Monte-Carlo Tree Search AI Program for Solving Multiple Types of Solitaire

130003128 Laura Brewis

University of St Andrews

7th April 2017

Supervisor: **Ian Gent**

1 Abstract

Extensive research has gone into creating intelligent AI programs to be able to play certain games perfectly. Games like Chess and Go have been focused on over the past few decades, and both games can now be played well by a computer (in Chess' case, being the best in the world). These are both examples of two player games with perfect information (where both players can see all of the game, and there is no randomness involved). A type of game that hasn't been focused on as much is one-player games with non-perfect information, such as Solitaire/Patience games.

The aim of this project is to create an Artificial Intelligence program to play and evaluate multiple different Solitaire games, each of which have some level of randomness to them: such as unknown cards or shuffling of the deck. Some examples of these types of games include Klondike, Spider and Sir Tommy.

This general AI player will compare the different games and compute the probabilities of winning for each type of game, provided the user was to play the optimal move in each turn.

The program will primarily use the Monte Carlo Tree Search (MCTS) method (known for its use in computer Go) to decide which moves to play. The MCTS method will be independent of the type of game so that it is interchangeable, with the hope that other types of solitaire can be played with relative ease.

2 Declaration

Contents

List of Figures

3 Introduction

The main aim for this project is to produce a program using Artificial Intelligence to solve different types of solitaire games. Specifically, the algorithm Monte Carlo Tree Search (MCTS) was chosen due to its success in many other games recently. The game Go has been the first main success for the MCTS method, which has been widely studied and improved upon over recent years. [7]

Go is a two-player, perfect information game, meaning both players can see all the information on the current state of the game. This is similar to games such as chess and checkers, which have both been extensively researched in AI, with computer programs having overtaken human players many years ago. Go is played on a large board (19x19) and at any point in the game there are up to 361 legal moves that can be made [17]. The search space here (which is all the possible combinations of games) is larger than the number of atoms in the universe [2]. This shows how difficult Go is, and how computer versions of Go have had a much slower progress than other games. However, the introduction of MCTS has extremely improved these programs, which is a remarkable achievement in artificial intelligence. [7]

A type of game that hasn't been studied in great depth is solitaire, which is a family of different card games. Many of these games are two-player, with non-perfect information. Examples of Solitaire include Klondike, Sir Tommy, and Spider Solitaire. These were chosen to study in this project due to their general popularity, yet computer solvers have not been extensively studied, especially when compared to other games, like Chess and Go. [10]

This project focuses more on creating an average solver that works for different types of solitaire, rather than creating a competent solver based on only one or two solitaires. Because the solver isn't tailored to a certain game, it was not expected to be the optimal solver, and it has solving rates lower than other specific solvers. Nevertheless, creating a general solver that can easily be adapted for new types of solitaire is more complex, as it cannot have any type-specific heuristics and relies solely on the MCTS method for its function.

One of the secondary aims of the project was to be to calculate the percentage of games that are winnable with perfect play for each type of solitaire. This is something that has not been achieved before. Not being able to calculate these odds has been deemed "one of the embarrassments of applied mathematics" [22]. Therefore, the aim was that this program would be able to calculate, albeit approximately, these odds of success. But because this project does not claim to have "perfect play", this objective focuses more on the solver's best play instead.

3.1 Objectives

The objectives of this project, as set out at the start of the project (September 2016) are as follows:

Primary Objectives

1 Build a program for humans to be able to play multiple types of solitaire

2 Create an AI to play good moves for these games

3 Create an AI program that uses MCTS to solve the game

Secondary Objectives

1 Modify the program to be able to give it a state of the game, and for it to select the best move/best x moves

2 Play the AI multiple times for multiple types of solitaire to work out the winnable games with perfect play

Tertiary Objectives

1 The AI can give the best x moves and the % the game can be won with each move

2 Have the AI check the % winnable each move is, and show how reliable itself is

3 Evaluate AI against itself/other solvers

4 Create a rule system to represent these different types of Solitaire

4 Context survey

This section describes how the algorithm Monte Carlo Tree Search (MCTS) works, and how it is adapted for one-player, non-perfect information games. It also describes solitaire games, and a way of representing the game. In addition, we will discuss related research in this field, focusing on research conducted on MCTS.

4.1 Monte Carlo Tree Search

4.1.1 Strengths of MCTS

The MCTS method works by playing the game with random moves, and building a search tree from those results. This tree is the search space that is explored by the algorithm. By using this method, we do not need to try every possible move to see which one is best, as we instead take the average result from many random plays.[5]

Another search method which has been very popular in the past is alpha-beta. This has been successful in games such as chess and checkers. However, this method doesn't work well when there is a high branching factor in the game; i.e., a large amount of possible moves in each state of the game. Chess has a low branching factor of 35, yet Go has a much higher branching factor of 250 [15]. This means that the alpha-beta method, and others like it, is insufficient for games such as Go. MCTS, on the other hand, works well with high branching factors. It allows us to search deep in the tree without searching every possible move, so we only search a selection of the overall search space. This is why MCTS has become extremely popular in computer Go. [9]

The second main disadvantage of alpha-beta, and other similar programs, is that it requires a function to evaluate the game at a each state. This is difficult to do without a lot of in-depth knowledge about the game. MCTS can be used in these cases, where little game-specific knowledge is known. For example, some of the best computer Go programs were created by people who did not have in-depth knowledge of the game and approriate tactics; they only knew the rules. This is exploited in this project, where a single MCTS program can be used to solve different types of solitaire. Here, only the rules are defined, and no additional information is included, such as game-specific heuristics.[5, 12]

MCTS has been used in different variations in the past for imperfect games such as scrabble and bridge. By using the tree structure, MCTS has greatly improved Go programs [5]. In March 2016, Google's AlphaGo beat one of the top human Go players in the world, using MCTS as part of the program. This shows how this method has greatly improved the modern front of Artificial Intelligence. [14]

4.1.2 How MCTS Works

MCTS is a best-first algorithm, which means it chooses the most promising node of the tree to explore, with respect to a certain rule. Figure 1 shows the four sections to this method:

1 Selection - here the algorithm chooses which node to explore by following the *Tree Policy* rule until it reaches a leaf node or a node with children that have not yet been explored.

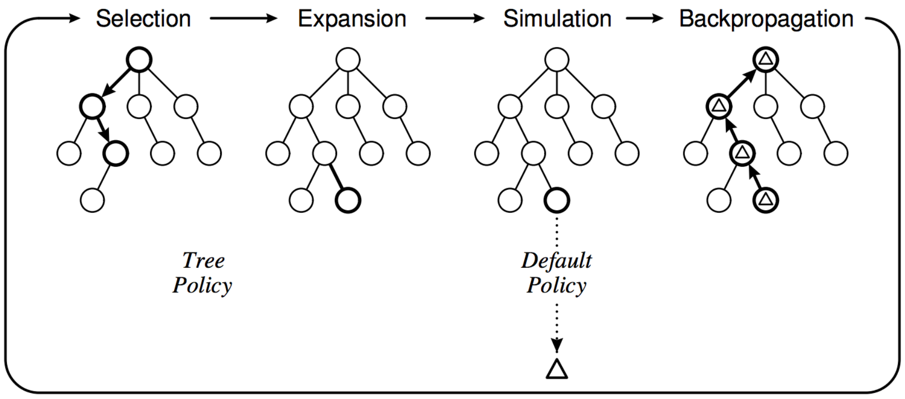
2 Expansion - once we have reached this leaf node, a new game state is created in a new node.

3 Simulation - this new state is then played with moves according to the *Default Policy* rule. This is often referred to as a *random playout*

4 Backpropagation - the result from the simulated game is stored and the parent nodes have their score modified according to this result.

[5, 12]

Figure 1: Diagram of MCTS Method [5]



By performing this algorithm multiple times, we can create a tree with a good representation of scores, which have been calculated by averaging these random simulations. Then we select the node which has the best score as the move to play. [12]

In the *Selection* part of the algorithm, the choice of *Tree Policy* will affect how well the program plays. The *Tree Policy* decides on a trade-off between exploring moves that have not yet been considered and looking at moves that look the most promising. This *Tree Policy* decides on the best node by looking at the score of that node. [5]

The other main part of the algorithm is the *Default Policy*, which dictates which moves are made during the *Simulation* part of MCTS. The simplest implementation for this policy is for it to make random moves. However, this would not give good results, as random play isn't realistic when aiming to perform well in a game. This *Default Policy* starts with a state of the game (that isn't an end state) and simulates the game to result in an estimate score at the end. This program's implementation of the policies is described in section 8.1.2. [5]

4.2 Solitaire

In this project, we have used some standard terminology for dealing with solitaire. Figure 2 shows a game state of a Klondike game, which is a snapshot of the game at a certain point. The game state will change if a move is played. This image is labelled with the four possible locations a card could be in:

1 *Stock* - where the deck of cards sit. This becomes empty when all the cards have been played to other locations.

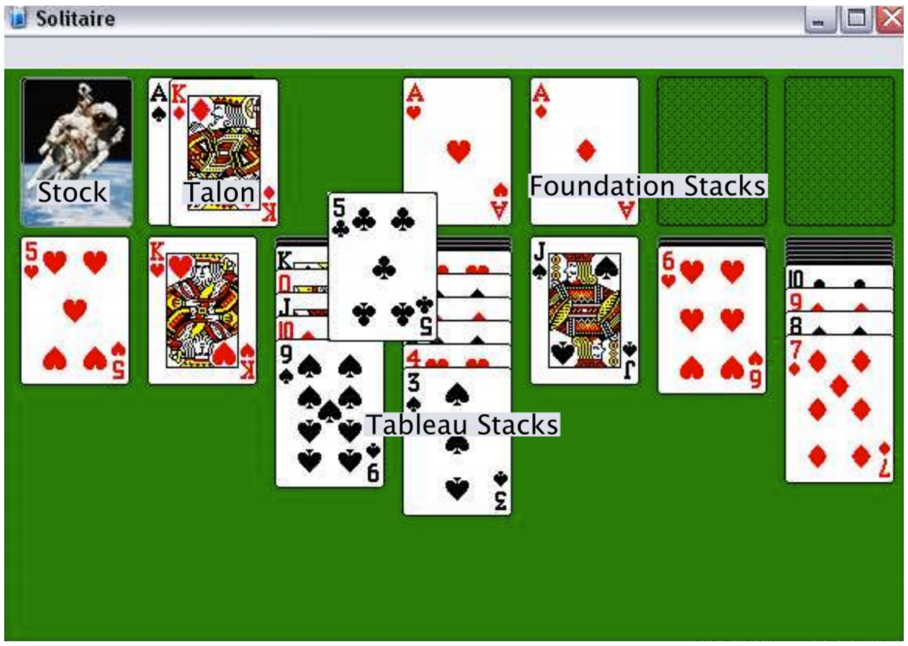
2 *Talon* - where the topmost card from the stock is placed when it has been removed from the stock.

3 *Foundation* - consists of four piles (one for each suit) where the cards are stacked, typically in suit from Ace to King.

4 *Tableau* - multiple piles where cards are placed during the game, often according to certain rules. Sometimes at the start of a game the tableau is empty or it contains cards.

[4]

Figure 2: Snapshot of a solitaire game [4]



This system of describing the locations of the cards is useful when dealing with multiple types of solitaire, as many types share these same locations, but with different rules as to how legal moves are defined. Towards the end of this project, a notation was devised to be able to describe some solitaire types, furthering this standardisation. This notation is described in section 8.3.

In order to win a game of solitaire, all of the cards must be placed in the foundation. For a game with 4 suits and 13 cards per suit (Ace-King), this means that there will be 52 cards in the foundation. The *score* of a game is defined as the number of cards in the foundation. This value is used to determine how well a game has been played.

