

École polytechnique de Louvain

Performance Evaluation of Counterfactual Regret Minimization and Monte Carlo Tree Search Variants:

A Case Study on Belote

Auteur: **Alexandre SERRAO**

Promoteurs: **Eric PIETTE, Achille MORENVILLE**

Lecteurs: **Eric PIETTE, Achille MORENVILLE, Aurélien BUCHET**

Année académique 2024–2025

Master [120] : ingénieur civil en informatique

If there were no tomorrow, I'd live today the same.

CONTENTS

Summary	v
Acknowledgments	vii
1 Introduction	1
2 Context	3
2.1 Belote: The game.	3
2.1.1 Rules.	3
2.1.2 Challenges.	6
2.2 Imperfect Information Games	7
3 Important Concepts	9
3.1 Game Theory	9
3.1.1 Nash Equilibrium	11
3.2 Monte Carlo Tree Search.	11
3.2.1 Algorithm	12
3.2.2 Upper Confidence Bound UCB	13
3.2.3 Limits to simple MCTS.	13
3.3 Item Set MCTS	13
3.4 CFR	15
3.4.1 Regret and Counterfactual regret.	15
3.4.2 CFR algorithm or Counterfactual regret minimization	16
3.4.3 Limits of CFR	19
3.5 ESMCCFR & OSMCCFR	20
3.5.1 External Sampling	20
3.5.2 Outcome Sampling.	21
3.6 DeepCFR.	22
3.6.1 Neural Networks.	22
3.6.2 DeepCFR for IIG's	24
4 OpenSpiel	25
4.1 Important code.	25
4.1.1 spiel.h	25
4.2 Belote in Openspiel	27
4.2.1 Implementation	27
4.2.2 Observation Tensor	29
4.2.3 Resampling.	31

4.3	Implemented algorithms	32
4.3.1	Implementation of IS-MCTS	32
4.3.2	Implementation of ESMCCFR	34
4.3.3	Implementation of OSMCCFR	38
4.3.4	Implementation of DeepCFR	42
5	Results	47
5.1	ISMCTS	47
5.2	ESMCCFR	50
5.3	OSMCCFR	51
5.4	DeepCFR.	53
6	Conclusion	59
6.1	Round Robin	59
6.2	Perspectives	60
6.3	Final words.	60
7	Glossary	61
	Bibliography	63
	Glossary	64

SUMMARY

This thesis tackles a few MCTS and CFR based algorithms for imperfect information games through a study case on the card game Belote. Belote is a trick taking game played by 2 teams of 2 players where the goal is to be the first team to reach 1000 points through trick playing and bets. The focus will be on the following algorithms: MCTS and its imperfect information variant ISMCTS, MCCFR with external and outcome sampling as well as DeepCFR, the neural networks version of MCCFR. To study these algorithms, OpenSpiel will be used as a framework to implement Belote and tune the algorithms to fit the game and vice versa. The algorithms studied will first be detailed theoretically and then through experimentation. The goal of these experiments will be to measure their performances and understand the results and how they work in the specific domain of the Belote game.

ACKNOWLEDGMENTS

I'd like to give my sincerest thanks to my supervisor Eric Piette for the opportunity to write about such an entertaining subject. To my assistant supervisor Achille Morenville, for his guidance, encouragements and help in time of needs. Thanks to my faculty for bearing with my imperfect organization. A special thank you to Sarah, who tries her hardest to do enough when she's doing the most. I am deeply grateful for the people who kept me going and gave me strength with those little gestures that mean the world. I'd never want to be someone else, for I am who I am thanks to them.

1

INTRODUCTION

Imperfect Information Games provide a challenging setting for artificial intelligence due to stochastic events, large exploration costs and hidden information. While algorithms such as MCTS and CFR variants are already widely covered in the literature, their applicability remains under explored for more unconventional games such as Belote. This thesis implemented the Belote game in OpenSpiel, providing a new research tool for IIG's, and evaluated the performances of known algorithms on this setting. Although traditional ISMCTS and MCCFR algorithms manage to provide a strong baseline and robust strategies, they still struggle with scalability. As for DeepCFR, although it shows promising learning signs, it still underperforms in a context as complex as the one of the Belote game. By providing tools and preliminary insights on the subject, this thesis opens the way for other researches on the adaptability of current IIG's algorithms in high stress environments.

1

Imperfect information games (IIG's) are very popular among the public but also among researchers. For researchers, it is very important to invest in finding solutions to deal with the imperfect information constraint. This constraint forces to find ways to deal with both stochastic responses and high volume decision trees. Opposed to fully observable games of the same scale, IIG's possess a lot of possible game states and it is a challenging setting to find good decision mechanisms. Furthermore, the algorithms applicable in IIG's context are not only powerful but also usable in other real life imperfect information contexts and in large perfect information contexts thanks to their abstractions and other approximation mechanisms.

Among others, Monte Carlo Tree Search (MCTS) is an algorithm that fares well in large scale deterministic games [1]. Its purpose is to produce a relevant decision tree without repeatedly exploring all the breadth of the tree. Counterfactual Regret Minimization, or CFR, is another algorithm that repeatedly computes the regrets and counterfactual regrets of the different positions of players, in a game context, to come up with a well refined strategy depending on the adversary strategy. While it is a theoretically perfect algorithm in the sense that, with enough iterations, it converges to a Nash equilibrium, it's scalability over large games is problematic.

This thesis will tackle algorithms that are derived from both of these well known principles in the imperfect information game setting of Belote. Using ISMCTS, the imperfect information variant of MCTS as the baseline, MCTS & CFR mix, MCCFR will be evaluated with external sampling and outcome sampling. DeepCFR will also be one of the main subjects as it also derives from both starting algorithms, with a deep learning combination.

Belote is a game that has a particular structure of betting and playing phases, with a lot of strategic plays and uncertainty. It is a very simple game as well as one of the hardest to solve and it is therefore an interesting subject to evaluate current existing algorithms and adapting them to the Belote game, which is a very well-known game but has yet to be studied. Similar games of trick tacking games such as Bridge or Whist, that have already been studied. However Belote brings its own challenges and interest with its own particular set of rules. In order to evaluate it, the Belote game will be implemented in the OpenSpiel Framework allowing to use the existing algorithm on it as the basis of this study. Adding the Belote game to OpenSpiel will also provide a mean for further researches to take place on the study case.

This thesis will first cover a theoretical part on both the study case context and the mentioned algorithms, as well as some important concepts necessary to understand them. Secondly, it will dive into the OpenSpiel platform through both the current existing algorithms and the implementation of the Belote game on it. Finally, the evaluation as well as the discussion on how they performed in this particular setting.

2

CONTEXT

This thesis aims to explore learning algorithms and agents for an imperfect information game. To support this endeavor, some groundwork must be laid about the studied case and the surrounding domain of the problem.

2.1 BELOTE: THE GAME

The Belote game is a 4 players card game that is played through betting and winning tricks. It is hard to retrace its origins as it has been transmitted by word of mouth for centuries. It also is the reason why it currently has so many official variants and even more unofficial ones. As it stands it is still a very popular game among people of all ages and environments. The variant covered by this paper is a simpler variant of the original game called the Coinche.

2.1.1 RULES

The variant that will be studied is more specifically a variant of the Coinche. Let's review its rules.

Player Order

The game is played with four players making two teams of two players playing in order. If player P_{ij} is player j of team i , the order circles like this :

$$P_{11} \rightarrow P_{21} \rightarrow P_{12} \rightarrow P_{22}$$

Where any P_{ij} can be first player of the phase.

Phases

The first player is the one right after the dealer following the *player order*. When all the phases are over, the first player becomes the dealer.

1st deal

The dealer deals 3 cards to each player starting with the first player and then another 2 cards totaling 5. The top of the deck card is then revealed to players to start the betting phase.

Betting

The revealed card's suit is the proposed trump for this round. Starting from the first player, each player can either accept the trump and take the revealed card, or pass. However, if everyone passes, a second round of betting starts.

Starting with the first player, each player can either announce a new trump or play the game in *all-trump* or *no-trump* mode and take the revealed card, or pass. If again everyone passes, the first player is forced to make a bet and take the card.

2nd deal

The dealer then deals 2 cards to the better and 3 to the others, starting with the first player to complete the eight cards hand of each player.

Play

The players will now play 8 tricks. The first trick is initiated by the first player (still the one after the dealer). After that, the player who has won the last trick is the opener for the next one. Each card has a value and each trick's value is the sum of the 4 cards that compose it.

When every trick has been played, the total points of the tricks won by the two teams is calculated and the points are added to the total of each team, depending on the bet that has been taken and if it has been won. If the total of one team reaches above 100 points, the team wins. If both team reach above 100 points at the end of the 8 tricks, the team with the highest score wins. If none reached 100 points, the cards are shuffled again and the game goes back to the dealing phase, changing dealers and first players accordingly.

Tricks

A trick is a set of four cards played by the four players. The players must follow the rules of the trick and can't always play any card of their hands.

1. The players must play a card of the same suit as the opener if they can.

2. If they can't, they must play a trump card, except when the current trick winner is their teammate.

3. Played trump card must always be of a higher rank than the other played trump cards unless the current trick winner is a teammate.

4. If a player can't play the opener suit and cannot play a trump, or is not required to, he can play any cards he want.

The player who wins the trick is the one who has played the card with the highest value in the opener suit or the highest trump. In *all-trump* or *no-trump*, it is always the highest value from the opener suit. The player who wins becomes the opener for the next trick.

Card Value

In Coinche, the cards range from 7 to Ace. However they are not ordered as in a standard card game and the order changes depending on the trump.

Normal order			A	10	K	Q	J	9	8	7
Value	20	14	11	10	4	3	2	0	0	0
Trump Order	J	9	A	10	K	Q			8	7

When a suit is elected as the trump, it follows the trump order, and the other ones follow the normal one. In *all-trump*, all suits follow the trump order and in *no-trump* all suits follow the normal one.

In a standard round of eight tricks, there are 162 points at play. 30×3 from the normal suit, and 62 from the trump suit and an additional 10 from the last trick winner. In *no-trump*, the value of the Ace is set to 19 instead of 11 so there is also a total of 162 points to be made. In *all-trump*, there is 258, 4×62 and the additional 10.

Belote and Re-Belote

It happens when one player possesses both the King and Queen of a trump suit. When playing one of them he can call *Belote*, announcing that he possess the other card of the pair and *Re-Belote* when playing it, granting 20 additional points for his team. In case of *belote* and *re-belote* there are some extra points to be made. 20 for normal games, 80 (potentially) for *all-trump* games and 0 for *no-trump* games.

Bets

The betting is the phase that will determine the value, the order and also the

strategy the player must adopt. The bets can take 4 forms:

1. Accept the proposed suit as trump.
2. Choose any other suit as trump.
3. Play every suit as a trump suit. Essentially if every suit is a trump in *all-trumps*, no suit is a trump but the order still changes.
4. Play every suit normally so there is *no-trump*.

2.1.2 CHALLENGES

Belote is a very challenging game to solve in many different aspects and has a unique structure that enhances the difficulty.

First and foremost, the strategies. Belote is a card game that does not follow the conventional card order. Furthermore, depending on the betting phase, the card order will change, making it harder to evaluate a good hand before the playing phase. Also, it is a team game, where the cards of the team mate as well as the opponents are unknown. So winning a trick is not always the better option. Each card has its own point value, that changes depending on the setting, making some tricks more important to win or avoid losing. Thus winning the most tricks is not always the answer either. The Belote game has a lot of subtleties that make defining a domain dependent heuristic a complex problem and finding a good strategy requires to explore every possible options.

The second point that makes the domain of Belote interesting is the separation of the game in two dealing phases. This means that a decision must be taken when there remains 3 cards to be dealt to every player. This adds a lot of uncertainty and it also is a complex factor to consider. The betting phase, while done before any tricks, is the single most important decision of the game. Many rounds can go unilaterally if the betting is in favor of a team.

Taking into account the team, hidden information, two-phased dealing, and rules complexity, it makes the Belote a prime study case for artificial intelligence and especially tree based algorithms.

2.2 IMPERFECT INFORMATION GAMES

Belote is an imperfect information game, meaning that players do not have access to all the information. In this case it is the cards of other players and the cards that have yet to be dealt in the betting phase. This adds a lot of complexity to the creation of an agent.

Compared to perfect information games, IIG are known to be harder to solve[2] because they are composed of stochastic plays in most of the moves, resulting in the inability of players to ascertain the result or the gain an action can provide. Also in many instances of IIG, the game can take an absurd amount of states, as defined in the Game Theory discussed in chapter 3, making it hard to describe a state by state policy.

Metrics

For the Belote game, the starting hand of a player is at first composed of 5 cards and then 8, giving roughly $\frac{32!}{26!} \approx 652 \times 10^6$ possible states for the betting phase and $\frac{32!}{24!} \approx 424 \times 10^9$ at the start of the playing phase. This is only taking the known cards into account.

Considering the number of possible bets: 4 suits, *no-trump* and *all-trumps*. Looking at the starting states of the playing phase, without hidden information, $32!$ possible states exists. This number multiplied by 6, the number of bets is over 10^{35}

The challenge lies then in adapting known algorithms to build agents such as MCTS to match this number of possible states and the stochastic nature of the choices each move has in an IIG. Thankfully, it is not imperative to explore the entirety of the non-hidden information tree, but exploring the information hidden tree remains very challenging.

3

3

IMPORTANT CONCEPTS

This chapter covers the main theoretical concepts that are important to the understanding of this thesis. It will present the algorithms studied as well as some other concepts at their roots.

3.1 GAME THEORY

To better illustrate games in the AI field, we can use game theory concept which is itself derived from the decision theory. Most games in a programming context can be described following a set of features[1].

- S a set of states which has terminal states forming its own subset S_T and an initial state s_0 which can be different depending on the game instance.
- n a number of players.
- A a set of actions that can be applied on a state.
- f a transition function that matches a state to another one through an action : $f(s_i, a) = s_{i+1}$.
- R an utility function that evaluates a state from S . This can be seen as a score for a given position for a player.

This list is not exhaustive but summarizes well which elements constitute a game. Depending on the complexity of the game, this list is sufficient to describe it completely or may need some more features.

Games can be categorized following these different properties :

- **Zero-Sum** Meaning that the gains and the losses of each player equals zero. The win of a party equals the loss of another.
- **Information** Whether the players have access to all the knowledge of the game or if some is undisclosed for a certain part or for the totality of the game.

3

- **Determinism** If there is a hidden chance factor somewhere and two instances of the same transition function, state and action may give a different result.
- **Sequential** Whether actions can be taken simultaneously or not.
- **Discrete** Whether actions are discrete or continuous. *eg: Poker with the raising mechanic.*

The Belote game can be defined as a *zero-sum* game where one team wins and the other one loses, where the *information* is incomplete many times until the last hand is played. The game is set in a *deterministic* landscape and is *sequential* where each player plays in turn. Each action is in a finite array of possibilities so we can say that it is *discrete*.

A state s can be defined by the numbers of cards in the hand of each player, the first player, the players hand, the cards on the table, the points each team made so far, the game mode and the player who won the last trick.

Since the game is in two phases, it may be important to consider two initial states for each phase. For the betting phase, an initial state s_0 can be defined as four hands of 5 cards, the card that is revealed in order to make the bets, the first player, the current player and how many turns it has been or the history of actions. The initial state of the playing phase slightly differs, as it has 8 cards hands and a trump mode.

