Information set Monte Carlo Tree Search for Two-players Belote

Thomas Boudier
GSSI
L'Aquila, Italy
Thomas.Boudier@gssi.it

***Abstract*—Monte Carlo Tree Search (MCTS) algorithms have been successfully used to design AI players that can beat the best players in the world in perfect information games as difficult as *Chess* or *Go*. For games with non-perfect information, a variant is Information Set Monte Carlo Tree Search (ISMCTS). We first describe how these algorithms works, then apply a ISMCTS algorithm on the two-players belote game and compare it's performance against random, greedy and human players.**

***Index Terms*—AI, Game Theory, Simulator**

I. Introduction

Monte Carlo Tree Search (MCTS) methods have been introduced in 2007 in [1] as an implementation for the game of *Go* and immediately proved extremely powerful in comparison to other existing algorithms. One of it's main strength is that it is *anytime*: we can stop the algorithm when we want and get a move that is good enough. During competitions, the time allowed for each algorithm to perform it's computation is limited, and prior algorithms sometime required to explore fully the game tree, which is impossible for games with high branching factors and high depth as *Go* or *Chess*. If fully exploring the game tree was not possible, these algorithms instead required a function to evaluate the state of a non-terminal game state, which is also sometime extremely difficult. As MCTS can stop is anytime, it was very successful in games with high branching factor. An other trait of MCTS to mention is that it never prunes a move: if a move seems bad at first look, it never puts to 0 the probability of playing it during the simulations; while previously used algorithms had to rely of such *pruning* methods [2].

Moreover, very little domain knowledge is required to run it, so it is very easy to apply it to a wide variety of games, as well as for real-world applications.

Most work on MCTS algorithms are about games with perfect information games, but less is known about non-perfect information games. Some works [3] [4] [5] [6] [8] suggested to use determinization: before each rollout, the informations that are unknowned to the AI player are sampled uniformly and the algorithm acts as for perfect information MCTS for the rollout. The determinization approach is not without flaws, as we detail in section IV-A. We take interest in this report on a family of algorithms called Information Set Monte Carlo Tree Search (ISMCTS) that aims to solve some of the problems caused by determinization.

In section II we will present the non-perfect information game of belote and it's version for two players. Unlike the games of *Klondike Solitaire* [5], *MTG: The Gathering* [8] and similarly to *Bridge* [6], a game of belote does not end with a winner and a loser but with scores ranging typically from 0 to 162. In more advanced versions of the game, players may make contracts that rewards bonus points at the end game if completed. It is not clear yet if and how MCTS-like methods may help to make the contracts, so we will not take into account these details. For section VI, we will only consider the simplest version of belote where players already have their cards in hand and on the board, and they try to maximize the number of points they get at the end of the game.

II. Belote

A. Four-player belote

Belote is a trick taking game for four players using a 32-cards deck: $\{A, K, ..7\} \times \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$. Similarly to Bridge, players are divided in two teams; North-South and East-West. They play in counter-clockwise order. The first team reaching 1000 points wins. A game consists in a dealing phase, the choice of the trump color, then 8 tricks. At the end of the last rounds, players count the points of the tricks they earned, and add this to their total number of points. They repeat the game until one team reaches 1000 points.

Dealing

One player is designed as the dealer. He gives each player five cards, they turn one card face up. The player directly to his right says if he wants it. If he refuses, the next player express it self. When a player accept a card, the its color becomes the *trump color* for the game. He automatically makes the contract of scoring a minimum of 82 points. If they fail, they will score 0 and their opponents 162. Then, the dealer completes the card of each player until everyone have 8 cards.

Rules

The player to the right of the dealer plays first. He plays one card of his choice, then the player on its right, etc. until every player played one card. Then, the player that played the strongest card wins the trick, put all the cards on his side, and play the first card in the next round.

