Third Year : Computer Science - Cognitive Engineering - AI

Semester Project Report

2019/2020

Hanabi Project

26/01/2020

Authors:
RABII Younès
EL KHAMSI Achraf
SAINDON Jean-Marie

Supervisors:
FIJALKOW Nathanaël
LAGARDE Guillaume

Acknowledgement

In this section, we wanted to thank our supervisors Nathanaël Fijalkow and Guillaume Lagarde. We had too few opportunities to meet with them during the time of the project but each time we met, our discussions were always interesting and productive. We are a bit saddened to not have more time to push the project further with them and we regret that this month of January was flooded of work on our side but we really appreciated to work with them and we wish them the best for the future in their researches, projects and lives.

Semester Project

The Hanabi project is a semester project that has been launched in the AI section of ENSEIRB-MATMECA / ENSC. We had the opportunity to participate to this project centered on the game of Hanabi that has been recently pushed by Deepmind as a new challenge in the AI domain. The goal of this project was simply to take on that subject and to try to create an Hanabi AI.

Abstract

In this report, we focus on the three main parts of our work during the project. First, we focus on the comprehension of the context, the game itself and the challenges that it brought to AI. Then, we present the environment in which we were invited to work and the information we gathered to help ourselves understanding better and faster the mechanisms. Finally, we finish the report with the heart of our work, which is our conception and implementation of our own Knowledge Representation. It aimed to be more complete than the base knowledge provided by the environment because of the deduction system we imagined. For us, creating a good Knowledge Representation was an important and necessary set up to train our agent with reinforcement learning in the future.

Contents

1 The Multi-Agent issue in AI

Throughout human societies, people engage in a wide range of activities with a diversity of other people. These multi-agent interactions are integral to almost everything nowadays. With such complex multi-agent interactions playing a paramount role in human lives, it is essential for artificially intelligent agents to be capable of cooperating effectively with other agents, particularly humans.

The challenge for an intelligent agent is to be able to understand how the other agents behave and respond appropriately. The problem is that the behaviour policies of other agents can be stochastic, dynamically changing, or dependent on hidden information. So it's hard to coordinate between them.

The general aim of this project is to develop novel techniques capable of imbuing artificial agents with a theory of mind, which is the ability to attribute mental states — beliefs, intents, desires, emotions, knowledge, etc. — to oneself, and to others, and to understand that others have beliefs, desires, intentions, and perspectives that are different from one's own. To reach this objective, several researchers (from Google [1] and then Facebook [2]) have examined the popular card game Hanabi.

2 The Game Hanabi

The firsts questions that could come to our mind when we think about this decision of pushing that game as a challenge for AI could be: Why the game of Hanabi and not another ? What makes this game special enough to be chosen ? The answer is that researchers were interested in the advantages it offers because: First, it presents the kind of multi-agent challenges that humans solve in cooperative environments and second, it is an imperfect information game. Which means that the player has to take decisions without being aware of all the possibilities the other players and himself have.

Building an Hanabi AI would give us more insight into how to create machines that cooperate with other agents and understand their intentions. We could even replace humans in some monotonous work and improve the performance in the industrial world for example.

2.1 Brief reminder on the rules of Hanabi

Hanabi is a card game for 2 to 5 players. The cards are divided into 5 colors, each color having cards numbered from 1 to 5. The deck is made up of 50 cards, 10 of each color: three 1, two 2, 3, and 4, and one 5. The game's imperfect information arises from each player being unable to see their own cards

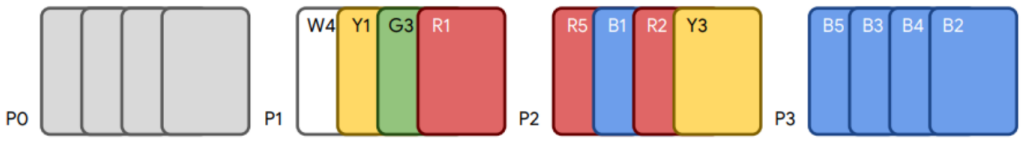

Figure 1: Vision of the game for the player P0 (he cannot see his own hand but he sees the ones of his teammates)

The goal of the game is to play cards so as to form five consecutively ordered stacks, one for each colour, beginning with a card of rank 1 and ending with a card of rank 5. To succeed, players must coordinate to efficiently reveal information to their teammates about the ranks and colors of their cards. By knowing as much as possible about their hand, the players will know whether they should play a card or not. Playing a card is successful if the card is the next in the sequence of its colour to be played.

