# Comparison of Monte Carlo Tree Search Methods in the Imperfect Information Card Game Cribbage

**Richard Kelly**

Department of Computer Science
Memorial University

Supervised by Dr. David Churchill

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science

April 2017

# Abstract

Non-deterministic, imperfect information games, in which there are random elements and players can not always observe the entire game state, pose challenges for AI design over deterministic, perfect information games. Monte Carlo Tree Search (MCTS), an AI technique that uses random sampling of game playouts to incrementally build a search tree, has been used successfully in perfect information deterministic games. MCTS does not rely on hand-crafted heuristics for evaluating non-terminal game states or on other domain-specific knowledge, and performs well for games with large branching factors. Several techniques for playing imperfect information games with MCTS have been used for various board and card games. Techniques include sampling over many possible determinizations of a given game state, and more complex techniques in which the information sets that players belong to are considered. In this dissertation, we introduce the imperfect information card game of Cribbage and the MCTS algorithm. We then describe our implementation of Cribbage for two players and several MCTS-based and non-MCTS-based AI players. We compare the performance of our players and find that Single-Observer Information Set MCTS performs well in this domain, beating a simple determinized MCTS in our implementation of Cribbage most of the time. We also present experiments with different parameters of the MCTS-based players.

# Acknowledgements

Thank you to my supervisor, Dr. David Churchill, for his advice and guidance through this research and helping to make this dissertation possible.

I would also like to thank all the staff, faculty, and my fellow students in the Department of Computer Science, Memorial University, for making it a great place to study. In particular, I'm grateful to Dr. H. Todd Wareham for his guidance and support in my final year of this program.

Thank you to my family and to my partner Gabriela, who has endured many late nights of work and has supported me greatly throughout this degree.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1:

# Introduction

Games with imperfect information, including many card, board, and video games, present interesting challenges for game AI. The card game Cribbage, a popular game for 2-4 players, is such a game. Some properties of Cribbage, shared by many other card games such as Poker, are:

**Non-Determinism:** Cribbage is a non-deterministic game, meaning that there is randomness in the game, and that actions taken by players do not deterministically lead to the next state of the game. Cards are dealt randomly to each player at the beginning of each hand and a card is drawn from the deck partway through each hand.

**Imperfect Information:** Cribbage is an imperfect information game, which means that players do not have knowledge of the complete game state. Each player holds a hand of cards which are hidden from the other player until they are played. Unseen cards in the deck are also hidden from each player.

Monte Carlo Tree Search (MCTS) is a game AI technique that has been used successfully in several perfect information, deterministic games, such as computer Go,

since 2006 [1]. It works through repeated sampling of game outcomes using elements of random play, while gradually building a search tree. Because MCTS uses random play, it works without relying on domain-specific knowledge such as hand-crafted heuristics for evaluating non-terminal game states.

Several techniques for handling imperfect information and stochastic game elements in MCTS have been used for board and card games such as Skat, Dou Di Zhu, Settlers of Catan, and Phantom Go [2], [1], [3]. Cribbage was chosen as a game to implement MCTS in for this research because it is relatively easy to implement, but designing a hand-crafted AI to play optimally for Cribbage would be somewhat difficult and time-consuming. It is possible to make a hand-crafted AI that plays reasonably like a player who knows the rules well, which was a useful characteristic of the game for testing purposes. We are unable to find previous similar work on the game of Cribbage. The purpose of the research was to focus on how MCTS can handle the unknown information in the game tree, from the perspective of the player to act.

In imperfect information games, it is hard to account for all the possible states that a player might be in, and to infer what an opponent's plan or hidden resources might include based on their actions. This makes techniques for handling these kinds of games interesting and worthy of investigation.

In chapter 2 we describe in detail the game of Cribbage, Monte Carlo methods, and the MCTS algorithm. Chapter 3 describes the implementation of Cribbage in Java and the different AI players implemented for this game. A scripted player, a Cheating UCT player, a Determinized UCT player, and a Single Observer-Information Set MCTS player were implemented. In chapter 4 we present the results of testing the AI players against each

other, tests on varying parameters of some AI players, and discussion about the performance of the AI players. In Chapter 5 we summarise the results of this research and present some possibilities for future research.

# Chapter 2:

# Background

## 2.1 Cribbage

Cribbage is a card game for 2-4 players using a standard 52 card deck and a playing board used for score keeping. For this project, only the 2-player version of the game was considered.

### 2.1.1 Description of Play

Cribbage is played in repeated hands until a player reaches 121 points at any time during a hand, thus winning the game. A hand consists of several ordered stages: the deal, throwing, the cut, play, and finally the count. The dealer position alternates between hands, and several game elements are affected by who the current dealer is.

At the beginning of each hand, both players are dealt six cards, then both simultaneously discard two cards face down to a form a third hand (the crib), which is scored by the dealer at the end of the hand. A single card, referred to as the ‘cut’ in this research, is drawn from the deck face-up after cutting the deck. The cut is a community card that forms the fifth card in each of the three hands for the ‘count’ at the end of the hand.

**Figure 2-1: Cribbage board with best hand in the game**

In the "play" stage of the hand, each player plays one card from their hand face up, alternating turns, while keeping a running sum of the ranks of cards played. When the sum reaches 31, or no one can play without going over 31, the play restarts from zero with the remaining cards. When the play is over, both players count the points earned by their hands (combined with the cut card), and the dealer also counts the crib as their second hand.

Table 2-1 summarizes the ways that a player can earn points in Cribbage. Players score points in both the play and counting for hitting fifteens and thirty ones, pairs, flushes, straights (runs), and other card combinations. Aces count as one and face cards count as 10 when creating sums from combinations of cards. Figure 2-1 shows a Cribbage board and the best counting hand in the game, which is worth 29 points.

**Table 2-1 Summary of ways to earn points in Cribbage**

| Game Stage | Description | Points |
|---|---|---|
| Cut | The cut card is a Jack of any suit | 2 points for the dealer |
| Play | Second/third/fourth card of same rank played successively by either player | 2/6/12 points for the player who plays the card |
| Play | Three or more successive cards whose rank form a 'run', played in any order (e.g. 7, 5, 6) [Note, a second run can be scored by the next player to play, and a third, etc.] | Points equal to the number of cards in the run for the current player |
| Play | Play count hits fifteen | 2 points for the current player |
| Play | Play count hits thirty-one | 2 points for the current player |
| Play | Last card played before count resets or play stage over ("one for last") | 1 point for the current player |
| Count | Any two or more cards that sum to fifteen | 2 points for hand owner |
| Count | Two/three/four cards of same rank | 2/6/12 points for hand owner |
| Count | Any three or more cards whose ranks form a 'run' (e.g. 10, J, Q, K) | Points equal to the number of cards in the run for the hand owner |
| Count | Flush (in-hand only or in-hand + cut for player hand; all five cards only for crib) | Points equal to number of cards in the flush |
| Count | Jack of same suit as cut card | 1 point |

Human players usually play an odd number of games constituting a match. If a player wins by more than 30 points the opponent is said to be skunked (or double skunked if the lead is 60 points or more), earning the equivalent of two or three game wins, respectively. Concepts of multi-game matches and skunking are not considered in this project. There are also other rule variations that some players use which are not considered here.