Non-trump			A	10	K	Q	J	9	8	7
Points	20	14	11	10	4	3	2	0	0	0
Trump	J	9	A	10	K	Q			8	7

Fig. 1. Points per card depending if they belong to the trump suit. This table also gives implicitly the relative strength of the cards: for the trump suit: $J > 9 > A > ...$ and $A > 10 > K > ...$ for the other suits.

The color of the first card played in a round is the *dominant color*. Players must at all time play a card of the dominant color if they have one. When they don't have a card of dominant color, if they have a trump card they must play it. Otherwise they can play anything in their hand. If the dominant color is a trump, players must play a stronger trump card if they have one, otherwise any trump they have. If they don't have a trump, they can play anything else in their hands.

If a player fails intentionally or not to follow a rule and their opponents notice it, the faulty player's team score 0, their opponents 162 and the game is immediately terminated[1].

The winner of the trick is the player that played the strongest card of dominant color if no trump has been played, or the strongest trump otherwise.

Points

Figure 1 gives the distribution of points awarded for each card as well as the relative strength of the cards.

Players also add to the total number of points of the cards in the tricks they earned some bonus: for instance, 10 points for the winner of the last trick; 10 points if they add both the queen and the king of the trump suit, etc. In some version of the game, player can also bet during the dealing phase. See [7] for more details.

Belote is by nature a game of partial information; however through the tricks players makes deduction on what remaining cards their opponents have. For instance suppose ♣ is the trump suit. If the dominant color of a trick is ♡ and the last player of the trick plays J♣, we can make the following deductions: first, they do not have any ♡ card. Second, they probably do not have any other ♣, otherwise they would have used a weaker one. Consequently, it is safe to assume that if we play any card as dominant color in future tricks, they will not be able to do anything. For the rest of this study, we chose to direct our study on a version of belote for two players.

B. Two-players belote

Two-players belote use the same deck of 32 cards and very similar rules. The difference lies in the fact that players have only 6 cards in hand, and 10 cards on the table organized in 5 piles. On each pile, one card is hidden by a visible card. When playing the visible card, the one below is unveiled. At the beginning of the game, each player knows 11 of his cards, 6 of which are invisible to his opponent. During the game, he eventually unveils 5 other cards. In Fig 2

[1]If a team wins all 8 tricks of a game, they score 250 points, so on extremely rare occasions it may be worth it to intentionally break the rules. However, it is an extremely bold play and not a very fair strategy.

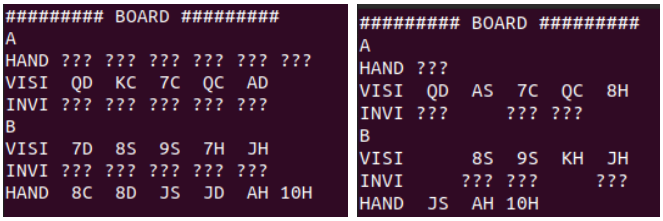

Fig. 2. Interface of two-players belote (a) at the beginning of the game (b) after a few tricks; from the point of view of player B.

The dealing phase is similar to four-players version: the dealer gives 5 hidden cards to each player, then one card is accepted by one of them. Then, the dealer gives an additional card such that each player have 6 cards in hand. Then, he distributed the cards on the board.

The rules and the points works in exactly the same way.

III. Perfect Information Monte Carlo Tree Search

We start by describing the algorithm of Perfect Information Monte-Carlo tree search (PIMCTS). This algorithm is applied on game for where no information is hidden, such as *Go* or *Chess*. The algorithm builds a tree iteratively. Each node correspond to a a game state, and each edge to a move done by a player. Nodes store several informations: number of times we've been traveling through them, sum of the values of the simulations, and the sum of the squares of the values of the simulations. The root node correspond to the state of the game where we call the algorithm, i.e. when the PIMCTS player has to make a decision.