The players can only communicate through grounded hint actions that point out all of a player's cards of a chosen rank or colour. Importantly, performing a hint action consumes the limited resource of information tokens (8), making it impossible to fully resolve each player's uncertainty about the cards they hold.

Finally, the team has 3 life tokens in case of misplay of a card and if all the cards in the deck have been drawn, a final turn takes place and then the game ends with a score equal to the sum of the cards played during the game (0 if all the life tokens are lost).

To recapitulate, on their turn, players have three possible actions:

- Giving a hint to another player (costs 1 clue token). A clue consists in either indicating all cards with a specific number or a specific color
- Discarding a card (earns 1 clue token) (then drawing another card)

- Playing a card (then drawing another card). The card played must either be a "rank 1" card (to begin a new stack), or be 1 rank higher than the last card played of the same color. If the card played does not meet these conditions, this card is discarded and the team loses a life token. After 3 life tokens lost, the game ends with a score of 0.

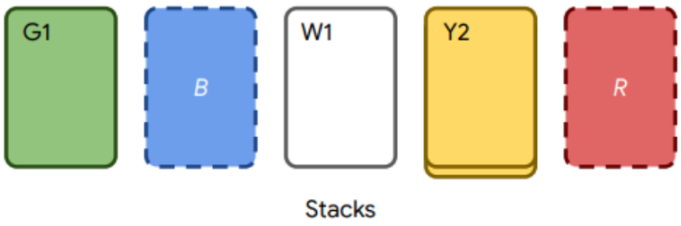

Figure 2: Vision of the game for the player P0 (he cannot see his own hand but he sees the ones of his teammates)

2.2 The Challenges

What is interesting about the game Hanabi is that it is different from other games where computers have reached super-human skills, like chess and go.

Hanabi's cooperative nature complicates the question of what kind of policy practitioners should seek. The Challenge is to discover a policy for the entire team that has high utility, which can be in some cases hard or impossible to find.

The presence of imperfect information in Hanabi creates another challenging dimension of complexity for AI algorithms. Most search algorithms as MinMax, Monte-Carlo or most reinforcement learning techniques will not work for the game Hanabi since they require a complete knowledge of the game state.

Lastly, Hanabi's communication protocol is limited due to the hint tokens. This will force our agent to communicate as efficiently as possible so as to maximize the score.

The combination of cooperation, imperfect information, and limited communication make Hanabi ideal if we aim to make our agents capable of reasoning about others and understanding their intentions and desires.

3 Work environment

At the begining of the project, we read Deepmind's paper on the Hanabi challenge [1] and decided to work with the environment they designed to play Hanabi and train agents.

3.1 Game loop and Structures

The game environment is the part that represents the game board (It is provided by Deepmind in a GitHub repository). Each turn, the environment apply the action provided by an agent on himself (auto-update) so that it can be used by the next players (agents). The agent, on its side, uses the current state of the environment to choose its next action.

The general loop of the game follows the structure (simplified):

```
Env = Environment.create(Configuration)
Players = ListOfAgents()
While not Env.gameFinished() :
    activePlayer = Env.nextPlayer()
    observation = Env.observation(activePlayer)
    move = activePlayer.act(observation)
    Env.applyMove(move)
```

In order to organize ourselves, we decided to examine the code of pyhanabi.py and to create our own tables of structures such as the **configuration** one (a python dictionary):

Parameters	Type / Value	Description
colors	int in [2 - 5]	Number of different colors
ranks	int in [2 - 5]	Maximal rank of cards
players	int in [2 - 5]	Number of players
hand_size	int in [4 - 5]	Number of cards per hand
max_information_tokens	int >= 0	Number of clues at the beginning
max_life_tokens	int >= 0	Number of lifes at the beginning
seed	int	Seed of the random module
random_start_player	bool	First player chosen randomly

