Department of Computer Science

Submitted in part fulfilment for the degree of BEng in Computer Science.

Solving Solitaire (Klondike) in Parallel

James Smith

2025 - April

Supervisor: Steven Wright

Acknowledgements

I would like to thank my supervisor Steven Wright for all of guidance during the completion of this project. I would also like to thank my friends and family for all of the support they have given me.

Contents

List of Figures

List of Tables

Executive Summary

The aim of this project was to investigate the solvability of Klondike solitaire, a well-known single player card game popularised by its inclusion in the Windows operating system. Due to the vast possibilities of different card configurations in any given game ($52!$ potential card combinations) and the complexity of the number of possible states a game can reach, the question of solitaire's solvability remains unanswered. Previous studies on the matter yielded varying results; the highest solvability rate found approximates that roughly 82% of games may be solvable. The aim of this project was to create a Java program capable of simulating a game of Klondike solitaire using automated solvers. This program was used to determine an approximate proportion of solitaire games that are solvable.

An evolutionary programming methodology (EP) was chosen for the development of the Java program. EP facilitates iterative development of the program, generating various prototypes at each stage of the project. Starting with the creation of a user-playable game of solitaire using a text based user interface, increasingly complex solvers were designed and implemented allowing for automated execution of the solitaire game. First was a random move solver, followed by a greedy heuristic solver and finally a Monte Carlo solver that used rollouts to determine the most favourable move to make. The final point of development incorporated parallel computation in to the program, allowing for the same game of solitaire to be solved multiple times without significantly reducing solver throughput. The aim was to reduce the chance that a solvable game was found to be unsolvable; games involving randomness produce varying move combinations for the same game, potentially resulting in a blocked card state . Manual black box testing was used to test each stage of development. The text-based UI developed in the first iteration was used to assess the behaviour of solvers and follow the flow of a solitaire game.

Large scale experiments were conducted on each solver using the University of York's high performance computing facility: Viking. The random move solver was tested using varying numbers of repeats permitted for each game: 1, 8, 16, 32, 64, 96, 128 and 256. The Monte Carlo solver was tested with 1, 8, 16, 32, 64 and 96 game repeats. For each number of repeated games, the Monte Carlo solver was tested with varying degrees of randomness used in the rollout simulations (10%, 30%, 50%,

80% and 100%). The random and heuristic solvers were ran on 10000 different games. Due to the increased complexity of the Monte Carlo solver, throughput was significantly reduced. The Monte Carlo solvers were ran on only 1000 different games.

When ran with no game repeats, the random solver performed poorly solving only 0.37% of games. However when allowed to run the same game 256 times, a success rate of 30.87% was achieved. The solvability rate and number of game repeats seemed to be linearly dependant; the success of a random solver is dependant on how many game repeats were permitted and as such, how long the solver is allowed to spend on each game. Due to the deterministic nature of the heuristic solver, running a game multiple times would provide no benefit: the same moves would be selected every time. The heuristic solver achieved a solvability rate of 8.06%. By fine-tuning the parameters of the Monte Carlo solver, a solvability rate of 80.7% was reached, closely aligning with existing estimates. However, using the Monte Carlo solver significantly increases the time taken to solve a game.

While the estimate generated by the Monte Carlo solver closely matched the estimates found during research of previous solutions and proves the effectiveness of rollout techniques, the results were constrained by the number of card configurations tested as a result of solver throughput. Future work should focus on optimising the Monte Carlo further to reduce the time taken to determine the solvability of a game. This could be achieved by caching previously tested game states, checking for blocking card configurations at the start of a game or investigating more effective heuristic strategies. By increasing the throughput of the solver, a larger subset of card configurations could be tested, generating a more accurate estimate for the solvability of Klondike solitaire.

Statement of Ethics

This section reflects the ethical considerations of this project and the measures put in place to comply with ethical principles. A copy of the ethical consideration checklist can be found in Appendix A.1

Avoidance of Harm

The avoidance of harm mainly refers to taking steps within the project to prevent any physical or psychological harm caused. This project requires no participants, so no harm will be caused during the project completion. Historically, versions of solitaire have been used as a gambling game. It could be argued that the results of this project could encourage gambling; however, the version of solitaire used for gambling is different from the Klondike version which is explored in this project, and so this should not be a problem.

Informed Consent

This project requires no participants, as such informed consent is not required.

Data Protection

No data regarding individuals will be collected, handled, or stored during this project, and so data protection is not applicable.

1 Introduction

Solitaire (Klondike) is a widely popular card game, particularly since it was included in the Windows 3.0 operating system in 1990. A lot of the time when playing solitaire, you will find that there are plenty of cards left on the table with no possible moves remaining. It is this regular outcome to the game that has led mathematicians to ask the question: How many of the games of solitaire that are played, are actually winnable? However, this question is not an easy one to answer. In fact, at this time there is no definitive answer, and the inability to determine the probability of winning a game of solitaire has been described as 'one of the embarrassments of applied mathematics' [1].

While the exact probability that any given game of Klondike solitaire being winnable is still unknown, it is currently approximated that around 82% of games are solvable [2]. The goal of this project is to create a Java program that can simulate a full game of Klondike solitaire and use this to test the theory that 82% of games are solvable.

Throughout this report, Klondike Solitaire will be referred to simply as Solitaire.

1.1 Project Objectives

This goal can be broken down into the following objectives:

- Evaluate existing solutions to extract the successes and failures of previous attempts.
- Investigate methods to solving games of Solitaire to apply to the developed Solitaire simulation.
- Develop a Java program capable of simulating a game of Solitaire.
- Optimise the created program ensuring the solution is as tractable as possible.

- Implement Monte Carlo techniques to parallelise many solitaire solves.
- Analyse results to determine an accurate bound for the probability of solitaire games that are solvable.

1.2 Project Outline

Below is a brief overview of the sections in this report and the contents of each:

Chapter 2 goes into the history of Solitaire. The rules of the game and terminology used throughout this report are explained and previous solutions to the problem are explored.

Chapter 3 looks at the software development methodology chosen for this project. The work completed in each iteration is outlined with key focusses highlighted. Finally, the inner workings of each solver are explored.

Chapter 4 looks at the experiments that were run on the various solvers created. The results of these experiments are then analysed.

Chapter 5 concludes the project, looking back at the original aims and evaluating the successes and failures of the project. Potential for future work on the project is explored.

2 Literature Review

2.1 The history of Klondike Solitaire

Playing cards and the games played with them have been around for centuries, however their exact origin is often disputed among historians. It is believed that they were introduced to Europe in the second half of the 14th century from the Islamic world [3]. Morehead believed playing cards were first introduced to northern Europe, likely Italy [4]. These early playing cards shared the same structure as those we see today but with different suit signs and court figures; playing cards did not originally display a Jack, Queen, and King. Alongside the cards, trick taking games (games played with a series of finite rounds) were introduced. Trumps and bidding games were a European invention that came later [3].

Solitaire only became known in the early nineteenth century, with the oldest known book published in Moscow in 1826. Over the course of the nineteenth century, much more literature regarding solitaire was written and spread across the globe. As an example, by the mid nineteenth century several collections on solitaire appeared in Scandinavian countries like Sweden [5]. The first piece of English literature about solitaire and patience games was created around 1870 by Adelaide Cadogan, titled "Lady Cadogan's Illustrated Games of Solitaire or Patience" [4]. Cadogan illustrates numerous variations of solitaire, and her work was later revised in 1914 to include American games. This included a game called Canfield. This is the closest example to the Solitaire most common today (the rules of which are discussed in Section 2.3) [6].

