Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London
Strand Campus, London,
United Kingdom

7CCSMPRJ

Individual Project Submission 2021/22

Name:	Weibo Zhao
Student Number:	21026399
Degree Programme:	Artificial Intelligence
Project Title:	AI for a social card game
Supervisor:	Murphy Josh
Word Count:	10006

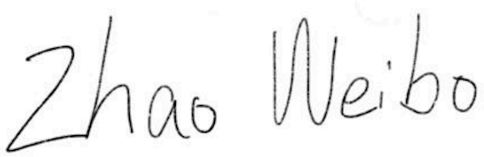
RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☑ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature:



Date: August 8, 2022

Department of Informatics
King's College London
United Kingdom

7CCSMPRJ Individual Project

AI for a social card game

Name: **Weibo Zhao**
Student Number: 21026399
Course: Artificial Intelligence

Supervisor: Murphy Josh

This dissertation is submitted for the degree of MSc in Artificial Intelligence.

Acknowledgements

Thanks to King's College London for providing relevant references, websites and learning platforms from which this project has drawn many valuable materials.

Finally, I would like to thank my tutor Murphy Josh for his careful teaching, which provided significant help for the completion of this project.

Abstract

There are many famous examples of AI applications in games, such as GO AI Alpha GO and Chess AI DeepMind. This project aims to build an AI that can play social card games with human players. However, this AI is different from previous games such as GO and Chess. Because the information available to the player in card games is incomplete, it is challenging to construct AI behavior22 patterns.

Based on the Japanese card game "Hanabi", this project builds Hanabi AI that can play the game with human players and achieve performance equal to or even better than human players in this game.

This project uses the Monte-Carlo tree search based on probability to simulate the following possible steps of the AI model and obtains the optimal action under the current situation. This project provides some ideas for the AI construction of incomplete information games.

Nomenclature

AI	Artificial intelligence
MC	Monte-Carlo algorithm
MCTS	Monte-Carlo tree search
PIMC	Perfect Information Monte-Carlo
UCT	Upper Confidence Bounds for Trees

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

Nowadays, artificial intelligence technology based on computer science is applied by more and more industries, and artificial intelligence technology plays its role in various industries.[1].

For hundreds of years, social gaming has been about humans' ability to show off their wits, deceive, persuade, threaten, charm or bargain with others. These behaviors are completely impossible for a machine that cannot think on its own, but the advent of artificial intelligence break it.

In 1997, IBM's Deep Blue supercomputer stunned the world by defeating Garry Kasparov[2], the then world chess champion, within a standard time limit of 3.5 to 2.5 points. This is the first AI to beat a human player in a social game. Since then, artificial intelligence technology has developed rapidly; until March 9, 2016, AI entered a new generation, Google's AlphaGo AI defeated the world Go champion Lee Sedol and created a new milestone in the era of artificial intelligence[3]. The amazing performance of artificial intelligence in social games is due to the computing power of computers and the clever algorithms that engineers have designed for them. As of now, CPUs can do 100 billion to 100 trillion floating-point calculations per second, so fast that computers can complete and calculate almost every possibility in a social game in seconds, so humans have lagged far behind AI in terms of fast calculation and calculation accuracy.

There is also a lot of AI technologies in social card games[4]. Social card games are different from the board games mentioned above. Card games not only require you to win by beating your opponent, but sometimes you also need to cooperate with your opponent to win. Moreover, in social card games, the player is more limited, because in most cases, the player does not have complete information about the game, only his own information. Since the player does not have complete information about the match, there is no way to choose the best strategy for the situation. So in a card game, the biggest challenge is how to cooperate with teammates and how to choose the best strategy with incomplete information.

The power of AI can also be applied to social card games. Unlike chess and Go, social card games involve not only battles between players but sometimes Requires cooperation with other players. Also, in social card games, the AI can only know its own hand in most cases. AI has no way of knowing other players' hand cards, and therefore cannot effectively choose strategies based on information on the field.

One of the key challenges for AI in social card games is how to choose the most beneficial strategy in the presence of incomplete information.

1.2 Aims and Objectives

This project will design AI for the Japanese boarding game 'Hanabi' and enable it to make rational strategy choices in the absence of complete information and to cooperate or compete with human player.

There are two main aims of this project:

- **Explore whether AI can be applied to card games with incomplete information and evaluate the performance differences between AI player and human player.**

  AI has outperformed humans in common perfect information games, where players can see all the information on the field in real-time, and each player has the same information, such as Go, chess and Sudoku. However, in incomplete information games, the AI is not able to process the information on the field and make optimal decisions with incomplete information in a very sound way, as is often the case in card games or board games.

- **Explore The Monte-Carlo Tree Search and its variants in the incomplete information game performance.**

  The Monte-Carlo is a general term for a class of stochastic methods. AlphaGo uses this algorithm, which works well in board games but does not mean it will do better in card games. Currently, more and more algorithms based on Monte-Carlo tree search have been used to solve incomplete Information games, such as Perfect Information Monte-Carlo (PIMC)[5]. This project will also explore the performance of its variant algorithms.

1.3 Background and Literature Survey

This section will briefly review relevant research in this research field and explore relevant research on AI applications and social card games around the world based on literature comparison.

More than a decade ago, researchers began to study the feasibility of AI in asymmetric information games and began to study relevant algorithms.

In the Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference paper, Filipa Correia et al. discussed the possibility of robots as social card game sueca players, which is a card game popular in the Spanish community that require participants to trade and perform in order to win. There are two modules in their search - the game module and the social module[6]. The game module showed the algorithms they designed for card AI. The social module represents the research they have done to improve the trust of human players towards robot players. In this paper, the Perfect Information Monte-Carlo (PIMC) has been applied as the core algorithm and achieved excellent performance, which is an algorithm used for incomplete information. Their team improved PICM for sueca's rules and time limits and created a hybrid player that dramatically increased the win rate.

In addition to using PIMC and related MC algorithms, deep reinforcement learning is also a good method that can be applied to social card games. Professor Johannes Heinrich's team try to use deep reinforcement learning and Fictitious self-play in the

perfect information game to make the behavior between the two players to the Nash Equilibrium[7]. However, the disadvantage is that this method is temporarily only suitable for card games played by two people, but it still provides ideas for this project.

Adam's team from the Facebook AI Research institute tried to use the Monte-Carlo algorithm and deep reinforcement learning to solve the problem of AI in the asymmetric information domain[8]. Their team's paper presented at the AAAI conference laid a solid foundation for the development of AI in card games. Its core idea is to combine the Monte Carlo algorithm with deep reinforcement learning, which is mainly divided into single-agent search and multiagent search. Single-agent search assumes that one agent uses online search, which is the strategy learned from the environment in reinforcement learning, and the remaining agents use the blueprint search strategy. Multiagent search is based on a single-agent search, each agent will learn, but the search space may be 10 million times that of a single-agent search; when the search space is too large, some agents retreat to useing blueprint search. They also optimized the model using Action-Observation history (AOH), Common Knowledge (CK) and other metrics, used RNNs as the basic model of Deep RL, and deployed it in the famous cooperative game Hanabi, and achieved far more than expected scores under the multiagent search.

In the research direction mentioned above, some observational games (mostly card games) provide a very favorable research direction for the field of multiagent cooperation and deep reinforcement learning[9], and the Monte-Carlo algorithm plays a decisive role. However, it is still very difficult for agents in games to cooperate with team members and fight against opponents at the same time. The next two papers provide feasible solutions for intelligent agents that consider both cooperation and confrontation through DouDiZhu[21], a traditional Chinese game.

DouZeor[10] is an AI designed explicitly for Doudizhu, a card game that requires both cooperation and confrontation and has far more action space than any other card game, up to the power of 10. Enhance the Classic Monte-Carlo Methods with deep Neural Networks, Action Encoding, and parallel Actors to be used to beating other Doudizhu AI players. It beat DeltaDou in just 10 days of training, which is a strong AI program which uses Bayesian methods to infer hidden information and searches the moves with MCTS. The success of DouZero proves that the combination of the classical Monte Carlo algorithm can be perfectly adapted to partially observed cooperative and antagonistic card games.