### 2.1.2 Properties of Cribbage

Cribbage has a relatively small game tree for a given deal of the cards. If the game were to be played with all cards face-up, as a perfect information game, each hand would have at most $\underbrace{15\times15}_{Throwing} \times \underbrace{4\times4\times3\times3\times2\times2\times1\times1}_{Play} = 129{,}600$ different possible ways to be played. Many of those would be equivalent. For example, if a player is holding 6♣ and 6♠ in hand, playing either one of them is equivalent during the play stage of a hand.

However, from one player's point of view, considering only the cards that player has seen, there are $\binom{46}{6} = 9{,}366{,}819$ different 6-card hands that the opponent could have at the beginning of a hand. Again, many of these hands are equivalent, but the effect is that the imperfect information nature of the game makes the set of possible states a player can be in at any time quite large.

## 2.2 Monte Carlo Tree Search

### 2.2.1 Monte Carlo Methods

A Monte Carlo method is any method which approximates a function through random sampling. In the case of a perfect information game, one way to select a move could be to repeatedly descend the game tree to the end of the game from the root node by taking actions for all players randomly, and then recording the result as a win or loss. Over an infinite number of samples the average results for each action from the root of the tree would approach their true values. The hope is that a move that results in better win rates given random play thereafter is a good move to take.

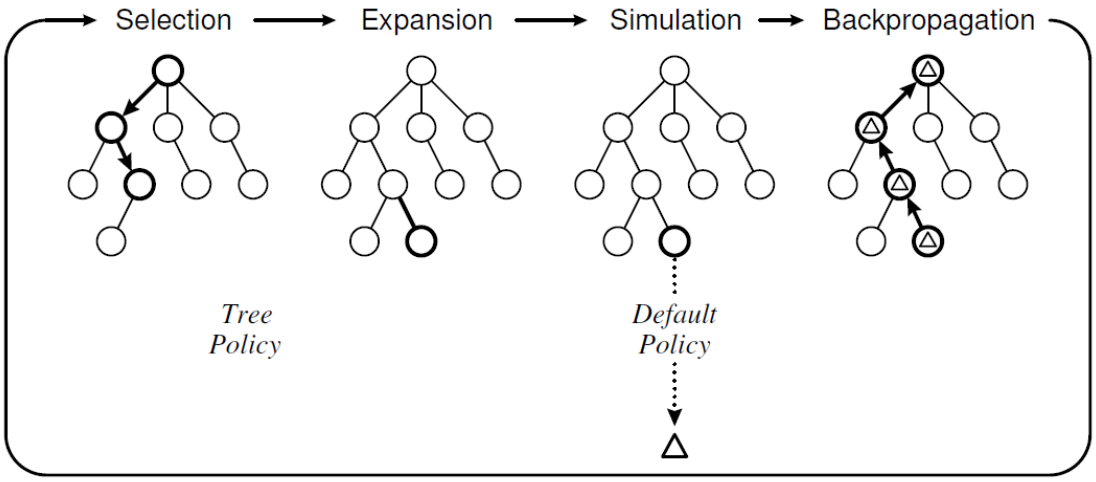


**Figure 2-2: One iteration of MCTS**

This is an example of a flat Monte Carlo search, because it accumulates results only at the root of the search tree. In this example, the move to try in the next sample is always random. MCTS, described in the next section, improves on these properties.

### 2.2.2 Monte Carlo Tree Search

MCTS was first described by Rémi Coulom in 2006 [4]. MCTS improves on a flat search method by building a search tree that is reused in each iteration of the search. MCTS does not explore every part of the tree systematically as in minimax, but instead builds a tree asymmetrically by focusing on more promising branches of the tree. Every node of the search tree in MCTS accumulates information about how successful it has been in previous iterations in the same way as the top-level nodes in a flat Monte Carlo search. That information is then used to bias the selection of child nodes at every level of the search in subsequent search iterations, though the initial selection criteria is still random.

MCTS is an on-line search—nothing needs to be precomputed to use MCTS—and it is an "anytime" search, meaning it can be stopped whenever a computational or time

budget is reached. When the search is stopped, the best move found so far from the root of the tree is selected. In practice selection can be calculated in different ways, and is addressed in the next section.

The steps for building the MCTS tree are summarized in Figure 2-2 from [1]. In more detail, the procedure for building the tree (and the one followed for this research) is as follows:

**Nodes:** The tree consists of nodes representing states in the game. A node can contain a copy of the corresponding game state or the action leading to the node's corresponding game state. Each node keeps track of its visit count and total score or value (win or loss in most games) from visiting that node, as well as a reference to its parent node and child nodes.

**Selection:** Descend the tree from the root node by following a selection policy until either a terminal node or a node with unexpanded children is reached.

**Expansion:** If the selected node is not a terminal node, expand it by creating a new node representing an action taken from the parent node and the state arrived at by taking that action.

**Simulation or Playout:** Play from the expanded node by following a "default" policy until reaching a terminal game state, which has a value (score for Cribbage) for each player associated with it. The default policy is usually random play but can be otherwise.

**Backpropagation:** Backup the simulation values to all the nodes visited in the selection and expansion steps.

A key benefit of MCTS is that all that is required is a working implementation of a game. Random playouts mean that no knowledge of how to play the game well, and no evaluations of non-terminal states, are needed. While not necessary, most applications of MCTS for competitive game play do use various enhancements to the algorithm described above, including the use of domain-specific knowledge. For instance, MCTS Go agents use knowledge about basic patterns of Go pieces to narrow down the search space, so that all moves do not have to be considered [1]. This is helpful for games such as GO which have very large branching factors (hundreds of possible moves per turn).

### 2.2.3 UCT Algorithm

The algorithm presented in the previous section omits the selection criteria used to descend the tree. The selection problem is an example of a multi-armed bandit problem, in which there are $k$ choices that each provide an unknown rate of return, and the goal is to maximize the return in a limited number of visits to the $k$ choices [5]. It can also be described as exploitation vs. exploration. After visiting some of the choices, or trying each one at least once, there is a choice which has the highest average return so far. The balance of exploitation and exploration is about when to select the best choice found so far and when to try other choices that have fewer visits to learn more about their actual rate of return.

The selection algorithm most commonly used for MCTS and used in this research is Upper Confidence Bound or UCB1. It was first applied to MCTS by Kocsis and

Szepesvári in 2006, which they call Upper Confidence Bound for Trees, or UCT [5]. The UCT algorithm for selecting the next child node $v'$ of a node $v$ to visit is:

$$\arg max_{v' \in children\ of\ v} \quad \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

where:

- $Q(v)$ is the accumulated total value *for the player to act at node* $v$ from the backpropagation step of previous iterations;
- $N(v)$ is the number of times $v$ has been visited in the selection step of previous iterations;
- $c$ is an exploration constant that can be adjusted for different domains.