Canfield was a proprietor of a gambling salon in Saratoga during the 1890s. A customer would pay $50 for a deck of cards and receive $5 for every card that was successfully positioned in the foundation piles at the end of the game. It was estimated that Canfield stood to make $25 per game [4]. Solitaire's popularity dwindled after the late nineteenth century. It was not until Morehead and Mott-Smith released their book: 'The Complete Book of SOLITAIRE AND PATIENCE GAMES' that interest in the game was re-piqued [7]. Morehead details several different solitaire games and how they can be played. In fact, he and Mott-Smith describe more than 150 solitaires in their literature [4].

Nowadays, solitaire is a widely known card game. This is partly due to its inclusion in the Windows 3.0 operating system back in 1990. Microsoft solitaire was developed by Wes Cherry, an intern working at Microsoft, in 1988 [5]. As of May 2020 Microsoft solitaire hosted 35 million players each month with 100 million hands played daily across the world [8]. It has been included in every instance of Windows operating systems since its creation, with the exception of Windows 8. An article written by David Platt in 2015 describes the public's outrage to its exclusion. Platt also discusses the cultural and technological impact of solitaire and the role it played in popularising the Windows operating system: "it was solitaire that brought ordinary users into Windows" [9].

2.2 Solitaire rules and definitions

As discussed in the previous section there have been several variations of solitaire throughout the years. Below is the full breakdown of the rules and terminology that will be adhered to in this report.

2.2.1 Terminology

Piles : There are 7 piles. Starting from the left, one card is placed face up. Next to this card, two cards are placed with only the top card being face up. Then three cards and so on following the same rule until the final pile contains 7 cards.

The stock : The 24 cards not dealt into the piles. These are placed face down. This is where cards are drawn from.

The waste : Space where cards drawn from the stock are placed. Only the top card on the waste can be used.

A build stack : Placed on the piles. A build stack contains cards placed in descending rank order, each card with alternating colour.

The tableau : Where the piles and build stacks are placed.

The foundation : There are four slots in the foundation, one for each suit (♡,♢,♣,♠). Cards can be placed here in ascending rank order.

Visual representation of initial game set up can be seen in Figure 2.1.

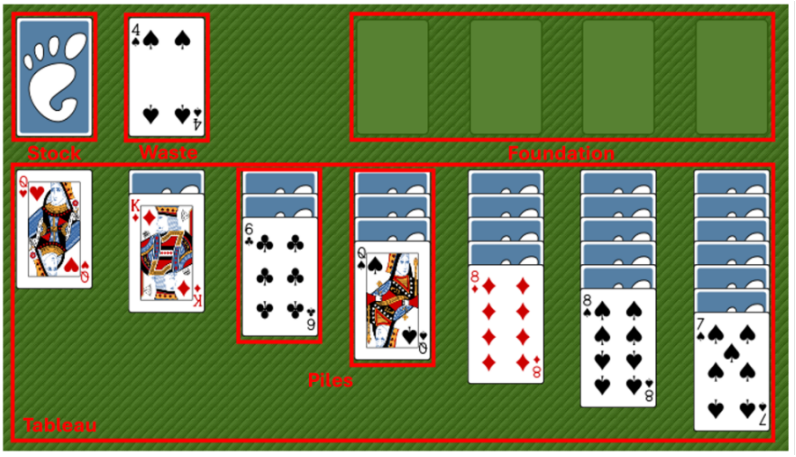

Figure 2.1: A figure showing the initial set up of Solitaire and the names of each component. Unannotated image source: https://en.wikipedia.org/wiki/Klondike_%28solitaire%29

2.2.2 Rules

- The game should be played with a standard deck of cards: 52 cards, 4 suits (♡,♢,♣,♠), 13 ranks per suit ($A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K$), ♡ and ♢ cards are red, ♣ and ♠ cards are black.
- Cards should be shuffled before play. 28 cards placed on the tableau in 7 piles (as per the piles definition above). The rest are placed face down in the stock.
- **Drawing cards from the stock**:
  - Cards are drawn in sets of 3 from the stock and placed face up in the waste.
  - Only the top card on the waste can be accessed during play.
  - Should the stock be emptied, the pile of cards from the waste is placed face down back onto the stock.
  - The stock can be replenished and cards drawn from it indefinitely.
  - If there are fewer than 3 cards in the stock, all remaining cards are drawn to the waste.
- **The foundation**:
  - A card of rank x can only be placed on the foundation if the card

of rank $x - 1$ is already on the foundation.

- A card must be placed on the foundation pile of the matching suit (e.g. a ♠ cannot be placed on a ♡,♢ or ♣ foundation pile).
- Once placed on the foundation, cards **cannot** be moved back onto a build stack.
- The game is won when all cards have been placed on the foundation.

- **Build stacks**:
  - A card x can only be moved onto a build stack if the existing card on top of the build stack has the rank $x + 1$ and is the opposite colour.
  - E.g. with build stack: 5♣, 4♡, 3♣ only a 2♡/♢ could be added.
  - A build stack can be moved to another pile in its entirety so long as the previously mentioned rule is followed. It can also be split and part of a build stack can be moved to another pile.
- A king can be moved to any empty pile.

2.3 Previous Solutions

2.3.1 Unsolvable and Unplayable Games

The question of Solitaire's solvability has been pondered by many mathematicians, computer scientists, and academics for years. As previously mentioned, the current chance any given game of solitaire is solvable is known as roughly 82%. In this section, the methods and practices used to reach this estimation are explored.

When creating a program that is able to solve a game of Solitaire, it will be helpful to understand card configurations that prevent a game from being solved. Ruiter [10] discusses in detail sets of card configurations that result in a game of solitaire being unsolvable. These card configurations are described as Type 1 and Type 2 deals, each is explained below:

A Type 1 blocking set is described as follows.

- A given card x is dealt onto the final position in a pile and so is placed

face up (e.g. a 7♡).

- Somewhere in the same pile as x are two cards of the opposing colour with value $x + 1$ (e.g. the 8♠ and 8♣).
- Somewhere in this pile there is also a card whose value is less than x and suit is equal to x (e.g. the 3♡).
- The 7♡ cannot be placed on the foundation before the 3♡. The 7♡ cannot be moved off of this pile because the only cards it could be placed on (the 8♠ and 8♣) are behind it in the pile and cannot be accessed.
- The 3♡ cannot be accessed without moving the 7♡ and so cannot be put on the foundation.

This formation of cards creates a "primitive" blocking set and so the game can never be won.

A type 2 blocking set is where multiple "primitive blocking sets are intertwined in a way that a lock card might not necessary lock up the remainder of its primitive blocking set within its own pile, but the lock cards together do lock up the remainders of their primitive blocking sets". An example of a type 2 blocking set can be seen in Figure 2.2.

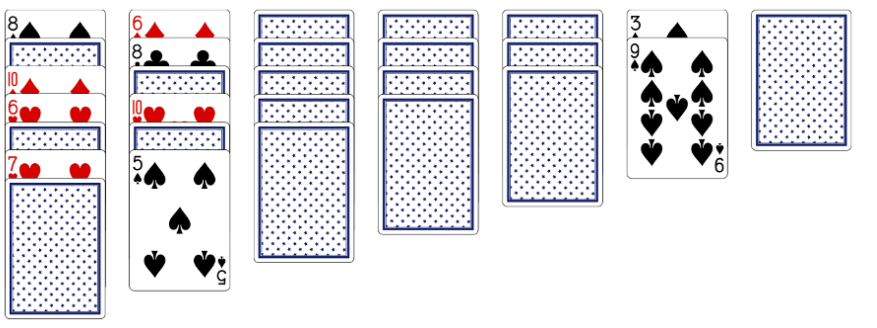

Figure 2.2: A figure representing a type 2 blocking set as defined by Ruiter [10]

While Ruiter's paper is not directly focussed on the solvability of solitaire, rather determining the number of unplayable games, it is useful to understand blocking sets that create an unsolvable game of solitaire. A game may contain a blocking set but still have other available moves. Identifying whether a game is unsolvable can be done immediately after each move, eliminating wasted time and computational resources on a game that will eventually have no possible moves.

