|  |
| --- |
| CITS3001 Algorithms, Agents and Artificial Intelligence |
| Moss Side Whist Agent |
| POMDP Monte-Carlo AI |
|  |

Martin Porebski, 21498791

Dylan Johnson, 21469038

26 October, 2017

|  |
| --- |
| a |

# 1.0 Project Specification

Moss Side Whist is a card game played with three players. One of the players, the *leader* is dealt 20 cards, while the remaining two receive 16 cards at the beginning of the game. The leader has to discard 4 cards before starting the first round. The game is played in *tricks*. In each trick, players place one of their cards face up in a clockwise order, starting from the leader. The winner of the round becomes the leader and starts the new trick by playing one of their cards. To win the trick, the player has to play a card of a higher rank (organized from 2 to Ace) or *trump* the current suit with a spade. The players have to play the cards of the same suit as the leader’s card, unless they do not have any, in which case they can play a spade to trump the card (and win) or a card from a different suit (and lose)[1].

In order to have a useful comparison of potential implementations, it is necessary to have a clear definition of the problem at hand. Specifically, we need to define the search space. This starts with defining the state of the environment at any given moment in time. The most important state variables are the locations of each card in the deck (referred to as "*deck state*"), with a card being either in one of the player's hands, on the table or out of play. This suggests we represent these locations as a subset of cards in the deck, with cards moving from one subset to another as the game progresses. The deck state is then the makeup of these five subsets at a given time (i.e. which cards are currently in each location).

The problem is complicated by partially observable nature of the environment due to the fact that during play, we only know the locations of some cards. This includes:

* The 16 cards that are dealt to us (the agent).
* The *t* \* 2 + *p* cards that have been observed by the agent. This means the cards that after *t* completed tricks, *p* being our position in the play order).
* The 4 cards discarded at the start of the game, if we are the leader (i.e. *p* = 0).
* If an opponent plays a card from a different suit to the lead we know that they can't have any cards from the lead suit.

With these rules governing how much of the state is known, we can use random variables to represent the unknown elements of each location. Assuming a truly random deck shuffle, we can derive the probability distribution of these variables by setting probability to 0 for cards we know they can't be, then evenly dividing the remaining probability across all other cards.

Not knowing one opponent's hand means we can't rule certain cards out of being in the other opponent's hand and we must consider both possibilities, implying the search space will be larger than the number of deck states.

The final, unknown piece of the model is the opponent’s strategy, which governs the state transitions that occur between our agent's turns. Whether we assume a simple fixed strategy or a probability distribution over their likely decision, not knowing their hand means transitions will be non-deterministic in nature. We can encapsulate both opponent moves with a single transition, the probability of which is that of the opponents playing two given cards in sequence.

The last component of the search space model is utility (point scoring). We could include the current score tally as a component of state, but there is a more elegant solution. Score is not dictated by the deck state but by the transition from one state to another (i.e. who played what cards). We can associate rewards (negative step costs) with state transitions, where the agent wins the trick, and do likewise for the opponents if we wish to measure the agent's performance against them.

## 2.0 Literature review

## 2.1 Greedy Model-Based Reflex

Greedy algorithms make the choice that seems most appropriate in a particular moment. They make a locally-optimal decision, with no knowledge or regard for past or future decisions. Their simplicity means that they are often fast to run and easy to implement, but also limits their usefulness. Failing to consider the future is often not conducive to complex scenarios such as strategy games [2].

A model-based reflex agent gathers what information it can from its environment and uses its model to make inferences about what it sees and update its internal state accordingly. It then refers to its condition-action rules and applies the action corresponding to its current internal state. Performance is dependent on the complexity of the model and updating the internal state, as the rest of the decision process is pre-calculated.

### Project Implementation

With our defined model of the environment, it is relatively easy for the agent to predict the optimal card for winning the current trick. However the approach varies depending on its position in the play order.

The third player, having the knowledge of all cards in the trick, would be able to always select either a winning card or dispose of a lowest card. The leader would either play the highest card to secure a winning trick, or the lowest card. Similarly, player 2 would have to beat the leading card and consider if they are able to defeat whatever player 3 will play. Picking the disposed card is less obvious, with options including the lowest number value or from the suit we hold the most of.