For this project, the first type of solitaire to be studied was Klondike. For the rules of how to play Klondike, see Appendix B. In our version, we only turn over one card at a time from the stock, and we allow unlimited re-deals of the deck. Cards can be moved between the Tableau, and cards can be moved from the Foundation to the Tableau. These rules sometimes vary in different interpretations of Klondike. This type of solitaire was chosen to study due to its popularity and simple rules.

The second type of solitaire focused on was Sir Tommy. The rules for this can be found in Appendix C. Sir Tommy was chosen as it was different enough from Klondike to need separate rules, so that the MCTS program had to be generalised, yet it still had the same locations (stock, talon, foundation and tableau).

4.3 Related Work

There is much research on the Monte Carlo Tree Search method, a lot of which is discussed in *"A Survey of Monte Carlo Tree Search Methods"* [5], published in 2012. This survey describes the method and the progression of AI in games, as well as discussing in depth different enhancements and variations to the basic MCTS method, especially improvements in the Tree Policy. *"Monte Carlo Tree Search and Its Applications"* [16], 2015, is another article which describes some of the main uses of MCTS, while also including some of the more modern developments, such as using MCTS in the board game *Risk*. This use of MCTS is described in more detail in *"Sample Evaluation for Action Selection in Monte Carlo Tree Search"* [8].

Solitaire has been researched under a multitude of search methods. For example, the solitaire game Freecell has been researched with the A* search algorithm, which has been found to produce "optimal solutions" [11]. In addition, a version of Klondike called *Thoughtful solitaire* has been studied intensively. This game is the same as Klondike, but with all the cards known to the player from the start. This research uses a *rollout* method, which uses simulated play and a separate strategy to pick a move that can win. [22, 4]. This rollout method can be adapted to work for MCTS, as described in *"Nested Rollout Policy Adaptation for Monte Carlo Tree Search"*[18].

There has been some research on using Monte Carlo methods with solitaire, without the tree structure. For example, the article *"Lower Bounding Klondike Solitaire with Monte-Carlo Planning"*[3] describes studying Klondike solitaire with different methods, resulting in an algorithm that can win over 35% of Klondike games. [3]

There is evidence of work with solitaire with Monte Carlo planning (and nested rollouts). However, research on using MCTS with solitaire was not found. In addition, there were no studies found on a general solver that could solve multiple types of solitaire. Despite this, the methods and algorithms looked at in these other research areas can be adapted and used as guidance here in this project.

5 Software Engineering Process

This project was conducted over a 7 month period, from September to April. In the first few weeks of the project, the majority of the work was defining the project details. Because the project was originally so open-ended, many decisions had to be made, such as which search algorithm to use. A lot of reading around the subject was conducted in these weeks, which is where a lot of the research from section 4 was found. From this reading, MCTS looked like a promising option to study in this project due to its success in other games.

After the initial research, the objectives and description of the project were defined. Once these details were decided upon, I created a rough plan to guide my work over the year. Because we had a long time to complete this project, I decided to focus my work week-by-week, in order to complete more, smaller tasks and to improve my efficiency.

After a couple of months of working on the project, my supervisor and I realised the plan was ambitious. The MCTS method was a lot more difficult than we first thought, which meant it took longer to implement than planned. Because of this, we concentrated on the prioritised, primary objectives.

Since the start of this project, during term time a meeting was conducted between me and my supervisor every week. These meetings acted both as a retrospect meeting to discuss the work completed the week before, and as a planning session for the work to be completed by the next meeting. Each week was treated like a sprint, with the higher prioritised work to be completed first, and any other to be done if there was still time.

Sometimes the work was not completed from the week before, due to unpredicted complexities, or other commitments and deadlines restricting the time I could spend on the project that week. In these cases, my supervisor and I discussed the problems behind this and kept the requirements for the next meeting as the same.

In addition, in some of the meetings we discussed the theory behind specific parts of the project. For example, after the MCTS method had been implemented, my supervisor was not entirely sure it was working as it should. We went through the theory of the method and after that meeting I realised I had implemented some of the method wrong and was able to re-write it.

This iterative process of working week-by-week enabled me to meet my targets. I could then focus on the project as a collection of smaller components (such as creating a working Klondike game, creating the MCTS algorithm, creating the Sir Tommy game, etc...), rather than trying to work on everything at once.

Also, working in this agile way allowed me to identify the parts of the project that weren't working as well. For example, the MCTS does not perform as well as thought on Sir Tommy games (shown in section 10.2). After realising this, my supervisor and I discussed how this is acceptable due to the generalising nature of the project; that it is more important to have a program that is able to solve many solitaires in a mediocre way than to be able to solve one or two types really well.

When beginning to write the code to create a user-playable Klondike program, some jUnit tests were created to test the functionality of the card structure. There was quite a lot of thought put into how to represent the locations of the cards in the game, so I wanted to ensure these were working correctly. In addition, it tested the dealing method for Klondike to ensure features such as the face down and face up cards were dealt correctly.

However, later in the project there was a lot of re-factoring to change how the program was run; from having a different class for each solitaire type to having the type be defined by a JSON file. Due to this, the unit tests no longer work. The class *SolitaireTest* holds these tests, most which have been commented out to remove the errors they were creating.

6 Ethics

The original ethics information mentioned possibly interacting with others to aid in testing the human-playable solitaire games, and also checking how much they agreed with the AI moves. However, this was not needed in the project, so there were no ethical concerns in user studies.

Also, there was a possibility of using existing solvers to compare my AI against, which involved ethical concerns in the case of needing to find a solver that was free, or buying it. There was not enough time to find other solvers for comparisons, so these ethic concerns were irrelevant.

7 Design

When designing the project, the first step was to create a game of Klondike solitaire that was playable by a user. There was a possibility of having a user interface, so that the user could click on the card and either drag it to the destination or click on the destination to move it there. However, having a user interface seemed unnecessary work, taking time that could be spent on the AI, which was the main aim of the project. Instead, the game was decided to be played in the command line, with the player simply typing the number of the move they wished to make. This meant that the game had to be able to detect all of the possible moves at any one point in the game. This, when implemented, was also used in the AI solver.

Originally, the code was designed just for the Klondike game. It was made to be game-specific and fairly simple, so that there was a minimum viable product that worked and could be built upon. After this, the code was adapted for Sir Tommy, and then all the code that depended on a certain type of solitaire was gradually phased out while building the solitaire notation.

7.1 MCTS Method

The major part of the project was creating the MCTS AI program. Because MCTS for solitaire has not been researched before, there was not much information to help with this, apart from the basic structure of the algorithm. Many decisions had to be made, such as the the implementation of the tree policy and default policy. Additionally, checking for the end of the game was also planned, which was harder than originally thought for Klondike.

The MCTS method was designed to be independent of the solitaire game. It was designed to be able to work with any game, provided it could see the state of the game and the possible moves available to it. The method chooses the move to make from the list of all possible moves, with the choice being affected by the score calculated in the MCTS tree. This means it does not need to know the specifics of a game to be able to work.

When using MCTS with two player games, the score stored at each node is the win/loss ratio for that node. To adapt this for solitaire, we can change this to be the score accumulated so far in the game; i.e. the number of cards in the foundation. This is used because the aim of the game is to get all cards to the foundation, which would be a score of 52. This is a reasonable indicator of how the game is progressing, as more cards in the foundation implies the game is going better, even though this may not always be the case.

7.2 Solitaire Notation

The notation for solitaire, described in section 8.3, was a way to represent the different types of solitaire so the program could differentiate between them easily. From my research, I did not find any other examples of this type of notation, so it can be assumed to be a new concept in computer solitaire. The only existing representation of solitaire was the standardised names of the locations: stock, talon, tableau and foundation. This meant that there was no concepts for me to base my notation off of, so I had to deconstruct solitaire games and represent all the details that distinguish them, which was a fairly lengthy task.

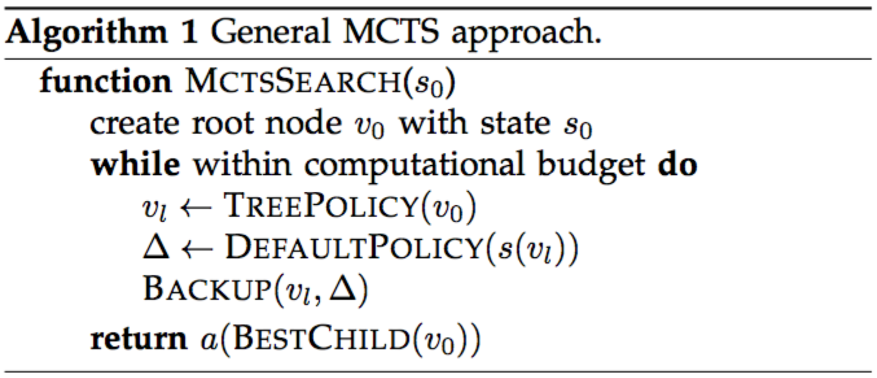
8 Implementation

Java was chosen as the language to code in as that is the language I have most experience with, and I was confident it would work well for the project. Because we are not focusing on efficiency, the speed of the language did not need to be taken into consideration. Also, the IDE IntelliJ was chosen to write it in due to my experience in it, and the features of the IDE to aid in programming.

8.1 The MCTS Method

The main structure of the MCTS algorithm used in this project was originally found in *A Survey of Monte Carlo Tree Search Methods* [5], and is shown in figure 3.

Figure 3: MCTS Algorithm Structure



Algorithm 1 General MCTS approach.

```
function MCTSSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0$))
```

8.1.1 Representing the Tree Structure

In this method, tree nodes are created to represent the game states. This node has multiple parameters. The parent node and an ArrayList of children nodes are stored, with the parent node being set in the node's constructor, and any children being added if and when they are created. The *isRoot* boolean represents whether the node is the root of the tree, and if the node is a root, then the parent is set to *null*.

The corresponding *GameState* object (which represents the state of the game) is also stored, as well as the MCTS score. This score is the score that is back-propagated up the tree when deciding which move to make.

8.1.2 Algorithm Implentation

The algorithm begins by creating the root of the tree. In the case of solitaire, this is a node of the starting game state, after the game has been dealt and no moves have been made. After this, the tree policy is used to determine which unexplored node to expand. While searching for an unexplored node, if there are multiple children we can choose to look at, the method *getABestMove* finds the child with the best score. If there are multiple children with the best score, then one of them is chosen randomly.

Once a node with at least one unexplored child has been chosen to expand, the method *expand* gets all the legal moves from the game state of that node and creates a child node for each one, removing any duplicates that occur from previously expanded children. Out of these newly created nodes, the one with the best score is selected to be the new node. Once again, if there are more than one node with the best score, a random one of these is chosen. This consists of the *selection* and *expansion* part of the MCTS method.

Next, the simulation is run on the tree multiple times in the default policy. Here, the default policy works by recursively calling the method *randomPlay*, until the pre-determined depth of the random play has been reached. For each call, if there are no more legal moves yet, then we have reached a terminating state of the game, so we return the score reached at the end of the game. Otherwise, a random move is chosen from the list of all legal moves and this move is played (and a new state created). After each move, the game is checked to see if it has been won, or if we have reached the maximum depth of play. If not, the program recursively plays the next move.

After this random playout for the expanded node has been ran multiple times, the MCTS score (in the *MCTSScore* field) is set for this node to be the mean score (the sum of all the scores for each playout divided by the number of playouts completed). This score is then backpropagated back up the tree in the *BACKUP* method. This works by setting the *MCTSScore* of the parent of the expanded node to this new score, and recursively setting each parent after that, until the root of the tree is reached.