In order to determine these blocking sets the player would need to be able

to see the face-down cards, meaning it would be impossible to do playing a standard game of solitaire. There is a variation of Solitaire known as "Thoughtful Solitaire" where the player can see the cards hidden below a build stack. This variation of the game is widely used when creating Solitaire solvers (for example by Yan et al. [1]); the solver created in this report will be for a game of thoughtful Solitaire.

It is also important to consider whether a game is playable: are there any legal moves immediately after the cards have been dealt? Kortsmit [11] outlines the conditions which need to be met in order for a game to be unplayable:

- There is no ace present amongst the playable cards (this could be immediately placed onto the foundation).
- No playable card on the piles can be placed onto another pile.
- No playable card in the stock can be placed onto a pile.

2.3.2 Monte Carlo Techniques and Rollouts

Almost all previous solutions to this problem have determined the solvability of solitaire by using Monte Carlo techniques.

"Monte Carlo simulation is a type of simulation that relies on repeated random sampling and statistical analysis to compute the results." [12]. It is useful when modelling probabilities of different outcomes in a scenario where said outcomes are difficult to predict due to randomness. Hence it is very useful when solving the problem of this report. In a standard 52 card deck, there are $8 * 10^{66}$ [13] possible combinations that can be made. Using brute force to solve each of these games is obviously infeasible. In Monte Carlo simulation we would take a random sample from this set of possible card combinations and run simulations (attempts to solve the game) on this sample to compute a predicted final percentage of winnable games.

Rollouts are a type of Monte Carlo technique. First used by Tesauro and Galperin [14] in their solver for backgammon. It was Yan [1] that first used rollouts in a solitaire solver.

At any point in a game of Solitaire, there will be multiple possible moves. For each move, simulate the rest of the game using calculated heuristic values to drive decision making, until the game is either won or lost. The move that performed the best will be chosen. It was with this strategy that Yan achieved a 70.20% success rate.

Later Fern [2] optimised the rollout strategy with his multistage nested rollouts. A nested rollout is similar to the rollouts used by Yan, however instead of simply using heuristic values to drive decision making, another rollout is recursively performed on the proposed move to determine its validity. In other words, say you are in game state x with two possible moves. Rolling out one of these moves takes you to game state x' which has three possible moves. A rollout will be performed on each of those moves to determine which is best.

This is developed further with multistage nested rollouts involving the application of multiple heuristics for different stages in the game (this is discussed in more detail in Section 2.3.3).

This optimised use of rollouts improved the bound found for the solvability of solitaire, concluding that 82% of games were winnable. It is clear that the use of rollouts greatly improves the decision making in a game of solitaire increasing the effectiveness of solitaire solvers.

2.3.3 Heuristics

While there is no known perfect strategy for solving a game of Solitaire, there are certain moves that are deemed more favourable than others. This has resulted in various heuristic approaches being applied to optimise game moves in an attempt to increase the likelihood that the move taken will lead to the game being solved.

Yan et al. [1] chose to base the heuristics used in their solution from the scoring system used in the Microsoft Windows solitaire, as can be seen in Table 2.1.

Table 2.1: Table outlining the heuristic rules used by Yan et al. [1]

Move	Points
Card moved from stock to pile	+5
Card moved from pile to foundation	+5
Card moved from foundation to pile	-10
Any other move	0

Yan then created his own rules for when multiple moves had the same heuristic scores. The rules for if a move is from one build stack to another can be seen in Table 2.2, where k is the number of originally face-down cards on the source stack. The rules for if a move is from the stock to

a build stack can be seen in Table 2.3. Any other card move would be assigned a priority of 0.

Table 2.2: Table outlining move priority when moving a card from one pile to another. [1]

Condition	Priority
Move turns an originally face-down card face-up	$k+1$
Move empties a stack	1

Table 2.3: Table outlining move priority when moving a card from the stock to a pile. [1]

Condition	Priority
Card being moved is not a King	1
Card is King and matching Queen is in the stock, in the waste, in the foundation, or is face-up in a pile	1
Card being moved is King and matching Queen is face-down in build stack	-1

The use of these heuristics was successful; alone the heuristics achieved a 13.05% success rate in 10000 games played. Combining heuristics with rollouts resulted in a success rate of 70.20%. [1]

Kortsmit [11] proposes a heuristic strategy that assigns a score to an entire game state, rather than each move. The use of this heuristic method resulted in 13.75% of games being found solvable. While this is an interesting approach it did not use Monte Carlo techniques and so will not be used in this implementation.

Fern et al. [2] also created their own heuristics to improve decision making within the solitaire solver. Their approach involved two different heuristics: H1 and H2. H1 would be used at the start of a game until a local maxima is reached, at this point H2 will be used. These heuristics are outlined in Figure 2.3.

Num	Description of Feature for Card x (# of elements)	H1	H2
1	x is in a Foundation stack (52)	5 - **rank value**	5
2	x is face down in a Tableau stack (52)	**rank value** - 13	**rank value** - 13
3	x is available to be moved from the K^+ Talon (52)	0	1
4	x and the other card of same rank and colour are both face down in some Tableau Stack (26)	-5	-1
5	x is blocking a suited card of lesser rank (4×(12+11+...+1))	-5	-1
6	x is blocking one of its two Tableau *build cards* (48×2)	-10	-5

Figure 2.3: A figure representing the heuristic strategy used by Fern et al. [2]

H1 favours revealing hidden cards while H2 favours moves that progress the game towards a solution. With the use of these heuristics and nested rollout techniques Fern concluded that 82.24% solitaire games were solvable.

It is clear that use of relevant and effective heuristics is essential to optimise the solving of a game of solitaire. The literature studied provides multiple heuristic approaches with varying success rates that will be considered when creating a solver.

3 Methodology and Implementation

3.1 Development Methodology

An iterative and incremental (I&I) methodology has been chosen for the creation of the solitaire solver. I&I development allows for the project to be organised into "small mini projects" [15]. These projects can be developed one step at a time in the form of an expanding model [16], facilitating changes to the requirements and the design. The specific I&I methodology chosen is evolutionary programming (EP), allowing flexibility in the software development lifecycle for a program that will be continuously evolving [17] throughout development. EP's iterative nature is the reason for its choice; it is known prior to starting that there will likely be changes to the design of the program as it is implemented, with new optimisations or changing class structures for each new feature that is added. The evolutionary programming model used in this project can be seen in Figure 3.1.

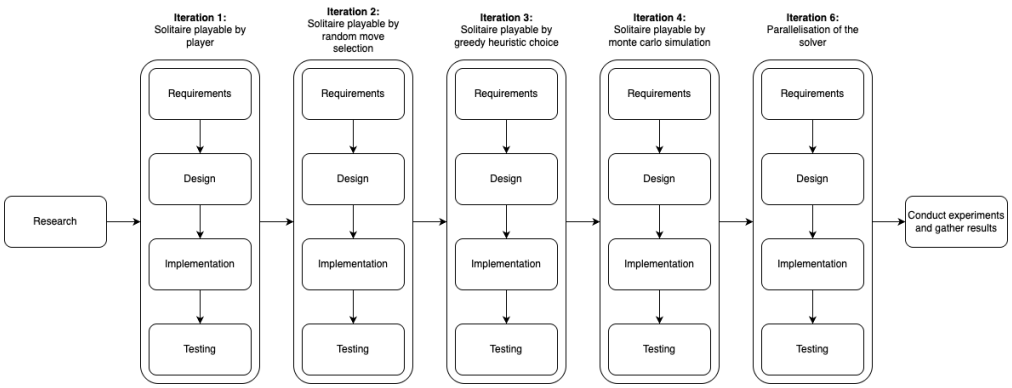

Figure 3.1: A figure showing the steps taken during the completion of this project in accordance to the evolutionary programming methodology.