In the selection step of the UCT algorithm, the child node which maximizes the above expression is repeatedly selected to descend the tree. The first term of the expression represents the average value observed when visiting that node in the past. The second term incorporates the ratio of visits to the parent node to visits to the child node. Intuitively, child nodes that have shown little value will be visited when the second term grows, which happens as the parent node is visited. The MCTS algorithm described in this chapter ensures that each child node is visited once before this formula is used to select a child node in future iterations. Kocsis and Szepesvári showed that the probability of selecting the optimal action when using UCT converges to 1 over an infinite number of search iterations [5].

The UCT algorithm used in this research is adapted from [1], and is presented in pseudocode here:

**Algorithm 1** UCT

**function** search(State $s$, c)
    root ← new Node($s$)
    **while** within computational budget **do**
        Node newNode ← treePolicy(root, c)
        value ← defaultPolicy(newNode.state)
        backup(newNode, value)
    **return** bestChild(root, 0).action

**function** treePolicy(Node $v$, c)
    **while** $v$.state is non-terminal **do**
        **if** $v$ is fully expanded **then** $v \leftarrow$ bestChild($v$, $c$)
        **else return** expand($v$)

**function** expand(Node $v$)
    $s \leftarrow v$.state
    randomly choose $a \in$ untried actions from $s$.actions
    $s' \leftarrow s$.applyAction($a$)
    $v' \leftarrow$ new Node($s'$)
    $v'$.parent ← $v$; $v'$.action ← $a$
    **return** $v'$

**function** defaultPolicy(State $s$)
    **while** $s$ is non-terminal **do**
        randomly choose $a \in s$.actions
        $s$.applyAction($a$)
    **return** $s$.value

**function** backup(Node $v$, value)
    **while** $v$ is not null **do**
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) +$ value
        $v \leftarrow v$.parent

**function** bestChild(Node $v$, c)
    max ← $-\infty$
    **for** $v' \in$ children of $v$ **do**
        score ← $\frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
        **if** score $>$ max **then**
            max ← score
            maxChild ← $v'$
    **return** maxChild

# Chapter 3:

# Methodology

## 3.1 Cribbage Java Application

For this project, a Cribbage game and AI players were implemented in Java. Both a text-based and Java Swing based Graphical User Interface (GUI) were made for human play against the AI players. In Figure 3-1, we see an image of the Java GUI after the first hand of a game. The program implements all the rules found in standard play described in Chapter 2.

## 3.2 Cribbage AI players

Two baseline AI players were implemented for the purposes of testing the MCTS-based players. These were a random player, and a simple scripted player. The scripted player takes any action which will give the most immediate points. In the case of discarding to the crib, it keeps the four cards which (on their own, without a fifth card) would have the highest point total at the end-of-hand score count. During play it plays the card which earns the most points immediately for that play, with ties broken by selecting the first such play scanned. The scripted player could be considered equivalent to a human player who knows the rules but never looks ahead to future moves in planning.


**Figure 3-1: Cribbage Java GUI**

### 3.2.1 Cheating UCT

The simplest version of a MCTS Cribbage player agent we implemented is a *cheating player*. It uses the UCT algorithm outlined in section 2.2.3 while considering the game to be a perfect information game. It plays as if all cards are known to both players, and the order of cards in the deck was known to all players. The Cheating UCT agent ignores the issue of imperfect information, but it serves as a useful benchmark for comparing to other AI players.

In all versions of MCTS used in this research the best move is chosen by comparing the visit counts of all the child nodes from the root of the tree after the computational or time budget is past, and selecting the move corresponding to the child node with the highest

visit count as the move taken. Another method that could be used is to choose the node with the highest average value.

Another decision needed for all MCTS-based players in this research is where to stop the search. Typically, MCTS simulations are played out until the end of a game. The value returned for those simulations is either 1 (win) or 0 (loss). In Cribbage, each player receives a certain number of points during the play and counting of each hand, and then play resets as a new hand is dealt with nothing but the total score and dealer position (which changes) carrying forward to the next game state. Running the simulation past the end of the current hand isn't necessary, since the value of actions taken during the hand can be measured at the end of the hand. Therefore, all MCTS-based agents in this research use the difference between the players' point gains at the end of the current hand as the value of a simulation. For example, if player A gains 3 points from a hand and player B gains 10 points from the same hand, that simulation would have a value of -7 for player A, and 7 for player B.

### 3.2.2 Determinized UCT

Determinization is a popular way to handle imperfect information in game AI when using Monte Carlo techniques. On any player's move when there is hidden information in the game state, from that player's perspective they may be in any one of many possible game states. That combination of states together form an information set for that player. A determinization of a game state is any random state from the player's information set. For example, at the beginning of a hand after the cards have been dealt to each player, a player

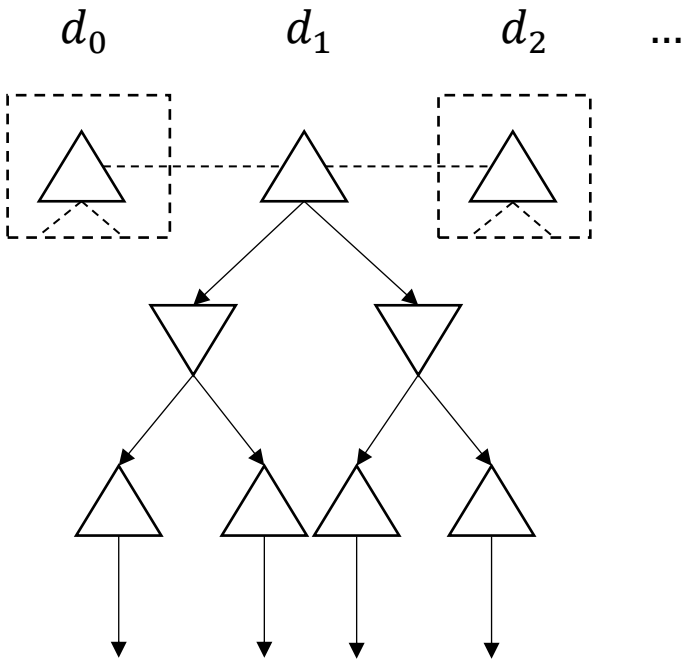


**Figure 3-2: Illustration of Determinized UCT**

knows their own cards but doesn't know the opponent's 6 cards. The opponent can have $\binom{46}{6} = 9{,}366{,}819$ different 6-card hands, so there are that many states in each player's information set in that stage of a hand.

Michael Buro, Timothy Burtak, and others use a technique called Perfect Information Monte Carlo (PIMC) to play games such as the German trick-based card game Skat [6], [7]. The technique involves repeatedly taking determinizations from the information set the player to act is in, and then using a standard AI technique for playing out that determinization as a perfect information game. The move chosen at the end of the search is the one that had the most success over all the determinizations. In Figure 3-2, we see a determinization, $d_1$, which is searched independently of other determinizations.