This provides a good starting point as an assumed fixed opponent strategy. However the evaluation requires comparing all cards in their hand, knowledge that our agent doesn't have. We will present a solution to this issue later on.

The greedy evaluation is unlikely to beat more intelligent methods. However each of the decision rules can be relaxed for use when we are less certain of the opponent’s cards, providing the possibility of optimising a sound strategy.

## 2.2 ExpectiMax Search

ExpectiMax (EM) is a variation of the MiniMax concept where some moves are governed by chance, rather than player logic. EM incorporates the chance-factor, by treating it as another player (“*nature*") [3]. The search-tree branches into potential outcomes whenever the chance rather than logic determines them ("*move by nature*") [3].

Evaluating a chance node is done not by choosing the value of a child node, but instead by averaging the values of all children weighted by their probability of occurring. This leads to the chance nodes taking the expected (hence the algorithm's name) value of all their descendent leaf nodes.

It is important to note that alpha-beta pruning can't be applied to chance nodes, which can significantly impact (decrease) the performance [3].

### Project Implementation

The random events in our model are the state transitions. We can represent these as moves by nature made following each move made by the agent. The collective score of the transitions leading to a node can be assigned to it as utility.

Unfortunately, ExpectiMax faces considerable performance hurdles. With branching factor ~48 at maximum viable search depth ~768 [4], the search is substantial. Mapping the space as a tree removes transition model and its potential advantages.

Although the ExpectiMax algorithm has too many drawbacks to seem a viable option, the player-versus-nature structure it revolves around is an informative method for arranging non-deterministic transitions and may prove useful in combination with another algorithm.

## 2.3 Monte-Carlo Tree Search (MCTS)

The Monte-Carlo method is a technique for approximating deterministic output when the input is known only as a probability distribution [5]. It generates many samples of possible inputs using the distribution, then runs a deterministic simulation on each sample and returns the expected result across all samples.

Monte-Carlo Tree Search is an application of this method to game trees. Rather than exhaustively search the tree, it samples paths down to the bottom (end state). The immediate child that produced the best win rate is then chosen as the move to make.

The path taken by each sample is not entirely random. Each node maintains a record of wins-to-losses ratio it has led to. After each sample is completed, the ancestors of the final node update their records with the result. Successive samples use the win rate to weight their "random" selection, weighting the search towards the most promising areas of the search space over time.

A function called the UCT ("Upper Confidence Bound 1 applied to trees") [5] can be used to fine tune the degree to which win rate guides the search. If wins have too much influence, the search may narrow too quickly and miss the optimal choice.

MCTS provides a great performance boost over ExpectiMax as it prunes the tree more intelligently [4, 5]. It also has the advantage of beginning and ending at anytime: the sampling process can be stopped at any point and the current estimates can be used.

### Project Implementation

MCTS would apply well to the previously discussed ExpectiMax tree. The win rate method would weight sample direction at max nodes, while our transition model would weight it at chance nodes.

The performance improvements are likely to make MCTS viable where ExpectiMax wasn't [6, 7]. We would have to formulate a UCT function, but a standard template exists that we can optimise for this application.

## 2.4 Markov Decision Process (MDP)

A Markov Decision Process frames a non-deterministic search space as a path problem [6]. It maximises the collective utility of the path the agent takes through the space while taking into account that the agent has limited control over said path.

The optimal choice to make in each given state can be calculated ahead of time and loaded into the agent as a policy, equivalent to the condition-action rules of a reflex agent.

A variant of this model is the Partially-Observable MDP (POMDP) [8], applicable to partially observable environments. Because the agent cannot know exactly what state it is in, it instead maintains a "belief state" that takes the form of a probability distribution over what the current state might be. Each action triggers an observation for the agent, allowing it to update its belief state.

### Project Implementation

Finding a policy for a POMDP involves mapping the potential belief states (the "belief space") rather than the actual state space. The reward for being in a belief state is unknown, so must be represented as the expected reward of the actual states corresponding to that belief [9].

In POMDP the information gathering can have utility in and of itself. If an action is likely to make the belief state more certain, the expected reward for that action will often be higher even if there is no immediate tangible benefit.

The transition model for a belief space revolves around potential observations rather than potential destination states. Each action available in a belief state may result in one of several observations, each leading to a new belief state once the belief is updated with the observation.