Note that I&I methodologies such as EP would usually end in maintenance. However as this is not an application that will be running consistently, but rather a program to be run a few times in order to gather results, the

maintenance step has been excluded.

3.2 Iterative Development Overview

As can be seen in Figure 3.1, each iteration involved a requirements, design, implementation, and testing section. Note: a full set of requirements for the project can be found in Appendix A.2. Below is a brief overview of each iteration and its core focus:

User-playable Solitaire game : Iteration 1 creates a simple solitaire game that is playable by a real user. The focus is to build the core foundation of the game. While later iterations do make some changes to the program created here, the underlying algorithms (used to move cards and check the legality of moves) remain largely the same.

Random solver : Iteration 2 builds on the previous iteration by changing the format of the main game loop. Rather than a user selecting the moves to make, the game is progressed automatically. The random solver finds all possible moves and selects one at random.

Heuristic solver : Iteration 3 implements an algorithm to assign each move heuristic values depending on their effectiveness. A greedy heuristic solver is created, selecting the move whose heuristic score is the highest.

Monte Carlo Solver : Iteration 4 combines the logic from both the random and heuristic solvers to create the Monte Carlo solver. For each possible move the rest of the game is simulated using an R-greedy algorithm (See Section 3.4.3). The move with the highest Monte Carlo score is selected.

Adding parallelisation : The final iteration adds parallelisation to the solvers, allowing the same game (same configuration of cards) to be run through a solver multiple times without significantly reducing throughput. This reduces the possibility that solvers involving randomness will produce a false negative result (determine a solvable game to be unsolvable).

3.3 Program Structure

The different aspects of a game of solitaire are used to structure the program; each fundamental component of the game (see Figure 2.1 for reference) is a class within the solver. A breakdown of the program structure can be found below: (A full UML class diagram of the program can be found in Appendix A.3)

Pile : Each pile is represented by two stack data structures, one for the hidden cards and one for the build stack.

Stock : Dealing form the stock to the waste is abstracted and the stock is modelled as an ArrayList<>. A pointer variable will be used to manage the stock card deals and card access.

Foundation : The foundation class will store integer values for each suit. These values will be incremented when a card is added to a foundation pile.

Card : Each card within the deck is an object, with rank (integer), suit (char) and colour (boolean) as attributes. A location attribute represents the cards position in the game.

Deck : Facilitates the creation and shuffling of the deck of cards.

Move : Moves in the game are represented by a Move object with two attributes: the card being moved and the destination Pile. There are two constructors, one requiring the Card and one requiring both the Card and the destination Pile (used to represent different move types). Move also stores attributes for scoring with the Monte Carlo and heuristic solvers.

Solitaire : The Solitaire class controls the game of solitaire, dealing the deck as outlined in Section 2.2.2. The Solitaire class facilitates the movement of cards (Stock to Pile, Stock to Foundation, Pile to Foundation, and an entire/partial build stack move to a different Pile) between components of the game, as well as containing methods for move determination used by each solver.

Run : Run is where all different solver types are kept. A method runSolver() determines and runs the relevant solver in parallel based on the parameters provided.

GameStateCopy : Used by the Monte Carlo solver to save a copy of the current state of the game. A Monte Carlo simulation is run on the game and once complete, the game is backtracked to the state of the

game when the copy was made, ready for the next simulation.

3.4 The Solitaire Solvers

For a game to be solved automatically the solver needs to be able to find all usable cards and for each of those cards find all possible moves.

Solitaire.getUsableCards()

The getUsableCards method is implemented using the K^+ state representation [2] where every reachable card in the stock is accessible at one time. The cards in the build stacks of each pile are also all playable and are added to the usable cards list. Figure 3.2 shows the algorithm used to find all usable cards in a game state.

```
ArrayList<Card> usableCards = new ArrayList<>();
try {
    Card card = stock.draw();
    while (!usableCards.contains(card)) {
        usableCards.add(card);
        card = stock.draw();
    }
} catch (EmptyStockException e) {
    // Do nothing, stock is empty
}
for (Pile pile : piles) {
    if (!pile.getBuildStack().isEmpty()) {
        usableCards.addAll(pile.getBuildStack());
    }
}
return usableCards;
```

Figure 3.2: Code snippet from getUsableCards() in Solitaire

Solitaire.getPossibleMoves()

getUsableCards() is called in the getPossibleMoves() method. Each card in the usable cards list is iterated through and checked against the foundation piles and the top card of each pile's build stack. If either (or both) moves are valid, a new Move object is created and added to a list which is later returned.

Solver structure

All solvers follow the same code structure (See Figure 3.3). Note: all solvers return the number of moves made. If this value is negative, the game is lost, if positive the game is won.

```
boolean end = false;
while (!end) {
    possibleMoves = game.getPossibleMoves();
    if (possibleMoves.isEmpty()) {
        return -movesMade;
    }
    //Determine move here
    game.makeMove(move);
    movesMade++;
    String currentState = game.getGameState();
    if (Collections.frequency(gameStates, currentState) > 5)
    {
        return -movesMade;
    }
    else if (game.getFoundation().checkWin()) {
        return movesMade;
    } else {
        gameStates.add(currentState);
    }
}
return -movesMade;
```

Figure 3.3: General structure of the solvers

A game is lost if there are no possible moves left or if the same game state is repeated 5 times (indicating the same moves are being made in an endless loop).

3.4.1 Random Move Solver

Java.util [18] contains a Random class that was imported and used to facilitate the random move selection (see Figure 3.4).

```
int randomInt = (int) (Math.random() * possibleMoves.size());
Move move = possibleMoves.get(randomInt);
```

Figure 3.4: Random move selection

3.4.2 Heuristic Solver

Table 3.1 outlines the heuristic strategy designed for the greedyHeuristic-Solver. The heuristic strategy devised has been created to favour moves to a foundation and moves that reveal cards, and prevent moves that create blocked cards. The priorities assigned are those explained by Yan et al. [1].

Table 3.1: Table of heuristics. Note that n = the number of cards revealed by a move.

Move	**Heuristic**	**Priority**
Foundation moves	$+10$	
If move reveals a hidden card	$+10$	
If card is an Ace	$+10$	
Let card have rank x. If a card of the opposite colour and rank $x-1$ is hidden	set to -5	
Moving from stock to a pile	$+5$	
If card is a King and a Queen of the opposite colour is hidden		-1
Else		1
Move entire build stack to pile		
If the move reveals a card	$+10$	$1+n$
If the move creates an empty pile	$+5$	
If card is a King and is being moved to an empty pile	$+5$	
If moving a King from an empty pile to another empty pile	set to 0	
Move partial build stack to pile	set to 0	
If move could create a move that reveals a hidden card on the next turn	$+5$	

Using the cards location along with the destination pile (or lack there of), the type of move is determined. Then for each move in the list of possible moves, the heuristic values and priority are set. Figure 3.5 shows the algorithm used to determine heuristics for entire build stack moves. (See Appendix A.4 for the full algorithm to determine heuristic values for all move types).

```
if (!piles[card.getLocation()].getHiddenCards().isEmpty()) {
    this.updateHeuristic(10);
    this.setPriority(1 + piles[card.getLocation()].
   getHiddenCards().size());
} // Full build stack move will reveal a hidden card
else {
    this.updateHeuristic(5);
    this.setPriority(1);
} // Full build stack move will create an empty pile

if ((card.getRank() == 13) && (dst.getPile().isEmpty())) {
    if (piles[card.getLocation()].getHiddenCards().isEmpty())
    {
        this.setHeuristic(0);
    } // Moving a king to empty pile creating another empty
   pile achieves nothing
    else {
        this.updateHeuristic(5);
    } // Moving a King to an empty pile
}
```

Figure 3.5: Code snippet from Move.determineHeuristic() showing the algorithm to determine heuristics and priority for entire build stack moves.