The algorithm is put forward, as the field of artificial intelligence applied to the asymmetric information in the field of AI algorithms are constantly changing and improving, the studies to the research direction are roughly the Monte-Carlo search tree and depth of reinforcement learning is very suitable for use to solve the asymmetric information social card games, and have good performance in the field of confrontation and cooperation.

1.4 Organization

This paper will introduce the project along multiple lines and is organized as follows:

- Chapter 2 (Background), this chapter will detail an introduction to related algorithms and their variants for solving non-social game fields (more suitable for non-perfect information game fields), and give the relevant mathematical theory proof.
- Chapter 3 (Objectives, Specification and Design), this chapter will introduce the purpose of this project in detail, the basic rules of Hanabi games and the related requirements of the project, and related technical specifications.
- Chapter 4 (Methodology and Implementation), this chapter will introduce in detail how to build an AI model and a game model and combine the two into one project, which will introduce many detailed classes and functions, and will also attach a detailed code snippet.
- Chapter 5 (Results, Analysis and Evaluation), this chapter is mainly about the results of the project operation and related brief analysis and gives the comparison of the performance of AI and human players in scoring the performance of AI and the reasons for obtaining such results.
- Chapter 6 (Legal, Social, Ethical and Professional Issues), this chapter mainly discusses whether the project complies with the British Computer Society Code of Conduct and Code of Conduct and the social and ethical issues that may arise from the project.
- Chapter 6 (Conclusion), this chapter mainly organizes the conclusions and inspirations of the conclusions of this project and provides a reference direction for future research.

2 Background Theories

This section mainly discusses the technical background principles used in the project and how they can help AI player achieve better performance. This section is divided into two subsections. The first section is about the Monte-Carlo method based on probability and statistics and some variants. The second section is the algorithm of the Monte-Carlo tree search and its realization process. The third section is the application steps of the Monte-Carlo tree search in the field of incomplete information games.

2.1 Monte-Carlo

Monte-Carlo[11] is a general term for a class of stochastic methods. The characteristic of this kind of method is that approximate results can be calculated on random samples. With the increase of samples, the probability of obtaining the correct result gradually increases.

Monte-Carlo method to deal with the problem of probability has very good performance[12]; it can be requested problem is converted into a certain type of random distribution, and the distribution of eigenvalues can be found, such as the frequency of random event or expectation variance of a distribution, etc., which can then be through these characteristic values as the key part of the experimental data.

The key to the excellent performance of the Monte-Carlo method in the field of incomplete information games lies in its ability to learn from experience when the environment and model are unknown. This experience includes the state, action and reward of the sample sequence. After obtaining the experience of several samples, The reinforcement learning task is solved by averaging the returns over all samples. Similar to the dynamic programming(DP) method, MC solution can also be regarded as a generalized policy iteration process; that is, the value function corresponding to the current strategy is calculated first, and then the value function is used to improve the current strategy, and the two steps are continuously cycled, so as to obtain the optimal value function and optimal strategy.

2.1.1 Prediction in MC

The goal here is to learn the value function from the experience of the sequence of states according to the policy π, and the value function is the expected value of the return. The following formula is the expectation formula under the policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \tag{2.1}$$

However, the Monte-Carlo method does not seek the expectation of return in policy evaluation but uses empirical mean return. The more samples, the average is going to converge to the expectation.

$$\bar{V}_\pi(s) = \frac{1}{N} \sum_{i=1}^{N} G_{i,s} \tag{2.2}$$

Here, 'i' is the state sequence index, and 's' is the state.

An episode can be viewed as a sample, assuming that for states s, given a policy π, the value function $v_\pi(s)$ is computed. In an episode, each appearance of s is called a visit. Of course, in an episode, s may appear more than once. The first occurrence of this state is called first-visit, so the first-visit MC method[13] is to average the returns obtained from all the first visits to s. According to the Law of large numbers, when the sample is large enough, the mean will approach $v_\pi(s)$. As the name implies, the every-visit MC method is to average the returns received by all visits to s.

2.1.2 Control in MC

Like dynamic programming, once the value function is obtained, the Monte-Carlo method can be used to search for the optimal solution.

The following is the strategy optimization formula of DP[14].

$$\pi'(s) = \arg\max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right] \tag{2.3}$$

This equation determines the optimal policy by finding the behaviour that maximizes the sum of rewards. But it's important to note that it uses transition probabilities, which is not very common in model-free learning. Since we do not know the state transition probability $p\left(s', r \mid s, a\right)$, we cannot conduct a forward search like DP. Therefore, all information is acquired through the experience of playing the game or exploring the environment.

2.2 MCTS

Monte-Carlo Tree Search (MCTS)[15, 19] is a process of gradually building an asymmetric Search Tree by randomly deducing the game, which belongs to a particular type of reinforcement learning.

The Monte Carlo tree search can be roughly divided into four steps.

- **Selection**
- **Expansion**
- **Simulation**
- **Backpropagation**.

In the beginning, the search tree has only one node, which is where we need to make a decision. Each node in the search tree contains three essential pieces of information: the situation it represents, the number of times it has been visited, and the cumulative score.

2.2.1 Selection

The node selection in the downward traversal of the tree is achieved by choosing to maximize some quantity, where the players must choose a slot machine to maximize the estimated revenue of each round. The Upper Confidence Bounds (UCB) formula [16]is often used to calculate returns.

$$UCB = v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{2.4}$$

Where v_i is the estimated value of the node, n_i is the number of times the node has been accessed, and N is the total number of times the parent node has been accessed. C is an adjustable parameter.

In 2006, Kocsis and Szepervari first constructed a complete MCTS algorithm by extending UCB to minimax tree search and named it the Upper Confidence Bounds for Trees (UCT) method[16]. This is actually the version of the algorithm used in many current MCTS implementations.

$$UCT\left(v_i, v\right) = \frac{Q\left(v_i\right)}{N\left(v_i\right)} + c\sqrt{\frac{\log(N(v))}{N\left(v_i\right)}} \tag{2.5}$$

Where v_i is the MCTS node whose UCT value is being calculated, and v is the parent node of this node. $Q(v_i)$returns the profit value of the v_i node, that is, the sum of simulated earnings of all considered child nodes under this node. $N(v_i)$returns the number of runs of an MCTS node on the backpropagation path. c constant is used to weigh the degree to which the UCT function is oriented on "Exploitation" and "Exploration". *log* is the natural log.

The UCT function can be divided into two parts. The first part is called "Exploitation" ($\frac{Q(v_i)}{N(v_i)}$). The second part is called "Exploration" ($c\sqrt{\frac{\log(N(v))}{N(v_i)}}$).

The exploration value is more significant when $N(v_i)$ is smaller. In other words, more compensation is given to nodes with fewer visits, which can only be selected if they compete with nodes with higher revenue ratios. If UCT only considers revenue ratios, nodes that happen to have a negative return on the first simulation will never have a chance to be selected again. Because UCT always chooses the node with higher revenue, this node is ignored. Therefore, with the addition of exploration compensation, even if the first simulation return is negative, with the addition of compensation, there is still a chance to be selected later to find the node with the best return through more simulations.

2.2.2 Expansion

After the selected node is obtained, the possible and untried action a is expanded with the state s of the current node and the new state s' is obtained. The newly expanded state node s' is added to the Monte-Carlo search tree, and this node is taken as the return value. This step is called "Expansion".

The essence of this step is to expand the confidence space further, and the state displayed is to add a new state s' to the confidence space. The expansion operation determines the specific action. For example, in neural network search, MCTS can decide to increase the number of neurons in the existing network's last layer and the current architecture's hidden layer.