And a terminal state is summarized by empty hands and the total score of each team when one of the said scores reach a given threshold.

The number of player would be 4 for 2 teams. The set of actions A would be the size of the player's hand truncated by the rules of the game. The player can play any card in hand unless he is forced to follow a color. Then again, we don't consider the betting phase which is a bit different from the trick trading part of the game.

The transition function is a bit more complex to be succinctly described. It allows the game to go from one state to another through an action. To put it simply, some "general state cases" are needed.

- s_1 is the state where every player has the same number of cards in hand. The trick and the rules of the trick have yet to be made public and *player one* has to make a move. Note that in this case player one is the one who needs to play first in this round. So it's either the first better at the start of the round or the last player that has won a trick during a round.
- s_2 is when player one has played and player two has to play a card. s_3 and s_4 are the same respectively for player 3 and 4.

- s_5 is a *temporary state* which is only there to highlight the transition from s_4 to different states. After the fourth player played its card, the trick is attributed to the winner and the points are calculated. Then there are different scenarios. If players still have cards to play, the game circles back to s_1 and switch the *player one* marker to the player that won the trick. If not, the total score of both teams is evaluated to see if there is a winner and reach s_t a terminal state, else the card are re shuffled and dealt again to all players and the state goes back to s_1 .

The transition function described above does not take into account the betting phase. Because it is only there for the notation and parallel with game theory. It is possible to derive the state transition function from the rules in chapter 2.

3.1.1 NASH EQUILIBRIUM

If we define $\mathcal{P}_i$, a policy for player i . $\langle a \in A, p \rangle \in \mathcal{P}_i(s)$, the list of possible actions and their probability to be taken in state s . A Nash equilibrium is a set $\mathcal{P}$ of policies such that altering the policy $\mathcal{P}_i$ is expected to worsen the expected results of player i [3].

In other words, it is a policy that is considered the best for all players in term of expected values and if one player tries to "take advantage" of the other policies it would only prejudice him.

Using rock-paper-scissors, a simple simultaneous, 2 players, zero-sum game as an example[4], such a policy would be to play each possible action with an uniform probability. At some point, if one of the player decides to play rock more often than the other two possibilities, the other player could adjust and profit from this change by playing paper more, which would lead back after adjustment to an uniform probability. This is an important concept as the purpose of the CFR algorithm and its variants is to reach such a set of policies.

3.2 MONTE CARLO TREE SEARCH

MCTS is a powerful algorithm that produces decision trees which is at the root of many Imperfect information algorithms[5]. It works by simulating actions on game states following different *decision* and *tree* policies to determine which node to explore and which action to take. Its strengths are that it does not require to update the score of intermediate game state nodes beside the terminal ones and can produce really good results on random decision policies for the tree building. The thinking process behind MCTS is to approximate the entirety of the search tree by exploring only a part of it and approximating the value of high order nodes by sampling enough leaf nodes. This allows to treat search trees that have exponential growth rate and have precise close to real value results. The algorithm's efficiency has been proven on a variety of large problems and games such as Go[6], and its concept is a sturdy base for many others.

3.2.1 ALGORITHM

The main idea of the algorithm is to build a decision tree for the problem it's working on by simulating episodes and outcomes. The process is repeated a for a specified number of iterations and during each iteration the tree is expanded following a tree exploration policy and a node selection policy. It works as follows:

1. If the node is a terminal state, back-propagate the value of the current state up the tree, which will reflect the quality of the leaf node.
2. If the node is not terminal but is a leaf that has no children yet, expand it following the rules of the game. The value of the expanded node is evaluated with a function evaluator. The evaluation can be done with a heuristic function or through a random rollout unfolding of the tree.
3. If the node is not terminal and expanded, choose a child node of the current node to continue the descend of the decision tree. The decision for the child node is done following a decision policy. There are many existing policies which can dictate how well the algorithm behave in different use cases.
4. Repeat the process on the selected child node until the resources run out.

Example 3.1: PSEUDO CODE MCTS

```

1 MCTS(node, t) :
2   while(t < threshold)
3     if(isTerminal(node)) : backPropagate(node);
4     if(node.children.isEmpty()) : node.expand()
5     else:
6       nodeToExplore -> select(node.children)
7       MCTS(nodeToExplore, t+1)
8   return node

```

The tree resulting from this building sequence will allocate higher values for actions that end up leading to preferable outcomes and lower for actions leading to undesirable ones. The important part of the algorithm is the policy that dictates which node to explore. In simple games it is possible to explore the entirety of the tree and have a clear value of the "best move" but it is often not possible to explore everything and have this certainty.

Problems that have multiple possible action choices and where it is uncertain which one is the best are part of the *bandit problems* class. In many cases it won't be possible to explore the entirety of the decision tree, and the difficulty here is to decide whether to explore promising nodes fully, or to explore nodes that are currently worse but could potentially lead to a better outcome. That is where the selecting policy comes in.

3.2.2 UPPER CONFIDENCE BOUND UCB

The selecting policy UCB[7] is a very straightforward one that tries to balance the dilemma between *exploitation* and *exploration*. The idea is that a promising node is always worth investing but it should still be possible to explore elsewhere hoping to find something better. Its value can be determined following this principle:

$$UCB_1 = \overline{X}_j + \sqrt{\frac{2\ln(n)}{n_j}} \quad (3.1)$$

Where $\overline{X}_j$ represents the mean value of the gains of this selection, n the number of simulations and n_j the number of simulations where node j has been selected so far. A good node will be selected because its value is high and the selection will grow the interest in other nodes that are not often selected. Thus exploiting first promising nodes, followed by exploration of under represented nodes/actions.

Depending on the characteristics of the problem it is sometimes good to upscale the constant factor of the logarithmic function of n in order to gather more exploration type iterations. The setting of this variable and altogether, the choice of the selection policy can have a significant impact on the performances of MCTS in different environments.

3.2.3 LIMITS TO SIMPLE MCTS

MCTS is a very good choice for any large game to approximate the decision tree of such large problem domains. However it suffers from scalability[8] problems when it encounters high branching factors trees and thus very large game trees. As it is, it is also unable to perform on imperfect information games because of the absurd amount of states and their representation.

3.3 ITEM SET MCTS

Also written as IS-MCTS[9] is a variant of the popular MCTS that can be applied to imperfect information games. MCTS is impractical to use on IIG's because its efficiency relies on the full state of a game which can't be disclosed with hidden information. One solution, to keep the powerful MCTS principle with imperfect information, is to use *item sets*. Item sets are sets of possible states described by a special partial observer state.

The idea remains the same but instead of doing straight simulations of the game to build a search tree, at each iteration, a sample state is generated through the partial observer state also known as the information tensor. The information tensor contains a representation of the current state seen by the observer. Given the information tensor, plausible states are generated to simulate the game as many times as needed and update the search tree and the item set each plausible state belong to.

Example for Belote

As stated above, a game state of Belote can be described by the hands of the players, the trick in play and other information. However in the case of item sets, the item sets are defined by an information tensor of the current player that contains the information accessible by the player.

For a Belote state, that would be the player's hand, the trick in play, and the history. Assuming each player has played 4 cards, which means there are still 4 cards left in each hand, a single item set could contain more than $4 * 10^8$ actual game states taking the cards played into account. This means that the total number of nodes will be divided by a big factor but the number of times each node need to be explored is still non-trivial.

Luckily the number mentioned earlier can be greatly reduced using domain dependent constraints on the plausible nodes, but it remains high.

An Item Set $\mathcal{I}$ can be defined as $\mathcal{I} \equiv \{s | I(s_i) = I(s_j) \forall i, j\}$ with $I(s)$ the information tensor of state s . Note that the information tensor is dependent of the player but for simplicity, the player that perceives the Info Set will always be the current player.

This method allows to reduce the size of the tree by fusing, depending on the original state, a lot of plausible states in one item set which can be intuitively both a good or a bad decision. A good consequence is that it allows to discover moves that are considered powerful in many different states because a said move will score highly in different plausible states. It might also backfire by highlighting moves that are considered losing or highly losing in some states. For example in Belote, the only needed information for a player to take an action are the legal possible actions at its current state. This alone could constitute a sufficient information tensor to proceed with the algorithm and would drastically reduce the space complexity of the search tree. However, doing this would fuse all the possible states and a good deal of impossible states into a single item set. This would lead to an uninformed decision evaluated with a lot of noise from impossible states in the current game. Adding information such as the current trick and the cards that has already been played creates a bigger number of possible Item Sets but reduces their size. Depending on the use case, an adapted information tensor can make all the difference by balancing precision and approximation for the problem in play.

Figure 3.1 shows that a less precise information tensor will lead to a bigger node approximation and thus less precision on the node value, and vice versa for a more precise tensor. As said above, both approaches have their pros and cons.

However, it remains primordial to fuse these nodes in information sets so that the algorithm can devise an unbiased strategy based upon the observation of the player and not a single sampled game state.

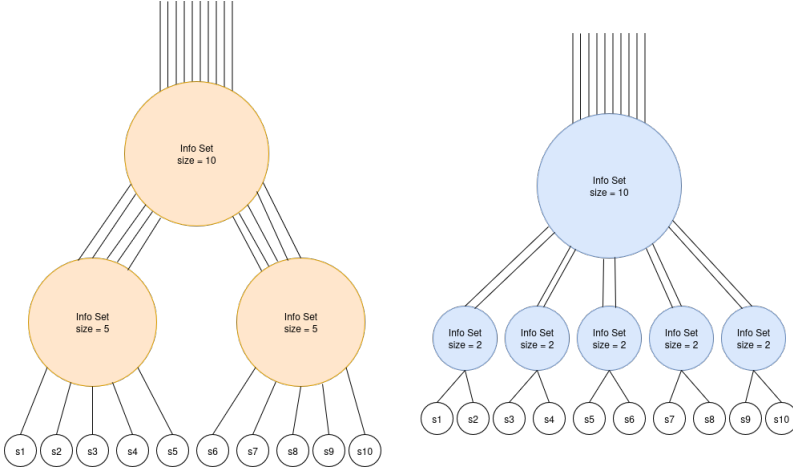


Figure 3.1: Difference in information tensor precision

3.4 CFR

3.4.1 REGRET AND COUNTERFACTUAL REGRET

Regret can be defined as a scale quantifying how well would one perform making a choice against the current strategy profile. Counterfactual Regret[10] is the regret weighted by the probability of the opponent playing a sequence of actions such that the current info state is reached. Regret of player i can be defined as follows:

$$r_i(I, a) = v_i(I, a) - v_i(I) \quad (3.2)$$

Where $r_i(I, a)$ defines the regret of taking action a at info state I , $v_i(I, a)$ defines the expected outcome value of taking action a at info state I and $v_i(I)$ is the expected outcome of reaching node I .

$v_i(I, a)$ and $v_i(I)$ can be defined as[11]:

$$v_i(I, a) = \sum_{z \in Z(I, a)} \mathcal{P}_{-i}(I) \mathcal{P}_c(I \rightarrow z) u(z) \quad (3.3)$$

$$v_i(I) = \sum_{a \in A(I)} \mathcal{P}_i(a) v_i(I, a) \quad (3.4)$$

Where $\mathcal{P}_{i,-i,c}(I)$ are respectively the probability to reach I following the policy of player i , the policy, and the chance node policy, $u(z)$ the utility of terminal state z .

To put it in other words, the expected value of choosing action a $v_i(I, a)$ is the sum of all the possible outcomes deriving from this choice, weighted by the probability of chance node and opponents to behave such that these outcomes are reached. The compound utility of the current node $v_i(I)$, is the sum of all the outcomes reachable from this node weighted by the probability of opponents and chance node playing such that they're reached, altogether weighted by the probability of current player playing these nodes in the current strategy

profile. The counterfactual regret of choosing action a at state I , $r_i(I, a)$, is the difference between these two values.

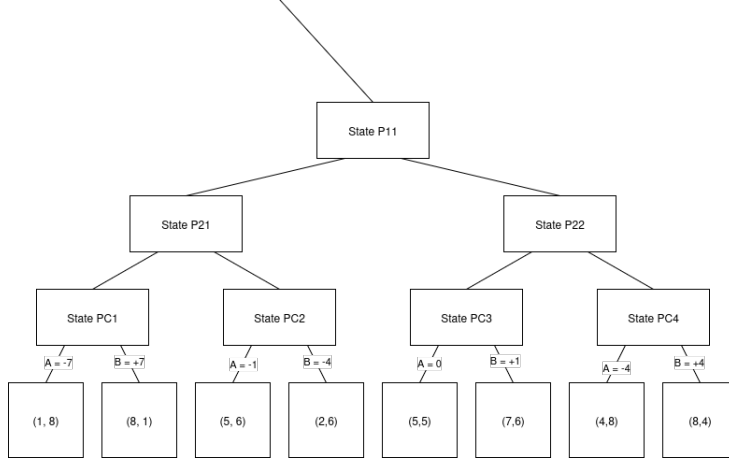


Figure 3.2: Simple decision tree

Figure 3.2 shows a simplified decision tree of a simple game.

In player one's point of view, assuming he plays a game where he takes action left, player 2 then always takes action left too, and the chance node always selects the left-most action too, the final score will be (P1:1, P2:8). Assuming in another play, the players and chance nodes always take the most-right actions, the result will be (P1:8, P2:4). If we define the Utility function for player one to be $U = u_1 - u_2$, we can compute the regret of choosing action left over the right one as $R = U_r - U_l = 4 + 7 = 11$. This can be interpreted as "if I played only right I would have had 11 more points now. The counterfactual regret of this value would be the value itself (11) multiplied by the probability of it being reached, the probability of State P11 being reached in the game.

3.4.2 CFR ALGORITHM OR COUNTERFACTUAL REGRET MINIMIZATION

The goal of this algorithm is to find a policy that minimizes regret while taking into account the regrets of the opponents. To do this, we compute the regret of each action using the counterfactual regrets. Starting from the root node, where the reach probability is 1 for every player, the counterfactual regrets are computed by traversing the decision tree.

Equation 3.3 can be rewritten as:

$$v_i(I, a) = \sum_{a' \in A(I_a)} \sum_{z \in Z(I, a)} P_{-i}(I) P_c(I \rightarrow z) u(z) \quad (3.5)$$

When the total sum is decomposed on n sums for each possible action in info state I . And now we encounter two possible cases : (i) I is a current player node, (ii) is an opponent node.

For (i), $\mathcal{P}_{-i}(I) = \mathcal{P}_{-i}(I_a)$ as well as $\mathcal{P}_c(I \rightarrow z) = \mathcal{P}_c(I_a \rightarrow z)$ because the policy of the player is not computed in these probabilities:

$$v_i(I, a) = \sum_{a' \in A(I_a)} \sum_{z \in Z(I_a)} \mathcal{P}_{-i}(I_a) \mathcal{P}_c(I_a \rightarrow z) u(z) \quad (3.6)$$

$$= \sum_{a' \in A(I_a)} v_i(I_a, a') \quad (3.7)$$

This gives the first recursive call of the algorithm. For (ii), $\mathcal{P}_{-i}(I) \mathcal{P}_{-i}(I \rightarrow I_a) = \mathcal{P}_{-i}(I_a)$ changes but $\mathcal{P}_c(I \rightarrow z) = \mathcal{P}_c(I_a \rightarrow z)$ stays the same because action a is chosen.

$$v_i(I, a) = \sum_{a' \in A(I_a)} \sum_{z \in Z(I, a)} \frac{\mathcal{P}_{-i}(I_a)}{\mathcal{P}_{-i}(I \rightarrow I_a)} \mathcal{P}_c(I_a \rightarrow z) u(z) \quad (3.8)$$

$$v_i(I, a) = \frac{\sum_{a' \in A(I_a)} v_i(I_a, a')}{\mathcal{P}_{-i}(I \rightarrow I_a)} \quad (3.9)$$

Which gives the second recursive call to allow the computation of the counterfactual regrets during the traversal of the tree. The same thought process can be applied to find the recursive call for $v_i(I)$. Subtracting these values will calculate the regrets at the decision nodes for player i .