In this research, we used determinizations, as in PIMC, but used the Cheating UCT agent to play out each determinization, as opposed to minimax or other methods. The visit

counts of all child nodes of the root node are summed across all determinizations, and the action corresponding to the child node with the most visits is returned as best move by the Determinized UCT agent.

To create a determinization, each part of the game state which the current player has seen is held fixed, and the rest of the game state is randomized. Since Cribbage is a card game, it is sufficient to shuffle the deck of cards (excepting the cards seen by the current player) and reissue cards to the other player per the cards' positions in the shuffled deck. No attempt is made to prevent repeats of determinizations, since the frequency of a card appearing in a determinization is equivalent to its odds of being in the necessary part of the deck to be in play. Only 13 cards of the deck are used in each hand of Cribbage (in our representation of a deck, cards 1-6 form one player's hand, cards 7-12 form the other player's hand, and the "cut" card is card 13), so there are many determinizations that result in the same cards being in play.

In addition to the parameters of execution time and exploration constant, Determinized UCT requires a fixed number of determinizations. Since MCTS does not guarantee convergence in a fixed number of iterations (it approaches the solution over time), the computational or time budget must be split over some number of determinizations. Experiments with different numbers of determinizations are presented in section 4.4.

### 3.2.3 Single Observer-Information Set MCTS

The algorithm for the final agent implemented in this research, Single Observer-Information Set MCTS (SO-IS MCTS), is adapted from a 2012 paper by Peter I. Cowling,

Edward J. Powley, and Daniel Whitehouse [8]. It relies on determinizations, like Determinized UCT, but it only builds a single MCTS tree in which nodes correspond to information sets rather than single states.

Before each iteration of the search, a determinization compatible with the current player's observable game state is created. Then as the selection, expansion, and simulation steps are conducted for the iteration, only actions which are compatible with the current determinization are considered. In each iteration, information from previous determinizations is used in selecting nodes to traverse, so the decision that a player would make is based on how good that decision has been in previous determinizations that included the same choice as a possibility.

In Cribbage, the actions available to the player to act (maximizing player) in a given node are the same in all determinizations in which that node is reachable, because that player's cards remain the same in all determinizations. The opponent (minimizing player), whose unseen cards are randomized in each determinization, may have actions available from a given node in one determinization but not in another, because the actions correspond directly with cards in the player's hand, which are different in each determinization. In Figure 3-3, the actions available in the $i^{th}$ determinization are shown in solid lines, while actions unavailable in the $i^{th}$ determinization are shown in dotted lines. This example reflects a situation that can occur in a Cribbage game because the maximizing player (triangles pointing upwards) has all actions available during all determinizations, whereas the minimizing player has some actions it can't take in some states in this determinization.

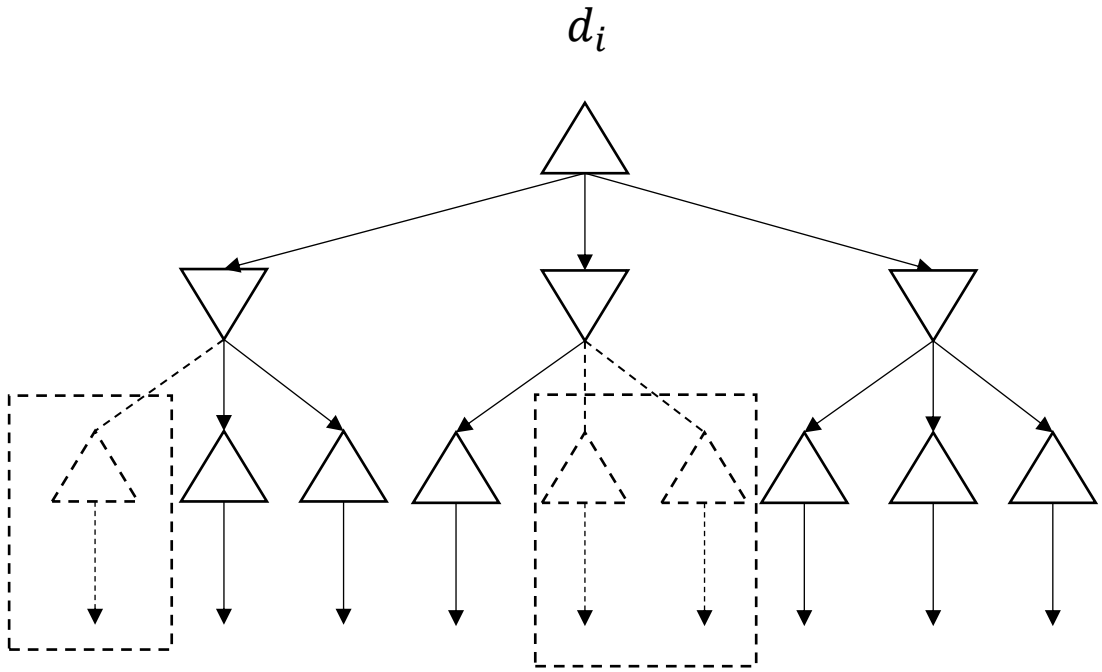


**Figure 3-3: Illustration of Single Observer Information Set MCTS**

In our implementation of SO-IS MCTS, each node stores the action that produced it rather than a state (each node is associated with an information set, which may include multiple states). References to child nodes are organized by keeping track of all actions that have been taken from that node. Unlike in the Cheating or Determinized UCT implementations, for an opponent decision node the possible actions include all legal card plays that the root player has not seen in a previous part of the hand.

The authors in [8] modify the selection formula for SO-IS MCTS by considering only the number of times a node has been available for selection, rather than the number of times its parent node has been selected. In normal UCT those numbers are the same, but in SO-IS MCTS, actions that are rarely available would be over-selected if the number of visits to the parent were used in the second term of the formula. The modified formula is:

$$\arg max_{\substack{v' \in \, children\ of\ v \\ consitent\ with\ d}} \quad \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln A(v')}{N(v')}}$$

where:

- $Q(v)$ is the accumulated total value *for the player to act at node* $v$ from the backpropagation step of previous iterations;
- $N(v)$ is the number of times $v$ has been visited in the selection step of previous iterations;
- $A(v)$ is the number of times the node $v$ has been available for selection;
- $c$ is an exploration constant that can be adjusted for different domains.

# Chapter 4:

# Results and Discussion

Tests were run to compare the performance of each of the AI players, as well as to compare different values for the parameters that can be changed for each of the MCTS-based players. The first player to act as dealer has an advantage in Cribbage, since that player benefits from the points available in the crib first. With random play by both players the dealer won 60% of one million games played. Therefore, all other tests described in this chapter were run by alternating the first player to deal, and even numbers of games were played in all cases so that the first dealer advantage would not affect the results.

Some experiments presented in this chapter comparing AI performance used number of search iterations as the search budget, while other experiments used time per move as the search budget. All testing was done on a machine with an Intel i5 7500 CPU running at 3.4 GHz. For comparison, one second of SO-IS MCTS search used between 50,000 and 150,000 search iterations, which is largely dependent on what stage of a Cribbage hand the search is starting from (a search iteration of a shallower tree is generally faster).