The default policy is currently simple, choosing the moves to make totally by random. This limits the effectiveness of the MCTS algorithm, as random moves generally result in weak play, which means the moves chosen to play may be suboptimal. By using heuristics we can improve on this, such as favouring moves which increase the score (which is a heuristic used in the tree policy), or adding game-specific knowledge such as favouring moves which allow us to turn over the top face down cards on the tableau in Klondike. [12]

However, if we were to include these heuristics in the default policy, then their effectiveness may vary depending on the type of solitaire game. By doing this, we create a search algorithm more focused on specific types of solitaire, and not solitaire in general. This was why conducting random moves was chosen here. [12]

8.1.3 MCTS Parameters

For the MCTS algorithm, there are three parameters used, each which affect a different part of the method. These are as follows:

- *maxNumSearches* - The number of times a new node is explored in an iteration of the MCTS method. Only after this number of explorations is the final move to play chosen.
- *maxRandomPlays* - The number of random simulations completed in the default policy.
- *maxRandomPlayDepth* - How many moves the random simulation will run for. If the simulation terminates before getting to this number, i.e. the game is won, then the method returns the result.

These three values can be changed to vary the different parts of the MCTS method. Larger values for each of these will theoretically result in better results, as more searches should find the better moves to play. For example, a larger *maxRandomPlayDepth* will mean the program will play more of the game in the simulation, meaning the results will come from game states further on in the game. Having *maxRandomPlayDepth* being large enough to reach the end of a game means that the prediction for the current move should have a positive effect on winning the game, as it is considering moves which have shown can win the game.

However, having larger values for these variables is not always ideal, as it increases the time taken for the games to run. Because of this, a trade-off is required, between the effectiveness of the program and the time the algorithm takes to run. Some testing of these parameters were carried out, varying each parameter one at a time to find the optimal values. The results of these are presented in section 9.

A parameter used in this program is *maxTotalMoves*, which sets a limit of how many moves to play for a game of solitaire. For Sir Tommy, this number does not need to be particularly high, as once one iteration of the deck has been completed, the game ends. However, for Klondike, the deck can be re-dealt indefinitely, and cards can keep being taken from the foundation and moved between the *tabstacks*. This means the number of moves to win could be fairly large. This parameter is set to be 500, which is reasonable considering many of the solved solitaire games take less than 200 moves.

Another parameter used here is *maxLoops*, which is a limit on how many times the program can run the MCTS method for a single move. Each time a possible move is chosen, if that move results in a loop or the resulting game state has occurred before, then the MCTS method is ran again to try and find another move. However, this can only be ran so many times before it can be assumed that there is no move that leads to a winning scenario. We have set this limit to 100, meaning 100 different moves will be tested before the program gives in.

8.1.4 Detecting the End of the Game

Because not all solitaire games are winnable, a way to detect this was added into the program. For Sir Tommy, it is simple to see when the game has been lost, as all the cards in the deck have been seen and no more cards can be added to the foundation. However, for Klondike it is more difficult to judge. Generally, Klondike is lost if there are no further moves that can be made that further the game, so there are no moves which will not end in a loop. For example, a card could be taken down from the foundation to the tableau and then put back up again, which could be done infinitely as a loop. Or a king could be constantly changed between two empty spaces.

To try and identify these scenarios in the game, a check was added to ensure game states are not repeated. This meant game states had to be compared, and if any two are equal, they can be identified. Every game state met in the game is added to an ArrayList of game states, called *PastNodes*. Then, after each iteration of the MCTS algorithm, the resulting game state is checked against every state in this list to see if it has already occurred. The method to compare the nodes compares each location inside the *GameState* object, checking that the ArrayLists of cards are the same.

In addition, a check for loops in the game was implemented, which first checks if a King is being moved between empty spot to another empty spot. This type of move will always be meaningless, as it cannot give us access to a card we did not previously have. Because it can occur quite frequently, we need to have a check in place to avoid the program looping with these moves. Building upon this, a check was also implemented to check that stacks of cards were not moved from one *tabstack* to another. Although this could be necessary for a game to be won, it is not feasible, as many moves can be permutations of moving one stack to another. When this check was not included in the code, some of the games ended up taking an unrealistically large amount of moves to win (over 500 moves), which is not feasible to keep checking when running many games of solitaire. This only affected a small number of the games, so seemed a reasonable limit to introduce.

A game of solitaire can end in multiple different ways. Sir Tommy either ends if the game has been won (all of the cards are in the foundation), or if we have ran out of legal moves to make. Klondike, however, is more complex. The different ways a game of Klondike can end in this program is:

- Winning the game - when all cards are in the foundation.
- No moves left that can lead to a winning game - this is when either game states have been repeated, or the only moves left are Kings being moved between empty spaces, and stacks of cards being moved between *tabstacks*.
- When a move selected by MCTS results in a loop - this happens when after *maxLoops* (currently set to 100) number of calls of MCTS, a suitable move cannot be found.
- No legal moves are left - this only happens in Sir Tommy. In practice, Klondike always has loops that can be made.
- Running out of moves - this happens in Klondike when *maxTotalMoves* (currently set to 500) number of moves has occurred. This doesn't happen in Sir Tommy because after one deal of the deck, Sir Tommy ends, which never surpasses around 200 moves.

Not all solitaire games are possible to be won, however the statistics of how many are winnable have not been discovered yet. This means that we do not expect this solver to be able to solve all of the games.

8.2 Solitaire Game

Many of the decisions made when writing the code was deciding on how to represent the different objects: i.e. cards, locations the cards can occupy, and moves that can be made. This section will detail how these were implemented, and will explain why certain data types were used over others.

8.2.1 Representing the Objects

The most fundamental object to think about is the card object. This constitutes of a value (from Ace to King) and a suit (Hearts, Clubs, Diamonds, Spades), which also determines the colour of the card. Value, suit and colour were created as enums, so they could be referenced easily throughout the program. Colour could have been omitted, as it can be inferred from the suit. However, because some solitaire games stack by colour, it seemed easier to have this as a separate field in the card object.

The deck is a collection of cards, here represented by an ArrayList. An ArrayList was chosen due to the ease of adding and removing cards. When creating the deck, parameters *numSuits* and *cardsPerSuit* determine which cards to create. These parameters work for four or less suits, and 13 or less cards per suit. Being able to reduce the deck helped with debugging the MCTS method, as well as adding to the diferent permutations of solitaire games.

When shuffling the deck, the *shuffle* method provided by the *Collections* library was used. This method puts the cards in a random order, with all orders having the same chance of occurring [6]. The deck is then dealt according to the rules of the solitaire game. For example, Klondike requires some cards to be dealt face down on the tableau, and some to be dealt face up, whereas Sir Tommy has no cards dealt at the start of the game. The program deals any necessary cards to the tableau, and then places the rest of the cards in the stock.

The four locations in solitaire (stock, talon, foundation and tableau) are each represented by an object. The stock is has an ArrayList of cards, which represents that pile of cards, with the integer *pileTop* being the number of cards in the pile, and corresponds to the ArrayList element which is the top of the pile. The stock also references the talon, which has another ArrayList of cards that stores the discarded cards from the stock. When the stock becomes empty, the cards in the talon pile get removed from the talon and are placed in the stock pile.

The foundation consists of four piles, one for each suit. Each pile has a corresponding pile top number, which acts like the *pileTop* in the stock and talon classes; telling the program which element of the ArrayList is the top. In the case that there are less than four suits, say n suits, only the first n foundation piles are used in the game. For example, when the program requests to have all the foundation piles (in the method *getAllFoundations*), the size of the ArrayList returned is n, the number of suits we are playing with.

The tableau is more complex, as it must hold both face down cards and face up cards. These each have different properties, such as we can't access the face down card to see what value and suit it has. In order to represent this, a class called *TabStack* was created. This class has two ArrayLists of cards; one for face down cards and one for face up ones, each of which has a pile top integer. The tableau class then has an ArrayList of these *tabstacks*, with the number of *tabstacks* being the size of the tableau needed for the solitaire game. For example, normal Klondike requires 7 tableaus, but normal Sir Tommy requires only 4.

8.2.2 Printing the State of the Game

To display the game to the user, the program prints the state of the game to the user, as well as printing out the current score of the game. When printing out a card in the *printCard* method, if the colour of the card is red, then the card is printed in red using ANSI codes. In addition, the symbol for each suit is printing using the unicode characters. These make the game more easily understood and played by a human.

8.2.3 Moves in Solitaire

There are multiple different moves that can be made in solitaire, some of which can only be made in some types of solitaire. Because the conditions for each move is different (for example, we can only remove the top card from the talon if one exists, so the talon must be checked that it is not empty), by introducing a *MoveType* enum, we can differentiate between these moves, so move-specific actions can be run. In the games implemented in this project there are 7 possible moves, listed below:

- *TABLEAUTOFOUNDATION* - moving a single card from one of the *TabStacks* to the foundation
- *TURNOVERTABLEAU* - if the top card in a *TabStack* is face down, then turn it over to be a face up card
- *TALONTOFOUNDATION* - move the top card on the talon to the foundation
- *TABLEAUTOTABLEAU* - move either a single card or a stack of cards from a *TabStack* to another one
- *FOUNDATIONTOTABLEAU* - move a single card from the foundation to a *TabStack*
- *TURNOVERSTOCK* - remove the top card from the stock and place it face up on the talon
- *TALONTOTABLEAU* - move the top card on the talon to a *TabStack*

For every state of the game, we check if we can find a legal move for each of these possible moves. Depending on the type of solitaire, only some of these will be checked. For example, Sir Tommy does not have the *TABLEAUTOTABLEAU* move, so we do not need to check for this type of move.

A method exists for each type of move to check for its occurrence in this game state. The *checkCardCanBePutOnFoundation* method is used for multiple types of move to check if a particular card can be placed on any foundation pile (according to the solitaire rules). Similarly, *checkCardCanBePutOnTableau* is used multiple times to check if a particular card is eligable to be placed on the tableau. This takes into consideration if there is an empty space on the tableau (there are no face up or face down cards on a *TabStack*) and if the card can be placed on this empty space.

The *checkTableauToTableau* method is fairly complex, as it also has to take into consideration that multiple cards on a *TabStack* can be moved as part of a stack to another *TabStack*. It does this by checking if any of the face up cards in a *TabStack* can be move to another *TabStack*. If this is possible, any cards below the card to move from the origin *TabStack* moves with the card to the destination as a stack of cards.

If a legal move for one of these move types is found, a new *Move* object is created. This object was created to represent all the different moves, and uses the enum *MoveType* to distinguish between which move it is. Certain parameters provide extra information about this move, such as integers *tab1* and *tab2*, which hold the number of the *TabStack* used in the move. *tab1* is the origin *TabStack*, and *tab2* is the destination *TabStack*, both of which are set to -1 if they is not applicable in the move.

This *Move* object holds either a card or an ArrayList of cards that are the object of the move. The ArrayList is only applicable when moving multiple cards in a stack from one place in the tableau to another. If this happens, then an integer *stackNum* holds the number of cards in this stack. In addition, the *Move* object has a field called *foundationType*, which holds the number of the foundation that the card will be placed. This is primarily used in Sir Tommy, when a card may be placed in a foundation that does not reflect its suit.

The method *makeMoveGetNewState* copies the original game state by copying each component of the game state. This deep copy calls the copy methods recursively, moving down through the objects until each individual card is copied. *makeMoveGetNewState* has a single argument of a *Move* object, which is the move to be applied to this new *GameState*. This move is implemented on the new state and then the new score is calculated (by summing the number of cards in the foundation).

When printing the moves, the description of the move is outputted, as well as any relevant parameters (such as the card (or stack of cards) and any specific *TabStacks* or foundations.