Structure of the **observation** object (a *pyhanabi.py* class):

Methods ()	Return Type / Value	Description
`current_player`	int	Id of the player
`current_player_offset`	int >= 0	Offset of the current player
`deck_size`	int	Number of cards remaining
`discard_pile`	[] of cards	List of discarded cards
`fireworks`	`{'B': 0,'G': 0,'R': 0,'W': 0,'Y': 0}`	State of the game
`information_tokens`	int	Number of hint tokens remaining
`legal_moves`	[] of moves	List of possible moves
`last_moves`	[] of history items (move)	Move-list of the last turn
`life_tokens`	int	Number of life tokens remaining
`observed_hands`	[] of [] cards	List of other player's cards
`num_players`	int	Number of players

3.2 Structure of an Agent

As we intended to work on agents, we did the same type of examination on this structure. The base of an Agent :

- Inherit from the *Agent* class (in the file *rl_env.py*)
- Has an *__init__* method
- Has an *act* method that is called at each turn and that returns the move chosen by the agent.
- Has a *reset* method

Parameters and return of the methods :

- The *__init__* method takes the configuration structure as parameter
- At each turn, the method *act* of the agent is called with the observation generated for it as parameter.
- A card is represented by a python dictionary :

Key	Value
`color`	`'R' 'Y' 'B' 'G' 'W'` or None
`rank`	1 2 3 4 5 or -1

- Finally, a move is represented by a dictionary with 4 formats possible :

Key	Value	Description
`action_type`	`'PLAY'`	Action of playing a card
`card_index`	`int`	Id of the card
`action_type`	`'DISCARD'`	Action of discarding a card
`card_index`	`int`	Id of the card
`action_type`	`'REVEAL_COLOR'`	To give a color hint
`color`	`'R' 'Y' 'B' 'G' 'W'`	Color to indicate
`target_offset`	`int >= 0`	Offset with the player concerned
`action_type`	`'REVEAL_RANK'`	To give a number hint
`card_index`	`int`	Number to indicate
`target_offset`	`int >= 0`	Offset with the player concerned

4 Modelling Knowledge in Hanabi

During our meetings with our supervisors, we rapidly stated that the knowledge representation was a critical issue when playing Hanabi. Indeed, in order to play or discard cards effectively, we need to have as much information as possible on our own hand. However, if we want to give relevant hints, we need to estimate how much information our teammates have on their own hands. In other words, an agent not only requires a good **self-knowledge** model for itself, but also has to estimate the self-knowledge of its partners. In the following section, we design a structure that is abstract enough not only to represent the knowledge we have on our cards, but can also be used to approximate other players self-knowledge. This structure is updated through the course of the game, and use every available information to deduce knowledge that wasn't explicitly given.

4.1 Knowledge Representation

The knowledge about our hand is represented by two structures:

- Two probability vectors for each card in our hand
- A table where we store the number of hidden cards

4.1.1 Probability Vectors

Color probability vector

A probability vector, each value being the probability for this card to be a specific color.

$$P_{color} = \begin{bmatrix} p_{Red} \\ p_{Blue} \\ \vdots \\ p_{Green} \end{bmatrix}$$

Rank probability vector

A probability vector, each value being the probability for this card to be a specific rank.

$$P_{rank} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_5 \end{bmatrix}$$

Joint Probability Matrix

A matrix containing the joint probability for each combination of rank and color.

$$P_{rank,color} = P_{rank} * P_{color}^{T} = \begin{bmatrix} p_{1,Red} & p_{1,Blue} & \cdots & p_{1,Green} \\ p_{2,Red} & p_{2,Blue} & \cdots & p_{1,Green} \\ \vdots & \vdots & \ldots & \vdots \\ p_{5,Red} & p_{5,Blue} & \cdots & p_{5,Green} \end{bmatrix}$$

4.1.2 Hidden Cards Tables

These tables keep count of the cards that are not yet revealed to us. We keep these tables updated by counting the number of revealed cards on the game, which includes:

- The visible information:
  - Stacked Cards (fireworks)
  - Discard Pile
  - Other player's hands
- Cards in our hand, whose rank or color is certain

