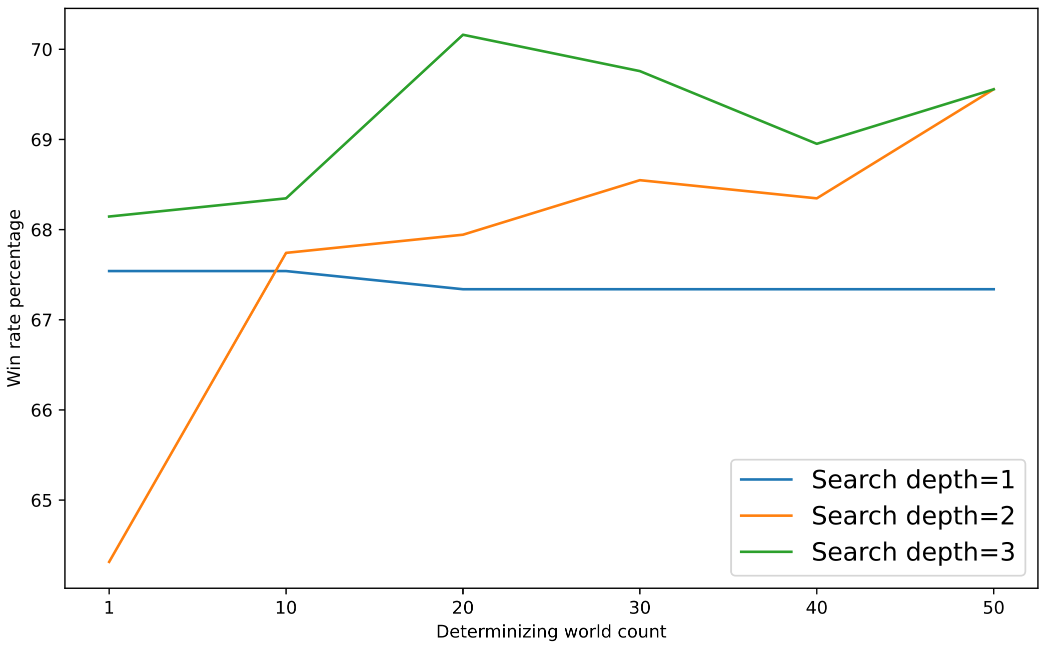


Figure 8.7 Performance of the minimax agent with fixed search depth and varying determinizing world count

Note that the minimax agent with a search depth of 1 does not encounter any piece of stochastic information because it does not go beyond the current round. However, in the different determinizing worlds, it faces agents with randomly chosen Private Objective Card colors that make the outcome for the varying world counts different. The performance of this agent is barely influenced by the determinizing worlds. On the other hand, in the case of the agent with a search depth of 2, the increasing world count spectacularly improves the performance. The agent with a search depth of 3 produces the highest average win rate of the tested search depths. The increasing world count does not show the same linear improvement of the win rate as in the case of the agent with a search depth of 2.

Next, I will illustrate how the search depth influences the performance. In this experiment, I am using a fixed number of 4 worlds. This experiment consists of 5 tournaments of 500 games with varying search depths of 1-5.

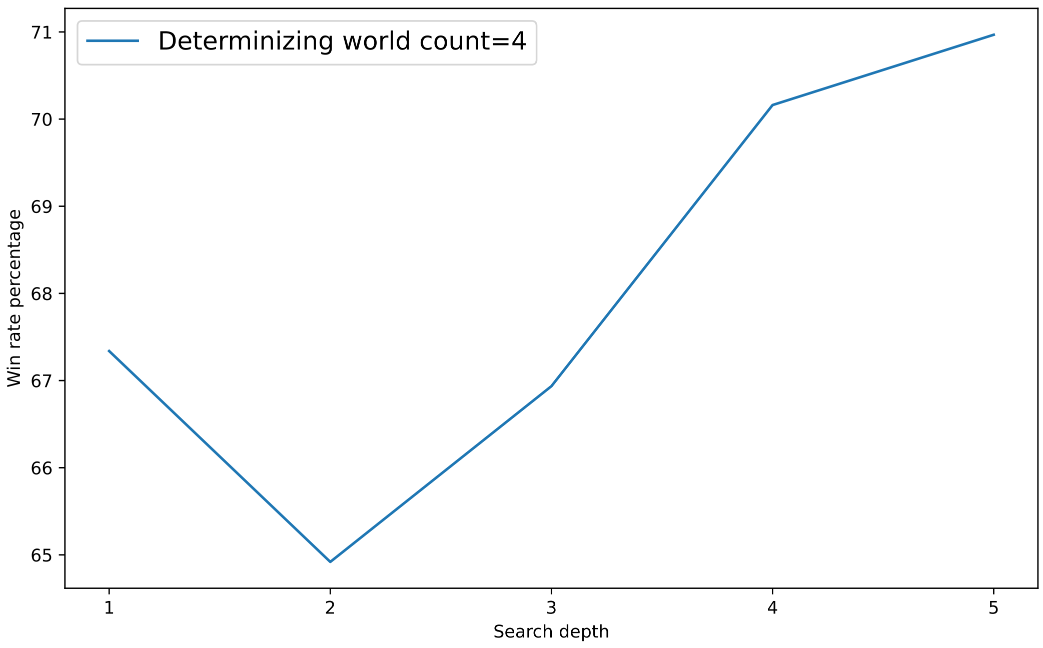


Figure 8.8 Performance of the minimax agent with fixed world count and varying search depth