8.3 Solitaire Notation

Originally, the project focused on only Klondike, which meant some of the functions to get the legal moves and implement these moves were based on the rules for Klondike, such as stacking in colour on the tableau. When introducing Sir-Tommy, these type-specific parts of the code were separated depending on the type of solitaire we were dealing with. By introducing an enum to hold the solitaire type, the program could decide which code to run by a switch statement on the solitaire type. This was used for methods such as dealing the tableau and deciding if the stock could be turned over.

However, since the aim of the project is to build a general solver, having the code type-specific undermines the generalisability of the game. In order to improve this, a new notation for solitaire was defined, with the different parameters representing the different possible rules in solitaire. This was encoded as a JSON file, which could be used as an input for the program. Listing 1 shows an example of this notation, with the parameters being set for a normal Klondike game, and Listing 2 shows this notation for SirTommy. The class *Parser* parses these JSON objects and sets the relevant information in the *Solitaire* object, which is then passed to the relevant card locations if applicable.

Listing 1: Normal Klondike example

```
{
    "type": "K",
    "numSuits": 4,
    "cards per suit": 13,
    "moves": [1,1,1,1,1,1,1]
    "maxOneInTalon": 0,
    "tableau" {
        "numTableaus": 7,
        "dealTableau": 1,
        "talonToAnyTableau": 0,
        "tableauToAnyTableau": 0,
        "stackAsc": 0,
        "cardsOnEmptySpace": [13],
    }
    "foundation" {
        "stackAsc": 1,
        "cardsOnFoundationBottom": [1],
        "stackByColour": 1,
        "stackAnySuit": 0,
    }
}
```

Listing 2: Sir Tommy example

```
{
    "numSuits": 4,
    "cards per suit": 13,
    "moves": [1,0,1,0,0,1,1]
    "maxOneInTalon": 1,
    "tableau" {
        "numTableaus": 4,
        "dealTableau": 0,
        "talonToAnyTableau": 1,
        "tableauToAnyTableau": 0,
        "stackAsc": 0,
        "cardsOnEmptySpace": [13],
    }
    "foundation" {
        "stackAsc": 1,
        "cardsOnFoundationBottom": [1],
        "stackAnySuit": 1,
        "stackByColour": 0,
    }
}
```

The definition of each parameter is as follows:

- "numSuits" - the number of suits to use in the deck of cards. This is a value between 1 and 4.
- "cards per suit" - the number of cards each suit contains in the deck. This is a value between 1 and 13.
- "moves" - an array containing either a 1 or a 0 to represent which types of move are legal. For example $[1, 1, 1, 1, 1, 1, 1]$ indicates all types of moves are legal (such as in Klondike).
- "maxOneInTalon" - a 1 or a 0 to represent a boolean of whether only one card can be placed in the talon or not (a rule which is true in SirTommy and false in Klondike).
- "tableau" - a JSON object to hold all the relevant information about the tableau.
- "numTableaus" - the number of *TabStacks* to have in the tableau. This is a number greater than 0, and there is no upper bound. However, for Klondike the number of cards in the deck must be greater than the number of cards dealt in the tableau, meaning the maximum number of *TabStacks* for Klondike is 9 (which equals 45 cards being dealt out).
- "dealTableau" - a 1 or a 0 to represent whether the tableau is to be dealt at the beginning of the game. By "deal" we mean the deal structure from Klondike, with 1 face up card for each *TabStack* and $n-1$ face down cards for each *TabStack* n.
- "talonToAnyTableau" - a 1 or a 0 to indicate whether the card currently in the talon can be placed on any *TabStack*, like in SirTommy.
- "tableauToAnyTableau" - a 1 or a 0 to show if a card in the tableau can be removed and placed on any *TabStack* in the tableau, irrespective of what the *TabStack* holds. This is not a rule present in SirTommy or Klondike, but is one that could be implemented for a different type of solitaire.
- "moveStacks" - a 1 or a 0 to represent if more than one card can be moved in the tableau (as a stack).

- “stackAnySuit” - a 1 or a 0 to indicate if the cards are the tableau can be stack in any suit or not.
- “stackAsc” - a 1 or 0 to tell if the cards in the tableau stack ascending or descending. Normal Klondike stacks descending from King to Ace.
- “cardsOnEmptySpace” - an array of numbers to instruct the program which cards can be placed on an empty space in the tableau (i.e. a *TabStack* with no face up or face down cards). A 1 represents an Ace, likewise up to 13 representing a king. In normal Klondike, only a king can be placed on an empty space. This parameter is rendered meaningless if *”talonToAnyTableau”* is true, and any card can be placed on any *TabStack*.
- “foundation” - a JSON object to hold all the relevant information about the foundation.
- “stackAsc” a 1 or a 0 to indicate if the cards in the foundation stack ascending or descending. Normal Klondike and Sir Tommy stack ascending from Ace to King.
- “cardsOnFoundationBottom” - an array of numbers to instruct the program which cards can be placed on the bottom of the foundation pile an empty space. A 1 represents an Ace, likewise up to 13 representing a King.
- “stackAnySuit” - a 1 or a 0 to represent whether cards on the foundation can be stacked in any suit, such as in SirTommy.
- “stackByColour” - a 1 or a 0 to show if the cards on the foundation can be stack by colour (so hearts and diamonds can be stacked on top of each other, and the same with clubs and spades). This is not a rule present in Klondike or SirTommy, but can be used to make an interesting alternative.

Some of these parameters, however, need to be co-ordinated with others to ensure that it results in a valid game. For example, it would be difficult to have 3 suits in Klondike, as when stacking by colour in the tableau there would be an uneven number of reds and blacks, therefore making the game more difficult. Also, there can’t be only 1 or 2 suits in Klondike if there are 7 tableaus, as 7 tableaus requires 28 cards to be dealt correctly. This means some additional thought needs to be put into these notations for different types of solitaire.

With these parameters, many different permutations of Klondike or sirtommy can be created, such as having only 3 suits in SirTommy, or allowing stacking by colour in the foundation. By restricting some types of move, such as *FOUNDATIONTOTABLEAU*, we can make Klondike slightly harder. An easy way to change the game slightly is to alter the size of the tableau, i.e. how many *TabStacks* are available in the game. By increasing this in Sir Tommy, we make the game easier to complete.

Another way of creating different variations of solitaire is by reducing the suits and the number of cards per suit. This was originally implemented for debugging of the MCTS method, as it took a while to finalise the method and have it working correctly. By having less cards to deal with, it was easier to complete the games and to check that MCTS was choosing good moves to make.

A different version of Klondike, called *"Klondike-Opposite"* was also created, with the rules similar but having some parameters set to be the opposite. For example, the foundation stacks from King to Ace (still stacking in suits), and the tableau stacks from Ace to King. This shows as example of how changing some parameters can result in a game that is slightly different, yet the solver still works. The notation for this example is shown in Listing 3.

Listing 3: Opposite Klondike example

```
{
    "numSuits": 4,
    "cards per suit": 13,
    "moves": [1,1,1,1,1,1,1]
    "maxOneInTalon": 0,
    "tableau" {
        "numTableaus": 7,
        "dealTableau": 1,
        "talonToAnyTableau": 0,
        "tableauToAnyTableau": 0,
        "stackAsc": 1,
        "cardsOnEmptySpace": [1],
    }
    "foundation" {
        "stackAsc": 0,
        "cardsOnFoundationBottom": [13],
        "stackAnySuit": 0,
        "stackByColour": 0,
    }
}
```

9 Empirical Evaluation

In order to test the MCTS method, some experiments were run with both the *Klondike-Normal* and *SirTommy-Normal* solitaire definitions. These experiments were ran to test the different parameters that affect the MCTS method: *maxNumSearches*, *maxRandomPlays* and *maxRandomPlayDepth*. Each parameter was singled out and tested in different intervals, with the other two parameters staying set at a specific value (so that only the parameter being tested is the one changing).

The tests ran were slightly different for each Solitaire, due to their different features. Klondike was tested to see what percentage of games could be solved, and how fast these solutions take. Theoretically, the larger the parameters, the longer the searches should take, and the better the results would be. This is because the MCTS should give better results the more times it is run and the higher number and depth of random plays. The times were only considered for solved Klondike games, as some of the unsolved games did not complete, so took less moves, and did not seem important to the results.

Sir Tommy was tested to find the average score for each setting of the parameters, and the average time taken for each game. We did not focus the number of won games here, as it was low (with 0-3 games being won ever 100 plays). However, the score in Sir Tommy is a good indicator of how well the game is going, so this seemed a good indicator to measure the MCTS method.

For Klondike, the parameters *maxNumSearches* and *maxRandomPlays* were tested between 10 and 190, in intervals of 20. For Sir Tommy, the parameters were tested between 10 and 200, with intervals of 10. Less parameters were tested for Klondike because Klondike takes longer to run. The average Klondike (solved) game took 8500 milliseconds to run, whereas for Sir Tommy the average game took 350 miliiseconds to run. This made a difference when running these experiments. For each Klondike test for each parameter, 50 games were tested, and for Sir Tommy, 100 games were tested. Because there were so few games in Klondike being used in the results (around 20/50 were solved), the Klondike experiment was ran 5 times, so each percentage is an average over 5 experiments, each running 50 games for each test of the parameter.

Klondike takes longer to run for MCTS due to the more moves it takes on average. When looking at only solved games, from 100 plays of both Sir Tommy and Klondike, the average number of moves was 138 (out of 3 solved games) for SirTommy and 231 (out of 25 games) for Klondike. Here *maxNumSearches* = 10, *maxRandomPlays* = 10 and *maxRandomPlayDepth* = 100. None of these Sir Tommy games took over 150 moves, whereas a few of the Klondike games took over our limit of 500 moves. Because of this, the maximum *maxRandomPlayDepth* tested for SirTommy was 160 (to allow for some outliers that may take more moves), and for Klondike was set to 500, to include as many Klondike games as possible.

An explanation of the data files for these results is found in Appendix D.

9.1 Klondike Experiments

9.1.1 Percentage of Games Solved

Figures 4, 5 and 6 show the effect of increasing the parameters against the average percentage of games won. All of the data points are the average of 250 games. These averages are all within the range 37-51%. These graphs have a large amount of variance in them, showing that 68% of the data points fall between 30 and 60 percent. This large variance implies that the scores of the Klondike games are spread out, and it is difficult to predict what any particular score could be.

These graphs show that the values of the parameters do not make a difference to how well the method performs, which is contradictory to what we had predicted. The linear regression lines show a slight increase or decrease, but it is not a good fit to the data points, so is not considered.