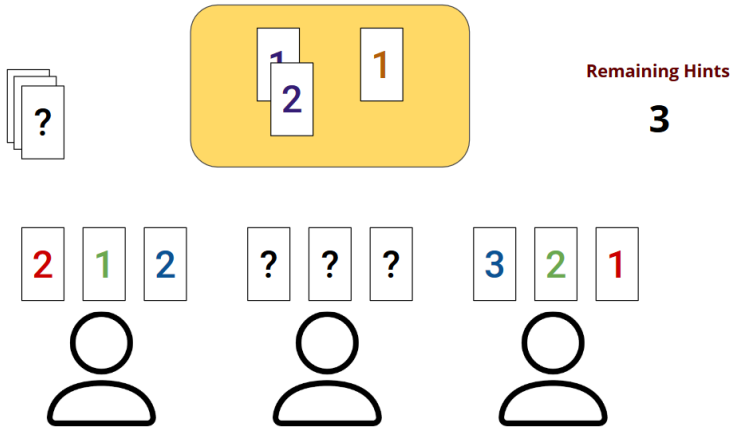

Figure 3: *Typical mid-game board of Hanabi. Player in the middle cannot see its own cards, but can see every other card not in the deck.*

The premise is that we will count the number of revealed cards for each rank and color, and subtract it from the total number of cards for each rank and color.

Remaining Hidden Cards - Color

Table keeping count of the number of cards that are hidden for the player for each color.

$$H_{color} = \begin{bmatrix} h_{Red} \\ h_{Blue} \\ \vdots \\ h_{Green} \end{bmatrix}$$

Remaining Hidden Cards - Rank

Table keeping count of the number of cards that are hidden for the player for each rank.

$$H_{rank} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_5 \end{bmatrix}$$

4.2 Initialisation

Hidden cards tables are initialised by putting the maximum number of each rank and color in the corresponding value. This is solely dependent on the initial configuration of the game.

Probability vectors are calculated by normalising the corresponding hidden cards tables so their sum is 1.

$$\forall c \in E_{Colors} = \{Ensemble\ of\ colors\} : p_c = \frac{h_c}{\sum_{i \in E_{Colors}} h_i}$$

4.3 Updating Knowledge

In section 3.1, we've described the game loop. In each round, we give the hand to the active player in order for them to "act", which means to deduce as much information as possible with their observation so as to make the best move. This chosen move is then applied in the game, which alter the game state, and so on. In the next sections, we will see how to update these sections based on the observation of the player.

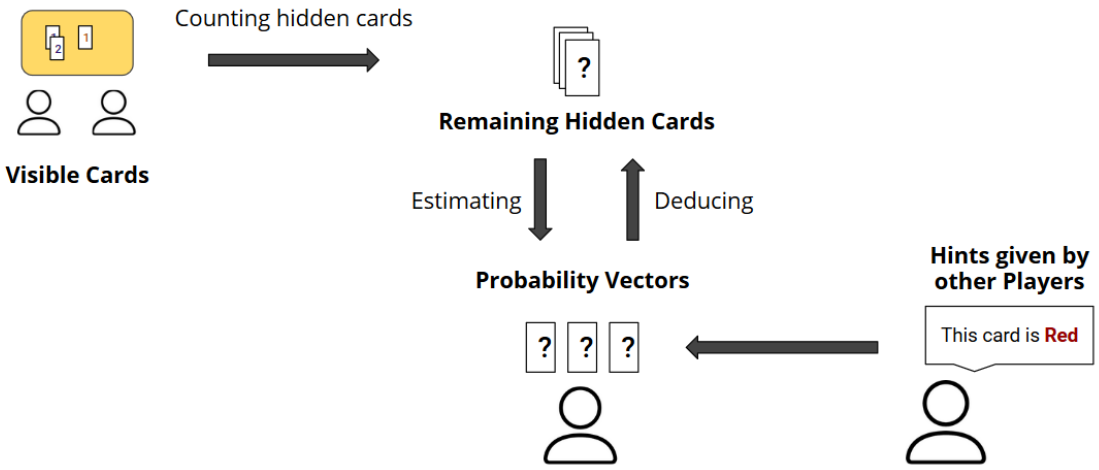

Figure 4: *Knowledge flow in the model. Fed by the Visible Cards and Hints, then enters a cycle of deductions and estimations, until every bit of knowledge is propagated through the model.*

