M2 IASD - Monte Carlo Tree Search for Games - Applying MCTS to French Belote, a Trick-Taking Card Game with Imperfect Information

Alexandre Olech

April 2025

Contents

1 Abstract - Very Short Full Summary

We implement 3 MCTS methods to solve French Belote :

- **Cheating UCT** (UCT in the perfect information setting).
- **Determinization/PIMC** (picking the best action in most possible worlds based on UCT). We implemented a sequential processing and a **parallel processing version**, which is three times faster to pick a move, given the same configuration.
- **Single Observer Information Set Monte-Carlo (SO-ISMCTS)** : using one search tree, one different world determinization for each trial, and counting availabilities of children instead of parent visits in the UCB1 formula.

We experiment with two action selection methods for Determinization/PIMC :

- **Max Times Best:** choose the action which is the best in most worlds.
- **Max Total Reward:** choose the action with the highest total reward when summing over possible worlds (this is the approach of Ginsberg et al. [4]).

We present results for two versions of Belote :

- **A simplified version with 12 cards only**. In this setting, our imperfect information algorithms explore all possible states once.
- **The full 32-cards game**. In this setting, our determinization UCT algorithms use **sampling methods** instead of exhaustively trying each possible world.

We evaluate the performance of each algorithm by comparing the win rates and average scores for 30 games (60 games for the 12 cards version) played by a team picking moves based on the algorithm vs. a team picking random moves. Algorithms were all tuned with their best observed configurations, with the condition that they choose a move in less than 25 seconds.

The results show a superior performance of SO-ISMCTS (with a performance almost as good as Cheating UCT) in the standard 32 cards game, compared to determinization, which is likely due to a better handling of strategy fusion. Interestingly, SO-ISMCTS becomes much more competitive when moving from the simplified 12 cards game to the full 32 cards game, which demonstrates a better scaling to exponential increase of nodes and possible worlds, compared to standard determinization / PIMC approaches. As we observed in the 12 cards setting, Determinization might still shine in some scenarios where shallow tree search is sufficient and parallel processing is highly available.

2 Introduction

2.1 Topic

The goal of this project is to study and implement MCTS methods for trick-taking imperfect information card games. Such games, like Tarot, Bridge or Belote, typically involve planning

and sequential decision-making, making them well suited for MCTS methods. However, due to the complexity of card games, the number of possible states can be extremely large, leading to search trees with very large number of nodes. The imperfectness of the information is also a challenging aspect to tackle, with interesting design choices for the search tree and handling of possible worlds.

2.2 Contributions

- This topic is similar to Kanupriya Jain's, another student of the course, who also applied MCTS for a card game with imperfect information. Together, we initially formed a team. We had many fruitful conversations that helped us define our approaches. However, our two games were very different and our implementation and experiments were done completely independently, so it made more sense to send two individual reports separately.

- All the implementations and experiments presented in this report were conducted by Alexandre Olech alone.

2.3 General approach

To solve imperfect information card games, we relied on the basic formulation of Perfect Information Monte-Carlo (PIMC), as presented in the slides of the course and summed up in algorithm 1 :

Algorithm 1 General formulation of determinization / PIMC

```
1: Given the set of all legal moves
2: for all possible worlds do
3:     Exactly solve the world
4: end for
5: Play the move winning in the most worlds
```

Approaches based on this algorithm were successfully applied to games like Bridge by Ginsberg [4]. This is also referred to as determinization in the literature [3], because it consists in solving the game by solving all the determined states of the information set. However, this method can be seen as "averaging over clairvoyance" [5] and is not without its flaws. The three main issues are :

- Inefficient handling of duplicated nodes across possible worlds.

- Strategy fusion : PIMC is able to adapt its strategy to each world because it has perfect information, but this is not true in the real world where the player cannot adapt its strategy.

- Non-locality : some determinizations may be extremely unlikely, due to the ability of other players to prevent the game from reaching the corresponding states.

Our goal was to implement this approach from scratch with Python, using UCT to solve the perfect information game. While other approaches are often preferred to UCT for this type of game, we chose to experiment with a "from scratch" UCT implementation in order to get a solid grasp on the core principles and practices of Monte-Carlo Tree Search algorithms.

Then, to tackle two core issues of determinization (strategy fusion and duplicated exploration), we experimented with an Information Set Monte-Carlo approach presented in [3].

We implemented and experimented with two settings :

- **A simplified version of Belote (12 cards)**, with exhaustive search nodes and possible worlds exploration.
- **The full-deck game (32 cards)**, where we had to adapt the method with sampling strategies.