The algorithm repeats the same four steps that we briefly summarize here:

- 1. Selection: starting from the root, selects a child of the current node until we reach a node that has not been fully expanded.
- 2. Expansion: add one children to our current node, corresponding to a move that has not been explored yet.
- 3. Simulation: simulate the game from this new move by using random, legal moves (or more advanced approach).
- 4. Backpropagation: modify the attributes of the nodes from the root to our expanded node according to the simulation results.

When we are out of time, we select a child of the root according to some heuristic. Here are some examples: *max-child* the highest average reward, *robust-child* the most visited, *secure-child* the one with highest lower confidence bound, etc.

The approach of PIMCTS has several advantages over conventional algorithms such has minimax:

- First, it is anytime: we can stop whenever we want and still have an adequate opinion on which move is a good one. This is important for AI-game competition where each player has limited time to play a game.
- For games that may require an high number of rounds before reaching a terminal state, classical algorithms perform poorly without having an evaluation function of a

non-terminal state of the game. For PIMCTS, we do not have this problem as we consider random legal moves during simulation.
- No game-specific knowledge is required to do PIMCTS: we do not need the aid of an expert to produce a strong AI player.

A. Selection

Selection consists in travelling the tree up to a node whose children is not fully expanded. The selection relies heavily on central-limit theorem, that gives us information of the probability of winning of each move. Before the introduction of MCTS algorithms, it was usual to use *progressive pruning*: when the estimated probability of winning a game after a move falls below some threshold, we completely cut the corresponding sub-tree from our tree. This allows a faster execution of the simulation process and may save a lot of memory, but may occult the fact that one move that may look bad may be followed by a killer-move we have not yet explored.

At each node, the algorithm compute the probability u_i of selecting one of the children of our node. In [1], Coulom suggest the following formulae to do the selection process for the game of *Go*:

$$u_i = exp(-2.4\frac{\mu_0 - \mu_i}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}}) + \epsilon_i \tag{1}$$

With ϵ_i being equal to:

$$\epsilon_i = \frac{0.1 + 2^{-i} + a_i}{N} \tag{2}$$

where μ_i is the estimated win probability of move i, N is the number of times we traveled through the current node, moves are ordered as $\mu_0 > \mu_1 > ... > \mu_N$, a_i is 0 or 1[2] and σ_i^2 is the variance of move i. u_i then represents the probability of choosing move i. As you observe, u_i never goes to 0 thanks to ϵ_i. The formula of ϵ_i was determined empirically.

Later, other formulae have been used to compute u_i, the most popular one being the UCT bandit formula:

$$argmax_j\left(\mu_j + 2c\sqrt{\frac{2\ln(N)}{n_j}}\right) \tag{3}$$

where j is a children of the current node, n_j is the number of visits of node j and c is the exploration factor[3].

B. Expansion

We say that a node is not fully expanded when at least one of his children is not currently in the tree. If we reach during the selection a node v that is not fully expanded, then we select one move that has not been tried yet, and add his corresponding children v' to the tree. The selection of the children to be added may be done uniformly randomly or up to some heuristic similarly to the ones described in the next section.

[2]depending if the corresponding move is an atari move.

[3]to be determined empirically. For most games where a simulation ends with a score in $\{0, 1, -1\}$, we take $c \in (0, 2)$.

C. Simulation

From v' is played a random game, meaning that the corresponding players play move at random among the legal ones available. A simulation is conducted up to when a player wins the game. In games like *Chess* where it is legal to have sequence of moves that repeat possibly infinitely, we prevent this by removing from legal moves the ones that would make us reach a previously visited state of the game. Such simulations are often also called *playouts*.

While the PIMCTS requires no game knowledge to work correctly, it is possible to tweak the simulation phase by adding some rules to modify the probability of choosing a move. For instance for chess, it is reasonable to assume that taking the enemy queen is often a good move. These modified simulations are called a *hard playouts* or *heavy playouts*.