Once the heuristic values are determined, the move with the highest heuristic score is selected and made. Should two scores be the same, the move with the highest priority is selected.

3.4.3 Monte Carlo Solver

The Monte Carlo solver uses rollouts to determine which of the possible moves will be made; a rollout involves taking a possible move and simulating the rest of the game after that move has been made.

As mentioned in Section 3.3, GameStateCopy is used to backtrack the game during simulations. The same happens when testing different moves: A copy of the game is created, a potential move is made, the move is simulated, and once simulations are complete the game backtracks to the copy made, ready for the next move to be tested.

During each simulation, a move is chosen by an R-greedyHeuristic algorithm, meaning a degree of randomness R is used when selecting a move. There is an R% chance the move chosen is random, otherwise the greedyHeuristic choice is made. (See Appendix A.5 for the full R-greedyHeuristic algorithm: greedyHeuristicPriorityWithRandom())

After each simulation, the number of face-up cards are returned by the R-greedyHeuristic algorithm and used to increase that moves Monte Carlo score. Once all moves have been simulated, the move with the highest Monte Carlo score is selected. Figure 3.6 shows the simulation loop used in the Monte Carlo solver.

```
1  for (Move move : possibleMoves) {
2      history.push(new GameStateCopy(game));
3      // Save game state before move
4      game.makeMove(move);
5      for (int i = 0; i < numSimulations; i++) {
6          history.push(new GameStateCopy(game));
7          // Save game state after move for simulation
8          int gameSim =
       greedyHeuristicPrioritySolitaireSolverWithRandom(game);
9          if (gameSim == 52) {
10             return movesMade + 1;
11             // If win found in simulation, this must be a
       winning configuration, no need to run more sims.
12         }
13         else {
14             move.setMonteCarloScore(gameSim);
15         }
16         history.pop().restoreGameState(game);
17         // Restore game to state after the simulated move was
        made
18     }
19     history.pop().restoreGameState(game);
20     // Restore game to state before move was simulated
21 }
```

Figure 3.6: Code snippet from Run.monteCarloSolitaireSolver() showing the simulation loop.

An optimisation to the solver can be seen on lines 9-10 of Figure 3.6. Initially if a simulation found a win, the rest of the moves would continue to be simulated. This computation was unnecessary; if a simulation wins the game then that game is solvable. So if the R-greedyHeuristic algorithm returns 52 (i.e no cards are left hidden in the game) then the solver returns the running total of moves made + 1 (the move that was simulated and won the game). This optimisation also saves computation time by removing the need to move all cards to the foundation. If all cards are face up then the game is always solvable.

3.5 Parallelising the Solvers

The java.util.concurrent [19] package was imported in order to facilitate parallelism in the solver.

3.5.1 Pile, Stock, Foundation and Card constructors

The Pile, Stock, Foundation and Card classes all handle mutable objects within the program. Parallelising the solvers (in iteration 5) means having multiple of the same game running at the same time; multiple copies of the same game need to be made.

In Java there are two different ways to copy objects: a deep copy and a shallow copy [20]. A shallow copy creates a new object, but does not create a copy of any objects it references (E.g. during parallelisation a new Solitaire object will be created, but not new Stock, Piles etc.). Instead, a reference to the original objects is created. If a shallow copy is used during parallelisation, race conditions on the mutable objects within the game will be created resulting in unpredictable behaviour and program crashes.

Instead, a deep copy is required: creating a new object with a new copy of all referenced objects. All mutable components (Pile, Stock, Foundation and Card) as well as the Solitaire class have additional constructors facilitating the creation of a deep copy. These constructors take their own class data type as a parameter. Figure 3.7 shows the deep copy constructor for the Solitaire class.

```
1 public Solitaire(Solitaire other) {
2         this.foundation = new Foundation(other.foundation);
3         this.piles = new Pile[other.piles.length];
4         for (int i = 0; i < other.piles.length; i++) {
5             this.piles[i] = new Pile(other.piles[i]); }
6         this.stock = new Stock(other.stock); }
```

Figure 3.7: Code from the Solitaire(Solitaire other) constructor

Deep copy constructors for Pile and Stock ensure new Card objects are created as required.

3.6 Testing

As individual aspects of the code are implemented, they are debugged using console outputs and custom exceptions.

Once each iteration is complete, manual black-box testing is conducted; the program is tested as if it is being run by a user with no knowledge of the internal code structure [21]. The tests conducted are done so according to the requirements table in Appendix A.2.

The text-based UI created in iteration 1 is used throughout development as a method of manual black box testing (See Figure 3.8). It allows the state of the game to be visualised and used to check algorithms within the program perform as expected.

```
QD      0C 0S 0H 0D
[6D]
[-, 2D]
[-, -, 7S]
[-, -, -, AD]
[-, -, -, -, 3D]
[-, -, -, -, -, 2C]
[-, -, -, -, -, -, 10D]
//A move is made//
QD      0C 0S 0H AD
[6D]
[-, 2D]
[-, -, 7S]
[-, -, QH]
[-, -, -, -, 3D]
[-, -, -, -, -, 2C]
[-, -, -, -, -, -, 10D]
```

Figure 3.8: Example of text-based UI used for manual black-box testing, showing a test for moving from a build stack to the foundation.

4 Results and Analysis

4.1 Experiments

The aim of this project is to test the theory that 82% of games of solitaire have a winning solution. This is done by creating a solitaire solver in Java.

Random move, greedy heuristic and Monte Carlo solvers will be run on several different card configurations of a solitaire game. The experiments to be run are as follows:

Random solver: Experiments will be ran to test the effect that the number of times a single game is repeated has on the solvability. The number of game repeats tested will be: 1, 8, 16, 32, 64 and 96, each ran on 10000 games.

Heuristic solver: Two different types of heuristic solver will be tested: a greedy heuristic solver and a greedy heuristic solver with priority. Each will be ran 10000 times. Each game will be ran only once due to the deterministic nature of the solvers.

Monte Carlo solver: The Monte Carlo solver will be ran 1000 times per experiment. Like the random solver, varying game repeats will be tested (1, 8, 16, 32, 64 and 96). Each repeat number will also be tested on a different degree of randomness used in the R-greedy heuristic algorithm (10%, 30%, 50%, 80% and 100%).

Success is measured by what percentage of games were successfully solved, with other metrics such as time taken and number of moves made recorded. Time will be used to determine the effect different parameters (number of game repeats and randomness percentage) have on the efficiency of the solvers. The moves per game will be used to analyse the effect the degree of randomness has on Monte Carlo solves.

All experiments will be run on the University of York's high-performance computing facility: Viking [22].

4.2 Results

4.2.1 Random solver results

Table 4.1: Table showing the results to experiments conducted on the random solitaire solver. Each experiment was run on 10000 games.

#	Wins	%	Moves (win)	Moves (loss)	Time (ms) (win)	Time (ms) (loss)	Total time (mins)
1	37	0.37%	314	43	2.84	0.65	0.17
8	288	2.88%	332.5	42	4.44	2.25	0.43
16	612	6.12%	339.5	43	5.19	2.82	0.55
32	1031	10.31%	339	43	6.29	4.3	0.80
64	1645	16.45%	336	43	9.97	7.97	1.47
96	2009	20.09%	335	43	15.06	12.6	3.30
128	2212	22.12%	333	43	19.41	17.44	3.08
256	3087	30.87%	334	43	42.63	39.72	6.87

Table 4.1 shows the results from the random solitaire solver. With only one of each game played, solvability is very low at 0.37%. As the number of games repeated increases, so does the solvability rate, reaching 30.87% with 256 of the same game played. This is illustrated in Figure 4.1.