2.4 Outline of this document

Section 2 presents the chosen game, its rules and our implementation of the game. Section 3 presents the MCTS algorithms we studied and implemented. Section 4 presents our main results and their interpretation.

3 Game

3.1 Setup and rules

The game follows the official rules of French Belote. It consists of two teams of 2 players, with players of the same team facing each other. However, for simplicity, the game only consists of one round, without bidding phase, and the cards are randomly distributed between the players at the start. By convention, we call "attack" the team of the first player to play in the beginning (positions 0 and 2) and defense the other team (positions 1 and 3).

We implemented two versions of the game, which correspond to two classes of cards :

- **A simplified version with 12 cards only**. It allowed us to experiment with our algorithms in a simple setup, with reduced nodes and possible worlds.
- **The full 32-cards game**, on which we applied the same algorithms, but with sampling methods.

After some experimentation on other formats with different number of players, teams and cards, we arrived at the 12 cards simplified setting, which we find interesting because it allows for some teamwork (ex : "giving" a good card to the teammate) and planning, while keeping the number of possible states relatively low. For example, in the perfect information setting, for a fixed set of player hands, at the beginning of the game, the maximum number of state/nodes in the search tree can be roughly bounded as follows :

$$n_{nodes} \leq \prod_{k=0}^{n_t-1} (n_c - k)^{n_p}$$

Where n_t is the number of turns, n_p is the number of players, and n_c is the number of cards in the hand of each player at the beginning of the game. This yields :

$$n_{nodes} \leq 3^4 \times 2^4 \times 1^4 = 1296$$

Which is relatively small for a typical application of MCTS.

For the 32 cards version, the number of nodes is exponentially larger, which means that we will usually not search the full tree of the game. We study both the 12 cards and 32 cards settings in our experiments.

3.2 Implementation

3.2.1 Three main classes : card, player, game

Our implementation of the game relies on three Python classes :

- The **Card class** represents belote cards, with four attributes :
  - id (ex: 0)
  - color (ex: pique)
  - label (ex: V)
  - value (ex: 20)
- The **Player class** represents a belote player. It has four attributes :
  - id (ex: 0)
  - hand : a list of belote cards
  - team : "attack" or "defense"
  - first_to_play : boolean variable indicating which player plays first for a given trick

  It also has three main methods :
  - get_legal_moves : returns the list of playable belote cards
  - naive_move : plays a random legal card
  - get_all_possible_states : returns a list where each element is a list of cards corresponding to a distribution of cards id for the other players, in the corresponding possible state
- The **Game class:** represents the main game. It has five main attributes :
  - players : the list of belote players
  - attack_tricks : tricks won by the attack in the current round
  - defense_tricks : tricks won by the defense in the current round
  - cards_on_table : belote cards currently on the table

- table_color : the color of the table (ex: carreau)

It also has 9 main methods :

- get_next_to_play_idx : returns the id of the player whose turn it is to make a move.
- reset_game : resets the game
- distribute_cards : distributes random cards
- give_trick_to_win_team : gives the trick to the winning team at the end of the turn
- reset_table : resets the table at the end of the round
- step : moves the game one step based on an action id for the current player
- random_step : moves the game one step based on a random action from the current player
- playout : move the game with random steps until it is finished
- get_state : returns the hash of the one-hot encoded current state

4 MCTS Algorithms

4.1 Building the tree and handling states

We represent the search tree using only nested nodes that reference their children. Concerning states, while we initially built integer representations based on one-hot vector and hashing, we found that we didn't need to explicitly reference states. We only store the information about number of visits, reward, incoming action and actions tried at the node level (and the availabilities as well for the SO-ISMCTS approach).

4.2 Perfect Information Setting : Cheating UCT

By perfect information setting, we refer to the situation where the player knows with certainty the content of the other players' hands. By "cheating UCT", we denote the application of UCT to this setting.

We chose to implement cheating UCT for two reasons :

- **Comparison benchmark vs. imperfect information approaches:** The performance of cheating UCT should be an upper bound on the performance of the imperfect information approaches. Therefore, the goal for an imperfect information UCT approach should be to be as close as possible to the performance of cheating UCT (in its best configuration).
- **Aggregating cheating UCT in all possible worlds:** Even if the agent does not have full knowledge of the state, he has access to the information set which contains all the possible states. Therefore, it is possible to use a perfect information algorithm such as cheating UCT on all of these states, aggregate the performance and keep the best action over all possible states.