Hard playouts rules can be coded by hand, but it is also possible to use an other approach. In [9], Drake and Uurtamo used a database for the game of *Go* to develop what they called the *pattern* heuristic: a pattern is a 3×3 region of the board, filled by either player point or vacant. Each pattern is associated with a weight representing the value for the white player to play in the center of the pattern. Their results shows that this approach does not perform very well in comparison to other heuristic when playing versus a no-heuristic player, and they suggests it may perform better when combinated with other hard-playout heuristics.

An other technique worth mentioning is *pruning with domain knowledge*. As we mentioned earlier, it is not profitable to remove moves that are *supposed* to be bad, at best we just reduce the probability of choosing them. The only exception is when we know by proof that a move is bad. For instance, for *tic-tac-toe* we know that playing in the middle case is the best strategy, so we can remove all moves that are different. In [10], Huang et al. shows that pruning with domain knowledge can improve the performance of their algorithm up to 50%.

D. Backpropagation

Backpropagation consists in travelling back to the root, modifying the information of each node we used. More specifically, we increase the counter of time we traveled through each node, and the counter of wins if it is relevant. For games where the result of the game is a simple win/lose we can simply count how many wins are associated with a node. When the result of a game is associated with a number of points we try to maximize, we can instead store the number of points earned at the end of the playout.

E. Parallel computation

An other improvement worth mentioning for the implementation of PIMCTS algorithm is the parallelization of the algorithm. Several approaches can be considered:

- *Leaf parallelization*: at the beginning of the simulation phase, run k simulation in parallel[4], then start the backpropagation once they are done.

[4]for ISMCTS, we of course have to consider different determinization for each of these k simulations.

- *Root parallelization*: build k different trees in parallel. Once we are out of budget, combines the trees together.
- *Tree parallelization*: different threads perform computation on the same tree. When a thread is working in a sub-tree of v_0, v_0 being a direct child of the root, we prevent other threads to work in the same sub-tree. Despite the increase number of communication of tree parallelization in comparison with other approaches, it slightly outperforms them [11].

IV. Information set Monte Carlo Tree Search

A. Determinization

Observe that for two-player belote, some information are hidden to the players. This raises a huge problem during the simulation phase: the cards hidden to the AI player cannot be the ones of the game, but they must randomized somehow or the algorithm will end acting like he knows what are the cards he should not being able to see.

A classical way to solve this problem is determinization [5] [8]. At each iteration, before the selection phase, the unknown parameters of the game are uniformly randomized[5], then we apply PIMCTS on this determinization of the game.

There are several weaknesses [4] behind the classical determinization approach:

- First, *strategy fusion*: a full example of strategy fusion is detailed in Fig 3
- Second, the tree often has nodes that are sort of equivalent but does not exploits this to speed up the computation.

B. Definitions

A game tree is a directed graph (S, Λ), where S are the states of the games and Λ are the sections of actions. For each state s, we define $p(s)$ as the turn of the player that is about to act[6]. The initial state of the game is denoted s_0.

The terminal states s_T are equipped with a reward vector μ. Classically the values taken by μ are in $\{-1, 0, 1\}$.

Two states may be connected by an edge, but players do not chose edges directly: they chose actions. An action may be interpreted in natural language as "player 1 plays an ace of spade". Actions can be seen as equivalence classes for edges; and the players chose among these classes. Naturally, if two edges start from the same state, they belong to two different equivalence class. The actions available at state s are denoted $A(s)$.

For games with imperfect information, players do not observe directly $s \in S$ but information sets, that are subsets of S. $\sim_i$ is an equivalence relation on the state set; its equivalence classes are the information sets of player i.