Modelling aside, formulating a policy with traditional MDP methods will not be feasible. Policy iteration and value iteration both require iterating through every state in the space and storing a solution for each one, a process which is not likely to be viable in terms of time or storage constraints. We could potentially shrink the space by grouping similar belief states together, leading us away from the MDP model and more towards a standard reflex model based agent.

## 2.5 Reinforcement Learning

Reinforcement learning is a method for optimising decision making through experimentation. The decision strategy (usually expressed as an evaluation function) is tested against many variations of input.

The agent incrementally modifies its evaluation function between tests, leading to a model where the search space is made of potential evaluation functions and the goal is to find the function with the highest expected return. Positive test feedback prompts the search to continue its current vector, while poor feedback leads to trying another approach.

Much like with Monte Carlo, a balance must be struck between deepening the current search vector and probing unexplored areas of the space.

### Project Implementation

Several of the previous algorithms have had parameters that leave room for calibration (probability ranges for reflex model based, UCT for Monte Carlo, and opponent strategy for all), a task well suited to reinforcement learning.

With only 16 tricks in a round of Whist we could simulate a great many rounds during the duration of the project, providing our agent with a massive sample size to test its strategy with. What's more, as the agent becomes more effective we can perform tests against opponents using the improved strategy, providing feedback on successively harder opponents and improving the predictive capabilities of the agent.

This leaves use with the issues of implementing the evaluation function in a performance friendly way, and devising appropriate functions for the feedback-update process and the exploitation-exploration balance.

# 3.0 Final Agent Design

The partially observed environment of Whist can be represented as a belief over the current deck state, contracting the probability distributions of the individual cards into one random variable. This solves the problem of modelling opponent strategy in an unknown state: we now have a collection of potential states each with a concrete value for all opponent cards. We can use the opponent strategy model to predict the moves for each deck state in the current belief, count the occurrences of each move weighted by the probability of the state they arose from, and then return the move with the highest aggregate as the expected one.

With the belief state covering such a massive number of potential states, iterating through and evaluating each possibility would likely take far too long and occupy too much memory. The probability of a state being true is derivable from our previously suggested model and can be calculated on the fly, negating the storage concern. Furthermore, we could instead evaluate the belief state from a sample of potential states to improve the performance.

### 3.1 Opponent Modelling

The agent will model the opponent’s strategy with an improved greedy algorithm, when sampling games in the MCST. It is very efficient and returns “good enough” approximations [5, 6, 8].

The simplest scenario to consider is when taking the last turn in a trick. With the opponent's cards on the table, the player can be certain which of its cards can win the trick. In this scenario it can select the lowest value winning card (saving higher value ones for later), or if winning is impossible take the opportunity to dispose of a low value card. If the player does not have the leading suit, it can play a trump or dispose of the lowest-ranked card.

When playing as the leader, the agent has the least information but also the greatest degree of choice. The greedy philosophy suggests playing the safest (i.e. highest value) card. In some situations we might be certain that the opponents hold no cards capable of beating one of ours (helped by the fact that leader knows which cards were discarded and therefore exactly which cards are in play), in which case the agent can safely choose that card. This is improved on by using the belief states modelled through the play. We specify the degree of “*risk*” the greedy opponent is willing to take when predicting what card to play.

The second player has arguably the hardest choice, facing both uncertainty and being forced to follow the lead card's suit. Similarly, it will use the maintained belief state to decide whether the player thinks that they can beat what player 3 will play. If it is able to beat the leader and it thinks that it is likely to beat player 3, it will play the higher card, otherwise it would play the lowest card.

### 3.2 States

On each turn, MCST will be run to determine the best move. It can maintain states that describe the observed environment to make informed decisions. Those consist of the game and belief states.

The belief state maintains the probability of each card being in a certain location (i.e. the probability of a particular card being held by a particular opponent). The data is stored as “bits” rather than arrays. This means that for each card there is a bit assigned in a *long* (64bits) or *int* (32bits) [10] to determine the position. Bitwise logic is then used to access the probabilities. This significantly improves the performance [11]. When evaluating the decisions, the belief state class can provide information about whether the specified player is *believed* to have a certain card or suit above the probability threshold.