Figure 4: Testing *maxNumberSearches* for Klondike for the Percentage of Games Won

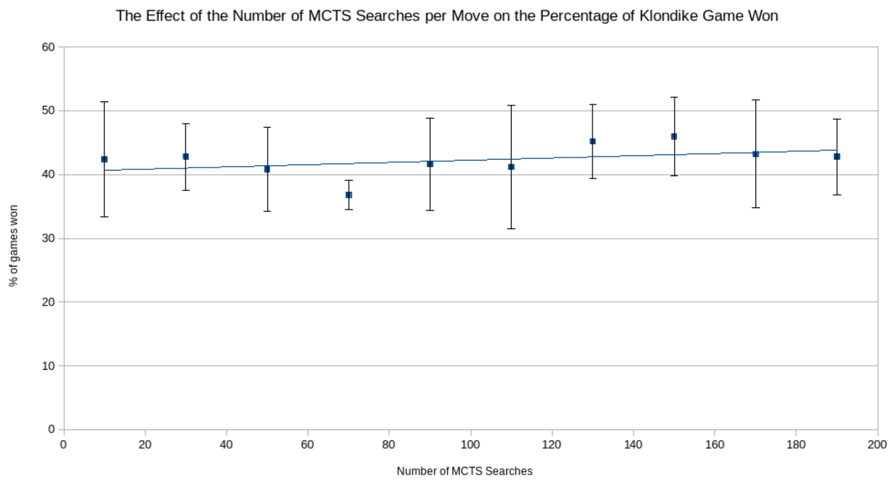


Figure 5: Testing *maxRandomPlays* for Klondike for the Percentage of Games Won

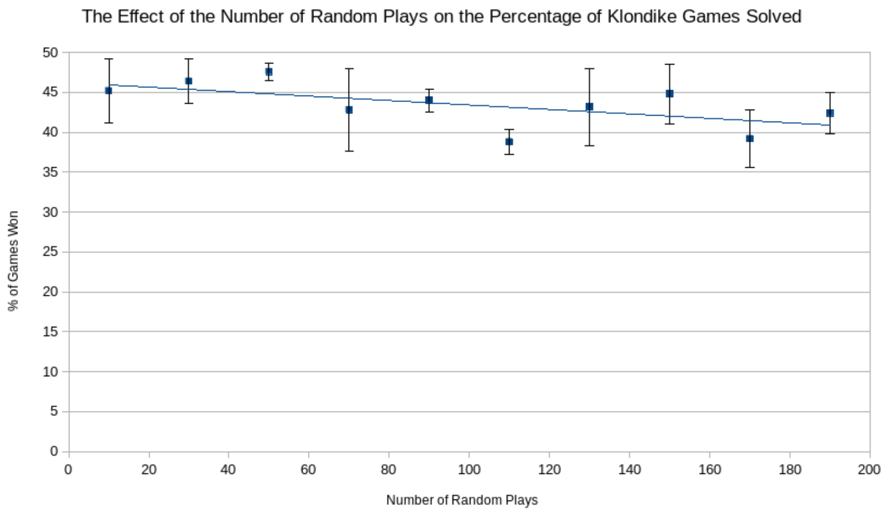
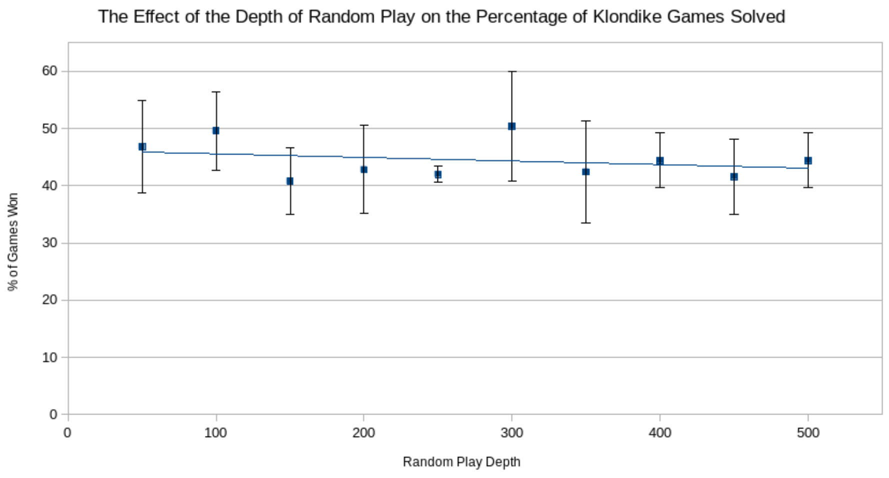


Figure 6: Testing *maxRandomPlayDepth* for Klondike for the Percentage of Games Won



9.1.2 Average Time of Solved Games

Figures 7, 8 and 9 show the effect of increasing the parameters against the average time of completion for the solved games of Klondike. All of the data points are the average of 250 games. These values vary a lot, with some of the times being twice as long as others. The variance here is very large, telling us that the different solitaire games make a difference in how long it takes to complete them. This could be because some games take more moves to solve the game than others, so this will naturally take longer to compute.

Once again, the size of the parameters does not seem to make any difference in how long the average game takes to solve, which is not what we expected.

Figure 7: Testing *maxNumberSearches* for Klondike for the Average Time of a Solved Game

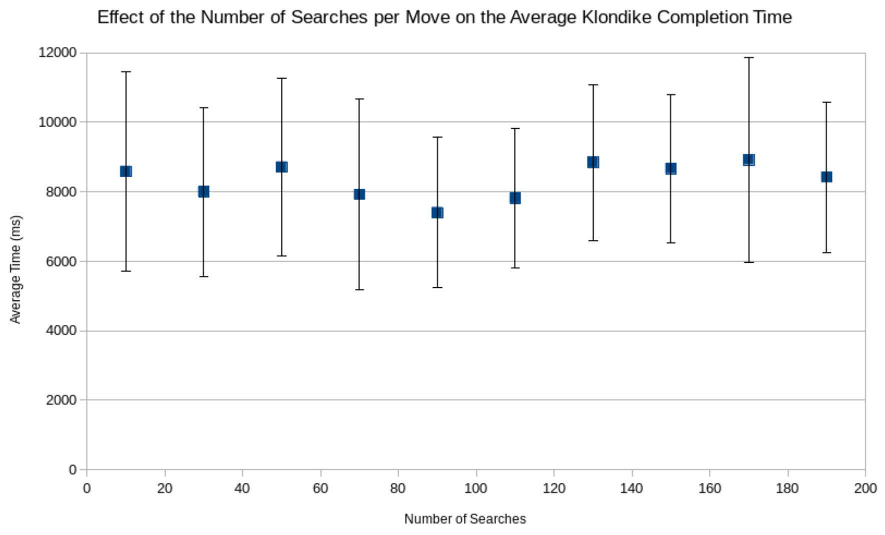


Figure 8: Testing *maxRandomPlays* for Klondike for the Average Time of a Solved Game

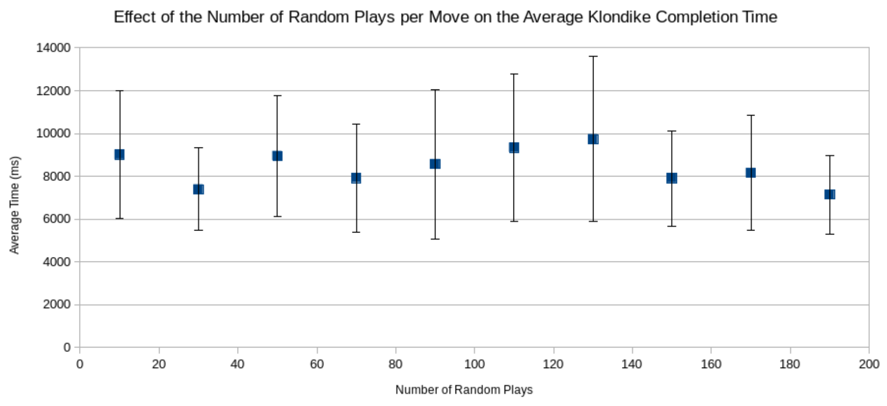
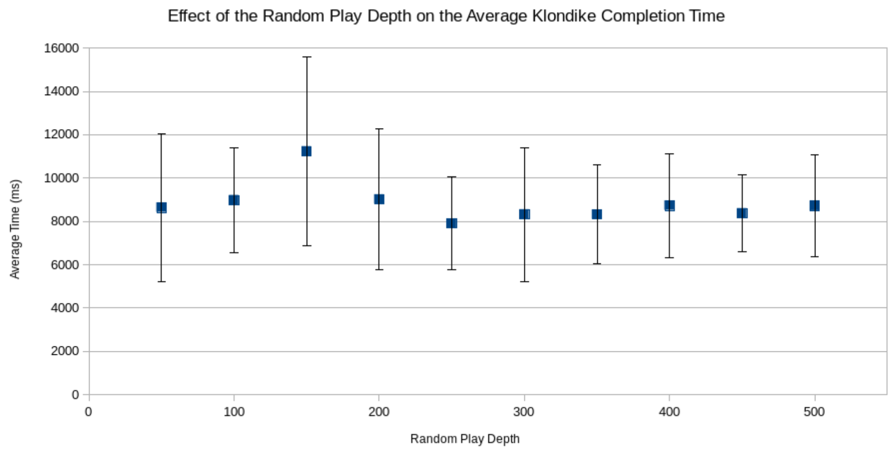


Figure 9: Testing *maxRandomPlayDepth* for Klondike for the Average Time of a Solved Game



9.2 SirTommy Experiments

We can also test how these parameters affect the average score of the SirTommy game. Figures 10, 11 and 12 show the results of these experiments. Each data point is the average score over 100 SirTommy games. From these graphs we can see that the average score ranges between 25 and 30, with a lot of variance in the data. From the error bars, we can see that 68% of the results fall between 16 and 38, which is one standard deviation away. These values are very spread apart, meaning there is a huge variation in the score possible for any given SirTommy game.

In addition, there is no obvious trend in how well the MCTS method works with the different parameters. In fact, if a linear regression line were to be added, it would be constant. This suggests that the size of the parameters do not have any affect on how well the method performs, however, more testing would be needed before we were able to conclude this.

Figure 10: Testing *maxNumSearches* for SirTommy for Average Score

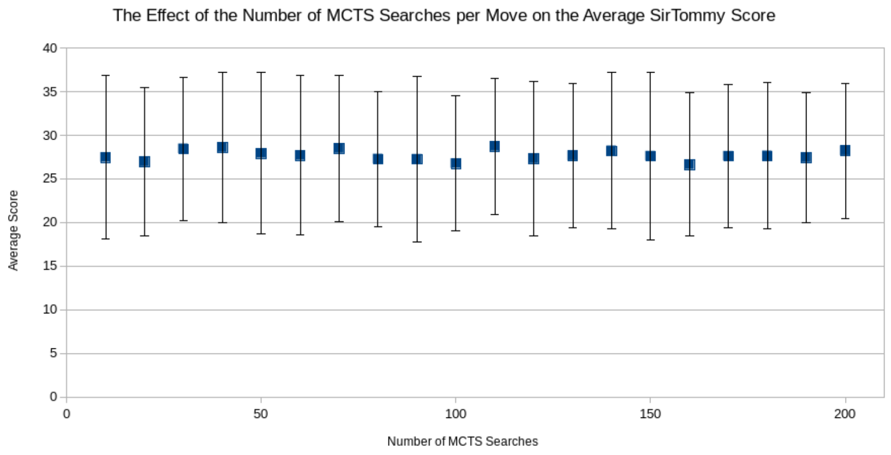


Figure 11: Testing *maxRandomPlays* for SirTommy for Average Score

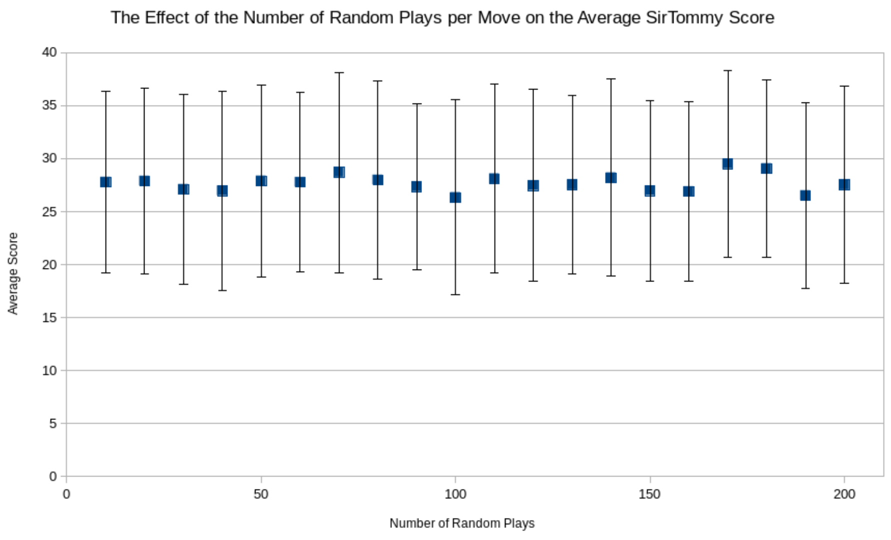
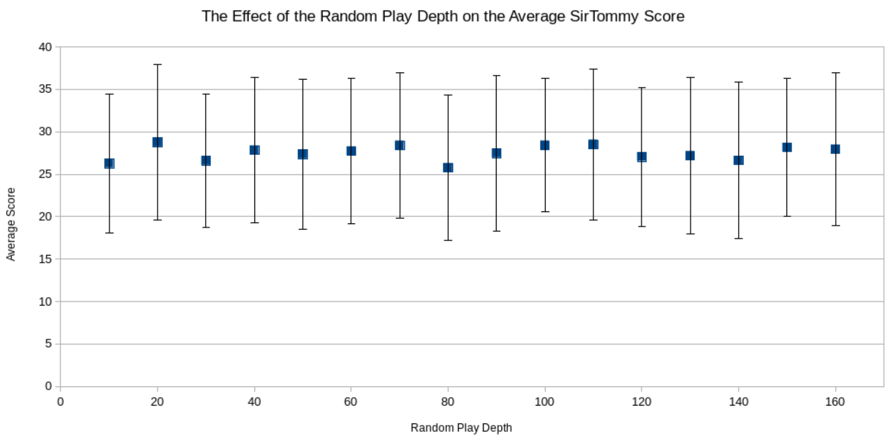