The algorithm is run a certain number of times, optimizing the strategies of different players each time. The end result is a policy for each player that reaches a **Nash equilibrium**. Put in a simple way, it is a back and forth thinking mechanism of: If my opponent acts following a certain policy, my strategy should be updated a certain way to minimize my overall regrets. The same thought process is applied to all players repeatedly. This leads to an average strategy or policy that minimizes regrets for all players. The equilibrium strategy is such that if a player tried to diverge from his own strategy to gain more it would only incur a loss in theory.

For example, if we define for the tree in figure 3.2 the policy of P2 and the Chance nodes to be:

	left	right
P2	0.5	0.5
Chance Node	0.5	0.5

For the State node PC1, the average outcome can be computed as the sum of the outcomes weighted by their probability of being reached. Thus if we define the utility of a player to be $U_p = score_p - score_o$, the utility score of PC1 for Player 1 is $U_{PC1} = 0.5 * 0.5 * (-7) + 0.5 * 0.5 * 7 = 0$. The same can be done for every other chance state:

PC1	PC2	PC3	PC4
0	-1.25	0.25	0

Assuming an uniform policy for player 2, the $P2$ expected utility can be computed using the expected utility from PC_i .

$$\begin{array}{cc} P21 & P22 \\ 0.5 * 0 + 0.5 * (-2.5) = -1.25 & 0.25 \end{array}$$

The counterfactual regrets for the action left and right at $P21$ and $P22$ can now be computed:

$$\begin{array}{cccc} P21 \rightarrow left & P21 \rightarrow right & P22 \rightarrow left & P22 \rightarrow right \\ U(PC1) - U(P21) = 1.25 & U(PC2) - U(P21) = -1.25 & 0.25 & -0.25 \end{array}$$

Note that these regret values are the ones from player 1's perspective and due to the nature of the utility function, the regrets from player 2's point of view are the exact opposite.

For these particular states, a policy can be computed using **regret matching**[12]. The idea behind **regret matching** is that if we regret not playing an action, it should be played more. So to compute a policy, the positive regrets are used. All the actions that have positive regrets will be considered and weighted by their regrets. Mathematically speaking, the probability of taking action a after regret matching will be:

$$p(a) = \max(0, cfr(a)) * \frac{cfr(a)}{\sum_n \max(0, cfr(a_n))} \quad (3.10)$$

Where $cfr(a)$ is the counterfactual regret of taking action a in the current info state, and the $\sum$ is the sum of all the positive regrets used to weight the regretted action to their regret value.

Using this new concept, policies for states $P21$ and $P22$ can be expressed as:

	left	right
P21	0	1
P22	0	1

In practice the policy taken is the average policy but we can see that this policy is what player 2 will tend towards to maximize its wins over the course of many games.

Finally the expected outcomes for $P11$ can be computed. However, these outcomes will vary depending on the policy considered for player 2. For the uniform strategy, the expected outcome have already been computed, so they can

be used to compute the expected outcome of $P11$ assuming player one also starts with a uniform strategy. This gives us an expected utility of $U(P11) = 0.5 * (-2.5) + 0.5 * 0.5 = -1$ with the current strategy. This allows the computation of the regret for each action:

$$U(P21) - U(P11) = -1 \quad \begin{matrix} P11 \rightarrow \text{left} \\ P11 \rightarrow \text{right} \end{matrix}$$

Assuming the freshly computed policy for $P21$ and $P22$, the utility values would have been $U(P21) = 0 * 0 + 1 * (-2.5)$ and $U(P22) = 0 * 0.5 + 0 * 0$. This result will push for this final policy:

	left	right
P11	0	1
P21	0	1
P22	0	1
Chance	0.5	0.5

Intuitively this seems like the most reasonable policy, because it leads to player one and player two taking +4 on a 50% chance at each turn. Any other move will lead to the other player taking a definitive advantage.

This example also shows how the policy attributed to other players affects the perception of the current player. In this case, the right-most strategy would always appear superior from the beginning but in more complex games, the action of updating strategies over and over again can show a lot of variance in the play styles across the training phase.

In practice, the game tree is traversed once for each player to update each player strategy. Still following the idea that one player update his strategy, then the second player adapts to that and the next player adapt to both, and so on. This is repeated a number of times and both the regrets and strategies are added and taken into account of into an average value at each decision point, so that the policy does not fluctuate too much and can smoothly converge to a **nash equilibrium**.

3.4.3 LIMITS OF CFR

CFR is proven to be quite effective but as it is demonstrated here, the process is quite tedious even for a single iteration in a little tree. Considering this algorithm on deep and broad game decision trees is difficult because of its time and memory consumption. If we take for example the Belote game, starting from the start of the play phase and assuming a mean branching factor of 4, we would already need to explore $4 * 10^{10}$ nodes for a single iteration for a single player, this has been proven to be nearing the limite of what CFR is capable of handling correctly[13]. To overcome this, there exist a few options that will be discussed below to approximate the expected outcomes and regrets without exploring the full breadth of the decision tree.

3.5 ESMCCFR & OSMCCFR

As it has been discussed earlier in this section, MCTS is an algorithm that allows to explore large trees by selecting worthy nodes to explore in order to redirect the resources used for the search in meaningful paths. MCCFR is a mix of the concepts of MCTS and CFR minimization. The idea is to compute the regrets for meaningful nodes and use the powerful strategies found by CFR algorithm with the smart abstraction done by MCTS.

3

The main difference with CFR is the coverage of the tree. Which node is useful, and how to explore enough of the tree for the abstraction to be relevant are the main problems. For this purpose, **External Sampling MCCFR** and **Outcome sampling MCCFR** come into play.

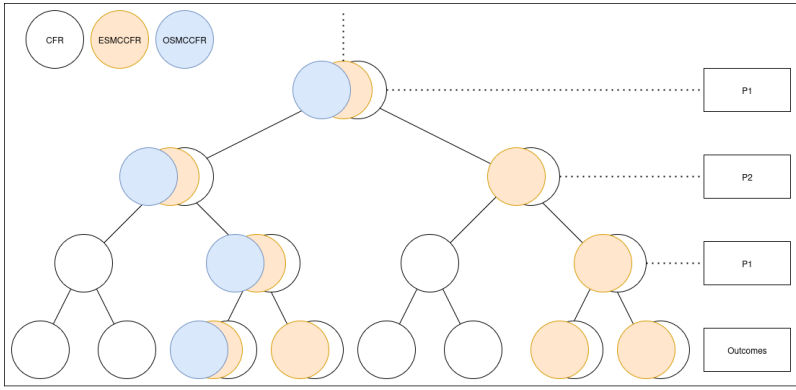


Figure 3.3: Coverage of a single iteration of CFR's algorithms

3.5.1 EXTERNAL SAMPLING

External sampling consists of sampling actions that are outside of the power of decision of the current player. As shown on figure 3.3, from the perspective of player 1, the info states visited follow this rule: expand to all possible actions at decision nodes, sample one from the opponents node. There are many ways to sample nodes, such as those previously described for MCTS and ISMCTS in order to both explore unvisited nodes and promising ones. One that has been proven to be effective is to simply use the current policy of the opponents and randomly sample using the weight each action carries in the current strategy. This allows to "prepare" for the worst case scenario. While this tends to highlight the more promising nodes, the exploration is done through the iteration where the regrets and strategy are updated for other players, where the algorithm explores all the possibilities for other players and samples the ones for the main player with its current policy.

In this new sampling setting, equation 3.3 and 3.4 can be rewritten as:

$$v_i(I, a) = v_i(I_a) \quad (3.11)$$

$$v_i(I) = \sum_{a \in A(I)} P_i(I \rightarrow I_a) v_i(I, a) \quad (3.12)$$

The probabilities fall since they have been sampled. Thus the expected value of taking action a becomes the expected value of $v_i(I_a)$. The recursive call is much simpler in this case. The counterfactual value is back propagated and serve as baseline in its own node and as expected value of reaching a node in ancestor nodes.

3.5.2 OUTCOME SAMPLING

As the name suggests, outcome sampling takes a single episode of the game, from the current info state to a terminal state. Since it doesn't explore as much the width of the tree and relies on the sampling policy to get relevant and varied outcomes, it can't rely on sampling with the current strategy alone. Its solution for the bandit problem is to add an uniform probability to all the nodes and then weight it with the policy, to achieve a policy that has higher probability for promising nodes and some probability to explore others. This makes the algorithm cost less on each iteration but makes it also less powerful per iteration.

Mathematically speaking, the previously defined expected values can in this case be re-written as [14] :

$$v_i(I, a) = \frac{P_{-i}(I \rightarrow z)}{q(z)} u(z) \quad (3.13)$$

for a single terminal state z . The value is reached at a terminal node and back propagated recursively using the proper probabilities at each nodes.

CFR, ESMCCFR and OSSMCCFR have their strengths and weaknesses that are summarized in table 3.1. While CFR is the most complete and powerful version of the algorithms, it requires to explore the entirety of the tree and for games such as Belote, it is computationally impossible for the early stages. ESMCCFR is already applicable to many more games, but its cost is still quite high, even for games like Belote that only uses 32 cards of the regular 52 sized deck card games. As the tree becomes deeper and wider, OSMCCFR becomes the only possible viable variant of these CFR algorithms. In terms of tree coverage, CFR can be compared to a Breadth Search, OSMCCFR to a Deep Search and ESMCCFR to something in between.

	Cost/Iteration	Coverage	Use case
CFR	Very high	Full	Small solvable games
ESMCCFR	High	Partial	Large games that need abstraction
OSMCCFR	Low	Low	Games with deep game trees

Table 3.1: CFR algorithms and their characteristics

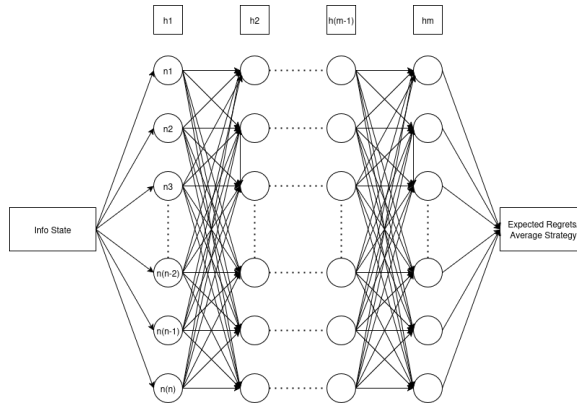


Figure 3.4: Neural network's blackbox

3.6 DEEPCFR

In the CFR abstraction category, there exists a neural network alternative beside external sampling and outcome sampling. The DeepCFR algorithm mixes the concepts of deep neural networks and counterfactual regrets. Its purpose is to model regrets and policies so that a playing agent can evaluate the best course of action for unknown states based on its info state.

To achieve this, it traverses the game tree using an external sampling coverage, collecting info states and their counterfactual regrets. The policy used for sampling is the policy derived from using regret matching on the regrets computed from the neural network in training. Just like ESMCCFR, the diverse coverage is assured by taking players as the "principal player" in turns to explore a variety of best options. Once the traversals of the trees are finished for every players, the collected data is used to train the advantage network, which goals is to output a regret vector using an info state as input. This process is repeated multiple times to mimic the "explore/update strategy" of CFR. The policies used during these numerous tree traversals are also stored and finally used to train a policy network. This network is the final product of the algorithm and returns a policy that should minimize regrets for a given info state in theory.

The way this algorithm works is very close to offline ESMCCFR. The main difference is that ESMCCFR relies on online calculation to compute a strategy for a given info state, unless it has already been visited in a previous iteration, in which case, it can still be good to redo a search for better results, while DeepCFR relies on lots of early off-line data and can work with new data thanks to the neural network's ability to give outputs with unprecedented inputs.

3.6.1 NEURAL NETWORKS

To better understand DeepCFR, one needs to grasp how a neural network operates. It is built upon lots of complex mathematical principles and formulas from derivative calculus, but this section will hover above the main concepts that are needed to understand DeepCFR.

Figure 3.4 shows a simple representation of one of the neural networks used by DeepCFR. Its architecture can be seen as a blackbox function that transforms an info state into a usable vector, either the expected regrets or the average policy for the info state provided at the entry of the black box.

$$\text{Expected Regrets} = \text{Neural Network}(\text{Info State}) \quad (3.14)$$

The neural network works as a function approximation for a complex function. In our case, the function that we want to approximate is the expected regret for an info state and its legal actions, and the average strategy at a nash equilibrium for a game.

The network is composed of layers, from 1 to m on figure 3.4, and each layer is composed of neurons, from 1 to n in the example. A network structure does not need to be homogeneous, but for simplicity, it is assumed to be in the example. In order to produce an output, the input traverses the neural network. Each neuron of a layer h is a linear combination of the neurons of layer $h - 1$. The more neurons by layer and the more layers there are, the more complex the function to approximate can be. Scaling is of course important and more does not always equal the better depending on the situation.

In order to figure out the weights of each neurons, the network is trained using simulated samples. Knowing the input and the expected output, the network can start the training phase by running the network with the sampled inputs, and minimizing the error metric, for example *MSE* with the output and expected outputs using derivative theory.

The idea behind the tuning of the network is that at each layer, it is possible to compute the impact of the neurons of the previous layer. In other words, the derivative of the layer in regard to each previous neuron. Using this, the error can be back-propagated through the network, updating neurons weights to minimize the error during training.

The training can vary in effectiveness depending on the available data, the number of times the data is propagated and back-propagated as well as the learning rate of the network. It is hard to give meaning to the value of neurons weight inside the neural network, hence the reason it is considered a black box. However, neural networks give good results across many different applications and, if tuned correctly, are very good approximations of difficult functions, such as the expected Regrets for a game or its average strategy.

3.6.2 DEEPCFR FOR IIG's

Across the literature, it has been shown that DeepCFR is one of the prominent algorithms for imperfect information games[15]. As a whole, the number of neural network applications is growing by the day. Its biggest asset is that if the approximation works for a large scale game, the high cost of the offline computation that is constituted by the training and the data sampling is compensated by its high efficiency online. Opposed to offline OSMCCFR and other MCCFR variants, DeepCFR approximation function can provide an estimate of unforeseen states that is not an average policy. The main goal of the neural networks is to uncover underlying mechanisms and strategies of the game through repeated simulations, such that when it comes across a new state, it can deliver a well thought policy because it recognizes similitudes with previous states it has been working on.

4

OPENSPIEL

4

A Framework for Reinforcement Learning in Games

OpenSpiel[16] is a framework that contains both algorithms and games. It allows to test different decision algorithms by generalizing games and agents. Doing this renders this framework usable as a base in which we can plug in new games implemented following the OpenSpiel generalization to test solving algorithms and/or plug in new agents too.