As discussed in the previous figure, the number of worlds does not highly influence the agent with a search depth of 1. This is the reason why the agent with a search depth of 1 produced better results than the ones with a search depth of 2 and 3. Otherwise, there is a clear performance improvement with the increasing search depth.

8.3 Final Results

In this section, I will present the final results of various agents. For the minimax and the MCTS agent, I will use the strongest configurations that were the results of the previous experiments. Namely, these are the search depth of 3 and world count of 20 for the minimax agent and iteration count of 80, exploration constant of 2.4, world count = 10 and rules-only DTFM playout strategy for the MCTS player.

The following table illustrates the results of an experiment using the Sagrada Round-Robin format with a starting game count of 488 and steps of 488 and the maximum game count of 2928:

	MCTS	**Minimax**	**Rules-based**	**Random**
MCTS		2928 46.4%-50.7%	488 21.5%-30.7%	488 0.5%-3.0%
Minimax	2928 49.3%-53.6%		488 23.8%-33.2%	488 0.1%-1.8%
Rules-based	488 69.3%-78.5%	488 66.8%-76.2%		488 1.3%-4.7%
Random	488 97.0%-99.5%	488 98.2%-99.9%	488 95.3%-98.7%	

Table 8.6 Minimax agent configuration comparison

These results indicate that the MCTS and the minimax agents are way stronger than the rules-based agent and the random agent. The MCTS agent won the tournament against the minimax agent but the tournament was played until the maximum amount of games defined for this experiment was reached. Also, it produced a higher win rate against the rules-based agent as well and for these reasons, the winner of this experiment is the MCTS agent.

8.4 Deterministic vs Non-Deterministic Game Version Results

The deterministic version is a version of the entire game where all the hidden and stochastic information is visible to the players. This means that the agents have access to not only the current round's dice but all the dice that will be selected in future rounds.

In this section, I will compare the results presented in the previous section with the same tournaments run in the deterministic version. The same type of agents in the non-deterministic version are compared with the agents having the same computational resources in the deterministic version. This means that the minimax agent will use the same search depth and the MCTS player will use the same amount of total iterations. The idea of this experiment is to compare how imperfect information affects the different players' performance.

I am comparing the performance of the MCTS agent with a total iteration count of 800, an exploration constant of 2.4 and using rules-based only DTFM as playout strategy, the minimax agent with a search depth of 3, the rules-based all moves agent and the random agent. This experiment had the same format as the one presenting the final results in the previous section.

	MCTS	**Minimax**	**Rules-based**	**Random**
MCTS		488 20.5%-29.6%	488 2.8%-7.2%	488 0.0%-1.5%
Minimax	488 70.4%-79.5%		488 24.0%-33.5%	488 0.3%-2.4%
Rules-based	488 92.8%-97.2%	488 66.5%-76.0%		488 1.3%-4.7%
Random	488 98.5%-100.0%	488 97.6%-99.7%	488 95.3%-98.7%	

Table 8.7 Deterministic version results

This experiment resulted in a straightforward order of the players. The MCTS player is the absolute winner of the experiment. The reason behind the performance difference between the minimax and the MCTS agents in the deterministic and non-deterministic version is connected to handling imperfect information. The minimax agent uses a maximum search depth of 4 in my experiments. This means that worst case, it has to evaluate 3 moves using dice from future rounds that are uncertain at the moment. On the other hand, the MCTS player in the simulation phase, plays every game until the end. This means that in the first rounds, it has to handle 30-45 uncertain dice according to the round. Using determinizing worlds tries to solve this problem but so many different dice combinations could appear that the real values and colors of the dice could be very different than the ones determined in the worlds. Using a higher number of determinizing worlds would probably strengthen the players but it is limited due to the available computational resources.

Conclusion

In this thesis, I implemented my virtual version of the board game Sagrada and different AI agents such as the minimax agent or the MCTS agent. A detailed analysis of the game provided a deeper understanding that is used in the agents' algorithms to make them as strong as possible. Through experiments, I evaluated the different configurations of the players trying to find the strongest ones.

The minimax and the MCTS agents were undoubtedly stronger than the other agents. The strongest playout strategy for the MCTS algorithm appeared to be the Rules-based ONLY DTFM strategy. On the other hand, the minimax player with almost any configuration was able to beat the MCTS players. This is most likely due to the hardness of handling the imperfect information for the MCTS player since in the deterministic world, the MCTS player was way stronger than the other players. The reason for the performance difference between the minimax agent and the MCTS agent in the deterministic and non-deterministic worlds is mainly the amount of imperfect information each player handles when choosing a move.