2.2.3 Simulation

Simulation is a single game strategy[17], a series of games starting from the current node (representing the game state) and ending at the end node after the game result is calculated. A simulation is a game tree node that evaluates approximate calculations starting at the initial point of a random game.

In the simulation, actions can be selected using the rollout policy function:

$$\text{RolloutPolicy: } s_i \rightarrow a_i \tag{2.6}$$

This function will input a game state s_i and produce the choice of the following action a_i. In practice, the function will allow many simulations to dash, and the default rollout policy function can be random sampling with uniform distribution.

The simplest form of simulation is simply a random sequence of actions in a given game state. Simulations always produce an evaluation, which in the case of a game is a win, a loss, or a draw. Usually, any value can be a reasonable outcome of the simulation.

The strategy thinks about how to play the game like human beings. The game tree usually represents a root node representing the initial state. When given a specific game rule, the rule can be used to traverse from the root node, and any unexpected node can be searched without storing the whole tree in memory. When an action is performed according to the rule, a new state node is obtained, which brings new information and affects the following action.

MCTS also uses the same features to build game trees. All nodes can be classified as visited or not visited. In general, if the simulation treats the node as an initial node, it has been evaluated at least once, and then it can be considered a visited node. In practice, the search starts with none of the root node's children being accessed. Then a node is selected, and the first simulation begins.

2.2.4 Backpropagation

After the first visited node simulation is completed, the result is backpropagated to the root node of the current game tree. The node where the simulation started is marked as visited.

Backpropagation traverses[18, 19] the child node (where the simulation begins) back to the root node. The simulation results are transmitted to the root node, and statistics are calculated/updated for each node along the backpropagation path. Backpropagation guarantees that each node's data will reflect the simulation results that started at all of

its children (since the simulation results are transmitted back to the root of the game tree).

The purpose of backpropagation simulation results is to update the total simulated reward $Q(v)$ and the total number of visits $N(v)$ for all nodes v along the backpropagation path.

- $Q(v)$, the total simulation reward, is a property of node v, which in its simplest form is the sum of the simulation results obtained through the considered nodes.
- $N(v)$, the total number of visits, is another attribute of node v and represents the number of times node v is located on the backpropagation path.

Each visited node preserves both values. Once a given number of simulations have been completed, visited nodes retain the exploited or explored information.

2.2.5 Pseudo code for MCTS

```
def monte_carlo_tree_search(root):
    while resources_left(time, computational power):
        leaf = traverse(root) # leaf = unvisited node
        simulation_result = rollout(leaf)
        backpropagate(leaf, simulation_result)
    return best_child(root)

def traverse(node):
    while fully_expanded(node):
        node = best_uct(node)
    return pick_univisted(node.children) or node # in case no children are present / node is terminal

def rollout(node):
    while non_terminal(node):
        node = rollout_policy(node)
    return result(node)

def rollout_policy(node):
    return pick_random(node.children)

def backpropagate(node, result):
   if is_root(node) return
   node.stats = update_stats(node, result)
   backpropagate(node.parent)

def best_child(node):
    pick child with highest number of visits
```

Figure 1: Pseude code for MCTS.

2.3 MCTS in the imperfect information game

The successful use of MCTS in perfect information games[20], such as Chess and Go, has prompted researchers to apply them to games with more complex rules, such as card games, real-time strategy games (RTS). In summary, three main features that challenge the practical applicability of tree search algorithms in such games:

- **Incomplete information** - The state of the game is not fully available to the player, meaning that decisions are made without full knowledge of the opponent's position. For example, consider the cards in the play's hand or the cards covered in the "fog of war." Depending on the game, other hidden elements can be hidden, such as a stack of face-down cards.
- **Randomness** - the randomness of the game mechanics. for example, in a game, a player gets a hand card randomly at the beginning of a random count.
- **Combinatorial complexity** - Each player's turn consists of a series of phases, or a decision consists of several actions at once. Typically, there are many actions generated by a continuous space, such as. A player's hand may be limited to multiple cards, and since each hand may be randomly generated, their combination creates a considerable amount of confidence.

These three properties significantly increase the branching factor, so classical MTCS have limited applicability. There are several ways to improve this problem significantly.

2.3.1 Certainty

One way to solve the problem of randomness in games is a certainty. Certainty involves a game that samples from the space of possible states and accordingly forms perfect information for each such sample. Sampling requires setting a specific instance (value) for each unknown feature to determine it so it is no longer unknown. For example, certainty might involve guessing which cards an opponent might hold in a card game. Searching the game tree on each state of perfect information is called PIMC. Perfect Information MCTS (PIMC) supposes that all confidential information is determined. Then the game is treated as a perfect information one with respect to a particular assumed state of the world.

This approach dramatically increases the branching factor when a project has more than one perfect piece of information, so enhancements such as mobile pruning using domain knowledge and the binary representation of the tree are employed. These improvements reduce CPU time per move and make computing budgets more manageable.

3 Objectives, Specification and Design

3.1 Problem Overview and Objectives

This project aims to design an AI that can play the famous Japanese card game "Hanabi"[22, 23, 24], and it needs to cooperate with human players through several actions to achieve higher scores or win. In addition, the performance of the AI player and the human player is measured by the final average score; The project also set up a control group of cooperation between human players and human players to measure the AI's performance.

The project is divided into two main components:

- **Design of rules for the card game "Hanabi"**, "Hanabi" is a very typical incomplete information game; each player can only see other players except his hand, so each player has a part of field information rather than the whole of the hand, and on the field, players need other players to speculate his hand, in addition to this, each player has a variety of action information to the field interference, Section 3.2 for details of the rules of the game.
- **Design of AI player**, The design of the AI player is the project's focus. The AI must choose the next course of action from imperfect information and cooperate with other human players to achieve higher scores. Since the search tree for imperfect information is too large, pruning is needed to reduce the search space for AI to achieve a balance between search time and maximum score.

3.2 Hanabi rules

3.2.1 Configuration of the game

- **Cards**: Game cards combine numbers and colors, such as blue number 1, red number 2, etc. The game has a maximum of five colors of cards, each with a number distribution from 1 to 5, where the default number configuration is three cards with the number 1, two cards with the number 2,3,4, and one card with the number 5. Number configuration can be changed arbitrarily depending on the gameplay.
- **Hand cards & deck cards**: Each player is initially given a hand of five or four cards, and the remaining cards are placed in random order on the deck, where no one can see the colors or numbers of deck cards.
- **Hint Tokens**: It can indicate to other players the color or number of their card. If the hint is color, then player will know all the cards with the same color; If the hint is number, then player will know all the cards with the same number.
- **Penalty Tokens**: It is used to count the number of times a player misses, and a penalty token is given when a player does not arrange his cards in the regular order, and the game fails when it reaches the prescribed upper limit.

- **PlayedCards**: It is used to store the cards played, there are as many colors as the corresponding stack, the stack of cards in numerical order.
- **DiscardCards**: It is used to store discarded cards, the cards are facing up here, which means that all players can see the colors and numbers of the cards.

3.2.2 Game process

1 : Select a random player as the initial player and move in clockwise order.

2 : When it is the player's turn to act, there are three actions to choose from:

Play, Play a card in hand to use; if the number of playing cards in PlayedCards can form a sequence, then play a booming score, and the hand was added to PlayedCards; Get a Penalty Token if the number of the hand card in PlayedCards does not form a sequence. And get a hand card from deck cards.

Discard, Player discards a hand to DiscardCards and obtains a hand from deck cards, replenishes a Hint Token, an action that cannot be performed if the Hint Token is at its maximum.

Hint, Use a Hint Token to hint at another player's hand. If the hint color is chosen, the player will know all the hand cards of the same color. If the hint number is chosen, the player will know all the hand cards of the same number.

3 : Conditions for the end of the game.

Hanabi is a cooperative game; as members of the same team, everyone must work together to create a beautiful firework. So there's a mutual win or loss.