**Table 4-1: Win rates of AI players against each other**

| **AI Players** | Random | Scripted | Cheating UCT (c=1.75, 1s) | Determinized UCT (c=1.75, 1s, 100 determs.) | SO-IS MCTS (c=2.0, 1s) |
|---|---|---|---|---|---|
| Random | - | 2.2% | 0.0% | 19.0% | 1.2% |
| Scripted | 97.8% | - | 24.0% | 87.5% | 35.5% |
| Cheating UCT (c=1.75, 1s) | 100.0% | 76.0% | - | 97.7% | 64.0% |
| Determinized UCT (c=1.75, 1s, 100 determs.) | 81.0% | 12.5% | 2.3% | - | 6.2% |
| SO-IS MCTS (c=2.0, 1s) | 99.8% | 64.5% | 36.0% | 93.8% | - |
| Note: Win rates given are for the player on the left-hand side, in 400 games played. | | | | | |

## 4.1 Comparison of AI Players

In Table 4-1, the win rates of each AI player against each other are given for a sample of 400 games of each pairing. Both the scripted and Cheating UCT players outperform the random player, with 97.8% and 100% win rates, respectively. Cheating UCT also outperforms all other players, as expected.

Determinized UCT performed worse than we expected, winning only 81% of games against the random player, and 12.5% of games against the scripted player. SO-IS MCTS performed better than all players except the Cheating UCT player, winning 99.8% of games against the random player, 64.5% of games against the scripted player, 36% of games against the Cheating UCT player, and 93.8% of games against the Determinized UCT player.

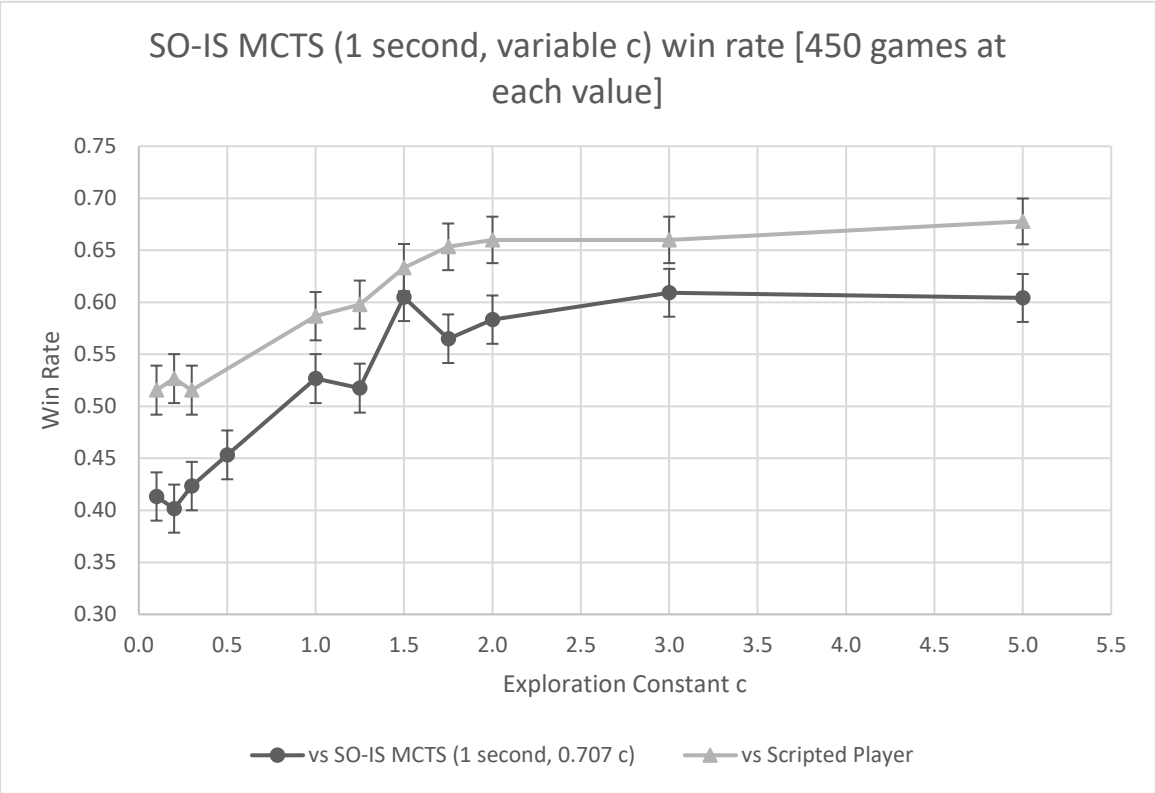


**Figure 4-1: Effect of exploration constant on SO-IS MCTS performance**

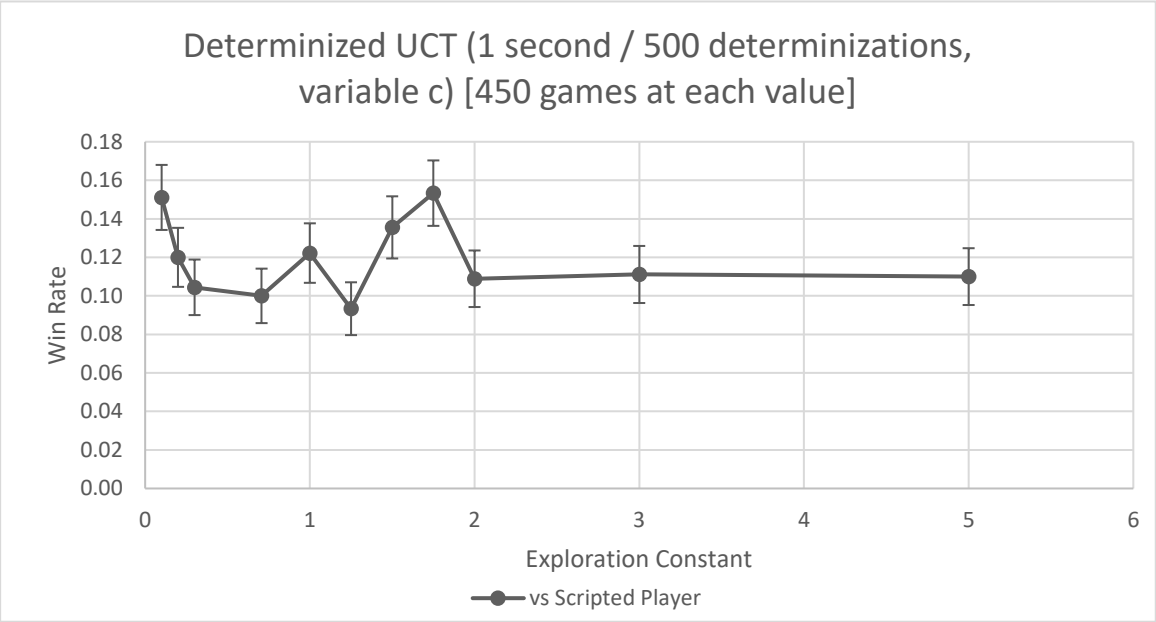


**Figure 4-2: Effect of exploration constant on Determinized UCT performance**

## 4.2 Variation of Exploration Constant

The exploration constant, $c$, used in applications where the simulation values are in the range $[0, 1]$ is often $1/\sqrt{2} \approx 0.707$ [1]. Therefore, we used $c = 0.707$ as a starting value when first performing tests.