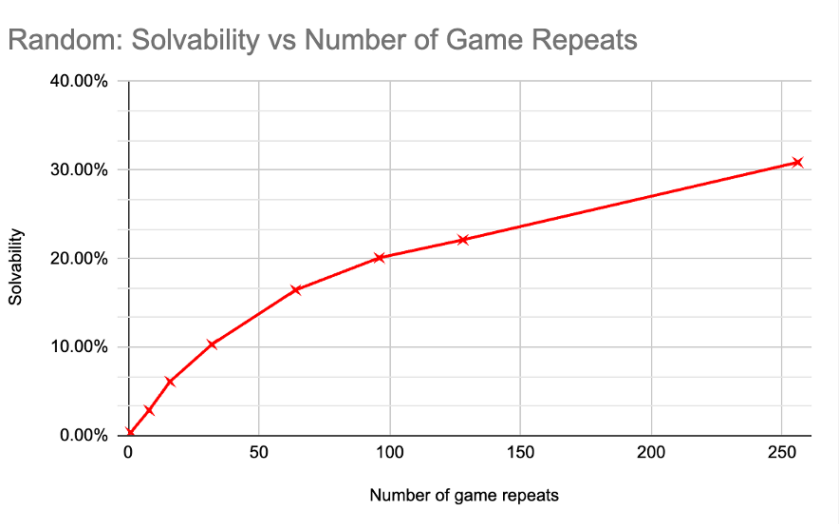

Figure 4.1: A graph showing the solvability of solitaire games with the random solver vs the number of times a game was repeated.

Figure 4.2 shows the rapid increase in solver execution time as the number of repeated games also increases.

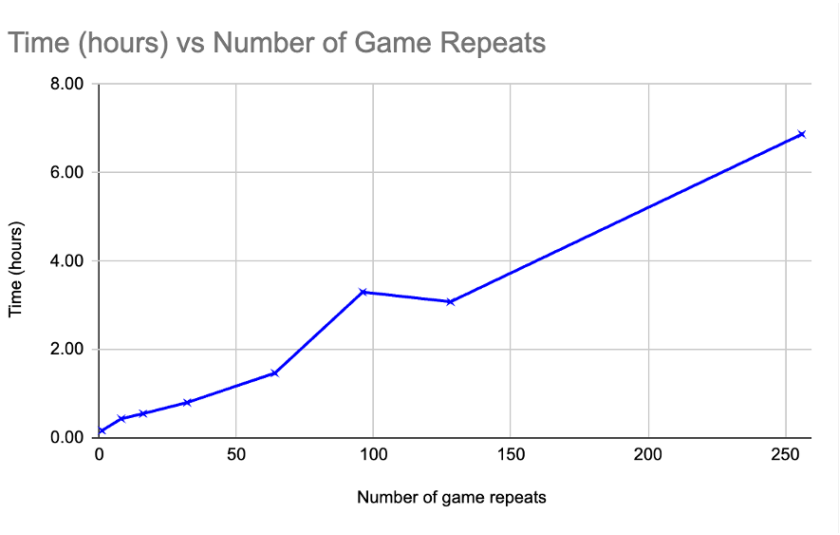

Figure 4.2: A graph showing the run time of the random solver vs the number of times a game was repeated.

4.2.2 Heuristic solver results

Table 4.2: Table showing the results of the heuristic solver with and without move priority taken into account.

Priority?	Wins	%	Moves (win)	Moves (loss)	Time (ms) (win)	Time (ms) (loss)	Total time (s)
No	621	6.21%	84	32	1.03	0.28	7
Yes	806	8.06%	85	32	1.03	0.3	8

Table 4.2 shows the impact of including a priority variable to the heuristic solver. Without, the solver reached 6.21%. With priority taken into account this increased to 8.06% with a negligible reduction in solver efficiency.

4.2.3 Monte Carlo results

Table 4.3: Table showing the most successful results conducted on the Monte Carlo solvers for each number of times the same card configuration was played.

#	Rand. %	Wins	%	Moves (win)	Moves (loss)	Time (ms) (win)	Time (ms) (loss)	Total time (hrs)
1	80%	765	76.5%	1.81	59.26	16	3293	0.66
8	100%	781	78.1%	2.73	57.53	170	7570	2.17
16	80%	807	80.7%	1.87	60.21	76	5400	1.83
32	30%	782	78.2%	2.06	59.19	50	5383	1.20
64	30%	800	80.0%	1.94	60.35	84	7164	1.75
96	50%	795	79.5%	1.79	63.83	115	20935	3.27

Due to the large volume of results for the Monte Carlo solver, Table 4.3 shows only the experiments that performed best for each number of times a game is repeated. The highest percentage of solvability achieved is 80.7%, when a game is run 16 times, with a randomness of 80%. A close second reaches 80% when a game is played 64 times with a randomness of 30%.

Figure 4.3 illustrates the full set of results for the Monte Carlo experiments.

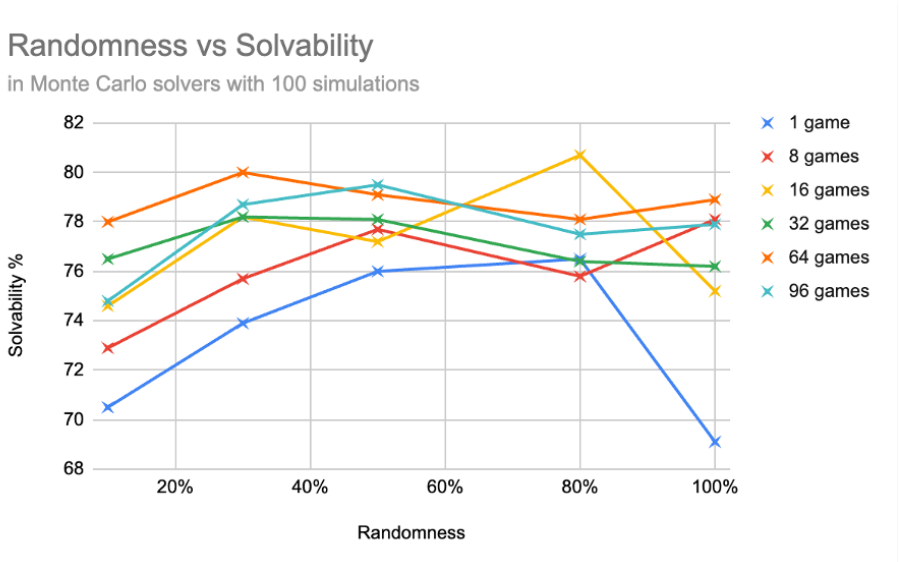

Figure 4.3: A graph showing the solvability of solitaire games vs varying degrees of randomness for the Monte Carlo solver.

The effect of randomness

As could be expected, increasing the degrees of randomness resulted in an increase to the time taken to solve a game (as is illustrated in Figure 4.4). Total program execution time shows the same correlation and the time it takes for a game to be determined as unsolvable slightly increases.