Future Work

Future work on this project could involve adding more options to play using the GUI for the human players. Such options could involve playing over a network or allowing multiple local players to play on a single computer.

There are multiple newer versions of Sagrada defining new Public Objective Cards and Tool Cards. Another direction of future development could include implementing the newer versions and allowing the player to choose one.

Experimenting with different modes such as one without Tool Cards or modified scoring principles could be interesting to have deeper understanding of the AI agents' decision-making principles.

Another area for future work could be to improve the AI agents. Defining a new rules-based agent could be beneficial not only for having a new agent but as a playout strategy for the MCTS agent as well. Finding a new strategy that makes decisions fast but uses domain-specific knowledge could make a strong opponent for the rules-based only DTFM playout strategy.

One important detail could be to implement iterative deepening for the minimax player. As described in Section 2.1, each Tool Card provides a different number of possible moves. This means that in general, the average time that the minimax agent spends to make the next move highly depends on the Tool Cards of the game implying that in some games the minimax agent exceeds the time limit for making a move.

Implementing other AI agents like one using deep learning is another option to consider when talking about future development.

Bibliography

1. SILVER, David; HUANG, Aja; MADDISON, Chris J; GUEZ, Arthur; SIFRE, Laurent; VAN DEN DRIESSCHE, George; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; PANNEERSHELVAM, Veda; LANCTOT, Marc, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016, vol. 529, no. 7587, pp. 484–489.
2. *Sagrada official website*. Available also from: `https://floodgate.games/products/sagrada`.
3. *Dire Wolf Digital official website*. Available also from: `https://www.direwolfdigital.com/`.
4. FLOODGATE GAMES. *Sagrada rule book*. 2017. Available also from: `https://media.floodgate.games/rule-books/Sagrada-Rule-Book.pdf`.
5. *gtkmm*. Available also from: `https://www.gtkmm.org/en/index.html`.
6. *nlohmann-json*. Available also from: `https://github.com/nlohmann/json`.
7. *argparse*. Available also from: `https://github.com/p-ranav/argparse`.
8. GOOGLE INC. *GoogleTest*. Available also from: `https://github.com/google/googletest`.
9. RUSSELL, Stuart J; NORVIG, Peter. Artificial Intelligence: A Modern Approach. In: 3rd ed. Pearson, 2016.
10. WINANDS, Mark HM. Monte-Carlo tree search in board games. In: *Handbook of Digital Games and Entertainment Technologies*. Springer, 2017, pp. 47–76.
11. KOCSIS, Levente; SZEPESVÁRI, Csaba. Bandit based Monte-Carlo planning. In: *European conference on machine learning*. 2006, pp. 282–293.
12. LONG, Jeffrey; STURTEVANT, Nathan; BURO, Michael; FURTAK, Timothy. Understanding the success of perfect information Monte Carlo sampling in game tree search. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2010, vol. 24, pp. 134–140. No. 1.
13. COWLING, Peter I; POWLEY, Edward J; WHITEHOUSE, Daniel. Information set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, vol. 4, no. 2, pp. 120–143.
14. WILSON, Edwin B. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*. 1927, vol. 22, no. 158, pp. 209–212.

List of Figures

2.1 Branching factor across the rounds for the Random agent 14

3.1 Model structure UML diagram 17

4.1 Incorrect evaluation board position example 23

4.2 The impact of the heuristic filter on the branching factor of two different AI players 25

5.1 Branching factor comparison between the minimax agent and the random agent 33

6.1 Branching factor across the rounds for the two rules-based strategies 36

8.1 Average branching factor for the MCTS agent's all playout strategies 46

8.2 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the First playout 47

8.3 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Minimax playout 47

8.4 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Rules-based ALL MOVES playout 48

8.5 Exploration constants producing the highest win rate in different iteration count-determinizing world count divisions for the Rules-based ONLY DTFM playout 48

8.6 Performance measurement with varying iteration count for the MCTS player 49

8.7 Performance of the minimax agent with fixed search depth and varying determinizing world count 53

8.8 Performance of the minimax agent with fixed world count and varying search depth 54

List of Tables

7.1 Results of the First and the Random agents in an experiment exploring potential player advantages 42

8.1 Iteration count that fits the time limit for the chosen playout strategies 45

8.2 Ranges for the possible values of the randomly chosen heuristic constant weights 50

8.3 Currently used heuristic weight constants 50

8.4 Minimax configurations fitting the time limit 51

8.5 Minimax agent configuration comparison 52

8.6 Minimax agent configuration comparison 55

8.7 Deterministic version results 56

List of Abbreviations

GUI Graphical user interface

CLI Command-line interface

DTFM Die-to-field moves - moves that place a die to a field

PrOC Private Objective Card

PuOC Public Objective Card

TC Tool Card

A Attachments

A.1 First Attachment