4.1 IMPORTANT CODE**4.1.1 SPIEL.H**

Example 4.1: GAMETYPE CONSTRUCTOR

```

1 struct GameType {
2     std::string short_name;
3     std::string long_name;
4     Dynamics dynamics;
5     ChanceMode chance_mode;
6     Information information;
7     Utility utility;
8     RewardModel reward_model;
9     int max_players;
10    int min_players;
11    bool provides_information_state_string;
12    bool provides_information_state_tensor;
13    bool provides_observation_string;
14    bool provides_information_tensor;
15    GameParameters parameter_specification;
16    bool default_loadable;
17    bool provides_factored_observation_string;
18 }

```

Each game must have a type that contains the name, different *game theory* characteristics, flags for the agents to know how to behave and some more parameters to define depending on the game.

Example 4.2: GAMETYPE CONSTRUCTOR

```

1 struct GameInfo {
2     int num_distinct_actions;

```

```

3  int max_chance_outcomes;
4  int num_players;
5  double max_utility;
6  double min_utility;
7  double max_utility;
8  absl::optional<double> utility_sum;
9  int max_game_length;
10 }

```

GameInfo class holds information that don't fit in the **GameType** class because they may depend on the **GameParameters** `parameter_specification` content.

Example 4.3: STATE CONSTRUCTOR

```

1  class State {
2  public:
3      virtual ~State() = default; // default destructor
4
5      // basic constructors
6      State(std::shared_ptr<const Game> game);
7      State(const State&) = default;
8
9      virtual Player currentPlayer() const=0; // return the current player
10
11     // apply an action and update the state accordingly
12     virtual void ApplyAction(int action_id);
13     virtual void ApplyActionWithLegalityCheck(int action_id);
14
15     // return the legal action of a player. If it's not his turn an empty list and
16     // if it's a chance node, the possible outcomes.
17     virtual std::vector<Action> LegalActions(Player player) const {
18         if (!IsTerminal() && player == CurrentPlayer()) {
19             return IsChanceNode() ? LegalChanceOutcomes() : LegalActions();
20         } else {
21             return {};
22         }
23     }
24     // return the legal actions of the current player
25     virtual std::vector<Action> LegalActions() const = 0;
26
27     virtual std::vector<double> Rewards() const {
28         if (IsTerminal()) {
29             return Returns();
30         } else {
31             SPIEL_CHECK_FALSE(IsChanceNode());
32             return std::vector<double>(num_players_, 0.0);
33         }
34     }
35     virtual std::vector<double> Returns() const = 0;
36     std::vector<float> ObservationTensor(Player player) const;
37     std::vector<float> ObservationTensor() const {
38         return ObservationTensor(CurrentPlayer());
39     }
40     void ObservationTensor(Player player, std::vector<float>* values) const;
41 }

```

The state constructor is the one that will shape the behaviors of the games and contains most of the information that will be used to devise game agents. It works by providing legal moves at its current state and contains the transition function *DoApplyAction*(`< Action > action`) that transforms a state s to s' through an action a .

$$s \xrightarrow{\text{DoApplyAction}(a)} s' \quad (4.1)$$

It also contains a return function for *step rewards*, *value returns* (contains the utility for each player at the end of a game) and *information* and *observation tensors*. All of these have uses in game agent designing.

4.2 BELOTE IN OPENSPIEL

As said just above, most of the important parts of what is happening is in the State structure, responsible for the behavioral patterns and the information of a game.

The states can be represented in different fashions depending on the game. For instance, if we have a game of *tic-tac-toe*, the only important components are the current grid, how it is filled and which player is to play. If we take a much more complex game such as the *Belote*, it becomes more ambiguous. As we have seen in the rules, if we decide to describe a state by its history (the set of actions that leads to the current state), which would be the most precise way to do so, we would have an astronomically large set of possible states and it would lead to long and wide searches or calculations for a game agent.

It also becomes interesting to discuss the scope of the agent. On OpenSpiel, most games are defined as step by step actions taken by the players or by the game itself. Which means that depending on the information we want to extract from a state, we could extract a lot more information that a player agent should be allowed to know. In the case of the *Belote* game, the most sensitive information would be the hands of other players and the cards left in the deck in the dealing phase.

4.2.1 IMPLEMENTATION

The implementation can be seen as a transcription of the rules in the C++ language with some added features to help the development of playing agents. The most important functions are the *DoApplyAction(< Action > action)* and the *LegalAction()* ones. The *ObservationTensors(< Player > player)* has its own value that will be discussed later.

Example 4.4: DOAPPLYACTION(<ACTION> ACTION) FUNCTION

```

1 void BeloteState::DoApplyAction(Action action) {
2     switch (phase_) {
3         case Phase::kDealOne:
4             return ApplyDealAction(action);
5         case Phase::kReveal:
6             return ApplyRevealAction(action);
7         case Phase::kBets:
8             return ApplyBetAction(action);
9         case Phase::kDealTwo:
10            return ApplyDealAction(action);
11         case Phase::kPlay:
12            return ApplyPlayAction(action);
13         case Phase::kGameOver:
14             SpielfatalError("Cannot act in terminal states");
15         default:
16             SpielfatalError("Invalid Phase!");
17     }
18 }

```

Example 4.5: LEGALACTIONS() FUNCTION

```

1  std::vector<Action> GameState::LegalActions() const {
2      switch (phase_) {
3          case Phase::kDealOne:
4              return DealLegalActions();
5          case Phase::kReveal:
6              return DealLegalActions();
7          case Phase::kBets:
8              return BetLegalActions();
9          case Phase::kDealTwo:
10             return DealLegalActions();
11          case Phase::kPlay:
12             return PlayLegalActions();
13          default:
14             return {};
15      }
16  }

```

4

Both of the above functions are in fact just dispatch functions that follow the *phase_* variable of the state. This variable can take different values and highlight the core idea of a *Belote* game.

Example 4.6: PHASE POSSIBLE VALUES

```

1  enum Phase {kDealOne = 0, kReveal, kBets, kDealTwo, kPlay, kGameOver};

```

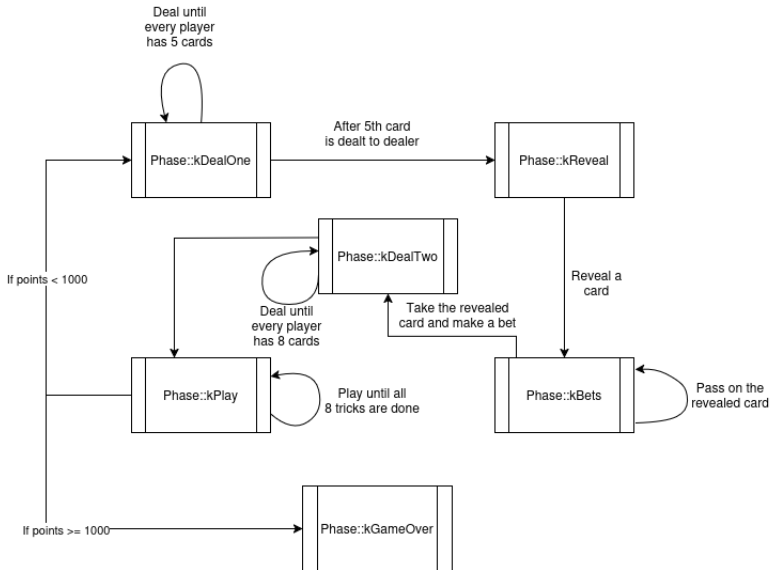


Figure 4.1: Phases diagram

The main principle is that an agent will use *LegalActions()* to fetch the set of possible actions and then for a chosen action *a*, it will call *DoApplyAction(a)* on the state to progress in the game. This set of functions provides the necessary tools to simulate games. Both these functions will behave depending on the value of the *phase_* variables which will be updated when *DoApplyAction(a)* is called, given that action *a* leads to a change

in game phase. The ability to simulate games is the key ingredient in devising playing agents for the game.

The implementation of the game is fairly loyal to the rules described in their own section. The game starts by dealing three cards followed by another two cards to each player, to total hands of five. A card is then shown and the current player can either take it and play it as a trump or pass and wait for the other options of bets: All trump, No trump, and any other suit as trump. The better always gets the card that is shown. Then the dealer fills the hands of each player and they all battle for eight tricks to honor the bet or to sabotage it. Tricks and points are calculated in regard to the bet that has been made. Once the eight tricks are played, the game starts over unless the score threshold is reached.

The main differences are that the shuffle of the cards is done in the dealing phase of the game instead a human biased shuffle before the game traditionally done in the real game and that, opposed to what is shown in figure 4.1, the game is implemented as single rounds of 8 tricks and then repeated as many times as possible until a team reaches an endgame amount of points. Because each round is independent from each other, it makes more sense to evaluate each round as an independent game rather than simulate all the possible dealing phases which contains a lot of hidden information and thus a lot of possible states.

In practice the game has been implemented as a single round of Belote. Algorithms in OpenSpiel rely a lot on the implementation structure of games to access the game's information. The problem with implementing the total game is that the game is structured so that each round is independent from the other. Meaning that combining all the Belote rounds in a single game would make the count of possible states explode. Considering each round as its own game simplifies the problem. Testing is however done on full games by having agents play as many rounds as needed to reach the winning point threshold, just like in the rules.

4.2.2 OBSERVATION TENSOR

For agent creation purposes, the observation tensor is primordial in an imperfect information game setting. Its role is to convey to the agent "what is player i seeing". Depending on the information tensor, search trees with information states rather than states will be wider or thinner. Learning algorithms such as DeepCFR will be able to use more or less state data to learn strategies.

Figure 4.2 shows the tensor used for the agents. It is composed of 285 one bit encoding value each allocated for a different purpose. It's size is in line with the information tensor of other card games such as Poker[15]. The values of the tensor are allocated as follows :

- **0-31** Hand of the player. Its encoding is in a vector of size 32, the number of cards in the game. For each card in the hand the value is 1, else 0.
- **32-127** Possible hand for each player. These vectors keep in track which card can other players have. The vector is updated in play. When a player plays a card, the associated value in the vectors is set to 0, because the card has been played, the possibility of it being in any hands drop to 0.

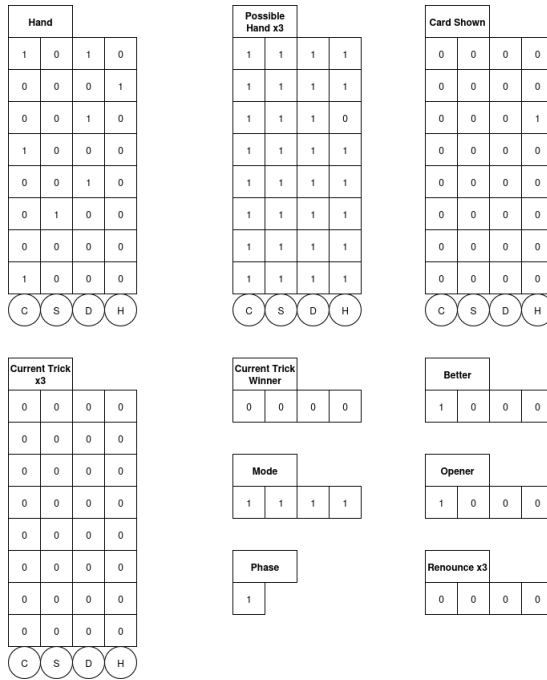


Figure 4.2: Belote's information tensor

- **128-159** Card shown during the betting phase. This allows to have the information during the betting phase. The card is then attributed to a player by setting its probability to be in any other hand to 0.
- **160-255** Current trick in play. 3 potential cards depending of the position of players, each in a vector of size 32.
- **256-259** Current trick winner. Important information for both legal actions and strategies.
- **260-263** Current bet in place. Uses a vector of 4, and set the bit of the trump to 1. No trump and all trumps mode are set by vectors of 1's and 0's.
- **264-267** Better, or which player set the bet. Its purpose is to identify which side is attacking/defending. Also in learning it might help to "guess" that the better has a good hand in the announced trump.
- **268-271** First player of the trick. Needed for trick re-building in resampling. Additional data too.
- **272-283** Renounce values for each other players. Redundant with possible hands but might help.
- **284** Phase bit for betting and playing phase.

Each of these values are kept in memory or are computed in $\mathcal{O}(1)$ complexity. The vectors only need to be built so its complexity is proportional to its size. The vector could have fewer or more information however it must respect one constraint for the agents work to be smoother. Each Info Set, which are determined by the information tensor must contain states that share the same legal actions and that are plausible. The plausible close is not necessary but helps avoiding dead states that are not reachable and reduce noise in agent training. The legal action clause is needed for agents such as MCCFR with sampling because they allocate regrets value only to legal actions. Having states belonging to the same set with different legal actions complicate the computation of regrets, or expected value for an action.

4.2.3 RESAMPLING

Resampling a state is rebuilding it respecting the information tensor of a player. It is needed in order to train the agents on different plausible states to avoid bias. If the agents were trained only on the existing state, they would use knowledge beyond the player's accessible information and it would not be relevant.

The number of possible states can vary a lot depending on the information tensor. If the information only contains the hand of the player for example, there would be factorial value of possible hands for other players and just as many states. This would cause the training to be diverse but not centered on the actual game and available information. In order to facilitate the training and keep the number of plausible states acceptable, the resampling function uses the possible hands vector stored in the information tensor. It then solves an assignment problem by dealing a possible hand of cards to each player under the constraints collected during the game. The problem is solved using a DFS with a branch and bound method using constraint programming concepts[17]. A random branching suffices in this case.

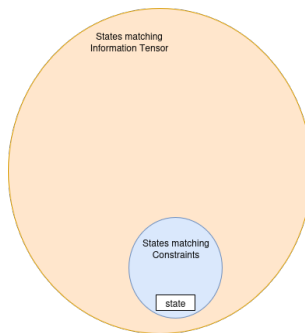


Figure 4.3: Quantity of possible states for a random information tensor

4.3 IMPLEMENTED ALGORITHMS

OpenSpiel offers a large panel of different algorithms thanks to many different contributors. It provides a large number of algorithms for all kinds of games. The ones of interest for this work are ISMCTS, external and outcome sampling MCCFR and finally DeepCFR. In this section, the OpenSpiel implementation of these algorithm will be detailed.