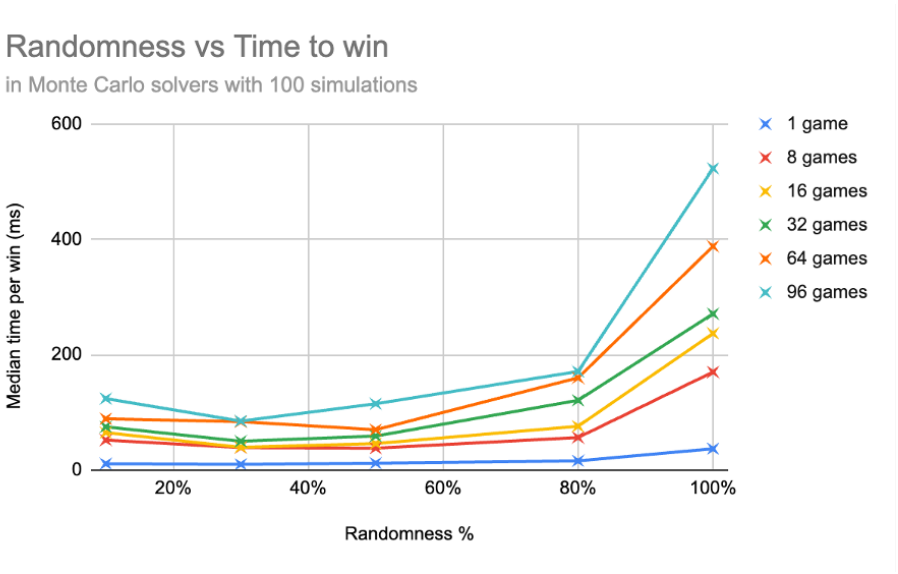

Figure 4.4: A graph showing the effect randomness has on the time taken to find that a game is solvable.

For losing games, there is a negative correlation between the degree of randomness and the number of moves made before the game ends (See Figure 4.5).

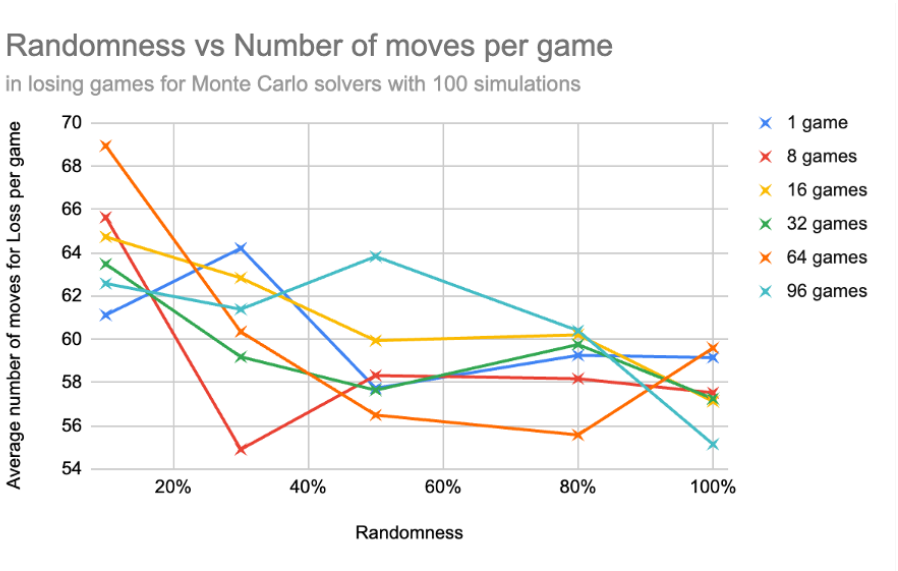

Figure 4.5: A graph showing the moves required to find a game is unsolvable vs the degree of randomness used in the Monte Carlo solver.

However with winning games, the moves required to find a game to be solvable are significantly higher at lower and higher degrees of randomness, and stay relatively similar elsewhere.

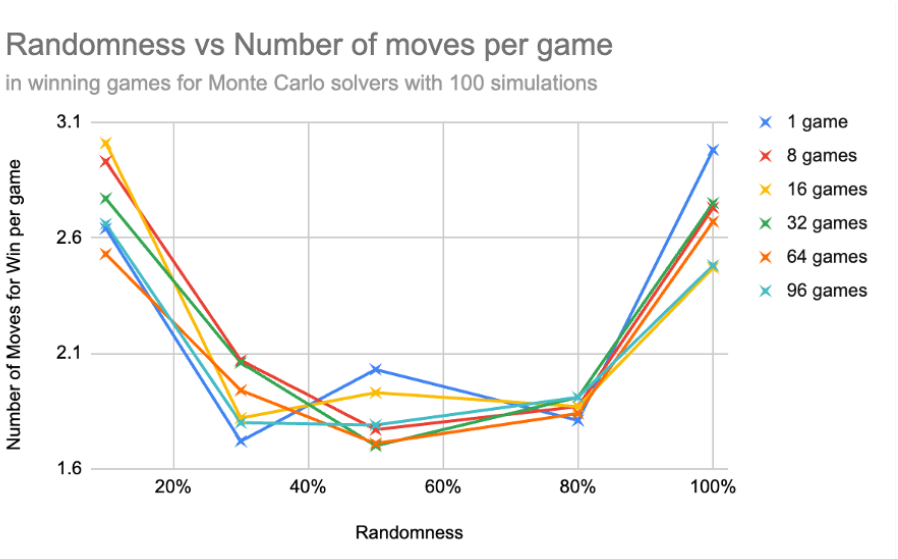

Figure 4.6: A graph showing the moves required to find a game is solvable vs the degree of randomness used in the Monte Carlo solver.

4.3 Analysis of Results

4.3.1 Random solver results

The random solver was expected to be the least effective in determining solvability of solitaire games. With a single game this assumption was correct, finding less than 1% of games to be solvable. However, when running the same game multiple times the effectiveness of the solver improved, reaching a solvability of 30.87% with 255 repeats of each game. With a linear correlation between the number of game repeats and the time taken to determine a game's solvability, the success of the random solver is dependant on how long it is allowed to run the same game before moving on.

4.3.2 Heuristic solver results

The heuristic solver out performs the random solver when games are not repeated. Due to the deterministic nature of the greedy heuristic algorithm, having the same game repeated would not yield new game outcomes. Therefore, it could be argued that the heuristic solver is the least effective at determining the solvability of a solitaire game.

4.3.3 Monte Carlo solver results

The use of Monte Carlo rollouts is undoubtedly the most effective way to determine the solvability of a game of solitaire. By fine-tuning the parameters used in the solver, a solvability of $\sim$80% was found. This success does come with the caveat of a significantly reduced throughput in comparison to the other solvers, meaning fewer card configurations could be run through the Monte Carlo solver (1000 per experiment vs 10000 on the other solvers). Had the experiments been ran on a larger subset of possible card configurations, a more accurate solvability rate may have been found.

5 Conclusion

The aim of this project was to develop a Java program capable of solving a game of thoughtful Klondike Solitaire. This would then be used to explore the question: "What percentage of solitaire games are solvable?". The $52!$ combinations for a deck of cards, as well as the incredibly large number of potential move sequences in any given game of solitaire, make it impossible to provide a definitive answer. The motivation of the project stemmed from a desire to further investigate solitaire's solvability, to verify the pre-existing estimate that $\sim$82% of games are solvable, and to perhaps produce a new, more accurate estimate.

This was achieved by creating a series of solvers using the evolutionary programming development methodology. This allowed multiple prototypes to be produced at different stages of development and resulted in the program evolving from a user-playable game of solitaire to increasingly sophisticated automated solvers. These included a random move solver, a heuristic solver and a Monte Carlo rollout solver. The solvers complexities increased one step further with the addition of parallelisation. This would ultimately reduce the frequency of false negatives by allowing the same game to be ran through the solver numerous times without significantly reducing solver throughput.

From the research completed prior to the solvers creation it was clear that Monte Carlo techniques were the most effective in producing a successful solver. The results of this report further reinforce this fact, with the Monte Carlo solvers reaching significantly higher solvability rates than the random and heuristic solvers. While the Monte Carlo solvers were not able to reach the 82% estimate already discovered, they were close behind reaching 80%.

5.1 Limitations

It is important to note that although the solvability rate achieved was relatively high, the subset of possible card combinations used by the solver was small in comparison to the total number of card combinations that exist. With only 1000 card configurations being used in each Monte Carlo solver

it is possible that the results achieved are slightly inaccurate. Nevertheless, the 1000 configurations tested provide a reliable estimate for the overall solvability that is achievable by the solvers. Using a larger subset of games with each solver would likely only alter the overall solvability rate by a few percent.