When Penalty Tokens reached the maximum limit, game over.

When there is no card on the deck, game over.

When the cards in player's hand couldn't make a sequence forever, game over.

3.2.3 Score

Each match has different card configurations, so the total number of cards is used as the total score for each match, and the number of cards in PlayedCards is the final score. Here are the achievements corresponding to the scores.

- **0% - 20%**: Hanabi beginner - Sucks!!!! The crowd is scattered!
- **21% - 40%**: Hanabi apprentice - Not bad! But no one will remember it!
- **41% - 60%**: Hanabi Master - That's great! The crowd was drawn in!
- **61% - 80%**: Hanabi Master II - Great! Tonight will be engraved in everyone's memory!
- **81% - 100%**: King of Hanabi - Perfect!!!!! The stars are pale by comparison!

3.3 Requirements

3.3.1 Functional Requirements

The following Functional Requirements Outline The functions necessary for AI to interact with The game and human players in Hanabi games.

- Access to all available information in the game, such as the current number of cards on the deck and other players' hands, the current number of hint and penalty tokens, and the distribution of DiscardCards and PlayedCards.
- Make three basic actions, play, discard, and hint.
- It can calculate and select the best one of the three actions according to the known situation.
- Be able to take cues from other players and make decisions based on other information.
- It can avoid making the wrong choice to the greatest extent and avoid getting a penalty token.

3.3.2 Non-Functional Requirements

In addition to meeting the above functional requirements, the model must meet the following non-functional requirements to complete the project. These include:

- AI model parameters are easy to tune.
- The whole game flow is clear and easy to play.
- The AI model can correctly prune the belief space to reduce computation time.
- It is easy to compare the differences between AI and human players.

3.4 Technical Specifications

The programming language used in this project is Python3.9, an object-oriented interpreted script programming language, and Python3.9 is the latest release of Python. Because Python has a beneficial AI community and rich documentation and development packages, it is very time to develop AI-related projects. Some development packages used in this project, such as NumPy, can significantly improve development efficiency.

In addition, the source code of many papers in this field has used Python as the primary programming language. To keep consistent with researchers in this field, Python is also the best choice.

The Integrated Development Environment (IDE) used for the project was PyCharm and Visual Studio Code, which were perfectly compatible with Python and made it easy to obtain the required facade installation packages. Their built-in command-line

tools make it easy to adjust the project, and the final interface with the game is the command-line interface (CLI).

This project uses version 4.13 of Anaconda, a development version of Python and R. Anaconda is widely used in machine learning, data statistics, predictive analytics and big data. It can effectively manage Python installation packages in different environments.

4 Methodology and Implementation

This chapter outlines the construction of Hanabi's basic architecture (rules of play), the design of the AI player, and how to apply the AI player to the game. The relevant technologies in this chapter are derived from Chapter 2 - Background, and are localized on the background of the essential technologies, making the relevant technologies more applicable to Hanabi.

The chapter is split into eight subsections, each subsection presenting the design and implementation of a critical aspect of the AI model or the game architecture. The subsections cover the following areas:

- Build the game architecture.
- AI player design
- The combination of AI player and Hanabi
- Test module

4.1 Game Architecture

This part is the foundation of this project. The design of AI is also based on the basic rules of Hanabi, so how to design a complete game without loopholes is the core of this part.

The design of this part of the code mainly depends on the Hanabi rules in section three. The Hanabi game mainly has the following classes to form the complete game architecture:

- **Player class**, it is used to store basic information about the player, such as the player's class (human player or AI player), the player's name, the number of cards in the player's hand, and the player's hand. The player also needs actions, such as the basic three actions (play, discard, and hint), which will be discussed in more detail later.
- **Card class**, it is used to store information about cards, precisely about one card rather than all cards, because all cards can be thought of as groups of many card classes. The main information here includes the color of the card, the number of the card, whether the color of the card is hinted, and whether the number of the card is hinted. Since cards are called by other classes, and for practical reasons, no action is assigned to the card class.

- **Desk Cards class**, it is used to store all the cards on the current desktop. The implementation of this class is based on the card class. It has the basic properties of the number of cards and the distribution of cards, and it also has the corresponding action method to adjust the current desktop cards.
- **Card stack classes**, it is divided into two classes, one is PlayedCards, and the other is DiscardCards, the most critical of which is PlayedCards, which is used to store the cards that have been played and formed into a sequence. DiscardCards is used to represent a discarded hand.
- **Token classes**, it is also divided into two classes, Hint Tokens and Penalty Tokens, where Hint Tokens are used to represent the token that can be used for hinting, and Penalty Tokens are used to represent the token in case of missed actions.
- **Actions class**, The most critical class in Hanabi Classes, which is based on all the above classes, implements three key actions: play, discard, and hint.

The details of each class design concept and basic code implementation follow.

4.1.1 Player class

Player Class is the simpler class in Hanabi Classes. It is easy to implement. This class is initialized with four essential attributes: Player name, Player type, Player hand number, and Player hand card(which is empty when initialized).

The following are some class action function, the first is to initialize the good players for empty hand allocation, main function realization is randomly selected from the deck class player hand the number of allocated to players; in addition, there are a class used to determine whether the player hand should be suggestion cards were suggested, such as Weber now have red 1 and red 2, When hint to the color of the first card, the two cards should be hinted to red, this function used to judge whether the all eligible cards have been hinted. The code snippet is shown in Listing 1.

```
1  class Player:
2
3    # Initialization functions
4
5    def __init__(self, name, order, numberOfCards):
6        self.name = name
7        self.order = order
8        self.cards = [Card(None, 0)] * numberOfCards
9        self.numberOfCards = numberOfCards
10
11
12   def setCards(self, cards):
13       self.cards(cards)
14
15   def drawHand(self, DeskCards):
16
17       for i in range(len(self.cards)):
```

```
18              self.draw(DeskCards, i)
19
20      # DISCARD
21
22      def draw(self, DeskCards, cardPosition):
23          newCard = DeskCards.removeRandom()
24          self.cards[cardPosition] = newCard
25          return newCard
26
27      # HINT
28      '''
29      @para hintType = color/number
30      @para hint = 1,2,3,4/yellow,red,blue
31      check all hint has been given in a hintType
32      '''
33      def allHintsGiven(self, hint, hintType):
34          if hint is None:
35              for i in range(len(self.cards)):
36                  if (self.cards[i].colorHinted == False) or (self.cards[i].
    numberHinted == False):
37                      return False
38              return True
39          elif hintType == "color":
40              for i in range(len(self.cards)):
41                  if (self.cards[i].color == hint) and (self.cards[i].
    colorHinted == False):
42                      return False
43              return True
44
45          elif hintType == "number":
46              for i in range(len(self.cards)):
47                  if (self.cards[i].number == hint) and (self.cards[i].
    numberHinted == False):
48                      return False
49              return True
```

Listing 1: Player class

In addition to the above functions, the Player class also has a visualization function, which stores the information that has been hinted. For example, if the number of a card has been hinted, a ”N” is added to the back of the card to indicate that the number of the card has been hinted; if the color of a card has been indicated, add a ”*” at the end of the card to indicate that the color of the card has been hinted. The code snippet is shown in Listing 2.

```
50      def storeInfo(self):
51          # Declaring rows
52          N = len(self.cards)
53          # Declaring columns
54          M = 2
55          # using list comprehension
56          # to initializing matrix
57          cardMatrix = [[0 for i in range(M)] for j in range(N)]
```

```
58
59          for i in range(len(self.cards)):
60              number = str(self.cards[i].number)
61              color = str(self.cards[i].color)
62              if self.cards[i].numberHinted:
63                  number = number + "N"
64              if self.cards[i].colorHinted:
65                  color = color + "*"
66
67              cardMatrix[i][0] = number
68              cardMatrix[i][1] = color
69          return cardMatrix
70
```

Listing 2: Visualization

4.1.2 Card class

The card class is the most basic class in Hanabi classes. It is the critical element that makes up the Hanabi game. All other classes and actions are based on it.