4.3.1 IMPLEMENTATION OF IS-MCTS

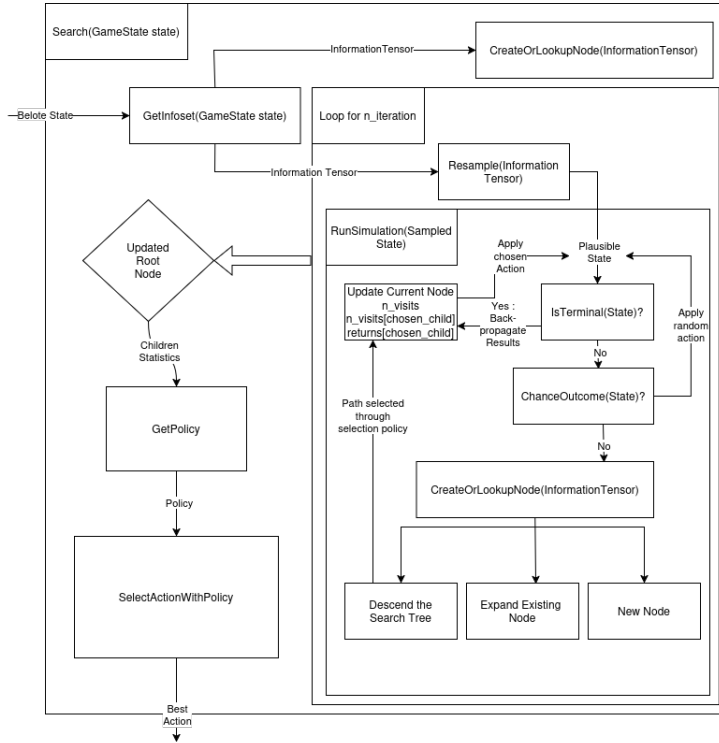


Figure 4.4: Diagram of the IS-MCTS algorithm

The algorithm behaves as described in 4.4 with some slight differences with how it is strictly coded for interpretation. Step by step, here is how it works for an agent that take as input a game state:

- The state has an information tensor matching the current player. This tensor is used to identify the information set it belongs to and allows to either locate an existing information set node or to create a new one. For a very basic implementation, the first step in a search will always be to create the root node, but as soon as it is implemented with a caching mechanism, the root node might be an existing node from a previous search made on an ancestor state of the current one.

- Using the information tensor as basis, the state enters a loop for a defined number of simulations. Each simulation is done on a plausible state computed from the observations of the current player. For each plausible state computed, the following step applies:
 - Depending on the step of the simulation, the tree search applies different actions. If the current state is a terminal node, which means it is the end of this n^{th} simulation, the result of the game is back-propagated among the information sets or nodes of the search tree, all the way up to the root node of the current search. If we come across a random state, such as a "draw a card" situation, the simulation progresses applying a random action following the probability of the situation.
 - For any other node where a player must take a decision, there are three observable cases. If the node is a fresh node, it is evaluated through an evaluator. A random evaluator in this case, it works by randomly executing the rest of the game a fixed amount of times and returning a mean value of the different results from these episodes. For an existing node that needs to expand still, it is expanded. The expansion follows a simple rule: if all the possible moves have been expanded as new fresh nodes, the information set represented by the node has been fully expanded. Otherwise, a new fresh node is formed with an unexplored move and will be evaluated by the random evaluator or if it's a terminal state, will return new results to the tree search. And finally if we are in an already visited node, the simulation continues by choosing an action following the policy used. In MCTS algorithms, *UCT* and *PUCT* are classics to determine what the next step of the simulation should be to balance the explore/profit ratio of each action as described in the MCTS section in the chapter 3.
 - When the policy takes effect and an action is chosen, the visited nodes are updated to keep track of their statistics: Sum of the returns value, number of visits, number of visits of their children. The action chosen is then applied to our current plausible state and the simulation continues. For other cases, the simulation stops there, and another plausible state is computed to restart a new simulation.
- When the simulations is finished, the result is the root node and the descending tree. Each visited node has had its values updated and can provide a temporary policy for the decision taken at their info state.
- Using this policy, an action is selected. It can be through random choices with probabilities updated in regard to the policy, or the action that returned the max value or simply the children that has been the most visited, which should, in theory, be the one that maximizes a good outcome.

There are some key points to take into account in this implementation since they that can alter the results to different extents. Firstly the caching mechanism can be implemented in different fashions. Not at all: Start fresh at each search, not a lot of memory used but

most of the search results are discarded and unused.

Cache a single round: Uses more memory but the search becomes stronger at each step of the round as states that may need to be explored are likely to have been already explored once. So it allows to deepen the search and provides a more informed decision.

Cache every round of the game: It takes about 15 rounds for a Belote game to end so it is slightly heavier in memory but does not change that much. However, the chances of information sets to be the same in two different rounds are very unlikely except at the end of the round where the game is mostly solvable because of the number of different plausible states that can easily be simulated.

4

Cache every game on multiple simulations: We could expect to explore more of the total game tree and for the agent to learn but just as many game cards, this would explode the quantity of memory needed due to the number of possible states. Also, without sufficient games played, the memory usage would grow without bringing sufficient coverage of the game tree to justify it. Assuming we explore 500 nodes per search, in a single game we would have explored around 5000 nodes. For a simulation of 500 different games, this would lead to roughly $2.5 * 10^6$ nodes. To put it in perspective, we would need $5 * 10^{35}$ games to uncover roughly a percent of the tree. Although the info set game tree is smaller than the real one, expecting results from keeping random states in memory seems hopeful.

The **information tensor** is also a factor. The one used is the one described in the previous section about the Belote implementation. It covers a lot of the game information and allows to have precise information set.

Finally, the evaluator used when reaching a new node is of utmost importance. The random rollout evaluator is a domain independent evaluator, it focuses on outcomes and provides an average outcome by sampling random episodes. On average, the time complexity of the ISMCTS agent can be defined in $\mathcal{O}(r \times i)$ where i represents the number of iterations and r the number of times the random rollout evaluator reaches a terminal node to evaluate a node. In practice, r holds a slightly lighter weight, probably because it requires less computation to choose actions randomly and the evaluation always starts in a lower part of the tree than the current state we are trying to evaluate. Simulations show that for the same time complexity, a better distribution of the ratio $\frac{\text{rollout}}{\text{iteration}}$ yields better results.

4.3.2 IMPLEMENTATION OF ESMCCFR

ESMCCFR's implementation has some similarities with ISMCTS because they both share some roots in Monte Carlo Tree search algorithm. Figure 4.5 shows a simple summary of the algorithm.

The tree exploration is nested in two main loops. The main one being the **number of iterations**, how many times the algorithm will loop. The other one is the loop that considers **each player** point of view. As mentioned in the chapter 3, this algorithm evaluates

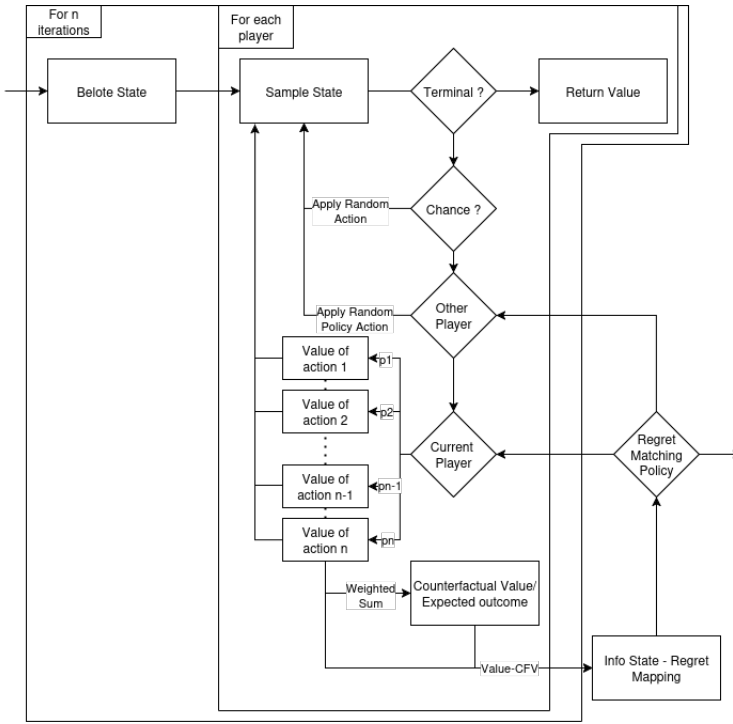


Figure 4.5: Esmccfr algorithm diagram

the best strategy for a player by computing best strategies for all players. So in turn, each player becomes the main focus of the algorithm.

Following the flow chart of the algorithm, the first input is the current state of the game. This state is perceived differently for each player, so the tree search is performed on a plausible state computed from the info state the current player is in. Depending on the state, different things occur, however in case of an agent, the first state will always be one where the current player has to take action.

In said case, the agent evaluates all the outcome values for each legal action for the current info state. These values will come from expanding the current info state to a new state after applying each possible actions and by recursively searching deeper in the tree. Once these values return, they are used with the current policy to compute the **counterfactual value** which is the expected outcome from this node. This value is calculated using the sum of each returned value weighted by the probability each action is taken in regard to the current policy.

$$cfv = \sum_{a \in n} p(a) * v(a) \quad (4.2)$$

Where $p(a)$ is the probability of taking action a in the current policy, and $v(a)$ the value returned by the tree search for this particular path. This counterfactual value is both returned for ancestor nodes and also used to compute regrets.

$$r(a) = v(a) - cfv \quad (4.3)$$

This equation of regrets means that for the current policy, choosing action "a" is a solid strategy if its value is superior to the current expected value cfv . If it is inferior, a negative value for regrets indicates a worse choice than the current strategy. All the regrets for each action a are then added to an Info state / Regret array mapping.

4

This mapping is the data base that contains all the visited info states and their current associated regret values. Its first initialization is set to very low equal values. This allows to have a starting uniform policy with low initial regrets that won't impact the algorithm after its first update. The policy is computed through **regret matching** and used for the final output, the weighting for the counterfactual value computing and the sampling for the adversary turns during the tree search.

To better illustrate how the mapping works, let's consider this example: Assuming player P is in state S and has 4 possible actions, a, b, c, d with an expected outcome of 1, 2, 3, 4. Here is how the Info State map would evolve:

- **Initialization of the state**

Info State	Regrets			
S	a	b	c	d
	≈ 0	≈ 0	≈ 0	≈ 0

- The first policy is a uniform one. This gives a counterfactual value of $cfv = .25 * (1 + 2 + 3 + 4) = 2.5$. With this, we can compute the first set of regrets for this info state.

Info State	Regrets			
S	a	b	c	d
	-1.5	-0.5	0.5	1.5

- The policy found using regret matching using the formula in chapter 3 is the following:

a	b	c	d
0	0	0.25	0.75

The value which had positive regrets has their strategy probabilities computed with $p(a) = \frac{r(a)}{\sum_{i \in \pi} \max(0, r(i))}$. With these, the counterfactual value can be computed : $cfv = 0.25 * 3 + 0.75 * 4 = 3.75$. This gives the following regrets for the second iteration:

Info State	Regrets			
S	a	b	c	d
	-2.75	-1.75	-0.75	0.25

These regrets are finally added to the existing ones. If the policy used is evolving in the right direction, it will converge to the optimal one. However if it is not, it will spread regrets to old actions that had negative regrets. Obviously, for this example, the policy will converge towards a probability of 1 for the best action d .

- At the end of the second iteration, the info state/regret mapping for the current state will look like:

Info State	Regrets			
S	a	b	c	d
	-4.25	-2.25	-0.25	1.75

The probability to choose d will be 1 and the regret of the other actions will keep diminishing, while d will stay the same because the expected outcome will be equal to the sampled one.

Now back to figure 4.5. If the state's current player is not the update player, a single action from the legal actions is sampled from the current player's policy and a recursive call on the progressed game state is called. The same thing occurs for a chance node which occurs when dealing cards for the Belote game for example.

Once the algorithms reach a terminal state, the value is returned and propagated through all the previous call of the tree traversing function. On its way, the value will help compute the counterfactual value of the last decision point, which will in turn be returned back to the root node of the first call.

4

4

4.3.3 IMPLEMENTATION OF OSMCCFR

OSMCCFR’s implementation is closer to ISMCTS than ESMCCFR is. However, they all are still based on the original MCTS algorithm. The main difference between external sampling and outcome sampling is the coverage of each iteration and thus the use cases for each algorithm. OSMCCFR can be is a kind of ISMCTS but with regrets instead of expected values.

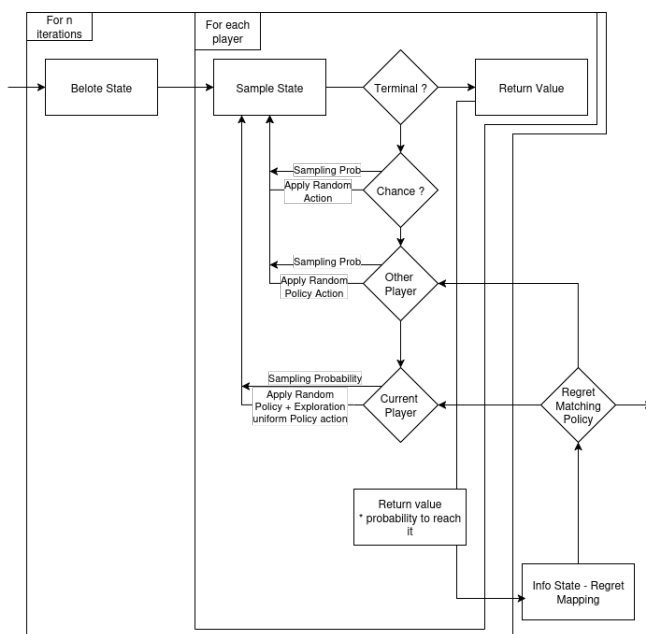


Figure 4.6: OSMCCFR's algorithm diagram

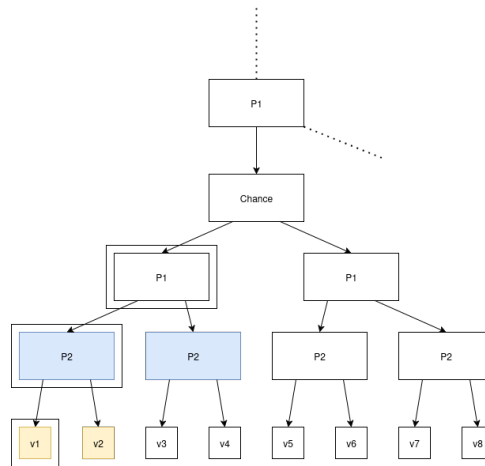


Figure 4.7: Game tree with OSMCCFR

Figure 4.6 shares a lot of similarities with figures 4.4 and 4.5. MCCFR based on outcome sampling behaves like the external sampling one until it reaches the sampling process of the player it's updating. At this point, instead of exploring each possible action and weighting their return values by the probability of the policy, outcome sampling samples only one action following the current policy weighted by a uniform policy that ensures a sufficient exploration rate in order not to trap the thinking process into a single good action.

This uniform sampling policy is needed in outcome sampling and not in external because policies, depending on the cases and the execution of the algorithm, are computed with regret matching. At many points during the iterations, policy will set null probabilities to actions, not because they are generally bad, but because they are bad in the current strategy profiles or because they have not been explored enough yet. This uniform sampling allows to solve the bandit problem in this case.

Example 4.7: OSMCCFR explore probabilities

```

1 policy = self._regret_matching(infostate_info[mccfr.REGRET_INDEX],
2                               num_legal_actions)
3 if cur_player == update_player:x
4     uniform_policy = (
5         np.ones(num_legal_actions, dtype=np.float64) / num_legal_actions)
6     sample_policy = self._expl * uniform_policy + (1.0 - self._expl) * policy

```

Because it samples a single outcome per iteration, each outcome is upscaled by their sampling probability before being multiplied by the probability of taking a decision in the current strategy profile. The same principle is applied to the estimated value of a node, where it is upscaled by the sampling reach of the node before being multiplied by the actual reach of the opponent on their node, or of the reach of the player on his node. The formulas for the expected outcomes for taking an action or for a single node are described in chapter 3.