Figure 12: Testing *maxRandomPlayDepth* for SirTommy for Average Score



9.2.1 Average Time of Games Completed

Figures 13, 14 and 15 show the effect of increasing the parameters against the average time of completion for all the games of Sir Tommy. All of the data points are the average of 100 games. These values are all within 300ms and 420ms, meaning all of the games take a similar amount of time to compute. This is as expected, as Sir Tommy games take a similar amount of moves to complete, as the deck can only be used once. This average time is a lot less than the average time taken for Klondike. One game of Sir Tommy takes roughly 1/3 of the time that a solved game of Klondike takes.

The variance here is high, similar to the Klondike games. This shows that there are still a lot of games that take more or less time than the average time, meaning the order of the deck still makes a lot of difference. There appears to be an outlier in the first test for *maxNumSearches*, evident by the extremely large variance. When looking at the data, one of the data points took 1062ms to solve (2.5 times the average for that test), and another data point took 2282 (over 5 times the average).

Once again, there does not appear to be a difference in the times according to the size of the parameters, as there is no clear increase or decrease in the average times.

Figure 13: Testing *maxNumSearches* for SirTommy for Average Completion Time

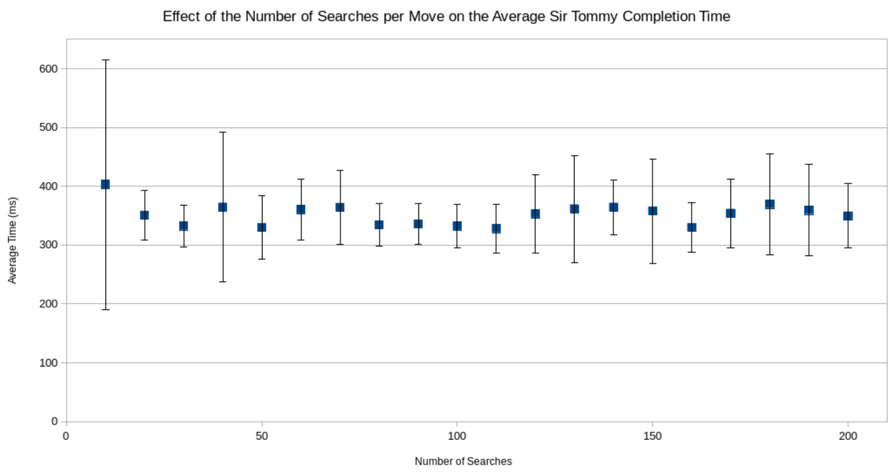


Figure 14: Testing *maxRandomPlays* for SirTommy for Average Completion Time

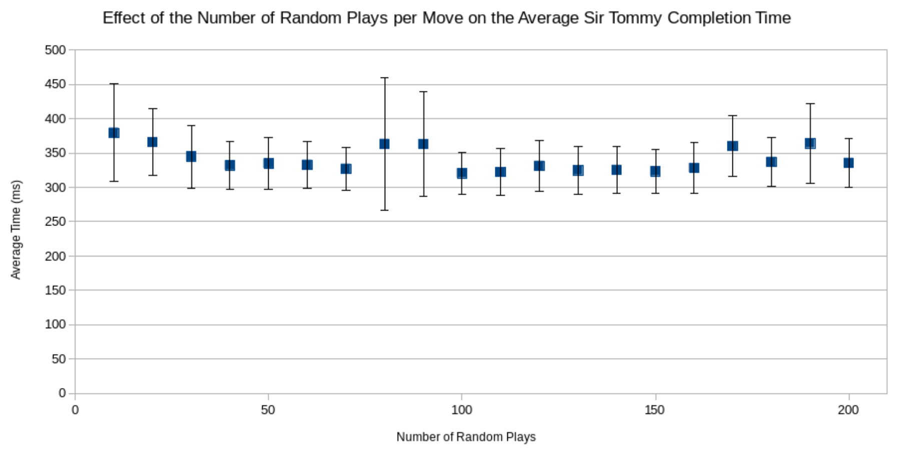
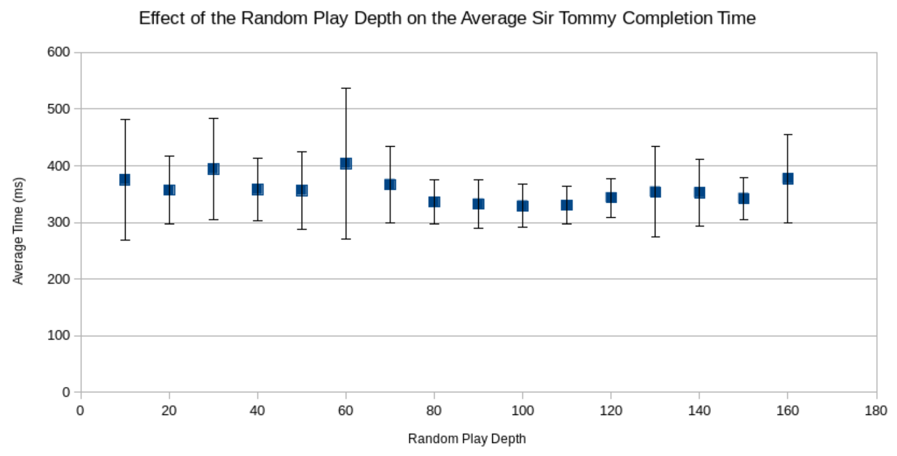


Figure 15: Testing *maxRandomPlayDepth* for SirTommy for Average Completion Time



9.3 Key Findings

For Klondike, the claimed percentage winnable games is 79% [21]. This program can solve between 30-50% of games. Since our program was not aimed specifically for Klondike, and we have omitted heuristics which could have improved the solved games, solving half the games is a good achievement.

For Sir Tommy, very few games were solved (<1% of games). From looking at related work on this topic, there is no relevant information available on the statistics of how many of these games are winnable. Due to the complexity of the game, we can assume this is significantly less than for Klondike. It is difficult to know if these numbers of solved games is considered good for an AI solver. However, being able to solve some Sir Tommy games is still promising, and perhaps with appropriate heuristics this number would be improved.

10 Testing

The majority of the testing in this project was done by frequently running the programs and looking at the results. When testing the MCTS algorithm, the different game states and moves chosen were printed and viewed to ensure it was running the simulations correctly. By reducing the number of suits and cards per suit, the method was able to be seen more clearly, as the games were simpler.

Part way through the project, my supervisor mentioned that the moves chosen for Sir Tommy did not look like 'good' moves to make. Because of this, I spent a while printing out the different stages (with reduced suits and cards) to see what was happening. From this, I realised that I was not creating the tree correctly, and instead was searching from the root every time, rather than creating a new node with each MCTS search.

To test the solitaire games, some jUnit tests were written at the start of the project, as mentioned in section 5. However, due to re-factoring of the solitaire structure, these were no longer applicable. The solitaire games were mainly tested by playing many of them. Because I have played both of these solitaires many times before, I could pick up any bugs in the program.

11 Evaluation of Project

The result of this project is a general solitaire solver that uses MCTS in order to calculate the moves that should be played. By focusing a lot of effort on creating the MCTS method, a reliable solver has been produced. From the research conducted for the context survey, there were no reports of a general solver for solitaire, implying this project is something new. Because of this, it is difficult to compare the solver to related work, as solvers available focus on the best play possible for that particular solitaire.

In addition, there does not seem to be any previous work on using MCTS with solitaire, hence to have achieved a working MCTS method that solves a decent number of the games is promising, and shows that this method could have the potential to be used in a wider variety of projects.

Since we did not expect to see 'perfect play' from the solver, it was expected to have sub-optimal results in the percentages of games solved. The more important part of the project was to be able to have a general solver. With the solitaire notation, this solver can also solve games such as *Shifting*, which is similar to Sir Tommy but with different rules in the tableau [19]. The notation of this is given in listing 4. This game has an average solve rate of 20%.

Listing 4: Shifting example

```
{
    "numSuits": 4,
    "cards per suit": 13,
    "moves": [1,0,1,1,0,1,1]
    "maxOneInTalon": 1,
    "tableau" {
        "numTableaus": 4,
        "dealTableau": 0,
        "talonToAnyTableau": 1,
        "tableauToAnyTableau": 0,
        "moveStacks": 0,
        "stackAnySuit": 1,
        "stackAsc": 0,
        "cardsOnEmptySpace": [13],
    }
    "foundation" {
        "stackAsc": 1,
        "cardsOnFoundationBottom": [1],
        "stackAnySuit": 1,
        "stackByColour": 0,
    }
}
```

Other solitaire games can easily be added to the repertoire of solvable games by modifying the methods to check for moves and implement these moves. The MCTS method has been designed to be interchangeable with any solitaires, providing it has been given the choice of moves to make and the state of the game. This means that many more solitaire games can be added with relative ease, expanding the project and proving the functionality of the general solver, which is the aim of the project.

The solitaire notation (outlined in section 8.3) was something not particularly focused on at the beginning of the project. However, when my supervisor and I were discussing how easily the solver could generalise to different games, a way to represent the games seemed more important, and very helpful to the project.

11.1 Primary Objectives

Primary Objectives

1 Build a program for humans to be able to play multiple types of solitaire

2 Create an AI to play good moves for these games

3 Create an AI program that uses MCTS to solve the game

For objective 1, having the game playable by humans was one of the first tasks completed, which also enabled testing of the game. If anything did not work properly then it would be picked up by myself while playing the game. The "multiple types of solitaire" part of the objective is based off of two types of solitaire, or three including *Shifting*. The different variations of these types allow us to create many different versions of solitaire. Originally, the aim was to have more main types of solitaire. However, the MCTS program was more complex than originally thought, and took longer than planned to get functioning correctly. This meant I had to decide on a trade-off between focusing on more types of solitaire or focusing on the functionality of the MCTS method. I chose to focus on the latter, as it was much more complex and cruicial to the project.

Objectives 2 and 3 were completed together. They were originally set as different objectives so that having a working MCTS algorithm was an objective in itself, as it was crucial to the project. For objective 2, it is difficult to say which moves are classed as 'good' moves. Because we can solve some of these solitaire games, it is reasonable to assume the moves chosen are classed as "good".