In this setup, we represent the game states exactly as presented in the previous section, taking into account the hand of all players, tricks won by attack and defense, cards on the table and index of the first to play player.

Then, we apply UCT, following very closely the pseudocode presented by the survey on MCTS methods by Browne et al. [1]. The pseudocode is presented in the annex 2. Our UCT.py script is based on this presentation of UCT.

Our best_child, expand, tree_policy and backup functions are closely aligned with this reference. In our UCT_search function, for every tree descent, we chose to create (and replace) a simulated game class which is an independent copy of the true game class. When copying the game class, it was crucial to use the deepcopy method from the copy library. With the basic Python copy, modifying the copy would have the unintended consequence of modifying the original game class.

Our default policy function uses the playout method of the game class in order to update the simulated game with random steps until completion.

Our node class is a basic Python class with only attributes for the state, total reward, number of visits, children, actions tried, incoming action (action which led to the corresponding state) and team.

The state of the node is computed as the hash of the binary state vector of the corresponding game class.

The "team" of the node corresponds to the team of the player which made the incoming action. This is what allows us to attribute rewards accordingly after a tree descent.

For each step of tree descent, we initialize a search list to which we append selected or expanded nodes. At the end of the playout, we backpropagate the rewards of each team to each node of the search list, based on their team attribute.

4.3 Imperfect Information Setting : All worlds UCT

By imperfect information setting, we mean the situation where the player knows with certainty his own cards, the cards on the table, and the cards already won by both teams.

We implemented a determinization approach inspired from algorithm 1, which we adapted to UCT (while the approach in Cazenave et al. [2] uses a Double Dummy Solver) with the last function (uct_all_possible_worlds) of the UCT.py script which uses the get_all_possible_states method of the player class, presented in the previous section.

To apply UCT in the imperfect information setting, we use the following method : using only the information available to the player, for each possible state, we create a simulated version of the game (again, using a deep copy of the game class), to which we apply our perfect information UCT algorithm.

Running UCT on each possible world can be computationally expensive as it multiplies the running time by the number of possible worlds (without using parallel processing), which can justify the use of sampling methods instead of solving each state once. In the simplified 12 cards game setup, the number of possible worlds was still relatively limited : it can be computed as follows :

$$n_{\text{worlds}} = \prod_{i=0}^{n_p-2} \binom{n_c \times (n_p - 1 - i)}{n_c}$$

Where n_p is the number of players and n_c is the number of cards by player in the beginning. In the simplified 12 cards / 4 players setup, this yields :

$$n_{\text{worlds}} = \binom{9}{3} \times \binom{6}{3} \times \binom{3}{3} = 1680$$

Since this number of possible worlds is relatively small, for this simplified game, we didn't use sampling methods and solve the game for each possible world at every MCTS move. For the full 32 cards game, we had to use sampling methods as we detail below.

4.4 Two Aggregation Strategies for Determinization/PIMC

After this process, multiple choices are possible to choose the best action :

- **Max Times Best:** choose the action which is the best in most worlds.
- **Max Total Reward:** choose the action with the highest total reward when summing over possible worlds (this is the approach of Ginsberg et al. [4]).

4.5 Enhancing Determinization/PIMC with parallel processing

In determinization, the computational budget can be expensive due to searching a great amount of possible states. However, since the explorations of all possible states in the information set are done independently, it is possible to run the search associated to each possible state in a different processing unit, and aggregate values at the end.

We thus implemented modified functions for the determinization approaches (see UCT.py), in order to handle parallel processing, using the `multiprocessing` library and the `ProcessPoolExecutor` tool from the `concurrent` library.

This considerably reduced the time needed to choose a move in the main game, with moves being **three times faster on our machine** given the same parameters.

4.6 Single Observer Information Set Monte-Carlo (SO-ISMCTS)

The implementation of SO-ISMCTS can be found under ISMCTS.py.

Single Observer Information Set Monte-Carlo [3] is a promising approach to solve the strategy fusion and inefficient budget allocation issues, compared to determinization/PIMC. Instead of searching the mini-max tree for each produced by each different determinization, we build a single tree where the nodes represent information sets rather than states. Then, we use a different determinization for each tree descent.