4.3.1 Updating Probability Vectors

As we have seen is section 4.1.1, the knowledge about each card is represented by two probability vectors: one for the color of the card, and one for its rank. There is two ways to update these probability vectors:

1. Update based on the hints given since the previous move of the active player:

   In this case, we iterate through the moves given since the player last acted. If one of the moves is a hint targeted to the player, The hint is applied to his probability vectors. The hint affect directly the targeted cards in our hands, and affect indirectly the remaining cards.
   Example: if our probability vector for the rank of the first card in our hand is [0.5, 0.25, 0.25] and that a teammate gives us a hint about the rank of this card (this card being of rank 1). The probability vector will become [1, 0, 0]. This hint will also affect the other cards (these cards being not of rank 1). Their probability vectors will be of the form [0, x, y].

2. Update based on the hidden cards tables:

   We will update the probability vectors of our cards according to the number of hidden cards in the game. We just update the probability vectors that are not specific, which means the probability vectors that don't have 1 in a single component and 0 in all others.

   Example: if the array of the hidden ranks is [2, 1, 0]:

- if the probability vector corresponding to the rank of a card is [0.33, 0.33, 0.33], it will become [0.66, 0.33, 0].
- if the probability vector corresponding to the rank of a card is [0, 1, 0], it will not change.

4.3.2 Updating the Hidden Cards Table

As we have seen is section 4.1.2, we keep count of the cards that are not yet revealed to us for each color and rank. The premise behind updating these tables is that we will count the number of revealed cards for each rank and color, and subtract it from the total number of cards for each color or rank.

4.3.3 The update loop

The update of our knowledge will follow the structure below:

```
update_proba_vectors_v1(obs)
    update_hidden_cards(obs)
        #update_proba_vectors_v2(obs)
            #update_hidden_cards(obs)
                #update_proba_vectors_v2(obs)
                    # ... (until the information
                       propagates completely)
```

Listing 1: Pseudo-code for the update loop

The function `update_proba_vectors_v1` (resp `update_proba_vectors_v2`) refers to the first way (resp the second way) of updating the probability vectors of our cards. The function `update_hidden_cards` update the hidden cards table according to the revealed cards in the game.

First, we call the first version of probability vectors update (we apply the hints targeted to us during the last moves. This way, we modify the probability vectors of our cards and we stock the hints related to them directly in their structures.

Second, we call the function to update the hidden cards table. If an update of these tables has occurred, this function call the second version of updating the probability vectors.

The second version of updating the probability vectors may as well call the update function of the hidden cards table if a card's color or rank was revealed.

This cycle continues until all the effects of the moves of the other players propagates through our knowledge structure.

5 Implementation & Future Work

Our Knowledge Representation was implemented in the Hanabi Environment provided by Deepmind. The commented code is available in our **GitHub repository**, it features:

- a script to easily instantiate and play a game with different agents in `game.py`
- an simple agent modeling and updating its self-knowledge the way we designed it, in `red_ranger.py`
- a separate class that can be used with a custom agent to model and update its self-knowledge, in `Knowledge.py`

We intend to test the effectiveness of our knowledge modeling by comparing the performance of agents playing with and without it. First experiments showed that even simple agents can deduce the rank and color of some cards without a single -direct or indirect- hint We conjecture that agents have a performance when using our knowledge modeling that's always greater or equal than when they don't use one.

We also hypothesize that this representation is fit for agents using Reinforcement Learning, as it is complete, compact and every uncorrelated variable is separated. A way to explore this, would be to see if RL agents learn to play Hanabi faster or diffrently with this way of representing knowledge, versus when they use the basic information provided by the environment.

References

[1] Bard Nolan and al. The hanabi challenge: A new frontier for ai research. feb 2019.

[2] Lerer Adam and al. Improving policies via search in cooperative partially observable games. dec 2019.