11.2 Secondary Objectives

Secondary Objectives

1 Modify the program to be able to give it a state of the game, and for it to select the best move/best x moves

2 Play the AI multiple times for multiple types of solitaire to work out the % of winnable games with perfect play

Objective 1 was unfortunately not met in this project. Being able to pass in a state of the game and have the program find all the legal moves could be performed easily, with just creating a parser for the state of the game. This could be completed with some more time on the project. The MCTS method could be used to select the best move, although selecting the best x moves is more complicated, and it would be difficult to order the possible moves according to their effectiveness. This could be done by using heuristics on the game: for example, in Klondike, moves that free up more of the face down cards is better than working the way through the stock. This is not ideal, however, as using heuristics undermines having a general AI solver for the different games, as each type of solitaire would need its own heuristics added.

Objective 2 has been completed, with the assumption that "perfect play" is the playing of the game by the MCTS method. This program does not claim to make perfect moves, and probably would not be able to without a better tree policy and default policy.

11.3 Tertiary Objectives

Tertiary Objectives

1 The AI can give the best x moves and the % the game can be won with each move

2 Have the AI check the % winnable each move is, and show how reliable itself is

3 Evaluate AI against itself/other solvers

4 Try to create a rule system to represent these different types of Solitaire

Objective 1 has not been completed, due to the difficulty in ordering the possible moves, which makes giving the best x moves impossible. In addition, it would be complicated to calculate exact percentages of games that can be won from a single move, as the winning of games in dependent on the future moves too, and not that single move.

Objective 2 has not been met for a similar reason as objective 1, as it is difficult to assess each move so in-depth, especially when the MCTS is not perfect, and we cannot guarantee the percentage of games won is representative of that type of solitaire, or is only the percentage our solver can win.

Objective 3 has partially been completed as the program has been tested against itself with different parameters for the values *maxNumSearches*, *maxRandomPlays* and *maxRandomPlayDepth*. These have been compared and evaluated in section 10. If there were more time available, research would be conducted into what other solitaire solvers are published online, and this program would have been tested against them. However, because this project is unique in that it is a general solver, rather than tailored for a certain type of solitaire, then it would likely perform worse than specific solvers available.

Objective 4 has been met in trying to define the types of solitaire. However, it has only been defined so far for the rules needed in Klondike and Sir Tommy, with a few extra rules added. To be able to define more types of solitaire, this JSON file would need to be expanded for all the possible variations, including information on how to deal the cards and the structure of the locations, as not all types of solitaire have all four of stock, talon, tableau and foundation, and may have additional locations.

11.4 Limitations and Improvements

This project had a few limitations, which became apparent throughout the year, especially in the MCTS method. For example, when looking at the *selection* stage, we could have an instance where we are checking down a particular path in the tree because this path has the best score, but the path may lead only to a dead end. A way to overcome this could be to modify the tree policy to include some random restarts in the method, where the policy will sometimes favour a node that looks suboptimal. This means that some possible moves that may have a small score and not look as favourable still have a chance of being chosen. This is a feature in a popular version of an MCTS method called *"Upper Confidence Bounds for Trees"* [5].

Another limitation in the MCTS method is that when performing the random playouts, the games being played are limited to the cards being in the position they are in the original game. However, because this is non-perfect information, the unknown cards at any point in the game could be any card not already seen, which means to make the random playouts comply with this, the unknown cards should ideally be shuffled in each iteration of the playout. This would be something important to improve upon if there were more time, as then the MCTS method would work closer to how it was designed.

In Klondike, the check for loops in the game disallows stacks of cards being moved between the tableau, as this is a problem that usually leads to loops. However, in a few games this is necessary to win, and the game can be won if this move was allowed. A possible improvement would be to allow some movement of stacks in the tableau. There is a trade-off between allowing all of these moves (which would take a lot longer than current games, but would find any moves which would allow a win if one existed) and not allowing any. This trade-off would have to be thought through carefully, to see if solving more Klondike games is more important than the increase in time in a lot of games.

Similarly, having repeating game states may sometimes be a good thing and could be necessary to win a game. Currently, if a state is repeated once then the program is set to exit, but perhaps allowing a state to be repeated a couple of times could improve the program. If there was more time for this project, I would have experimented with a few different values to see how many times states can be repeated until the game is definitely lost. To keep things simple for this project, this number was kept to just allowing one of each state.

When testing the parameters of MCTS, the games we tested have totally random decks of cards. Some of these are solvable and some are not, the proportions of which are unknown. This means that our tests rely on these random shufflings of the deck evening out after many plays, to give us reliable values of the data. An improvement which could fix this is by having a seed for decks, which would mean we could test the same decks with the different parameters, to check if by increasing the parameters we find more solutions of the game. However, this would pose the problem that the deals are not random, and may not be representative of the solitaire games as a whole.

The parameters available to change in the JSON notations for solitaire is fairly limited. These mainly include only the rules needed for Klondike and SirTommy. With more time, extra parameters could be implemented, which would therefore result in a larger set of solitaire variations being able to be solved. For example, in the tableau, an option to stack only in the same colour (rather than in alternating colours) could be added, as well as restricting this further to only stacking in the same suit. The latter could be flawed, however, as a stack from King to Ace of the same suit would be just as easily added to the foundation with no effort. Another idea for the notation is to add extra information about the deal at the start of the game. Currently, the only dealing of cards is for Klondike, which follows the pattern of $n-1$ face down cards for the *nth TabStack* with one face up card on top of it. This could be modified so that, for example, $n-2$ face down cards were dealt, with the 1st *TabStack* having no face up cards and being an empty space. This would require more thought, however, of how to represent a solitaire deal, especially if more types of solitaire were to be added to this project.

12 Conclusions

12.1 Overview of Objectives

The main aim of this project was to create an AI solver for multiple solitaires using the Monte Carlo Tree Search method. This is represented by the primary objectives, where the ability to play the solitaire games and the AI solver are split into different objectives. These objectives were met, focusing on two types of solitaire, Klondike and Sir Tommy, that could be varied by changing certain rules and parameters of the solitaire, which was also one of the tertiary objectives.

Other aims of the project involved using the MCTS method to rate the possible moves in a game, which was not able to be completed due to lack of time and the complexity of the problem.

Another aim was comparing the solver created in the project against itself and other solvers. The experiments on the parameters of MCTS allowed us to compare the solver against itself, and try to calculate the optimal settings of the method.

One of the aims that has become a main part of this project is creating the solitaire notation. This is something unique in solitaire, and has greatly helped this general solver. Hopefully the idea of this notation can be can used in future work on solitaire.

12.2 Key Findings

The key findings of this project is based upon how the algorithm affects solving the solitaire games. We can see the method works due the success rates in Klondike. On average, throughout all the experiments that are discussed here, 30-50% of Klondike games were solved. For Sir Tommy, less than 1% of games were solved. However, Sir Tommy games were averaging a score between 25-30, showing that the MCTS is sophisticated enough to complete over half of the game on average.

One of the main conclusions drawn from this project is that the parameters defined in the MCTS method do not have a noticeable effect on the percentage of games solved in Klondike and the average score in Sir Tommy. This came as a surprise, as it was assumed the more the method is ran (and at a larger depth), the better it would perform. Perhaps only a limited number of MCTS searches and random plays are needed to make the MCTS method effective, or perhaps these parameters would make more of a difference if the tree policy and default policy were more sophisticated.

In addition, the large variance in these findings indicate that the randomness of solitaire deals make them difficult to run experiments on. The possibility of controlling the solitaire deals by creating a seed for each deal could help eliminate this.

12.3 Future Work

It would be fairly easy to implement some more types of solitaire for this program, by implementing the new rules that may be needed. After doing this, if the program solved the new types of solitaire relatively well, then this would further show the success of the general solver.

Further improvements would involve the implementation of the tree policy and default policy, which affect how the MCTS method performs. In this project, we kept these simple in order to focus on having the functionality of the method working. However, by putting more time into these we could have received better results.

Future research on this project would focus on improving the tree policy and default policy, and observing any difference these make, especially on the experiments with the parameters. It would take a lot of research and experiments to try and find the optimal policies, so this could be a large topic to be looked into. In addition, by adding new types of solitaire, different games can be researched. There is a lot of research into the popular games (such as Klondike and free cell), but less research has gone into the less well-known games. This research could provide more insight in using MCTS with one player, non-perfect information games.

12.4 Summary

Personally, I think this project has been a success, as I have have produced a working solver and a solitaire notation that has not been attempted before. I feel like this area of research has a lot of potential, and I am confident that MCTS and other similar methods will keep improving the state of AI, until one day we can finally state the odds of winning every different solitaire game.

References

[1] URL: `https://www.solitr.com/klondike-turn-one`.

[2] *AlphaGo: Mastering the ancient game of Go with Machine Learning.* URL: `https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html`.

[3] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. "Lower Bounding Klondike Solitaire with Monte-Carlo Planning". In: *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling* (2009).

[4] Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. "Searching Solitaire in Real Time". In: ().

[5] Cameron Browne et al. "A survey of Monte Carlo tree search methods". In: *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES* (2012).

[6] *Collections.* URL: `https://developer.android.com/reference/java/util/Collections.html#shuffle(java.util.List%3C?%3E,%20java.util.Random)`.

[7] Adrien Couëtoux, Martin Müller, and Olivier Teytaud. "Monte Carlo Tree Search in Go". In: ().

[8] Steve Kroon Dirk Brand. "Sample Evaluation for Action Selection in Monte Carlo Tree Search". In: (2014).

[9] Ian Gent. *AlphaGo beats Go Champion.* URL: `https://studres.cs.st-andrews.ac.uk/2015_2016/CS3105/Lectures/L18-AI-Learning3AlphaGo.pdf`.

[10] Ian Gent et al. "Search in the Patience Game 'Black Hole'". In: ().

[11] Malte Helmert Gerald Paul. "Optimal Solitaire Game Solutions Using A Search and Deadlock Analysis". In: (2015).

[12] Istvan Szita Guillaume Chaslot Sander Bakkes and Pieter Spronck. "Monte-Carlo Tree Search: A New Framework for Game AI". In: (2008).

[13] *How to Play Klondike Solitaire.* URL: `http://www.solitairecity.com/Help/Klondike.shtml`.

[14] Christof Koch. *How the Computer Beat the Go Master.* 2016. URL: `https://www.scientificamerican.com/article/how-the-computer-beat-the-go-master/`.

[15] Alan Levinovitz. *The Mystery of Go, the Ancient Game That Computers Still Can't Win.* 2014. URL: `https://www.wired.com/2014/05/the-world-of-computer-go/`.

[16] Max Magnuson. "Monte Carlo Tree Search and Its Applications". In: (2015).

[17] *Number of Possible Go Games.* URL: `http://senseis.xmp.net/?NumberOfPossibleGoGames`.

[18] Christopher D. Rosin. “Nested Rollout Policy Adaptation for Monte Carlo Tree Search”. In: (2011).

[19] *Shifting*. URL: `http://www.solsuite.com/games/shifting.htm`.

[20] *Sir Tommy Solitaire*. URL: `http://www.solitairenetwork.com/solitaire/sir-tommy-solitaire-game.html`.

[21] *The Application of Human Monte Carlo to the Chances of Winning Klondike Solitaire*. URL: `http://www.jupiterscientific.org/sciinfo/KlondikeSolitaireReport.html`.

[22] Xiang Yan et al. “Solitaire: Man versus machine”. In: (2004).