As mentioned above, the card class initialization has four essential attributes: card color, card number, whether card color hinted, and whether card number hinted. The last two attributes are Boolean types set to False when initialized.

Although the Card class does not have an action function, it does have a visualization function that shows the color and number of the current card. The code snippet is shown in Listing 3.

```
71      class Card:
72
73      # Initialization functions
74
75      def __init__(self, color, number):
76          self.color = color
77          self.number = number
78          self.colorHinted = False
79          self.numberHinted = False
80
81      # Visualization functions
82
83      def sayInfo(self):
84          print("Color: {}, number: {}.".format(self.color, self.number))
85
86      def storeInfo(self):
87          return ("Color: {}, number: {}.".format(self.color, self.number))
88
89      def __eq__(self, other):
90          return self.__dict__ == other.__dict__
91
```

Listing 3: Crad class

4.1.3 Desk Cards class

The card class is used to represent the cards currently on the deck. Two parameters are passed during initialization, the number of colors and the card distribution. The total number of cards(NOC) can be obtained by the product of the number of colors and the card distribution. The formula is as follows:

$NOC = sum\{color\} * sum\{distribution\}$

For example, if there are 3 colors (red, blue, and green) and the card distribution is 1,1,1,2,2,3,3,4,4,5, then NOC is $3 * 10 = 30$ And during initialization, the color and card number distribution will be combined and added to the array of cards as all the cards in this game.

It also has two action functions, one of which is used to randomly select a card from deck cards that will be assigned to the player's hand at the start of the game; Another function removes the card just selected from deck cards and subtracts the total number of cards by one.

Finally, the visualization function shows the cards on the deck, but this information will not be shown to the player during the game.

4.1.4 Card stack classes

The first thing to look at is the PlayedCards class. Its initialization only needs the parameter of the number of colors. It can be understood that there are several colors, and there are stacks corresponding to several colors. Each stack can store the cards corresponding to the color. All players get as many points as there are cards in a card stack.

It also has an action function that adds cards that meet the rules to the PlayedCards. For example, the newly added cards must match the color of the card stack, and the card stacks are added in the order of numbers.

Another DiscardCards class is used to store the cards discarded by the player, and the discarded card needs to display its color and number.

Generally, these two card stack classes are empty at the beginning of the game. After each play, if the played card matches the rules, a card will be added to the PlayedCards, otherwise, a Penalty Token will be gotten; A card will enter the DiscardCards and receive a Hint Token.

4.1.5 Token classes

The Hint Tokens class defines the tokens used for a hint. Its initialization is the maximum number of tokens defined by the game rules, and the current number of tokens is initialized to the maximum token number. It has two action functions: one is removed token, which removes a token when the player hints; the other is added token, which adds a token when the player discards, but the total will not exceed the maximum token number.

The Penalty Tokens class is similar to the Hint Tokens class, except that the current number of Tokens is 0 during initialization. A Penalty token will be added when the wrong card is played. There is no function to reduce it.

```
92      class HintTokens:
93
94      def __init__(self, tokenNumber, tokenMax):
95          self.tokenNumber = tokenNumber
96          self.tokenMax = tokenMax
97
98      def tokenRemove(self, human):
99          if 0 < self.tokenNumber:
100             self.tokenNumber -= 1
101         else:
102             if human:
103                 print("\033[31mError. Not enough tokens. You need to discard
    to get token.\033[0m")
104
105     def tokenAdd(self, human):
106         if self.tokenNumber < self.tokenMax:
107             self.tokenNumber += 1
108         else:
109             if human:
110                 print("\033[31mError. Tokens maximum limit reached. You need
    to give hint to use token.\033[0m")
111
112
113 class PenaltyTokens:
114
115     def __init__(self, tokenNumber, tokenMax):
116         self.tokenNumber = tokenNumber
117         self.tokenMax = tokenMax
118
119     def tokenAdd(self,human):
120         if self.tokenNumber < self.tokenMax:
121             self.tokenNumber += 1
122         else:
123             if human:
124                 print("\033[31mError. Tokens maximum limit reached. You lost
    game.\033[0m")
125
```

Listing 4: Tokens class

4.1.6 Actions class

The action class mainly implements three essential functions: play, discard, and hint.

The hint type and content are entered when the hint function is initialized. For example, the hint type is color, and the hint is red. This function will call the hint token class and card class and convert the colorHinted or numberHinted parameter in the card class to True.

When the discard function is initialized, it will input the card position, an integer data representing the index the card player wants to discard from the player's hand. It also calls the hint tokens class, which gets a hint token when the discard is complete.

The card position will also be input when the play class is initialized. This function will judge whether the card played by the player matches the rules, and then judge whether the player should score or get the penalty token in turn.

The code blocks for the Play in Listing 5.

```
126     def play(self, cardPosition):
127         # cardPosition: Int. Index of the card you want to play from your
    hand.
128 
129         # initializing variables (extracted from state)
130         global activePlayer
131         newState = copy.deepcopy(self)
132         newState.parent = self
133         newState.depth = self.depth + 1
134         human = newState.human
135 
136         if newState.turn == 1:
137             activePlayer = newState.Player1
138         elif newState.turn == 2:
139             activePlayer = newState.Player2
140         DeskCards = newState.DeskCards
141 
142         playedCard = activePlayer.cards[cardPosition]
143 
144         # check if the card was correct somehow
145         # use functions from the playPile
146         verif = newState.PlayedCards.addCard(playedCard, human)
147 
148         if verif == False:
149             newState.penaltyTokens.tokenAdd(human)
150             newState.DiscardCards.addCard(playedCard)
151             if human:
152                 red = ';'.join([str(1), str(31), str(28)])
153                 message = ("\033[31mYou got a penalty token!\033[0m")
154                 print("\x1b[%sm %s \x1b[0m" % (red, message))
155 
156         if len(DeskCards.cards) > 0:
157             activePlayer.draw(DeskCards, cardPosition)
158         else:
159             activePlayer.cards.pop(cardPosition)
160 
161         newState.switchTurn()
162 
163         return newState
164 
165     def switchTurn(self):
166         pass
167 
```

Listing 5: Play

4.2 AI design

This project divides the design of AI into two parts; the first part is the calculation of AI belief space; the other part is the implementation of the Monte-Carlo search tree.

4.2.1 Belief space

The calculation of the belief space is not complex but very critical because the Monte-Carlo tree search needs to rely on the current belief space search. It needs to input the game's current state during initialisation, which includes the current state of all game architecture classes such as players, decks, tokens, and card stacks.

When the above state is obtained, it needs to rely on the above state information for pruning. For example, when a particular card is hinted at, the AI knows part of the information of this card and can reduce the belief space.

The final belief space class will help the AI get the possible distribution of its hand. The code snippet is shown in Listing 6.

```
168     class BeliefSpace:
169     def __init__(self, state, hand_size):
170         self.states = []
171         state.depth = 0
172         DeskCards = state.DeskCards.storeInfo()
173 
174         # transform the list into tuple
175         DeskCards = [(i, j) for [i, j] in DeskCards]
176 
177         # Reduce the belief space if the AI cards have hints
178         hinted = []
179         unhited = []
180         for i in range(len(state.AI.cards)):
181             if state.AI.cards[i].colorHinted or state.AI.cards[i].
    numberHinted:
182                 hinted.append((state.AI.cards[i].number, state.AI.cards[i].
    color))
183             else:
184                 unhited.append(i)
185 
186         player2 = [(i, j) for [i, j] in state.AI.storeInfo() if (i, j) not in
     hinted]
187         unknown = player2 + DeskCards
188 
189         all_combination = set(list(combinations(unknown, hand_size - len(
    hinted))))
190         for i in all_combination:
191             newstate = copy.deepcopy(state)
192             DeskCards = [item for item in unknown if item not in i]
193             newstate.desk = [Card(k[1], k[0]) for k in DeskCards]
194 
195             new = [Card(k[1], k[0]) for k in i]
196             for ind, k in enumerate(unhited):
197                 newstate.AI.cards[k] = new[ind]
```

```
198
199             self.states.append(newstate)
200
```

Listing 6: Belief space

4.2.2 AI model

After solving the belief space problem, the AI model's design can be officially started.