Another key modification comes from the use of Subset-Armed Bandits instead of traditional Multi-Armed Bandits. This is due to the fact that, in one determinization, at a given node, all children nodes are not necessarily available. The standard UCB1 algorithm calculates the score of a node as :

$$\bar{X} + c\sqrt{\frac{\ln n}{n_j}}$$

where $\bar{X}$ is the average reward of the simulations passing through the node j, n is the number of times parent j has been visited by the algorithm, and n_j is the number of times the node j was selected from its parent. In the Subset-Armed Bandits setting, to avoid biasing the exploration, we count availabilities of the children, instead of visits to the father node, which means defining n as the number of trials in which the parent was visited *and node j was available*, instead of just counting the number of trials in which the parent was visited.

4.7 Sampling Methods to solve the full 32 cards game

To solve the full 32 cards game, we adapted the imperfect information methods such that they sample possible worlds instead of exhaustively trying all possible worlds. Instead of pre-computing a list of all possible states, we sample a list of $n_{samples}$ possible states and sample from this list to perform each perfect information UCT or SO-ISMCTS trial. If the number of trials is higher than the number of samples (which is the case for SO-ISMCTS), we randomly sample a new list of possible states every time the number of trials exceeds the number of samples (and reset the count to zero).

5 Results and Interpretation

5.1 Evaluation Method

All the evaluations are implemented in the Evaluate.py script.

As presented in section 2, the configuration of the game used for our experiments consists of two teams of two players each, with 12 cards in total. Since the number of cards is very limited, results depend heavily on the initial cards. In a lot of cases, a team with a strategy can lose to a random strategy due to imbalance in the initial hands. Also, we observed that naive heuristic strategies like always playing the strongest card were worse than a random strategy in this configuration. We made two conclusions from these observations :

- Performance against random strategy is a decent baseline evaluation method.
- But it requires averaging win rates over many games to be meaningful.

Over a number of games n_g, we make two players of one team take MCTS moves, and the two other players make random moves. To avoid the bias of having the MCTS team being always first (positions 0 and 2 in the table) or second (positions 1 and 3) to play in the first turn, we make the MCTS play first in half the games and second in the other half.

In this configuration of the game, the maximum number of points is 80, and we consider that a team wins one game when it gets more than 40 points (there is no bidding like in typical Tarot or Belote, and cards were distributed randomly at the start).

If both teams reach a score of 40, we consider the outcome as a draw. Thus, we also report draw rates, because it's hard to interpret win rates without taking either draws (or the other team's win rate) into account.

Algorithms were all tuned with their best observed configurations, with the condition that they choose a move in less than 25 seconds.

5.2 Main Results

5.2.1 Simplified Belote with 12 Cards

Table 1 shows the average performance over 60 games of the different algorithms presented in section 4, with parameters used in the caption. We make the following observations :

- Over 60 games, Cheating UCT, All worlds UCT (with "max times best" action selection) and SO-ISMCTS teams performed better than their random opponent, with win rates above 50%.
- As expected, the imperfect information approaches (All Worlds UCT, SO-ISMCTS) perform worse than their perfect information counterpart (Cheating UCT). All worlds UCT with "max times best" action selection manages to be the closest to the performance of cheating UCT, losing only 3% win rate compared to cheating UCT.
- Interestingly, the "max total reward" action selection method performs worse than the "max times best" method, which is not better than random opponents. While we observed better performance of the "max total reward" method in a previous Tarot setting (with different rules for playing trump), we observe the reverse relationship in this present experiment for Belote, which illustrates the importance of tuning the aggregation method to the specific problem.
- Imperfect information methods take more than 10 times the time needed with a perfect information method, which illustrates the cost of imperfect information compared to perfect information.
- Although the SO-ISMCTS was expected to have a better time efficiency, due to sharing the exploration information of all nodes in a single tree and avoiding redundancy, this is not what we observe in this simplified 12 cards setting, where SO-ISMCTS is slower than All Worlds UCT. This is likely due to the fact that we were able to efficiently use parallel processing for All Worlds UCT, turning the independence of the search trees into a computational advantage. With less time and better win rate, All Worlds UCT is much more efficient since it manages to do considerably more trials than SO-ISMCTS (400 trials $\times$1680 possible worlds = 672000 trials for All Worlds UCT, vs. only 50000 trials in a single search tree).