Figure 4.7 shows a decision tree with a chance node and two players nodes. Assuming we were at framed node P2, updating the P2's strategy. In an external sampling setting, V1 and V2 would be sampled and the expected value of P2's node would then be the weighted sum of these values. Later used for regret computation and then regret matching to devise a strategy. In outcome sampling, only one of these values is sampled, and to compensate for the non sampled values and the times it is not sampled, the sampled value is scaled by its sample probability.

Assuming strategy P_2 at this node:

V1	V2
0.2	0.8

4

Assuming a uniform sampling strategy and that V1 is sampled. In external sampling, the expected value would be the weighted sum of all sampled values, however here there is only one. To compensate the sampling, an unbiased return value is computed by calculating $\frac{V_1}{p_{V_1}}$. For a uniform sampling like in this example, this can be summarized as "Let's count this value twice since it has only 1/2 chance of being sampled". In other sampling schemes such as the one in code 4.7, it allows to represents more values that are less likely to be sampled, a problem that does not occur in external sampling. All in all, when the node is reached enough times, this result is the same as having sampled every sample and weighing them by their sample probabilities.

The expected value is now computed by multiplying the unbiased outcome by the probability of choosing it in the current policy. Which gives us the value estimate $V_e = V_s * \frac{p(V_s)}{s(V_s)}$, where $p(V_s)$ is the policy probability of sampling V_s and $s(V_s)$ the sampling probability.

To compute the counterfactual regrets, the counterfactual expected value is computed by multiplying our expected value by the opponent reach probability as stated in chapter 3. And then by up scaling by the sample reach, following the same principle as for the sampled value. The counterfactual value can be written as:

$$cfv = V_1 \times \frac{p(V_1)}{s(V_1)} \times \frac{p_o(P_2)}{p_s(P_2)} \quad (4.4)$$

where the term $\frac{p(V_1)}{s(V_1)}$ is the local weighing and bias handling of the sample policy and the current policy, and $\frac{p_o}{p_s}$ is the global one on the whole tree which weights the value by the probability of the opponent playing the sequence of move that leads to this state and up scaling by the sampling bias across the history generated.

In the example, we can define starting from the above most node P1 the following probabilities:

The following probabilities can be explained literally as:

	my reach	opponent reach	sample reach
P1	1.0	1.0	1.0
Chance	$p_1(C)$	1.0	$s(C)$
P1 (framed)	$p_1(C)$	$p_c(P_1(\text{framed}))$	$p_c(P_1(\text{framed})) * s(C)$
P2	$p_1(C)p_1(P_2)$	$p_c(P_1)$	$s(P_2)p_c(P_1)s(C)$

- my reach: The probability that I played the action to reach the current state
- opponent reach: The probability that the opponent and that the chance node played the actions in order to reach the current node
- sample reach: The global probability of reaching this node with the current sampling profile

4

The recursive call of the algorithm differs depending on whether the current node is the player's, an opponent's node or a chance node.

```

1  # a chance node :
2  return _episode(state.apply_action(action), update_player, my_reach = my_reach,
3                opp_reach = opp_reach * prob(action), sample_reach = sample_reach *
4                prob(action)
5
6  # an opponent node with its current policy :
7  return _episode(state.apply_action(action), update_player, my_reach = my_reach,
8                opp_reach = opp_reach * policy(action), sample_reach = sample_reach *
9                policy(action)
10
11 # update player where policy != sample_policy :
12 return _episode(state.apply_action(action), update_player, my_reach = my_reach
13               * policy(action), opp_reach = opp_reach, sample_reach =
14               sample_reach * sample_policy(action)

```

The counterfactual value computed, it can now be used to compute the regrets for each action. As described in the other CFR algorithms, regret can be expressed as the difference of the sampled value and the expected one. So for the counterfactual regret, we use the counterfactual value of each possible action and the counterfactual expected value computed.

$$cfr_i = \frac{V_i}{s(V_i)} \times \frac{p_o(P_2)}{p_s(P_2)} - cfv. \quad (4.5)$$

By simplifying and developing cfv using equation 4.4 and then factorizing, the regret for action i can be written as:

$$cfr_i = \frac{V_i}{S_i} (1 - p(V_i)) \quad (4.6)$$

For unsampled actions, the computed regret is $-cfv$, and for the sampled action the regret found is proportional to $(1 - p)$ with the current policy. This means that a good value and a low probability in the current policy will create a larger regret for the current action and incur a larger negative regret on the other actions.

TRAVERSE TREE FUNCTION

Figure 4.8 shows the traverse tree function of DeepCFR[18]. As mentioned above, it is based on external sampling so it has many similarities with figure 4.5. As per usual in tree walking functions, there are different cases depending on the node. If it is a chance node, the tree expands a sampled outcome. If it is an opponent node, the tree also expands a sampled outcome, except here the sampling is done with a computed policy by the advantage network. The advantages (regrets) are then passed through a mask for the legal actions and through regret matching to compute a sampling policy. This policy is stored inside a policy buffer that stores <Info State, Policy> pairs that will be used to train the average policy network.

Note that for the precedent MCCFR algorithms, the legal action mask was not needed. Indeed, an <info state, policy or regrets> pair was dictated by the rules of the game, so each info state has its own set of legal actions. In the DeepCFR's algorithm case, the advantage network has to work on all possible states and thus return an estimated advantage value for each possible action in the game. This vector is then passed through a mask to output a policy fit for the current info state.

Also note that in the other MCCFR algorithms, the state examined is often a sampled state. In OpenSpiel they were defined as offline algorithms so that was not initially the case, but this change will be discussed in the section 5 of this thesis. In the case of DeepCFR, it is evaluated as a pure offline algorithm, meaning that it is not needed to go through the re-sampling phase. The training is done in an "omniscient" manner because it is assumed that the training phase will cover a wide enough range of info sets to be populated by different state values. In Online decisions, it is imperative to not use a fixed state and use plausible state resampled from the info state to avoid the bias of acting upon a single belief of the current initial state.

Back to the traverse tree function, the same difference applies for the player's node. It samples all the actions but to compute the expected value, the used weights are the ones computed from the current advantage network with regret matching. The expected value is then back propagated to the previous recursive call. The same thing happens when a leaf node is reached. Both of these values are used to compute other expected node values, which are then used to compute the sampled regrets, which will be different than the regrets computed from the network, unless it already converged. These regrets are then stored in the advantage buffer that stores <Info States, Regrets> pair to use on the network training.

SOLVE

The solve architecture is actually very straightforward once the traverse tree function has been detailed.

Example 4.8: Solve function of DeepCFR

```

1  def solve(self):
2      """Solution logic for Deep CFR."""
3      advantage_losses = collections.defaultdict(list)
4

```

```

5  for j in range(self._num_iterations):
6      for p in range(self._num_players):
7          for k in range(self._num_traversals):
8              self._traverse_game_tree(new_initial_state(), p)
9              adv_loss = self._learn_advantage_network(p)
10             advantage_losses[p].append(adv_loss)
11
12         self._iteration += 1
13         # Train policy network.
14         policy_loss = self._learn_strategy_network()
15
16     return self._policy_network, advantage_losses, policy_loss

```

Code 4.8 shows a three loop implementation that take into account different things. Firstly, the number of traversals. This can be seen as the data collection phase. For *num_traversals* times, the algorithm traverses the tree with external sampling, collecting regrets and opponent policies. The collected policies won't be very meaningful at first, but as the advantage buffer becomes more populated, the policies derived from it will become more relevant and replace the first ones.

The second outer loop is the CFR part. After the traversals, the advantage network, which predicts the regrets, is updated using the new populated regrets. An *advantage_loss* metric is also computed to keep track of mean square error (MSE) of the network and the sampled used for training. This part is the one discussed in previous sections, where the regrets for a player are collected and used to update the strategy so that other players can learn to play against the updated strategy.

Finally the last loop serves the purpose of training the average policy network. It is trained based upon the $p * k$ traversals and the policies collected during this time.

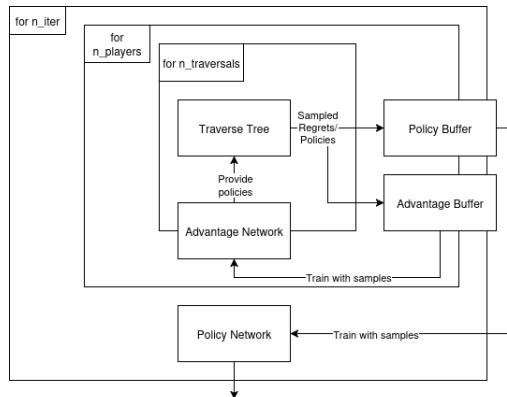


Figure 4.9: DeepCFR diagram

Figure 4.9 shows a larger picture of the algorithm. The end product is the policy network that allows to create a playing agent. The main idea is to refine an advantage network so that across all the traversals, the sampled strategies become sharper. These strategies will then be used to train the network that returns an average strategy for a given info state. All of this, based on the principle of counterfactual regrets minimization.

TRAINING AND RESERVOIR BUFFETS

To manage the space complexity, the algorithm uses a data structure called Reservoir Buffet. The main idea is to have a variety of different samples across all the sample collected during all the trainings. However, depending on the capacity one can or wants to use for training, keeping this sample in memory is constraining since the data cannot be directly used for training at reading time. To do that, the data structure used serves as a rolling mat for data. It works by accepting data as long as the capacity doesn't go over the set boundary, but once it does, each new data is treated to either go in or be discarded.

Example 4.9: Add method of a Reservoir Buffet object

```

1  def add(self, element):
2      """Potentially adds 'element' to the reservoir buffer.
3      Args:
4          element: data to be added to the reservoir buffer.
5      """
6      if len(self._data) < self._reservoir_buffer_capacity:
7          self._data.append(element)
8      else:
9          idx = np.random.randint(0, self._add_calls + 1)
10         if idx < self._reservoir_buffer_capacity:
11             self._data[idx] = element
12         self._add_calls += 1

```

As shown in code 4.9, the add/remove mechanism is dependent of the number of times the algorithm has added a data to the buffer. To choose whether the new data replaces an old one, the algorithm chooses a random integer index, if it is within the boundaries of the Reservoir, it is replaced, otherwise the new data is discarded. Following the formula in the code 4.9, as soon as the buffer is full, the probability of replacing starts at one, so that it allows new data to be used in the network training. As the times passes and the algorithm keeps collecting new data the probability will keep decreasing down to 50% when the number of nodes that needs to be saved reaches twice the buffer size, and 10% when it reaches 10 times the buffer size. This allows, when the algorithm enters the last phase of training, to have a variety of early stage states and new states from across many different games.

5

RESULTS

5

In this section different results are presented to advocate for the rest of the thesis. Simulations and collects of data allow us to confirm and infirm theoretical concepts and also to deepen the understanding of both the game and the algorithms.

5.1 ISMCTS

ISMCTS, although being cheap in computation resources, presents satisfying results for a game as complex as the Belote game. The first tests were done on the simplest version of it with the following parameters:

- Playing against a random agent
- Poor evaluator with a random rollout that goes down the game tree only once
- Increasing number of iterations/explored nodes per step taken in game: 10 - 10000
- Cache in between cards but not in between tricks. As explained earlier, it is very unlikely to reach similar states across different rounds, except in the end where the game is easily solvable

As we can see in figure 5.1, the win rate has a steady logarithmic increase following the number of iterations, up to 85% for 10000 iterations. While this result alone doesn't allow for much analysis, it already shows some limits of the algorithm, and the difficulty of the game. We can expect a little increase in performances for a very costly increase in computation resources. Intuitively, this logarithmic line looks much alike a depth/number of nodes behavior and may mean that the ISMCTS agent is doing well when exploring tiny trees, thus at the end of games, but has trouble finding the right moves when the tree search is larger. In order to help the agent, upping the number of rollout for the evaluator for nodes that are high up in the tree proves to be quite efficient.

For the experiment in figure 5.2, the ration $\frac{rollout_count}{n_iter}$ was the variable. A slight increase of the number of rollouts already outperforms a naive agent with less computation

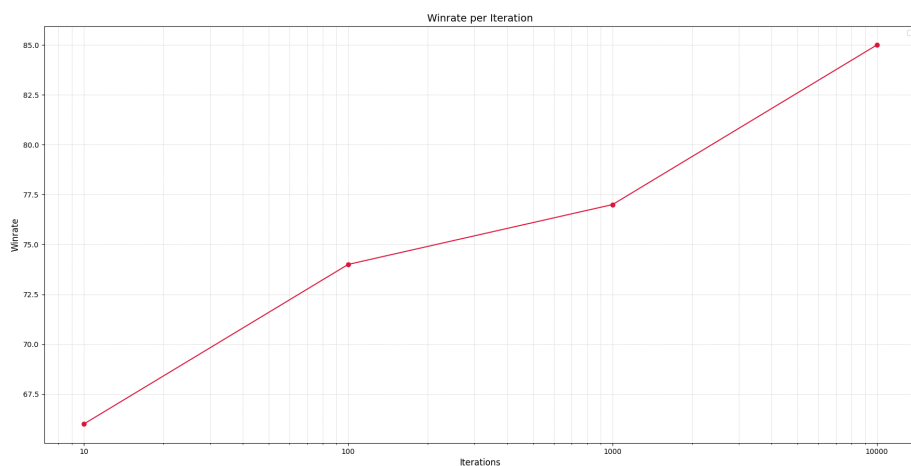


Figure 5.1: Win rate vs number of iterations

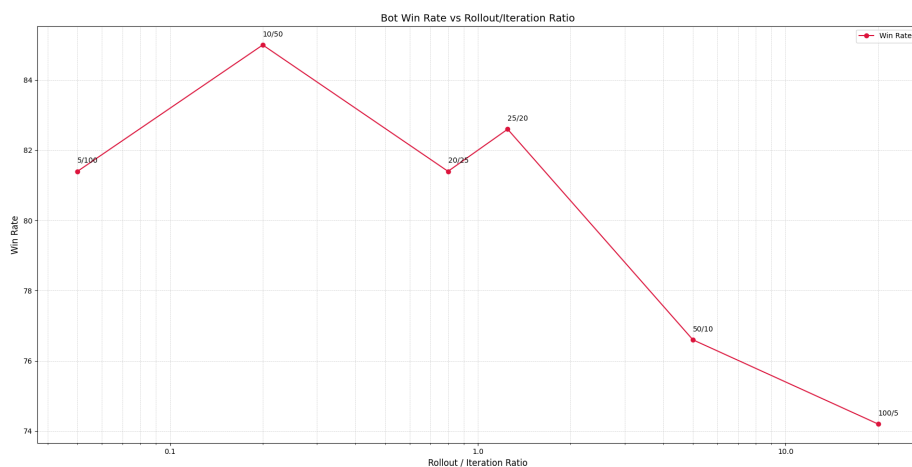


Figure 5.2: Win rate for a different repartition of computation costs between Rollout count and iterations

cost. On another note, it is notable that the increase in rollouts implies the decrease of iterations and thus when the ratio reaches a certain value, increasing this ratio becomes a loss. This can mean one of/or two things :

- Too few iterations hurts the precision of the agent
- Increasing the rollout count becomes less meaningful as it has fulfilled its goal to approximate a state.

The first point is already shown earlier and the second one is likely to be true. While it is not perfect, a few random rollouts can already showcase the average outcome of a game and knowing that the situation is good, a little less good or a little more good is of less importance than knowing it is good, bad or neutral.