First, the most crucial task of the AI model is to find a combination of states and actions that can maximize the utility of AI from the belief space just obtained. Therefore, conducting a traversal search on the state space. A State search class is set here to initialize it. Input Player, deck, card stack, token and other parameters and set the current status of the parent node, depth and value to 0.

Since states need to be updated based on different actions, the AI model must design three state transition functions based on three different actions. Those functions accept the current state and get three different states based on three actions: play, discard, and hint.

The next class is Solver, the core class of the AI model, which is responsible for calculating the downward searching utility for each state and selecting the action and state of the largest utility. Its initialization is very simple. The most critical parameter is the maximum search depth, which determines the maximum depth of the search tree, that is when to stop the tree search algorithm. This parameter is set by the human. If you want AI to have more control over the game well, this parameter should be appropriately increased. Still at the same time, it will also significantly increase the AI search time and occupy more resources. Therefore, it is necessary to make reasonable adjustments to the hyperparameters to balance the game computing resources and score.

The utility function is used to calculate the utility of the current state. There are also many hyperparameters here, such as the weight of the cards that have been scored, the weight of penalty tokens and the weight of discarded cards, but these hyperparameters have little impact on the entire project; there will be a detailed comparison in the Results section. The utility formula:

$Utility = \sum \left(card_{color}^{number}\right) * cardStackWeight + numberOfPenaltyTokens * penaltyWeight + numberOfDiscard * discardWeight$

Where $\sum \left(card_{color}^{number}\right)$ is the sum of number of the card in the PlayedCards, and the cardStackWeight is positive, the penaltyWeight and discardWeight are negative. In the default configuration of the AI model, cardStackWeight is 10, penaltyWeight is -5, and discardWeight is -1.

The maximum value function is to return the maximum value of play, discard and hint in the current state. If the current state is empty, it returns 0. If the current state gets more penalty points than the parent state, it returns -10 (negative number). Each downward search will cause the state depth to increase by 1. When the start depth is greater than the maximum depth, the utility value of the current state is returned.

The weight function is to return probabilities for what the player might play; if there is a hint on a card, playing this card is more probable than other actions; if there are no hints, giving a hint is the most probable. When the color or number of a card in the cards is hinted, the card's weight is set to 1; when both its color and number are hinted, its weight is set to 2.

When no card is hinted, the weight of hint is 1, and the weight of play is 0; when a card is hinted, the weight of hint and play is calculated according to the following formula:

$$hintWeight = \frac{1}{3 * \sum cardWeight} \tag{4.1}$$

$$playWeight = \frac{2}{3 * \sum cardWeight} \tag{4.2}$$

Where cardWeight is the weight of hinted card mentioned above (1 or 2). It can be seen that the weight of the play is twice that of the hint, which means that when there is a hinted card, the AI model should be more inclined to play the card.

The maximum value function and weight value function are coupled to each other and depend on each other.

The final function is Evaluate, and this function will traverse the entire belief space. Each card in each state in the belief space will calculate the hint color, hint number, play and discard weight value and add it to the array Children, in addition to the above four actions to the corresponding array named Actions. Finally, return to the action and card with the most prominent utility.

In this part, we need to combine the AI model with Hanabi[26] so that the AI can play Hanabi like a human player and cooperate with the human player. And this part should also be the entrance of Hanabi.

First, it is necessary to make a selection interface for the basic configuration of the game. Two modes are designed in this project, one mode is player and player, and the other mode is player and AI. The purpose of this design is to facilitate the comparison of AI performance.

At the same time, two game configurations are also set. The default configuration is to use the pre-set configuration to play the game, including 2 cards colors (red, blue), 4 hand cards upper limit and the card number distribution is [1, 1, 1, 2, 2, 3, 3, 4, 4, 5] The total number of cards is 20, so the total score is 20, 8 maximum hint tokens, 3 maximum penalty tokens. The other is a manual configuration, where the above parameters can be set by the player for the game before the game. The initial configuration of the above games will be packaged into an initial state and passed to the state space and AI model for calculation. After the calculation is completed, the AI will give its actions and cards, so Hanabi will update the state according to the actions and cards and hand it over to the next player to act.

The above is the idea of how to combine AI and Hanabi.

4.3 Test module

This part tests the AI module, the game module and the combination of the two modules. It only describes the relevant design, not the design details.

The game module test includes inputting each keyword's initial parameters to test whether the current state can be obtained correctly; the test for AI includes whether the current state can be given the correct belief space and whether the optimal action can be searched in the belief space.

4.4 Project architecture

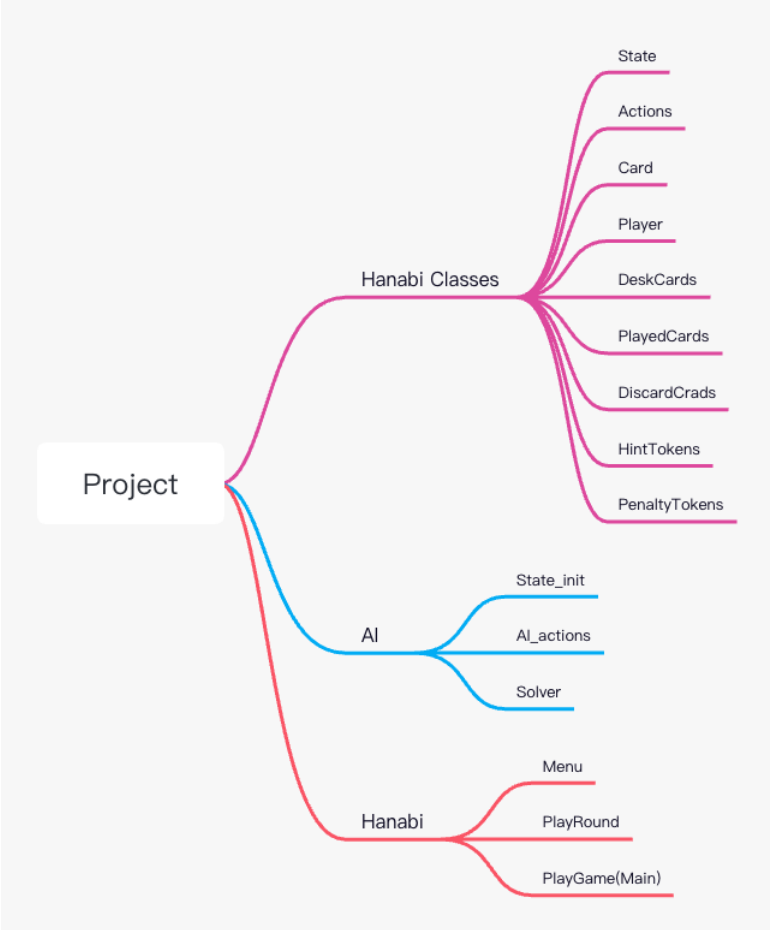


Figure 2: Project architecture

5 Results, Analysis and Evaluation

This chapter reports the results and evaluates the performance of the AI model. The evaluation will start from the following aspects: score, computing resource consumption, error rate, and control experiments with human players. In addition, this section also considers the influence of some hyperparameters on the experimental results[25], such as the maximum search depth, etc.