A game Γ for two players is:

$$\Gamma = (S, \Lambda, s_0, \mu, p, (\sim_1, \sim_2))$$

The legal moves from s from player i point of view are:

$$M_i(s) = \{[(s,u)]^{\sim_i} \mid (s,u) \in \Lambda\}$$

The legal moves from s are :

$$A(s) = M_{p(s)}(s)$$

C. PIMCTS Algorithm

Let s_0 be the state of the game where the AI player has to make a move. Suppose without loss of generality that the AI player is player 1. By our previous definitions, $[s_0]^{\sim_1}$ is the information set of player 1, that is containing all the possible dealings of the cards from its point of view. The algorithm works as follows:

Algorithm 1 Function ISMCTS

```
create a tree consisting of a single root node v_0
for n iterations do
  //Determinization
  pick d ∈ [s_0]^{~1} uniformly at random
  //Selection
  v = v_0
  while v is non-terminal or fully expanded do
    pick v in the children of v compatible with d according
    to a chosen formula
  end while
  //Expansion
  if v is non-terminal then
    choose a random action a untried action available from
    v under d
    add a child v' corresponding to the action a to the tree.
  end if
  //Simulation
  run a random simulation from v' using d
  // Backpropagation
  for each node u on the v'v_0 path do
      update u visit count and simulation reward for each
      sibling w of u do
      if w was available for selection under d then
        increase w's availability count
      end if
    end for
  end for
end for=0
```

The structure of the ISMCTS algorithm is very similar to a classical MCTS. However it has a few differences:

- Observe the determinization d is picked at the beggining of each loop, and is used during all four phases.
- During selection, we use it to only consider parts of the sub-tree that are relevant with d.

[5]Note that for some games as *Belote*, this is not an optimal way to do it. Suppose that ♣ is the trump suit and the first round went like that: I play A♣ and the adversary played 8♣. Then, we know that our adversary does not have in hand any of the following card:{J♣, 9♣}, else he broke the rules. These logical conclusions can be done after each trick, and allows us to use determinization that are more relevant. For this project, we did not implemented such logical rules because of time limitation, and the suspicion that the AI player will be very strong anyway.

[6]For games like *Chess*, players playing one after the other so we don't really need this as knowing how far from the root we are is sufficient. For Belote, a player may play twice in a row when winning a trick.