In Figure 4-1, we see that SO-IS MCTS performance improves with higher values of $c$ up to a value of 2.0. The test was conducted against a SO-IS MCTS player using $c = 0.707$ and against the scripted player, with 450 games played at each value. For the main AI comparison tests a value of $c = 2.0$ was used for SO-IS MCTS.

In Figure 4-2, we see the results of varying the exploration constant for Determinized UCT. Performance is highest between 1.5 and 2, but the results for very small values of $c$ are also high. That may be because each determinization gives a relatively small game tree to explore, causing the exploration constant to be less important. For the main AI comparison tests a value of $c = 1.75$ was used for Determinized UCT.

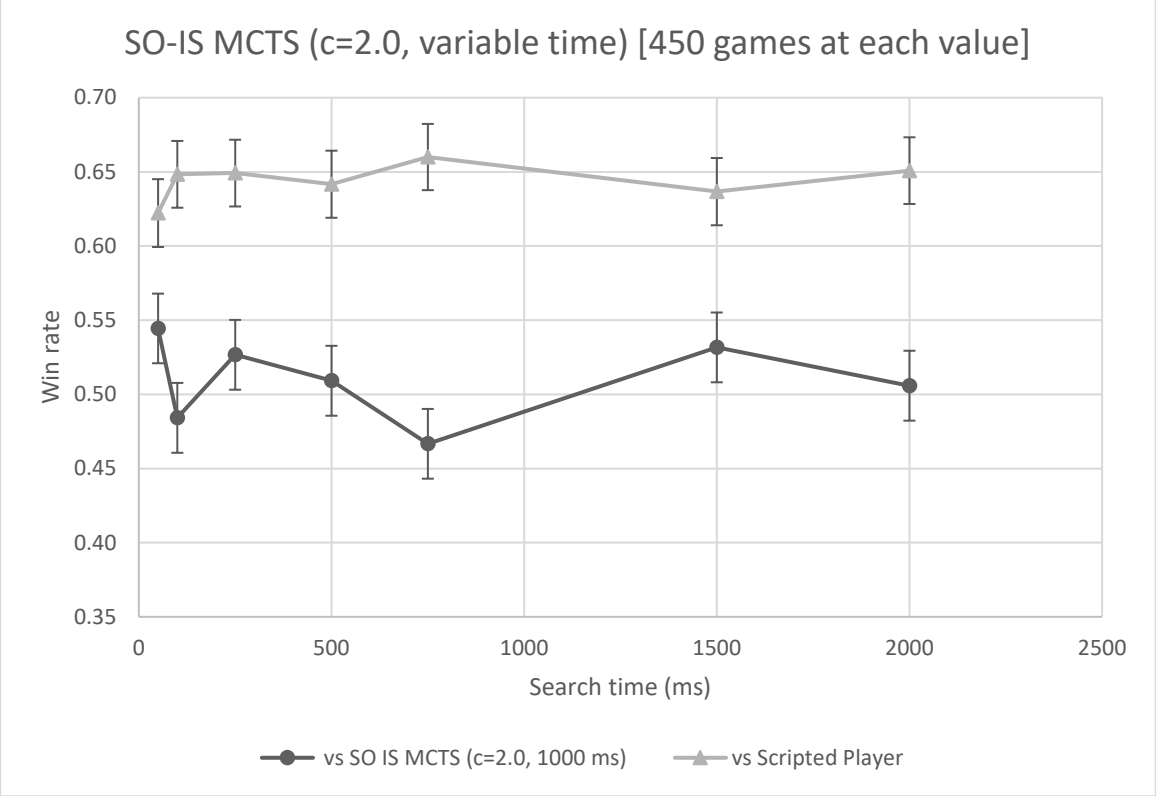


**Figure 4-3: Effect of search time on SO-IS MCTS performance**

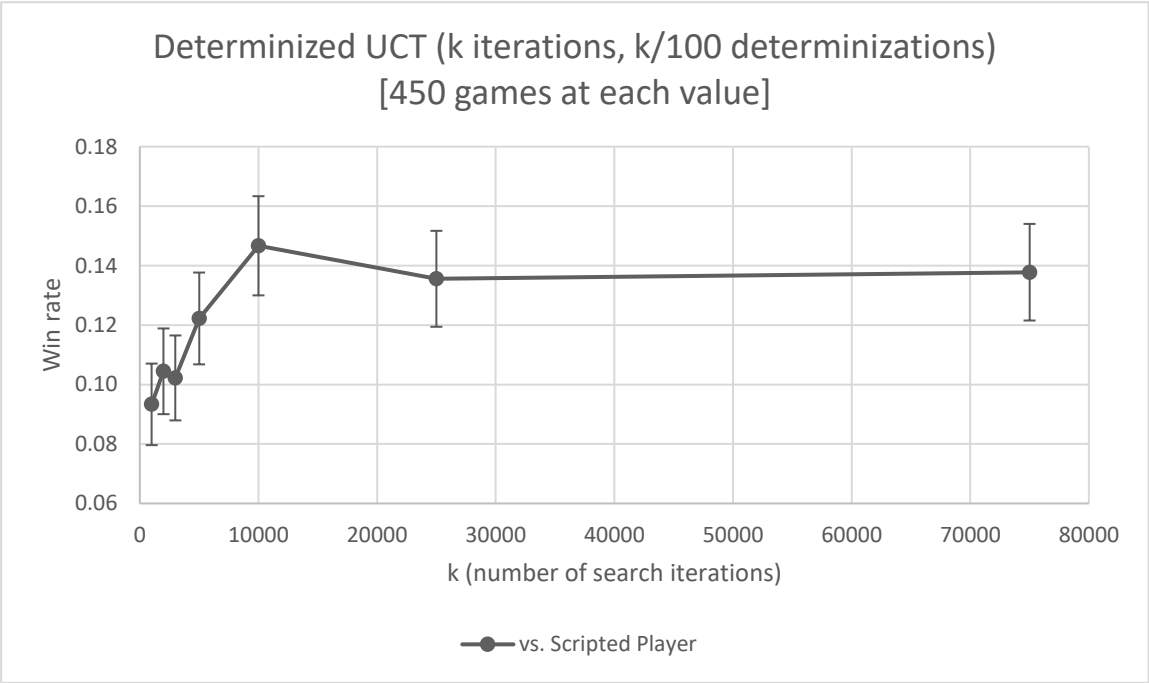


**Figure 4-4: Effect of search iteration budget on Determinized UCT performance**

## 4.3 Variation of Search Iterations

Intuitively, devoting more time or search iterations to a move should produce better results. In Figure 4-3, we see that SO-IS MCTS performance improves little beyond 100-250ms per move in test of 450 games at each amount of search time against the scripted player. At both 100ms and 2000ms the SO-IS MCTS player wins 65% of games against the scripted player. Against a SO-IS MCTS player set at 1000ms searches, there is no clear trend, and the 50ms player wins 54% of games. We believe that a single Cribbage hand is too small of a game to see a large impact above a few thousand search iterations.