The chapter is split into four subsections which cover the following areas:

1 : Game Score and Penalty token

2 : AI performance in calculation

3 : Hyperparameters

5.1 Game Score and winning rate

The final score of the game is one of the most critical factors in measuring the performance of the AI model. Under the premise of using the above default game configuration, the game's total score is 10. AI players scored an average of 8.444 points in 100 tests. And the AI model got stuck in the loop 3 times and couldn't make a decision, which means that the AI's error rate is about 3%. Among them, 23 rounds of AI and human players have won the game (total score), and it can be estimated that the winning rate of the game is about 23%.

In the default configuration, the average score between human players and human players in 100 games is 7.63 points, which is lower than the 8.44 points between human players and AI players. In 100 games, only 4 games are won by human players, which is lower than AI's 23 games.

In the manual configuration game mode, set the card color to 3 and other parameters remain unchanged. At this time, the total score is 15. After 50 games, the average score of AI and player group is 13.68, 8 of the 50 games are won, and the winning rate is about 16%. Although only one color is added, the computing time of AI has increased several times or even dozens of times. The discussion on performance is in the next section.

The human player group had an average score of 13.44 over 50 games in the manually configured game, which was not as good as the AI group but narrowed the gap relative to the default configuration.

Many factors cause this result. In games where AI and human players cooperate, human players are more inclined to give hints to AI, while AI is more inclined to use the Play action, which may be due to the hyperparameter settings when setting the utility. AI is more inclined to consider hinted cards when acting, while humans are likelier to trust the AI's accuracy in calculating probabilities.

In a game where human players and human players cooperate, each human player has a different personal strategy, some human players prefer to give hints to others, and some human players tend to play hands 100% of the time.

	Default(100 games)		Mutual(50 games)	
	score	win rates	score	win rates
AI & human	8.444/10	23%	13.68/15	16%
human & human	7.63/10	4%	13.44/15	3%

Table 1: Scores and win rates

5.2 AI performance in calculation

Before discussing AI performance, we first need to introduce the hardware configuration of the computer deployed in this project. This project uses two computers for comparison. On the first day, the computer used a MacBook Air (M1, 2020) with 16 GB of memory and the other one. The machine uses the Windows operating system; the memory is 16 GB, the CPU is i7-7500, and the GPU is GTX 1060 (1280sp).

Using the default configuration, the AI takes an average of 7156.52ms to process each action on a MacBook; the shortest time is 108ms, the longest time is 33015ms, and most action processing times are below 10000ms. Under the default configuration of 100 times, the probability of each action of AI executing play is 55.74%, the probability of executing hint is 24.59%, and the probability of executing discard is 19.67%. To sum up, it can be seen that AI prefers to use the action of the play; of course, this is also related to the fact that human players and teammates prefer to give AI player hints.

In the manually configured game (3 card colors, the rest of the parameters remain unchanged), the AI player's response time has increased significantly. In 50 games, the average response time of each action is as high as 100126.64ms, which is much higher than the default. The configuration is 7156.52ms, and the longest step takes up to 423897ms. This means that each game is closer to 15-20mins. This phenomenon is caused by the exponential growth of the AI belief space. When the critical parameters of the game, such as the number of cards in hand, the distribution of card numbers, and the number of card colors, increase linearly, the data that AI needs to calculate will be exponential or even higher. The final result is that each step of AI takes dozens of times more time to calculate.

The conclusions of the control experiment on another Windows computer are as follows. Using the default configuration, the average response time of AI players in 100 games is 8043.56ms, of which the longest response time is 43723ms. Compared with the MacBook, the time consumption is slightly increased but not obvious, which is within the normal range. However, under the configuration of adding a card color, the average time consumption of the Windows machine is 16850ms, which is about 6 times lower than that of the MacBook, which has been significantly improved, which means that the hardware device limits the performance of AI rather than the algorithm. Computation times within 20000ms are acceptable. In the subsequent experiments, another color was added to reach 4 colors. This time, the processing time was significantly longer. The average calculation time per calculation reached 133498ms, which was 6 times that of 3 colors in Windows.

Counting the decision-making ratio of each AI in the Windows computer, it is found

that the proportion of AI executing play is 58.43%, the proportion of executing hint is 23.74%, and the proportion of executing discard is 17.83%. The results of the two control experiments are not much different, which means that The action distribution designed by this algorithm is that play account for about 55%-59%, hint account for about 22%-25%, and discard account for about 17%-20%.

	Default(100 games)		Mutual(50 games)	
	MacBook	Windows	MacBook	Windows
AI & human	7156.52ms	8043.56ms	100126.64ms	43723ms

Table 2: Running time

	MacBook			Windows		
	Play	Hint	Discard	Play	Hint	Discard
AI & human	55.74%	24.59%	19.67%	58.43%	23.74%	17.83%

Table 3: AI action

5.3 Hyperparameters

We all know that hyperparameters are crucial in artificial intelligence and other fields involving models[25], because hyperparameters need to be added when the model is configured. It determines the performance of the model from the beginning. A good hyperparameter can greatly improve the performance of the model. Improve the accuracy of the model and shorten the training and calculation time. This section will evaluate the impact of hyperparameters on the model.

The number of hyperparameters involved in this project is not many. The most critical hyperparameter is the maximum search depth. Increasing the maximum search depth allows AI to see more steps to obtain more information about the possible occurrence of the current state. Effectively improve the score, but at the same time, it will also significantly increase the computing time of the AI model. In the control experiment, the maximum search depth was increased from 2 to 3, which means that each time the AI model will be able to see the utility of each possible state in the belief space after 3 rounds of actions to choose a better action.

After changing the maximum depth, 50 experiments were carried out, and the average score was 8.63, which was a slight improvement compared with the search depth of 2, but the average time required for each calculation increased to 127520ms. At the level of default configuration, the relative time increased by 16 times. It can be seen that the score improvement in exchange for sacrificing computing time is not acceptable. In addition to the increase in calculation time, the number of times that AI players execute Hint has increased, which has increased to about 31%, and the relative play ratio has dropped to about 50%.

50 games in Windows with default configuration		
search depth	score	running time
2	8.444	8043.56ms
3	8.63	127520ms

Table 4: The impact of search depth on AI

In addition, this section also changes the weight parameters used by the utility function to explore the impact of the weight on the AI's result judgment. First, try to increase the weight of the cards to 10 times, which means that the AI will be more inclined to choose the ability to score after 20 experiments, the average score of the cards was 8.64; then reducing the weight of the cards to one-tenth, the average score was 8.24 after 20 experiments. It can be seen that the weight of the cards does not significantly affect the final result. Changing both the weight of penalty tokens and the weight of discarded cards alone will not change the final result significantly, but increasing the weight of cards while reducing the other two weights will make the AI more aggressive, meaning that the AI will not hesitate to get Penalty tokens are also played as it deems reasonable.

But the conclusion is that the weights in the utility function do not significantly impact the final result.

6 Legal, Social, Ethical and Professional Issues

This chapter outlines the legal, social, ethical and professional issues within the context of the project.

6.1 British Computer Society Code of Conduct and Code of Conduct

All code throughout the project conforms to the British Computing Society's Code of Conduct (COC) and Code of Good Practice. Some open source third-party libraries are used in this project, and the holders or authors of these code libraries agree to disclose their code for everyone to use for non-profit-making activities.

This report also cites opinions in related field articles and conference reports. This citation model is widely adopted in the research field. At the same time, the authors of this report respect the protection of intellectual property rights, fully respect the owners of all cited resources, and guarantee their Results are not violated.

6.2 Social and Ethical Issues

The discussion based on AI and ethical issues has always been the focus of this project. Projects that use AI to replace human orientation will involve ethical issues. The ethical issue involved in this project is whether the use of AI to replace humans in social card games. The division of responsibilities is unclear, and whether AI has the ability to

take responsibility. For example, in a world-level card game in which AI and humans cooperate, AI should also bear the same responsibility as human players.

Compared with other AIs applied to the medical industry and the automotive industry, AI applied to social games will not cause serious and substantial harm to human beings, so as long as users in related fields strengthen supervision, ethical issues can be avoided.