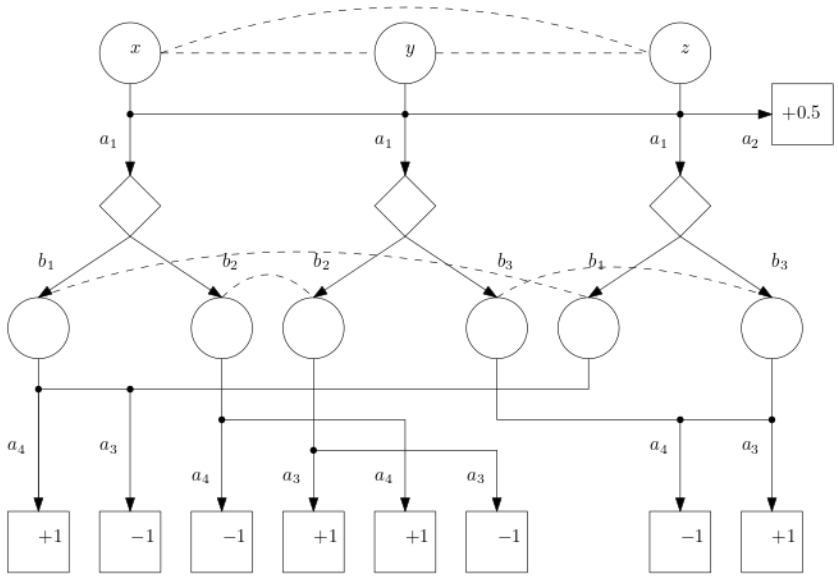

Fig. 3. Example of a game tree with two players. Decisions of player 1 are circles, decisions of player 2 are diamonds. The squares correspond to terminal states, and they hold the rewards for player 1. x, y and z are three different determinization of the game state. The dashed lines between two states correspond to information sets of player 1, so they cannot distinguish two states linked in this way. If player 1 decide to play the action a_2, he immediately earns a score of 0.5. If they choose a_1, player 2 choose a move among b_1, b_2 or b_3 depending on the determinization. If b_1 (respectively b_3) is observed by player 1, then they can decide the action a_4 (respectively a_3) and earn the maximum reward of +1. If player 1 instead observe the move b_2, then they theoretically cannot decide which action leads to a +1 score. However, using determinization, the algorithm "knows" whether it is in x or in y, thus it wrongfully concludes that choseing the action a_1 leads to a gain of +1. This is an exemple of strategy fusion.

- During Expansion, we can only extend to moves allowed by d. For *Belote*, it may happen that a node that was considered fully expanded under some determinization has new moves available under d.
- During simulation, d is used as for PIMCTS.
- During Backpropagation, we consider the siblings of the nodes visited that were available under d. This is a crucial to apply the UCT formula. We use here a modified version of it:

$$argmax_(j,d)\left(\mu_j + 2c\sqrt{\frac{2\ln(n'_j)}{n_j}}\right)$$

where j must comply to d and n'_j is the availability count of j. When a node is made available a large number of times, the number of time it is visited thus increases.

V. Rule-based players

Note: sections 5 to 8 are currently not written as I'm conducting the experiments. Currently, the ISMCTS algorithm is implemented and seems to be working correctly. The two-players belote game is implemented as well, and I can run simulation of an ISMCTS player with a random player or a human player.

I'm working on the following aspects:

- making sure the implementation is correct,
- and decorating the code to add measurement tools.
- implementing a rule-based player

Around the 10th of May I plan to distribute the code for voluntary players to play against the ISCMTS-player, and I'll compile the results in sections 6-7. As soon as the greedy rule-based player is ready, I'll make it play games against both the random and the ISMCTS players.

VI. Experiments

VII. Results

VIII. Conclusion

References

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in Proc. 5th Int. Conf. Comput. Games, Turin, Italy, 2006, pp. 72–83.

[2] Bruno Bouzy. Move pruning techniques for Monte Carlo Go. In Advances in Computer Games 11, Taipei, Taiwan, 2005.

[3] J.R.S. Blair, D. Mutchler, C. Liu, Games with imperfect information, in: Games: Planning and Learning, Papers from the 1993 Fall Symposium, AAAI Press, 1993, pp. 59-67.

[4] I. Frank and D. Basin, "Search in games with incomplete information: A case study using bridge card play," Artif. Intell., vol. 100, no. 1–2, pp. 87–123, 1998.

[5] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower bounding Klondike solitaire with Monte-Carlo planning," in Proc. 19th Int. Conf. Autom. Planning Scheduling, Thessaloniki, Greece, 2009, pp. 26–33.

[6] Cowling, Peter I., Powley, Edward J. and Whitehouse, Daniel (2012) Information Set Monte Carlo Tree Search. Computational Intelligence and AI in Games, IEEE Transactions on. 6203567. pp. 120-143. ISSN 1943-068X

[7] https://en.wikipedia.org/wiki/Belote

[8] C. D. Ward and P. I. Cowling, "Monte Carlo search applied to card selection in magic: The gathering," in Proc. IEEE Symp. Comput. Intell. Games, Milan, Italy, 2009, pp. 9–16.

[9] P. D. Drake and S. Uurtamo, "Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go," in Proc. 3rd North Amer. Game-On Conf., Gainesville, FL, 2007, pp. 35–42

[10] J. Huang, Z. Liu, B. Lu, and F. Xiao, "Pruning in UCT algorithm," in Proc. Int. Conf. Tech. Appl. Artif. Intell., Hsinchu, Taiwan, 2010, pp. 177–181.

[11] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo tree search," in Proc. Comput. Games, Beijing, China, 2008, pp. 60–71.