The game state maintains the variables related to the game itself. This includes the order the game started with, the current turn order, cards played in the trick and statistical information for MCST. The agent maintains the beliefs from the start of the game. This includes the starting hand as well as what cards have been played. On each turn, it updates the information. Furthermore, it maintains a set of beliefs from a perspective of its opponents. This allows on more reliable evaluation when modelling their turn in MCST.

UCT? tree structure? Backpropagation? Probablity calculation?

### 3.3 Discarding

The default discard strategy is to discard the lowest cards in our hand that are not spades, to increase our chance of winning a trick with a higher-ranking card. However it can be improved on in three ways:

* Check if there is a suit which we can completely discard that is not a spade. The advantage of that is being able to increase our chance of trumping (and winning) a trick. This done by checking how many cards of each suit we have in our hand. If suit size is smaller than 3 and the cards are low enough, they are discarded.
* Check for the lowest cards in each suit. If those cards can be used as “*backups*” to higher cards e.g. 2 can be used as backup to King, when an Ace is played. Again, this is done to increase the chance of winning a trick.
* The reinforced learning can be used to assign the likeliness of winning, when certain cards were discarded. This factor can be used in an evaluation function when deciding whether to discard a potential card or not.

# 4.0 Performance

After running a number of games, the following table has been obtained:

|  |  |  |  |
| --- | --- | --- | --- |
| Opponent | Sample(games) | Wins | Average time/game |
| Greedy vs. Random | 250 | 50% | 20s |
| Agent vs. Random | 200 | 95% | 20s |
| Agent vs. Greedy | 300 | 90% | 20s |
| Agent vs. MCST | 250 | 50% | 20s |
| Agent vs Martin | 5 | 100% | 20s |

It can be observed that the agent beats any common implementations. This suggests that it is making “*intelligent*” decisions, which enable it to win at a high rate. Furthermore, considered the time taken, it is very efficient. It does 351 samplings a round which take less than 200ms.

Tweaking of the parameters further produces the following results:

# 5.0 References

[1] Tim French. 2017. CITS3001 Algorithms, Agents and Artificial Intelligence Project. (September 2017). Retrieved October 01, 2017 from <http://teaching.csse.uwa.edu.au/units/CITS3001/project/>

[2] Keith Schwarz. 2013. Greedy Algorithms. Stanford University. Retrieved October 13, 2017 from <https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/13/Small13.pdf>

[3] Dan Klein. 2010. Expectimax Search Trees. UC Berkeley. Retrieved. October 14, 2017 from <https://inst.eecs.berkeley.edu/~cs188/fa10/slides/FA10%20cs188%20lecture%208%20--%20utilities%20(6PP).pdf>

[4] Marco Chiarandini, Stuart Russell, Peter Norvig . 2011. Games and Adversarial Search. University of Southern Denmark. Retrieved October 13, 2017 from <http://www.imada.sdu.dk/~marco/DM828/Slides/dm828-lec17.pdf>

[5] Cameron Browne. 2010. What is MCTS? Retrieved October 11, 2017 from <http://www.cameronius.com/research/mcts/about/index.html>

[6] Yao Yujian. 2016. CS3243R Project Report, Hearts as a Partially Observable Markov Decision Process. (2016). Retrieved October 12, 2017 from <http://yjyao.com/res/hearts.pdf>

[7] Joshua C. C. Chan, Dirk P. Kroese. 2010. Rare-event Probability Estimation with

Conditional Monte Carlo. (April, 2009). Retrieved October 10, 2017 from <https://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps.pdf>

[8] David Silver, Joel Veness. 2010. Monte-Carlo Planning in Large POMDPs. (2010). Retrieved October 20, 2017 from <https://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps.pdf>

[9] Geoff Hollinger. 2007. Partially Observable Markov Decision Processes (POMDPs). Carnegie Mellon University. Retrieved October 12, 2017 from <https://www.cs.cmu.edu/~ggordon/780-fall07/lectures/POMDP_lecture.pdf>

[10] Oracle. 2017. Primitive Data Types. Java Documentation. Retrieved October 20, 2017 from

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

[11] Sean Eron Anderson. 2009. Bit Twiddling Hacks. Retrieved October 20, 2017 from

<https://graphics.stanford.edu/~seander/bithacks.html>