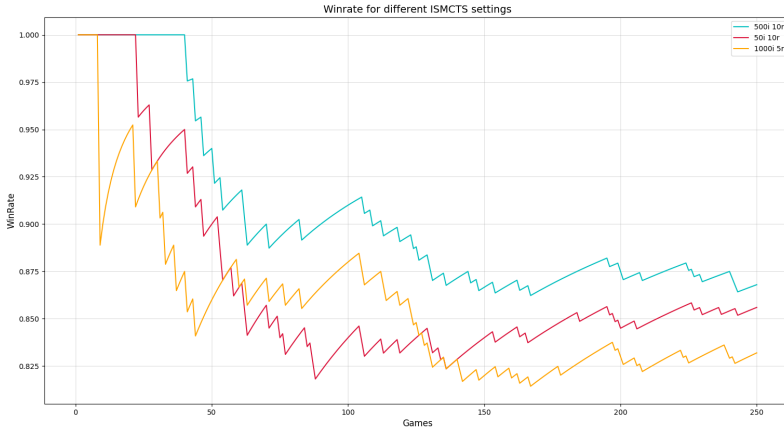


Figure 5.3: Best agents vs Random agent

This last figure 5.3 illustrates well our results. Speaking in $\mathcal{O}$ notation, $\mathcal{O}(5000)$ is showing the worse results with a distribution of 1000 iterations for 5 rollouts for the estimator. Following him, $\mathcal{O}(500)$ is doing better with 50 iterations and a rollout count of 10. And as expected, with 10 times the computational cost, $\mathcal{O}(5000)$ performs slightly better with a repartition of 500 iterations for a rollout count of 10.

To get an agent that constantly wins against a random bot would require probably 100 or 1000 times the current order and thus would be impractical. However, this agent shows that the difficulty probably lies in the higher branches of the tree search and probably on the betting phase of the game for Belote. This concept is most likely not exclusive to this game but the difficulty is greatly enhanced in the particular case of the Belote game. The shape of the tree is wider at the top and makes it hard for algorithms that generally have more exploration in the bottom levels of the decision trees.

The information tensor value also affects the results. Experiments show that ISMCTS performs 10% better for an agent performing 100 iterations per step with a rollout count of 5 when using a good information tensor. It is hard to quantify this increase in performances for more efficient ISMCTS agents but it shows that it matters. The information tensors used where the naive one where the player only knows what it can do and the cards played, and the other one was the one described in its own section in chapter 4. It is probable that the information tensor's abstraction in the naive case allowed the resampling to create too many different states and unreachable ones, causing the agent to perform poorly.

5.2 ESMCCFR

ESMCCFR was tested against the previous ISMCTS agent for relevant results. Its runs against the random agent yielded results that were very close to the limit every time for different settings which made the differences between each agents too tiny to discriminate.

As mentioned in chapter 4, ESMCCFR implemented as an offline algorithm is too weak to cover enough of the tree search to be relevant. In order to make it work as an online algorithm, a few steps are needed to adapt it.

- **Resampling** Because it worked with the intention of covering all the info states, the original OpenSpiel ESMCCFR did not implement resampling and tried to compute a decision tree by repeatedly computing new initial states. Knowing the number of possible initial states for Belote, this method was sure to not bring good results. To implement it in the existing architecture, the main loop was altered to call for the traversing tree method on a sampled state and to call it at every step of the game where the agent needed to provide a policy. Doing this also allows to use the caching mechanism discussed in the MCTS section in chapter 3. Since ESMCCFR depends on policy sampling, the observed nodes in the previous iterations were more likely to be used in future iterations and made caching even more effective.

For computational facility, the runs were made against the second best ISMCTS agent since it had close results for a much lower computational time. The results in figure 5.4 showcases the agents ISMCTS with the following parameters:

- 50 iterations per step
- 10 rollouts per iterations
- Caching during a round

Against the following ESMCCFR agents:

- 2, 4, 6, 8 and 10 iterations for each player
- Caching during the round

The ESMCTS agents show positive results from the 2 iterations mark already. However, when the iterations count increases, the results seem to plateau a little above the 80% win rate marker. It is important to note that the 2 iterations agent already stands several magnitude higher in computation cost than the second-best ISMCTS agent. This showcases once again that to achieve progress in such a complex game, a lot of resources are needed, at least for the current algorithm that does not depend on domain dependent heuristics.

The plateau as well as the imperfect score against random bots can be the consequence of the very high complexity of the initial states. Although better, ESMCTS still fails to grasp what is an optimal decision early on in the game and this leads to unwinnable games even if the action later on during a round are optimal or close to optimal.

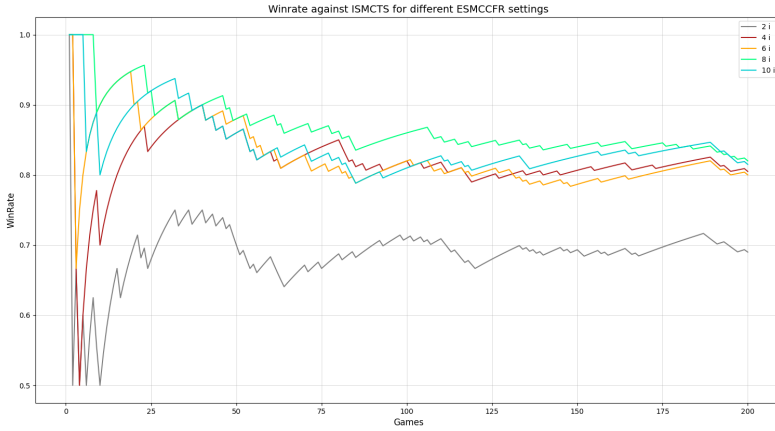


Figure 5.4: ISMCTS vs Different settings of ESMCCFR

It is likely that to break through the plateau, ESMCCFR would require a lot more iterations in early stages of the rounds, which are the most costly in computation resources. It is likely that the first calls of the agent, when the Belote state is in the betting phase, lead to noise more than anything else. The branching factor of the second phase of dealing is around 1320, which makes it, when sampling randomly, very unlikely that the agent in the betting phase simulates a state close to the real one. The most likely case is that with only 10 iterations, which represents 40 total tree traversals, the info state after the dealing phase has not been reached once.

The 20% missing in the win rate probably come from what is described above. This can be derived from the oscillations of the win rates of the 4 to 10 iterations agents that shows that the policy is doing well but reaches a bottleneck somewhere. This also means that the second part of the game is highly solvable, but depends a lot on the first part which makes it the main challenge.

5.3 OSMCCFR

Unsurprisingly OSMCCFR shows results close to ESMCCFR when pitted against ISMCTS. The main difference in these algorithm is the way the regrets are sampled. In ESMCCFR, info state nodes will be visited less times but the regret values inputed will be more precise and have less variance. For OSMCCFR, the idea is to pass through those nodes a lot more time and more efficiently, battling the variance and trying to achieve convergence through a lot of iterations.

The OSMCCFR agents parameters used in the following experiments where:

- 10, 50, 500, 1000, 1166 iterations per player per round.
- Caching between rounds.

For the ISMCTS counterpart, the parameters are the same as for the ESMCCFR measurements. The 1166 iterations count is an approximation to reach the same computation costs of the 10 iterations ESMCCFR, in order to have a fair comparison.

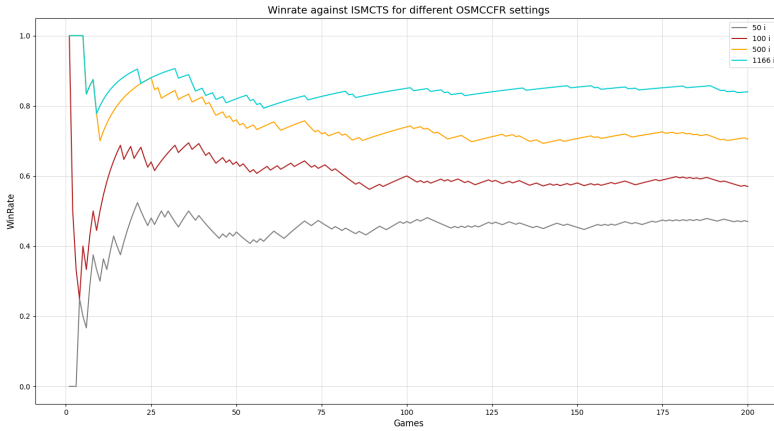


Figure 5.5: ISMCTS vs different settings of OSMCCFR

The first noticeable thing on figures 5.6 and 5.5 is that the win rate evolution is more steady than with ESMCCFR. On figure 5.6, the evolution seems to be logarithmic, akin to the one from ISMCTS. Which makes sense considering that OSMCCFR is a mix of concepts of both ISMCTS and MCCFR.

OSMCCFR performs roughly as well as ESMCCFR for the same computation time. However, it reaches the same limitation of ISMCTS, where, although the growth in performances being steady, it becomes hard to make it viable past the point of 10.000 iterations per step.

Both MCCFR alternatives gives satisfying results however not perfect utilizing different strengths. External sampling returns strong estimation for each iterations but does not evaluate the top nodes of the game tree enough because of the complexity. A single counterfactual value in the betting phase of the Belote game is composed of hundreds and thousands of values from the terminal and near terminal nodes. While outcome sampling traverses each level of the tree equally, passing much more through the same nodes in the early stages, its results are a bit lacking to compute the right policy in the upper nodes.

Comparatively speaking, OSMCCFR shows more promises than ESMCCFR by the evolution of its win rate curve. The problematic of having a high branching factor at the

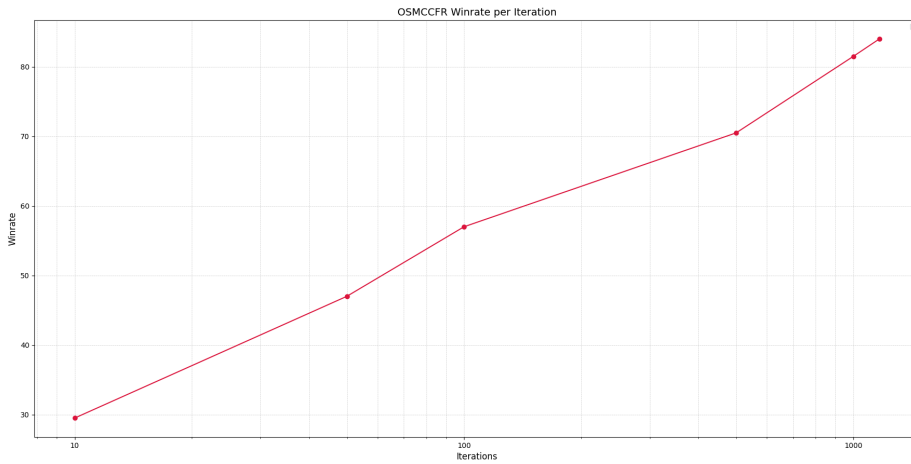


Figure 5.6: OSMCCFR's win rate per iteration

second phase of dealing seems to impact OSMCCFR less as it passes through the betting nodes 100 more times than ESMCCFR. This phase remains a bottleneck even for OSMCCFR, but given the time resources, it might achieve better results. Which is less likely for its external sampling counterpart.

5.4 DEEPCFR

While praised by many, DeepCFR remains a very greedy algorithm and under certain constraints fails to perform. Opposed to its online counterparts that have been discussed above in this section, DeepCFR has failed to bring significant results on the Belote game. While the quantitative amount of time that has been spent on simulations has not been rewarding in win rates, it is still interesting to review it, the thought process and discuss hypothesis on what went wrong.

DeepCFR is an algorithm that requires a lot of tuning. Its parameters span across different parts of the algorithm and thus depending on the use case, it has to be tuned differently. The tunable parameters are as follow:

- **Policy Network Layers:** The shape of the policy network layer. The number of hidden layers and their size as described in the chapter 3.
- **Advantage Network Layers:** The shape of the advantage/regrets network layer.
- **Number of iterations:** As discussed in the algorithm implementation in chapter 4, this number impacts directly the number of times the policy network is trained. It is also a factor of the total number of samples collected in the buffers.
- **Number of traversals:** This is correlated to the sample size and the amount of data collected.

- **Learning rate** : Factor that speeds up or down the convergence of the networks. A too great value will prevent from converging to a solution, while a too little value will require more time and risks not achieving a solution.
- **Batch size advantage**: During the training of a network, the network is effectively trained $n_network_step$ times. Each of these times, the samples used for training are a batch of the buffer used. The size of this batch is determined here. A small batch and low training will result in too little data for relevance, while too big of a batch and many training steps may lead to overfitting on data.
- **Batch size policy**: The batch size used for policy network training.
- **Memory capacity**: Memory constraint for the buffer to limit RAM usage. Constraint used by the Reservoir Buffet data structure mentioned in chapter 4.
- **Training steps (advantage or policy)**: Number of training steps used per iteration and for each player for the advantage network.
- **Reinitialization of advantage network**: Choose whether to clean the advantage network in between traversals and network training. This change is subtle and the algorithm should work with both values (true/false). However in the case of low data quantity, it might be good not to refresh it so that it is trained over a larger chunk of game states.

Since there are a lot of different parameters, it is important to define a range of usable parameters for the use case and the limitation faced. Both the **policy** and **advantage network** must have a decent amount of nodes in their layers. Function approximation of a complex game requires for the network layer to be able to interpret the data in many different ways. The number used for the experiment varied from 256 to 1024 for both networks layers

The number of iterations and the number of traversals must be as high as possible to ensure that there is enough info state to have a good estimate of the total game. Since the agent doesn't need to have been trained on all the nodes, the coverage does not need to be complete as it is for offline ESMCCFR, but it should be sufficient and diverse enough to come up with strategies on unvisited nodes. These parameters will mostly be dictated by the time complexity of the training.

The batch size should be a fraction of the size of the buffers. Since the networks are trained a number of times, there is no need to span across the whole buffer at each iteration. It should still be sufficient to take advantage of the data collected. So its value will stay in a range that fits $Buffer\ size \approx Batch\ Size \times n_{train_steps}$. This allows to have a range for the training steps too.

Using these constraints, several agents were trained with different settings. However, due to time and space constraints, the number of iteration and traversals as well as the memory capacity had to remain within low values for a game as complex as Belote. To have a comparative idea, Poker uses over 20k traversals and as lots of iterations in the

literature[19] to achieve results, unfortunately with the resources at disposition, the experiments had to be done with values close to one hundred. This is a probable cause for why the DeepCFR agent couldn't perform well even against a random agent.

In order to make sure that the lack of data was the cause of the problems, it was needed to explore other possible causes. Despite having testes a large subset of all the possible combinations of parameters, the agents produced fared poorly against a random agent, scoring between 45% win rate to 60%. Some parameters, excluding the iterations number and the number of traversals, seemed to have more influence such as the size of the networks, however after training the same agents repeatedly or on different sample data, it still produces results with too much variation to conclude that it could compensate for the lack of data.

Most of the agents have run for at least 8 hours of training time up to 5 days, which was the limit of the clusters used for simulations for a single run. Some have been run for a longer period of time by saving the state of the agent at the end of a simulation, and restarting a training from the saved state. For reference, a day's worth of simulation represents a little over 10^4 instances of the `traverse_tree(root_state)` function. Which represents $4 * 10^7$ visited states, which is not even close from a full tree size after the 2 phases of dealing.

To increase the sampling range, some agents were trained repeatedly to compensate for the time constraint. However, they all returned win rates similar as those on figure 5.7. The results show an irregular curve of learning which goes below 50% at times. The fact that it stagnates slightly over and slightly below the middle value shows that it is learning something even if it is bad.