In figure Figure 4-4 we see that the Determinized UCT player improves in performance as the number of search iterations increases from 1,000 to 10,000, winning 9.3% and 14.7% of games against the scripted player, respectively. Above 10,000 search iterations the Determinized UCT player does not improve for the values that we tested.

For the search budgets tested, a higher budget improves performance for both Determinized UCT and SO-IS MCTS, but in each case there is a plateau of performance improvement.

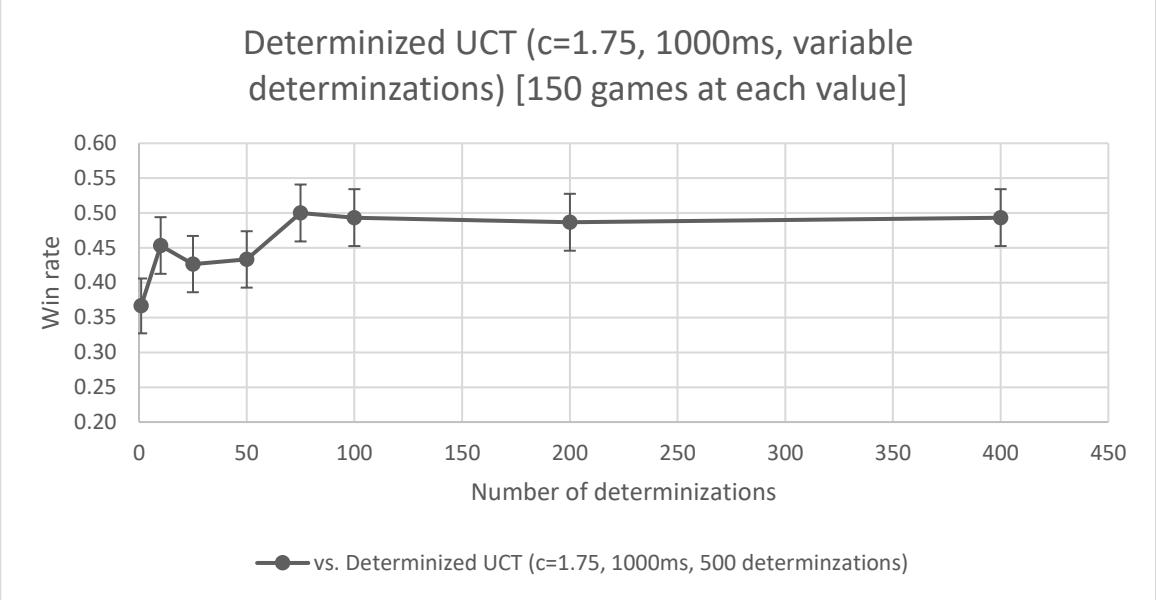


**Figure 4-5: Effect of number of determinizations on win rate**

## 4.4 Variation of Number of Determinizations

In Figure 4-5, we see the effect of varying the number of determinizations for Determinized UCT on performance against Determinized UCT with 500 determinizations. Both AI players searched for 1000ms in all tests, with exploration constant $c = 1.75$. Setting the number of determinizations lower than 75 resulted in worse performance than the 500 determinizations player, ranging from 36% to 43% in 150 games. Setting the number of determinizations to 75 or higher (up to 400) resulted in a win rate of 50% or very close to 50% against the player playing with 500 determinizations.

## 4.5 Discussion

The poor performance of the Determinized UCT player was surprising. Intuitively, it makes sense that a move that is good in many determinizations would be a good move to take on average. However, playing a given determinization as if it were a perfect information game

means giving both players access to information that they wouldn't have when evaluating the optimal moves for each player. It is also possible that using MCTS for the playouts in the determinizations is ineffective, since it's hard to choose the right amount of computational or time budget to give to each determinization.

### 4.5.1 Strategy Fusion

Cowling et al. note that strategy fusion is a problem for both Determinized UCT and SO-IS MCTS [8]. Strategy fusion is an effect in which the searching agent assigns incorrect values to nodes in the tree, because it searches as though it can distinguish between different states in an information set and make a choice based on which state it is in. SO-IS MCTS prevents some situations of strategy fusion, but not all [8]. Long et al. propose measurable properties of games that can indicate to what extent game properties that cause strategy fusion and some other errors are present [9]. Strategy fusion and other errors of this nature may account for why Determinized UCT performs so poorly for Cribbage.

### 4.5.2 Information Leak

One consequence of only considering information sets from the maximizing player's perspective in SO-IS MCTS is that information about that player's hand "leaks" to the model of the opponent represented by opponent action nodes in the search tree. Since in all determinizations the maximizing player's cards are held constant, the nodes of the tree where the opponent is to act will converge to optimal play against the maximizing player's actual hand. The opponent model is that of a player who happens to always know what the other player has in their hand.