A User Manual

There are three programs which can be run in this project:

- *PlaySolitaire* - this program runs the human-playable solitaire.
- *runMCTS* - this program asks the user how many games to run, and whether or not to print the results. Then it runs the games using the MCTS the decide on which moves to play.
- *SolitaireStatistics* - this program works the same as the *runMCTS* program, except runs the program multiple times for each paramter to be tested.

For each of these programs, the type of solitaire to run is decided by the input file in the field called *file*. This can be changed at the top of these classes. There are four different solitaires pre-defined in the *solitaires* folder, which can be modified to induce different behaviours, or more notations can be created and added.

In *SolitaireStatistics*, there are multiple fields which decide the MCTS parameters to set, which are changed at the top of the class. These fields are:

- *minParam* - the minimum value of the parameter we are testing.
- *maxParam* - the maximum value of the parameter we are testing.
- *interval* - the size of the intervals to increase by
- *whichParam* - indicates which parameter to test. 0 = *maxNumSearches*, 1 = *maxRandomPlays*, 2 = *maxRandomPlayDepth*

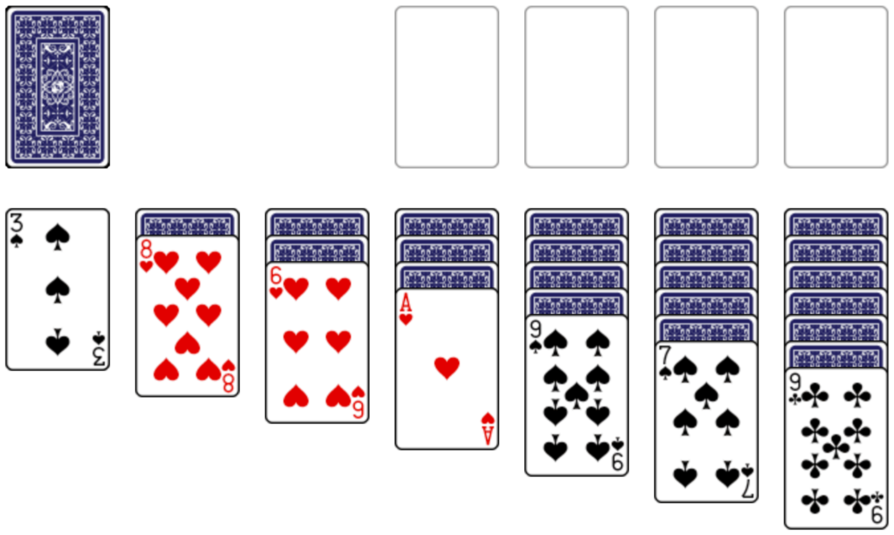
B Rules of Klondike Solitaire

B.1 Deal

The card layout at the beginning of a game of Klondile involves 7 stacks. The first stack has 0 face down cards, the second has 1 face down card, the third has 2, and so on, until the 7th and last stack has 6 face down cards. Each stack has an extra face up card on the top of these. The collection of these stacks are called the *tableau*.

After these cards have been dealt, the rest of the cards in the deck are placed in a pile called the *stock*, typically placed to the top-left of the tableau. To the right of the stock there is an empty pile called the *talon*, which will hold the cards once they have been taken from the top of the stock pile. In addition, to the top right of the tableau are four empty piles called the *foundation*. These represent the four suits, usually in the order hearts, clubs, diamonds, spades. Figure 16 shows an example of a deal in Klondike.

Figure 16: A Klondike Deal [1]



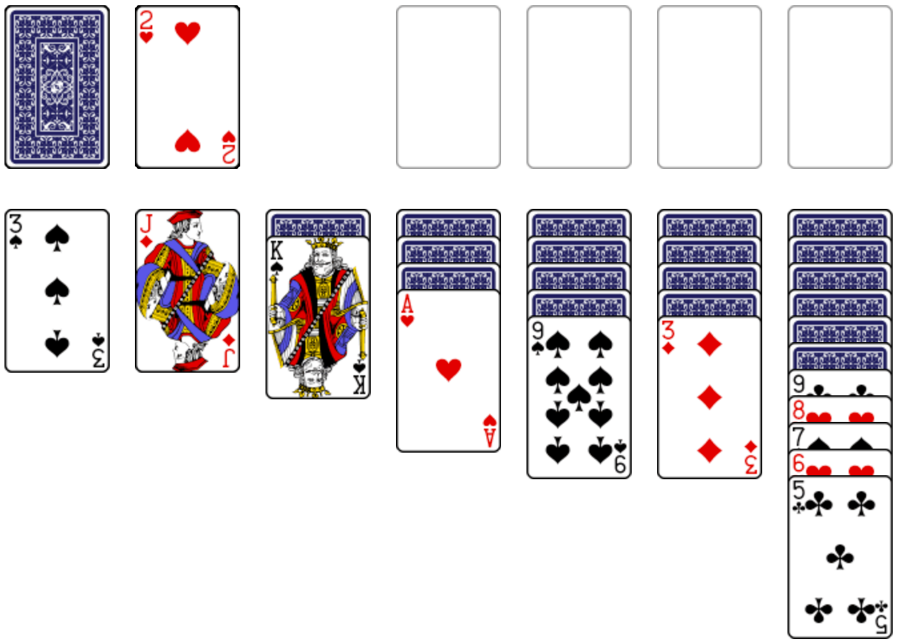
B.2 Objective

The objective of Klondike is to move all the cards to the four foundation piles. The cards must be stacked on the correct pile for their suit, and are stacked in ascending order from Ace to King.

B.3 Rules

Cards are stacked on the tableau in descending order (from King to Ace) in opposite colours. One example is shown in figure 17, where on the 7th tableau stack, the cards from 9 clubs to 5 clubs are stacked on top of each other. Because they are stacked by colour, not suit, the 7 spades is acceptable to be on the stack. Stacks of multiple cards may also be moved to another stack in the tableau, providing the face up card on that stack is the suitable card for the highest-value card (the bottommost card) on the stack.

Figure 17: A Klondike Stack [1]



If any of the tableau stacks have face down cards and no face up card, then the topmost face down card can be turned over to become a playable face up card. Similarly, if there exists cards in the stock, the topmost may be turned over at any point, with the card being placed face up in the talon.

Any topmost face up card (in either the talon, the tableau stacks or the foundation piles) may be moved to the tableau if there exists the card with value one above the object card's value in the opposite suit. Or, this card may be moved to the foundation pile of that suit if the card with value one below the topmost card is there.

If the stock pile becomes empty (and we have finished the deck of cards), the talon pile (if there are cards in the pile) is turned over and placed on the stock as a new pile. In the tableau, if any stack has no cards (either face up or face down), then the stack is empty and only Kings are allowed to be placed there.

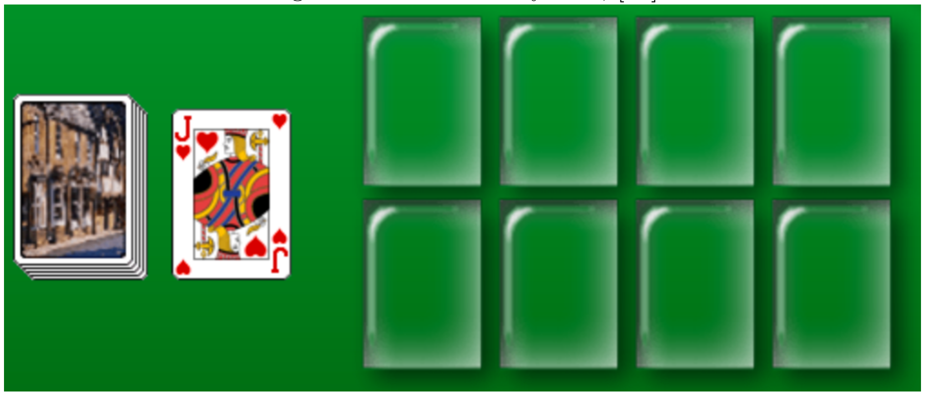
The game is played until the game is won (all cards in the foundation) or the game eventually blocks.[13]

C Rules of SirTommy Solitaire

C.1 Deal

There are no cards dealt out at the start of the game. Instead, all of the cards in the deck are placed to the side in a pile called the *stock*. To the right of the stock are four empty piles, called the *foundation*. In addition, there a four empty spaces below, for the *tableau*. The figure 18 shoes an example of a deal with the first card taken from the stock.

Figure 18: A SirTommy deal, [20]



C.2 Objective

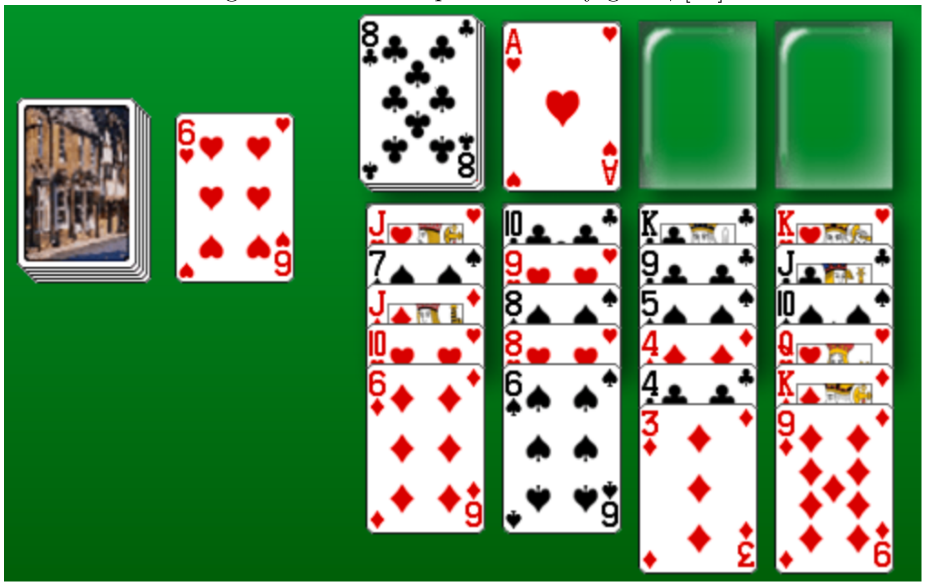
The objective of the game is to move all the cards to the foundation piles. These piles are build up in ascending order from Ace to King. However, the suit does not matter, so if there are , for example, a 2 on each pile, any 3 could be placed on any of the piles.

C.3 Rules

Any card taken from the stock is placed in the *talon*, the space next to it. This card is then placed in one of the four tableau stacks, no matter what cards are in the stacks. Once the cards are are placed on the tableau, only the topmost card on a stack can be moved to foundation. The cards cannot be moved between the stacks.

Once a card has entered the talon, it must be played. This means that when stock is empty and no more cards can be placed on the foundation, then the game is over. Figure 19 shows an example game part way through.

Figure 19: An example SirTommy game, [20]



D Experimental Data

The data used in section 9 were the result of many runs of the program. Originally, *SolitaireStatistics* was run for Sir Tommy with 100 runs for each interval of the parameters, with the results being stored in the folder *original-SirTommy*. Then this was run for Klondike with 50 games being ran for each parameter, which was ran 5 different times, with the results being stored in the folder *original-Klondike*. However, due to an error in the time calculation code in *SolitaireStatistics*, the times recorded in these results were incorrect (showing the milliseconds that has passed since 01/01/1970). This error does not affect the rest of the data accumulated.

These experiments were ran again, with the results being stored in the folders *measuringTime-SirTommy* and *measuringTime-Klondike*. Because the error in the first tests was not picked up for a while, there was not enough time to run as many Klondike games, hence a reduced number was run. These results are the data used for the graphs describing the average times of completion.

The original data was used for the percentage of games solved for Klondike and the average score for Sir Tommy. This is because there was a lot more data available for Klondike games, which made the results more reliable.