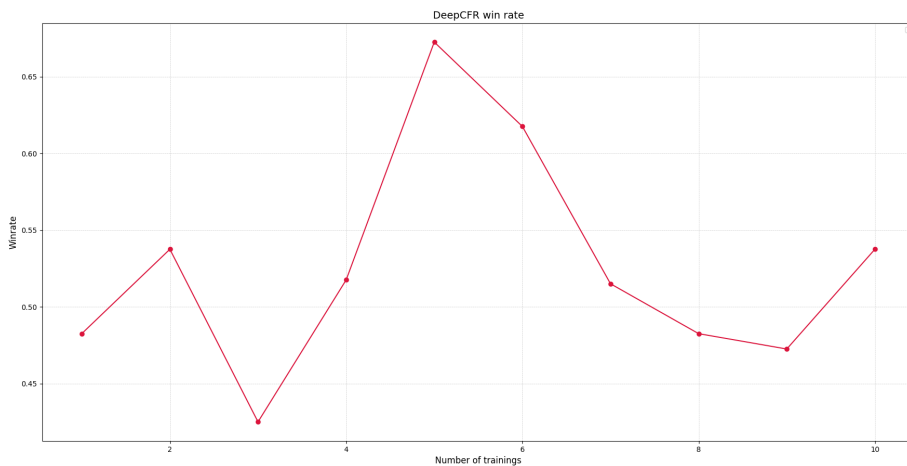


Figure 5.7: DeepCFR agent's win rate across multiple trainings

Looking at the problem and analyzing game histories, it came to light that DeepCFR was indeed learning but in a very skewed manner. Info States and associated action probabilities would either make a lot of sense, none at all or clearly have no significant values. Some bots were able to choose actions that seemed logical and advanced, strategically speaking, such as discarding a good card when the ally is winning the trick and a bad card when the opponents is, or even chase the trump suit. However there were as many other states where the preferred choice either didn't make sense or would highly point toward an illegal action. This last case would leave really low regret values for the possible actions but that had a high scale difference. For example values close to 10^{-5} and 10^{-8} would indicate actions that should not be played, or that are at least not relevant. But when scaled after passing through a mask of legal actions, the probability would completely tip toward the action that had a regret value of 10^{-5} although an uniform policy would function better at this point.

5

Another notable problem discovered in the policy per info states was that the bot did not understand that it was in the betting phase and thus did not propose any intelligent solution and fell into the above mentioned problem. While this was bound to happen for playing states because the restrictions for playing cards are rather hard to grasp from an info state, it should not be the case for the betting phase. Indeed, the rules for playing a certain card and thus determining the legal actions in play phase are multiple in the Belote game because it depends on the first card played, the trump suit, the cards in hand and the current winning player, but for the betting phase, it is directly encoded in the info state and should be understood.

The origin of the problem is not sure as it was not solved across the different things tried but the first intuition was that the betting phase info states look a lot like the playing phase at the 4th trick when there are still 5 cards left to play. Except that the phase bit is different, and that there are no cards played. This fact alongside with the fact that betting phase info states are under represented in the info states buffer lead to these states to be drowned in the sea of info state that look like them. Taking into consideration that the buffers are filled with external sampling, it can be estimated that it is populated by 1 betting phase info state for every 2000 other states. This value was estimated by analyzing the average increment in population in buffers for each traversal. Such a low ratio probably made learning difficult for the early phases of the game. This problem is less problematic in ESMCCFR algorithm because the less represented state's regrets and policy are computed using all the states below. However in DeepCFR learning, this rare states are likely not to be chosen at times for training, and even if they are, their weight will become insignificant compared to the rest.

To try to fix this, readjusting the daunting ratio of $\frac{\text{betting states}}{\text{play states}}$ could be a solution. In order to do this, one idea came to mind after noticing that many states had zero vectors as their regrets vector. Which meant that there were nodes where the decision didn't matter as the outcome would be the same regardless of their actions. These nodes are the ones where an action is forced by the rules, or when at some point of the game, it is obvious that a single team will win every last tricks. To prune these useless nodes, a mask was added to

the sample collection part of the code to avoid sampling nodes where the choice didn't matter. Doing this reduced by a factor of 3 the number of nodes sampled for each traversal, but increased the quality of the samples. Also it helped reduce the under representation of early nodes, which was still a bad ratio.

Unfortunately this did not bear fruit in the win rate aspect. Analysis of info states and choice made did show a few changes, with unimportant decision nodes having very few regrets value and often times an uniform decision pattern. Also there were less situation where the agent would propose illegal actions as the best output. But the betting states were still behaving as playing states and there were still a lot of mistakes with bad policies for playing nodes.

Another way to even the sampling profile is to use outcome sampling instead of external sampling. In external sampling each exploration of the tree gives a sampling profile that is increasing with the depth of the tree. However, with outcome sampling, each episode gives exactly one sample at each level of the game. The problem lies in the regrets computation. External sampling provides directly a regret profile for each action at every node it passes through, which facilitates the sample collection for the neural networks. With outcome sampling the samples are only about a single actions, thus it takes longer for the advantage network policy to compute realistic policies. This method did not bring any upgrade in the win rates of the agents.

After experimenting on the problem from different angles, it becomes more likely that the problem lies indeed in the total sampling and time allocated for training. As shown by the analysis of the <info states, actions probabilities> some states are well evaluated while others make the agent completely clueless, most likely because resembling states have not been visited.

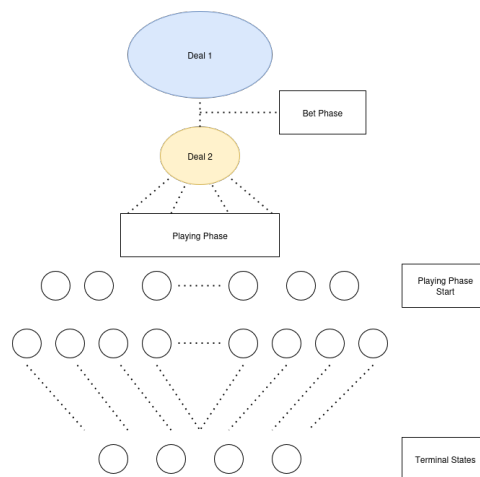


Figure 5.8: Belote's game tree diagram

Figure 5.8 shows a simplified version of a Belote game tree. Deal 1 phase and Deal 2 phase are two extremely high branching factors. Combining the 2 dealing phases allows to compute roughly $2.5 * 10^{12}$ info states for a single level of the tree. Luckily, lower levels tend to have fusing info states because the terminal layer has roughly around 10^6 info states. However to be able to perform, it is important for DeepCFR to go through a single info state multiples times, otherwise it is not able to converge to a strategy and samples regrets blindly with random strategies for every other states. This proves very difficult to put into application, at least for an offline version of the algorithm. Using external sampling, visiting 1% of the possible initial states at the betting phase would require 241.651 traversals. Or 22 days of simulation. Before the second phase of cards dealing that splits all those info states into more info states. It is not easy to determine what fraction of the tree should be visited during training, but whatever the fraction, it remains a heavy computational task for the Belote states.

5

One last effort in order to bring results with DeepCFR has been to train repeatedly on a few selected games, skipping the 1st dealing phase to control the sampling, in hope that the few games sampled would be enough to devise strategy for different types of games. The idea was to put more emphasis on learning correctly a few type of games, to make sure the agent understood the rules and some strategic points instead on focusing the coverage to have a large variety of states. However when sampled randomly, the games used to test the accuracy of the bot seemed to differ too greatly from the one used in training and thus didn't yield better results.

DeepCFR has also slightly been tested as an online algorithm, but the execution time for "easy" parameters was scales above the MCCFR counterparts for mediocre results, so it was not pursued further.

6

CONCLUSION

6.1 ROUND ROBIN

The results collected allow to make a final comparison of the different MCTS and CFR variants that have been used to solve the Belote game.

Vs	Random	ISMCTS	ESMCCFR	OSMCCFR
Random	50%	86.8%	99%	99%
ISMCTS	13.2%	50%	81.5%	84%
ESMCCFR	1%	18.5%	50%	50%
OSMCCFR	1%	16%	50%	50%

Table 6.1: Round Robbin Tournament : Random, ISMCTS, ESMCCFR & OSMCCFR agents

Table 6.1 shows that outcome sampling MCCFR comes slightly on top of the other algorithms. While it tied external sampling in their bouts, it still achieves a better score against the ISMCTS base-line. External sampling comes in a close second position, followed by ISMCTS that served as a good baseline for experiments. Unfortunately, DeepCFR did not qualify for this round robbin due to a lack of results.

	ISMCTS	ESMCCFR	OSMCCFR
Average thinking time	3s	40s	40s

Table 6.2: Thinking time per move

As shown in table 6.2, outside and external sampling are also very close when it comes to time complexity. Although when looking at the evolution of the win rates depending on their parameters, outside sampling seems to show more promising evolution if it were to use more resources, while external sampling seems to have reached a plateau. While ISMCTS shows some results with a low execution time, it is likely that it would need to multiply its time complexity several times to reach its MCCFR’s counterparts which would probably prove it to be somewhat less efficient than them.

6.2 PERSPECTIVES

Because it is the one that showed the less results, it is also the one with most possible developments: DeepCFR has proved itself against many simpler games and other larger scale games. However the Belote's system and complexity seems to be a tricky partner for the algorithm with a limited amount of time and resources. Although it did not perform in these experiments, it is likely that it will in different settings and with the help of other tools such as domain dependent or independent heuristics or different sampling methods. DeepCFR aside, MCCFR, although not fit to be used as an off line algorithm for Belote, yielded good results online. However they remain a bit lacking if the thinking time is the measurement. To increase performances, it could be interesting to investigate different concepts of tree searching beside the CFR and Monte Carlo concepts such as intelligent pruning through heuristics and/or function approximations.

6.3 FINAL WORDS

Belote is an extremely complex game with a particular game tree shape that adds a lot of complexity and different possible states. Although it uses less cards than an original card game that often uses a full deck, its organization does not make it fall behind in space complexity. The multi-phase game play and the lots of hidden rules of the game make it hard for an algorithm to properly learn its rules working with it and comparing it to other games of the imperfect information games class forces the realization that bringing solutions and working with algorithms on some imperfect information game or the other may be two completely different ordeals although they belong to a same class of problems. Large scale games are very hard to deal with but manageable in an online manner, and very well so below a certain number of possible info states. Even more so that the algorithms tested managed to achieve some results on a complex game without the use of domain specific heuristics. When it comes to the offline algorithms, it is still very hard to allocate the time and space resources to make a full analysis of it on Belote. Working with these algorithms for the Belote game would require to further optimize the game, the sampling schemes, resources handling and so on.

This thesis tackled a variety of imperfect information games algorithm on a very particular game of its own class. It showed how to use those algorithms in this particular use case, how they fare against each other and their different specificities. In order to experiment on these algorithms, the Belote game has been implemented in the OpenSpiel framework. Based on MCTS and CFR roots algorithm, simulations have been made by applying such algorithm to the game. Although DeepCFR is hailed as the leading algorithm in imperfect information game, it still comes out inferior to online counterparts when resources are limited. While DeepCFR did not work as intended, there is much learning to take from trying to make it work in a really constrained environment. As discussed previously, DeepCFR principally, but the other MCCFR algorithms too, can be improved in the particular complex case of the Belote game. And bringing solutions to this singular problem might lead to global improvements in the development of CFR and Deep based algorithm for imperfect information games and other domains.

7

GLOSSARY

To understand the sentences we read and write we must first understand the words that compose them. Here's a little glossary to help understand some vocabulary and concepts that are gonna be mentioned in this thesis.

Agent Playing entity that relies on a algorithm to provide its playing policy.

Bandit problem Class of problems characterized by the bandit dilemma, which concerns the balance of the trade-off between exploration and exploitation in tree searches.

Decision Tree A decision tree is a tree like structure composed of nodes and their successors. Successor nodes are linked to their parents node by a transition function. It is used to travel the unfolding of the algorithm across the domain of the problem it's working on.

Episode Describe the unfolding of a game from a certain state to its end.

Heuristic [method] is a method that gives satisfactory results or approximations. It can be domain dependent which means only applicable in the use case, or not.

Incomplete Information Game Game where all the information are not accessible to the players, making many decisions stochastic and widening/deepening the decision tree(s).

Nash Equilibrium [*in a game*] is found when no player can benefit from changing their strategy or their decision policy. It is computable in theory but hard to do so in practice.

Node Represents a decision point in a tree. Depending on the search tree it is a part of, it can be a state or an information set.

Offline Search Opposed to *Online Search*, search performed at the beginning that explores all the possible outcomes and produce a complete decision tree.

Online Search In the context of a game, search performed at each step of the game to form a partial decision tree when the search space is too big to explore fast enough or at all and/or because of the incomplete information characteristic of the game.

Outcome Value of an episode. Can be a final score for example.

Policy Set of probabilities and action at a given moment in a game.

Recursive Characteristic of a function that calls itself. Often linked with recursive mathematics, where one value allows the calculation of the next.

Stochastic Characteristic that implies randomness and probabilities. In a game sense, it is opposed to deterministic.

Uniform Policy Policy that allocate the same probability of all of its choices.

BIBLIOGRAPHY

REFERENCES

- [1] IEEE Edward Powley Member IEEE Daniel Whitehouse Member IEEE Simon Lucas Senior Member IEEE Peter I. Cowling Member IEEE Philipp Rohlfshagen Stephen Tavener Diego Perez Spyridon Samothrakis Cameron Browne, Member and Simon Colton. A survey of monte carlo tree search methods. 2012.
- [2] Huale Li, Xuan Wang, Zengyue Guo, Jiajia Zhang, and Shuhan Qi. Nncfr: Minimize counterfactual regret with neural networks. *CoRR*, abs/2105.12328, 2021.
- [3] Charles Z Zheng. Nash equilibrium. 1990.
- [4] Marc Lanctot Todd W. Neller. An introduction to counterfactual regret minimization. JULY 2013.
- [5] Viliam Lisý, Marc Lanctot, and Michael H Bowling. Online monte carlo counterfactual regret minimization for search in imperfect information games. In *AAMAS*, pages 27–36, 2015.
- [6] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *5th International Conference on Computer and Games*, MAY 2006.
- [7] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [8] ARTEM NOVIKOV and VOLODYMYR YANOVSKY. Exploring the limits of mcts in pac-man: Maze size, simulations, and performance. *Herald of Khmelnytskyi National University. Technical sciences*, 341(5):351–359, 2024.
- [9] IEEE Edward J. Powley Member IEEE Peter I. Cowling, Member and IEEE Daniel Whitehouse, Student Member. Information set monte carlo tree search. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, VOL. 4, NO. 2, JUNE 2012.
- [10] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.
- [11] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

- [12] Martin Schmid. Search in imperfect information games. 2021.
- [13] Jeffrey Long, Nathan Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):134–140, Jul. 2010.
- [14] Marc Lanctot. *Monte Carlo sampling and regret minimization for equilibrium computation and decision-making in large extensive form games*. University of Alberta (Canada), 2013.
- [15] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.
- [16] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019.
- [17] Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.
- [18] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *International conference on machine learning*, pages 793–802. PMLR, 2019.
- [19] Eric Steinberger. Single deep counterfactual regret minimization. 2019.