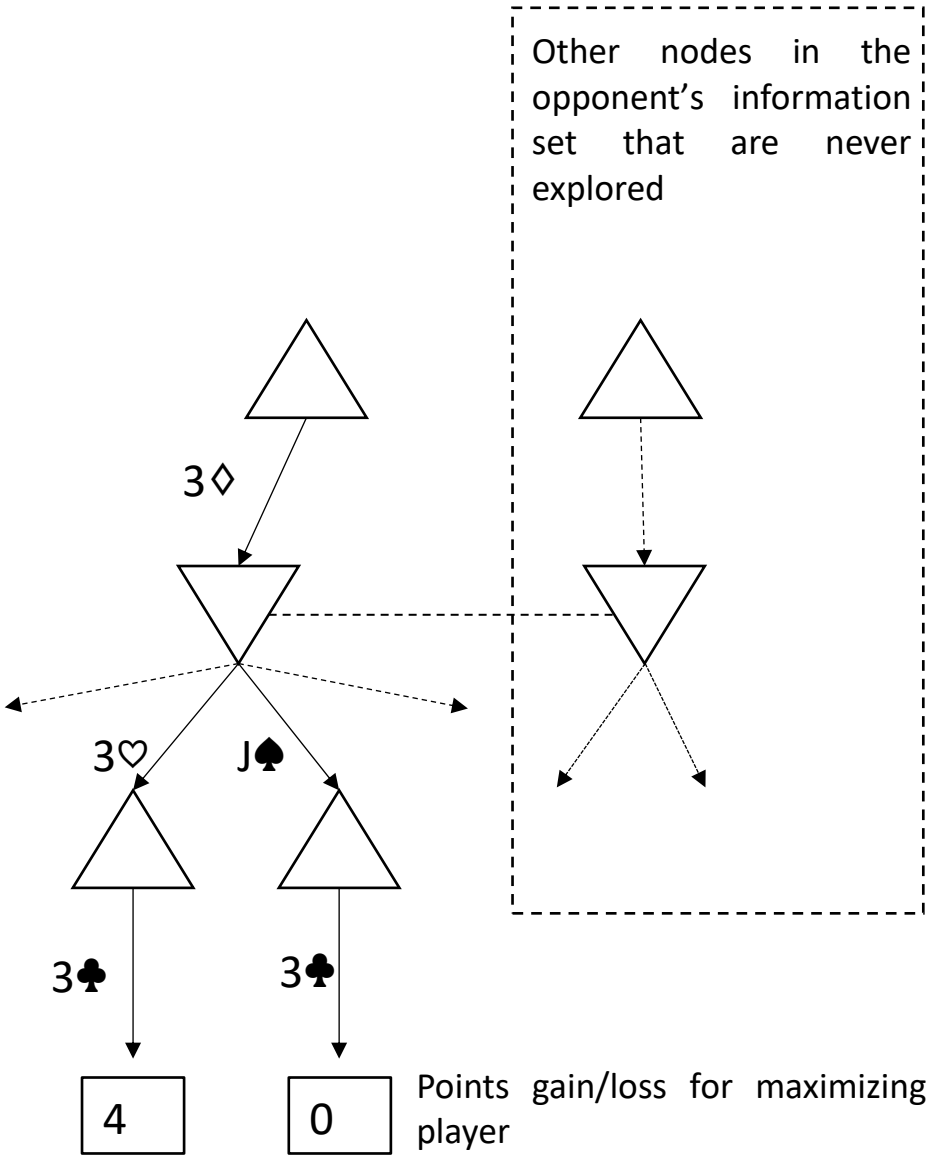


**Figure 4-6: Example of Information Leak in Cribbage**

In Figure 4-6, we see a partial search tree in which the root or maximizing player holds only a pair of 3s in the play stage of a hand. In the subtree shown, if the root player plays the 3♢, then in the current determinization the opponent holds two legal cards to play, a J♠ or a 3♡. In Cribbage, playing a second card of the same rank gives that player two points, but the first player can then play a third card of the same rank for six points, for a

4-point net gain. For that reason, “leading with a pair” is a common strategy by humans. In this example, the root player gains four points if the opponent plays their three, and zero points if not. The optimal move found for the opponent node in this example will be to play the J♠, because in every determinization in which the opponent has a three card, the maximizing player will also have their second three card. Therefore, the search tree will find a high value for the opponent action of playing the Jack, and a low value for the maximizing player’s node at the root of this subtree. This contrasts with what we believe to be a good strategy in Cribbage; depending on the opponent’s strategy, leading with a card that is part of a pair is often a good strategy because the opponent doesn’t know what cards the other player has.

# Chapter 5:

# Conclusion

In this research, we have implemented a Java version of the game of Cribbage, a scripted Cribbage player, and several MCTS-based players for Cribbage. We showed that SO-IS MCTS outperforms Determinized UCT and the basic scripted player in Cribbage without using any game-specific enhancements. SO-IS MCTS wins against the scripted player in 64.5% of games, and against the Determinized UCT player in 93.8% of games. SO-IS MCTS wins 36% of games against the Cheating UCT player. The strength of the algorithm for this game appears to be in its model of the player to act's information set and the opponent model, since increased search time, or variation of other parameters, could not improve the performance of Determinized UCT.

## 5.1 Future Work

In this research, the emphasis was on investigating the performance of MCTS algorithms for an imperfect information game while using a minimum of domain-specific knowledge. There are several simple enhancements that could be made to the algorithms already implemented that would likely offer some improvement at little cost.

### 5.1.1 Endgame Play

Playing MCTS simulations only until the end of a hand is convenient, but in fact the current implementation should lead to bad strategy in the last one or two hands of a game. Cribbage games end as soon as any player reaches 121 points, even mid-hand, so the optimal strategy in a hand that could be the final one is often to prioritize cards that are good for scoring points in the play stage of the hand, rather than the count stage. Also, because the dealer counts their hand and crib last, often the non-dealer can win in the final hand before the dealer counts, even if the dealer could also reach 121 points in the same hand.

The solution would be to switch the algorithm to a version that values simulations as either a win or a loss instead of net point gain whenever the score of one player gets within a certain range of 121 points. Rather than having to simulate future hands with completely random deals, future hands could just award a fixed number of points to each player. The amount per hand could be drawn from a database of games played by a good AI player. The current implementations of the MCTS-based AI players will instead play to maximize net point gain over the entire hand, regardless of how likely it is that a player may win part-way through the hand.

### 5.1.2 Simulation Policy

The simulation default policy used in all MCTS players for this research is random play. That isn't a requirement for MCTS. One easy enhancement to try would be to play with the scripted player instead of random for the simulation "default" policy.

### 5.1.3 Analysis of Game Tree Properties of Cribbage

In [9], Long et al. propose properties of game trees that can be used as predictors of the success of Perfect Information Monte Carlo (PIMC) Search. The Determinized UCT player used in this research is a type of PIMC search. Future work on the game of Cribbage could involve analyzing the properties of Cribbage game trees to explain the performance of the Determinized UCT player.

### 5.1.4 Other Ways to Handle Player Information Sets

Cowling et al. propose enhancements to SO-IS MCTS, including Multi Observer-IS MCTS (MO-IS MCTS), which builds a separate tree for each player in a game [8]. Each node in a tree in MO-IS MCTS corresponds to an information set for that player. This algorithm would address the strategy fusion and information leak issues highlighted in section 4.5 [8]. Implementing the MO-IS MCTS algorithm for Cribbage could be considered for future work on Cribbage. Implementing SO-IS MCTS and MO-IS MCTS for more complicated games with larger search trees is another area for potential future work.

# Bibliography


[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[2] Jan Schäfer, Michael Buro, and Knut Hartmann. "The UCT algorithm applied to games with imperfect information." Diploma, Otto-Von-Guericke Univ. Magdeburg, Magdeburg, Germany. 2008.

[3] Szita, István, Guillaume Chaslot, and Pieter Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*, pp. 21–32, 2010.

[4] Coulom, Rémi. "Efficient selectivity and backup operators in Monte-Carlo tree search." in *International Conference on Computers and Games*, pp. 72-83, 2006.

[5] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *European Conference on Machine Learning: ECML 2006*, pp. 282–293, 2006.

[6] Timothy Furtak and Michael Buro, "Recursive Monte Carlo search for imperfect information games," in *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, 2013.

[7] Michael Buro, Jeffrey Richard Long, Timothy Furtak, and Nathan R. Sturtevant. "Improving State Evaluation, Inference, and Search in Trick-Based Card Games." In *IJCAI*, pp. 1407-1413. 2009.

[8] Peter I. Cowling, Edward J. Powley, and Daniel Whitehouse, "Information Set Monte Carlo Tree Search," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 120–143, 2012.

[9] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search." in *AAAI*. 2010.