Algorithm	Average Score	Std	Win Rate	Draw Rate	Time per Move
Cheating UCT	50.27/80	25.83	66.67 %	0.0%	0.14s
All Worlds UCT (action: max total reward)	39.8/80	25.34	45.0%	0.0%	6.81s
All Worlds UCT (action: max times best)	46.0/80	27.18	63.33%	0.0%	6.91s
SO-ISMCTS	42.37/80	29.0	53.33%	0.0%	9.85 s

Table 1: 12 cards Belote results : performance of the different algorithms we implemented vs. random strategy. For each algorithm, the performance was averaged over 60 games, where one team picks moves based on the algorithm, and the other team picks random legal moves. Cheating UCT has a budget of 1000 tree descents per move, while All Worlds UCT uses only 400 tree descents per move, but does so over all 1680 possible worlds, hence the increased computational cost. SO-ISMCTS uses 50 000 tree descents per move, in a single search tree. Time per Move measures the time it takes for a player of the non-random team to play one non-final move.

5.2.2 Standard Belote with 32 Cards

Table 2 shows the average performance over 30 games of the different algorithms for the standard Belote with 32 cards, where sampling methods were used to consider possible worlds (instead of considering all worlds like we did in the 12 cards version). Parameters used are indicated in the caption. We make the following observations :

- As expected, Cheating UCT has the best score on average. However, Sampled SO-ISMCTS comes very close, and actually has a superior Win Rate over 30 games. This suggests that SO-ISMCTS is on par with Cheating UCT versus random strategy opponents, and therefore performs quite well.
- The two versions of standard determinization (Sampled All Worlds UCT) have a lower performance (given more time), which is consistent with the results of [3].
- When compared to the 12 cards game results, these results suggest that the performance hierarchy between determinization and SO-ISMCTS inverts when moving from the 12 cards game to the actual 32 cards game. This suggests that the advantages of SO-ISMCTS (better handling of strategy fusion and more efficient search) really prove their worth when the search tree has many nodes ($>> 10^3$) and the number of possible states is very high ($>> 10^3$). In other words, SO-ISMCTS demonstrates a better scaling to exponential increase of nodes and possible worlds, compared to standard determinization / PIMC approaches.

Algorithm	Average Score	Std	Win Rate	Draw Rate	Time per Move
Cheating UCT	98.47/152	37.29	76.67%	0.0%	13.0 s
Sampled All Worlds UCT (action: max total reward)	97.9/152	31.75	70.0 %	3.33%	20.23s
Sampled All Worlds UCT (action: max times best)	95.73/152	35.46	70.0%	3.33%	21.2 s
Sampled SO-ISMCTS	93.2/152	35.37	80.0%	0.0%	11.07s

Table 2: 32 cards Belote results : performance of the different algorithms we implemented vs. random strategy. For each algorithm, the performance was averaged over 30 games, where one team picks moves based on the algorithm, and the other team picks random legal moves. Cheating UCT uses 30000 trials. Both versions of Sampled All Worlds UCT use 200 trials for each of 1000 sampled possible worlds. Sampled SO-ISMCTS uses 3000 trials, each corresponding to a randomly sampled possible world.

6 Annex

Algorithm 2 The UCT Algorithm

```
1: function UCTSEARCH(s_0)
2:     Create root node v_0 with state s_0
3:     while within computational budget do
4:         v_l ← TreePolicy(v_0)
5:         Δ ← DefaultPolicy(s(v_l))
6:         Backup(v_l, Δ)
7:     end while
8:     return a(BestChild(v_0, 0))
9: end function
10:
11: function TreePolicy(v)
12:     while v is non-terminal do
13:         if v is not fully expanded then
14:             return Expand(v)
15:         else
16:             v ← BestChild(v, C_p)
17:         end if
18:     end while
19:     return v
20: end function
21:
22: function Expand(v)
23:     Choose a ∈ untried actions from A(s(v))
24:     Add a new child v' to v
25:       with s(v') = f(s(v), a) and a(v') = a
26:     return v'
27: end function
28:
29: function BestChild(v, c)
30:     return arg max_{v'∈children of v} [Q(v')/N(v') + c sqrt(2 ln N(v)/N(v'))]
31: end function
32:
33: function DefaultPolicy(s)
34:     while s is non-terminal do
35:         Choose a ∈ A(s) uniformly at random
36:         s ← f(s, a)
37:     end while
38:     return reward for state s
39: end function
40:
41: function Backup(v, Δ)
42:     while v ≠ null do
43:         N(v) ← N(v) + 1
44:         Q(v) ← Q(v) + Δ(v, p)
45:         v ← parent of v
46:     end while
47: end function
```

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[2] Tristan Cazenave and Véronique Ventos. The $\alpha\mu$ search algorithm for the game of bridge, 2019.

[3] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[4] M. L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, June 2001.

[5] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 2009.