7 Conclusion

In short, this project has achieved its intended goal of building an AI model that can cooperate with human players to play Hanabi games. The outcome of this project shows that AI models cooperating with human players can replace better performance than human players cooperating with human players.

The following conclusions can be drawn from the results. First, the AI model can only play a simple two-player game. With the improvement of the game configuration (increase the color of the card, increase the number of cards in hand), the running time of the AI will increase significantly, even to the point of intolerance, which means that the AI model is related to There is still a lot of room for improvement in the algorithm, which is the next research direction.

In addition, hardware devices have a significant impact on the performance of AI models, and AI may be deployed on machines with more robust hardware performance for testing in the future.

Interestingly, regardless of the game configuration, the scoring performance opportunity of the human player group is always lower than that of the AI group in this project, which is primarily related to the game strategy of the human player. It also shows the feasibility of applying AI to social card games. This project also demonstrates that AI can achieve excellent performance in games for complete information (chess, Go) and outperform human players in card games with incomplete information.

A Appendix

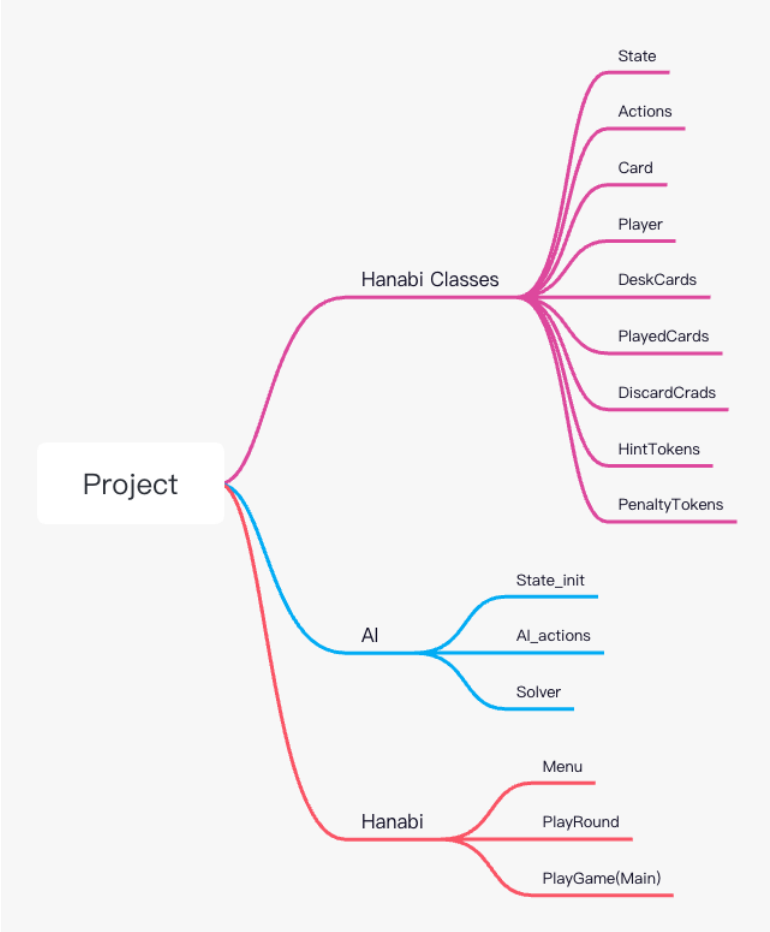


Figure 3: Project architecture

For the code, see Supplement File.
HanabiWithAI.zip

References

[1] Boyd M, Wilson N. Rapid developments in artificial intelligence: how might the New Zealand government respond?[J]. Policy Quarterly, 2017, 13(4).

[2] Hsu F H. Behind Deep Blue: Building the computer that defeated the world chess champion[M]. Princeton University Press, 2002.

[3] Lee C S, Wang M H, Yen S J, et al. Human vs. computer go: Review and prospect [discussion forum][J]. IEEE Computational intelligence magazine, 2016, 11(3): 67-72.

[4] Niklaus J, Alberti M, Pondenkandath V, et al. Survey of artificial intelligence for card games and its application to the Swiss game Jass[C]//2019 6th Swiss Conference on Data Science (SDS). IEEE, 2019: 25-30.

[5] Long J R, Sturtevant N R, Buro M, et al. Understanding the success of perfect information monte carlo sampling in game tree search[C]//Twenty-Fourth AAAI Conference on Artificial Intelligence. 2010.

[6] Correia F, Alves-Oliveira P, Ribeiro T, et al. A social robot as a card game player[C]//Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference. 2017.

[7] Heinrich J, Silver D. Deep reinforcement learning from self-play in imperfect-information games[J]. arXiv preprint arXiv:1603.01121, 2016.

[8] Lerer A, Hu H, Foerster J, et al. Improving policies via search in cooperative partially observable games[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2020, 34(05): 7187-7194.

[9] Zha D, Lai K H, Cao Y, et al. Rlcard: A toolkit for reinforcement learning in card games[J]. arXiv preprint arXiv:1910.04376, 2019.

[10] Zha D, Xie J, Ma W, et al. Douzero: Mastering doudizhu with self-play deep reinforcement learning[C]//International Conference on Machine Learning. PMLR, 2021: 12333-12344.

[11] Metropolis N, Ulam S. The monte carlo method[J]. Journal of the American statistical association, 1949, 44(247): 335-341.

[12] James F. Monte Carlo theory and practice[J]. Reports on progress in Physics, 1980, 43(9): 1145.

[13] De G C S. Statistical Properties of Monte Carlo Estimates in Reinforcment Learning[J]. 2021.

[14] Bellman R. Dynamic programming[J]. Science, 1966, 153(3731): 34-37.

[15] Browne C B, Powley E, Whitehouse D, et al. A survey of monte carlo tree search methods[J]. IEEE Transactions on Computational Intelligence and AI in games, 2012, 4(1): 1-43.

[16] Liu Y C, Tsuruoka Y. Modification of improved upper confidence bounds for regulating exploration in monte-carlo tree search[J]. Theoretical Computer Science, 2016, 644: 92-105.

[17] Chaslot G M J B, Winands M H M, Herik H J, et al. Progressive strategies for Monte-Carlo tree search[J]. New Mathematics and Natural Computation, 2008, 4(03): 343-357.

[18] Xie F, Liu Z. Backpropagation modification in monte-carlo game tree search[C]//2009 Third International Symposium on Intelligent Information Technology Application. IEEE, 2009, 2: 125-128.

[19] Chaslot G, Bakkes S, Szita I, et al. Monte-carlo tree search: A new framework for game ai[C]//Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. 2008, 4(1): 216-217.

[20] Cowling P I, Ward C D, Powley E J. Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering[J]. IEEE Transactions on Computational Intelligence and AI in Games, 2012, 4(4): 241-257.

[21] Whitehouse D, Powley E J, Cowling P I. Determinization and information set Monte Carlo tree search for the card game Dou Di Zhu[C]//2011 IEEE Conference on Computational Intelligence and Games (CIG'11). IEEE, 2011: 87-94.

[22] Goodman J. Re-determinizing MCTS in Hanabi[C]//2019 IEEE Conference on Games (CoG). IEEE, 2019: 1-8.

[23] Bard N, Foerster J N, Chandar S, et al. The hanabi challenge: A new frontier for ai research[J]. Artificial Intelligence, 2020, 280: 103216.

[24] Eger M, Martens C, Córdoba M A. An intentional ai for hanabi[C]//2017 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 2017: 68-75.

[25] Wulff E, Girone M, Pata J. Hyperparameter optimization of data-driven AI models on HPC systems[J]. arXiv preprint arXiv:2203.01112, 2022.

[26] Osawa H. Solving hanabi: Estimating hands by opponent's actions in cooperative game with incomplete information[C]//Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.