It should also be mentioned that each solver was given a different subset of games to solve. It is possible that one solver was given games that were very difficult to solve (only one possible sequence of moves resulting in a win) whereas another had games that were much more straight forward. One solver may be more likely of producing false negatives than another. However, To explore every possible combination of moves in a game would make this problem intractable. With each game being simulated by the Monte Carlo solver 100 times and the entire solver being run on the same game numerous times, it is safe to assume that most possible move combinations in a game were explored.

5.2 Further Work

Further development to this project could combat some of the limitations previously discussed. Namely, a pre-defined subset of card configurations could be created to ensure each solver is accurately comparable to one and other.

Further more, future work could involve investigating and implementing various methods of improving solver throughput without compromising its accuracy. For example, caching game states that have already been tested: further computation of game states already explored is unnecessary. Also, checking the blocking states discussed in Section 2.3. This would result in a larger subset of games being solved and increase confidence in the solvability.

Finally, implementation of different heuristic strategies or rollout techniques could result in an improved solvability rate. For example Fern et al. [2] used nested rollouts and a heuristic strategy that changed depending on the stage of the game. Adopting similar methods to this project will likely further improve the probability that a solver will successfully solve a game of solitaire.

A.1 Ethical Consideration Checklist

Ethical Considerations Checklist and Declaration

Project Description

Briefly explain the methodology of your study. Give sufficient detail that a non-expert in the subject can understand what you are proposing to do.

Solitaire (or Klondike) is a single-player card game that has been popularised by its inclusion in the Windows operating system. The probability of winning any given game of Solitaire is currently unknown, though a reasonably tight bound has been discovered of ~82%. This project is focussed on confirming this discovery using HPC resources to converge on a similar bound.

External Datasets

If you are using external datasets, please (a) identify the dataset(s) giving sufficient details; (b) identify the relevant licence or terms of use for the dataset(s) and justify why your project would be compliant with those terms; (c) if the data is about humans, also provide evidence that the data was initially collected with consent.

N/A

Potential Ethical Issues

Does your project involve any of the following? Please mark Yes or No for **all** issues.

Issue	**Yes / No**
Human participants (adults or children)	No
Human data (e.g. data collected through surveys and questionnaires on issues such as lifestyle, housing and working environments, or attitudes and preferences, or datasets including human data)	No
Datasets that require permission from the data provider	No
Applications that could potentially involve unethical practice, including potential dual-use applications (e.g. projects involving tools or data that can be used for unethical purposes e.g. to attack systems)	No
Funding sources or collaboration with potential to adversely affect existing	No

relationships or bring the University or Department into disrepute (e.g. projects related to gambling, dark market, etc.)	
Restrictions on dissemination (e.g. not being allowed to publish certain datasets or results)	No
Military or defence context	No
Overseas countries under regimes with poor human rights record or identified as dangerous by the Foreign & Commonwealth Office	No
Human material (e.g. tissue or fluid samples), vertebrates, especially mammals and birds, or any other organisms not previously mentioned	No

If you answered **No** to all the above, you do not need ethical approval.

If you answered **Yes** to any of the above, you must complete a Fast-Track Ethical Approval Form, get it signed off by your project advisor, and submit it for approval to the Departmental Ethics Officers.

Student Declaration

I have considered the ethical implications of this project, all the terms and conditions and permissions of any datasets being used, and I have identified no significant ethical implications requiring an ethical approval application.

Student Name	James Smith
Student Signature	*JSmith*
Date	21/03/2025

A.2 Full Requirements Table

Requirements have been organised according to the iteration of development they belong to. If a requirement has been changed from a previous iteration, it will be included in the new iterations list. Changes to requirements from the previous iteration are shown in **bold** and requirements that have been removed are shown by ~~strike through~~ text.

Table .1: Table of requirements.

ID	Type	Requirement
		Iteration 1: Playable by player
1.00	Functional	Game should start with a standard deck of 52 playing cards (cards A - K in each suit ♡,♢,♣,♠).
1.01	Functional	Each card should have a suit ('C', 'S', 'H' or 'D') a colour (Black or red) and a rank (1-13).
1.02	Functional	Cards should be shuffled into a random order at the start of each game.
1.03	Functional	Cards should be dealt into 7 piles on the tableau, the first with 1 card, the next with 2 and so on.
1.04	Functional	The cards not dealt into the piles should be placed in the stock.
1.05	Functional	The foundation piles will be represented as integers, one for each suit.
1.06	Functional	The top cards on each pile should be set to be usable ("turned face up"). The rest of the cards should not be usable ("turned face down").
1.07	Functional	Player should be able to deal three cards from the stock to the waste.
1.08	Functional	If there are not 3 more cards to deal from the stock to the waste, the player should be able to deal the remaining cards onto the waste.
1.09	Functional	The top card in the waste should always be usable.
1.10	Functional	If the stock is empty, the cards in the waste should be placed back into the stock such that the top card in the waste becomes the bottom card in the stock.
1.11	Functional	Cards can be moved onto build stacks in the piles if the card at the top of the pile has the opposite colour and has a rank that is one higher than the card being moved.

Continued on next page

1.12	Functional	An entire build stack can be moved onto another pile as long as the bottom card on the build stack follows the rules of requirement 1.10.
1.13	Functional	A partial build stack can be moved onto another pile as long as the bottom card on the partial build stack follows the rules of requirement 1.10.
1.14	Functional	A card of a given suit can be moved onto the foundation pile of the same suit as long as the card on that foundation pile has a rank one lower than the card being moved
1.15	Functional	When a card is moved to a foundation pile, the integer value in the foundation for that suit will be incremented.
1.16	Functional	If a pile is empty, a king (or build stack with a king as the bottom card) can be moved onto an empty stack.
1.17	Functional	If all suits in the foundation have integer values of 13, the game is won.
1.18	Non-Functional	A text based user interface will be output after every move, to provide visual feedback for each move.
1.19	Non-Functional	The user interface will resemble a real solitaire layout.
1.20	Functional	When output, a card will be displayed with its rank (A-K) and its suit. E.g. "AC" or "10D".
1.21	Functional	The text based user interface will display the top playable card in the waste, the foundation values and the piles.
1.22	Functional	The top card in each pile will be displayed, with all cards behind hidden, in the following format: [-, -, AS].
1.23	Non-Functional	The game will be controlled by keyboard, the user will select a choice of possible moves from a list displayed with the UI.
1.24	Non-Functional	If the player selects a move that is not possible, the game will not crash and the user will be able to make the choice again.
		Iteration 2: Random move solitaire solver
2.01	Functional	The program will represent the stock/waste as one single element: the stock. This will be a list of cards with a pointer to the current top card.
1.07	Functional	**When a card is dealt from the stock, the pointer will be increased by 3 and the card at that index of the list returned.**

Continued on next page

1.08	Functional	If there are not 3 more cards to deal from the stock, **the deal will return the card at the end of the list, and reset the pointer to smaller index of the 3rd index in the list and the final index in the list.**
1.09	Functional	**The card at the index of the pointer in the stock should always be usable.**
1.21	Functional	The text based user interface will display the top playable card in the **stock**, the foundation values and the piles. **(use for testing only)**
1.22	Functional	The top card in each pile will be displayed, with all cards behind hidden, in the following format: [-, -, AS]. **(use for testing only)**
1.20	Functional	When output, a card will be displayed with its rank (A-K) and its suit. E.g. "AC" or "10D". **(use for testing only)**
2.02	Functional	The solver will be able to assess the current state of the game and find all cards that could be used for a move.
2.03	Functional	Using the usable cards from requirement 2.02, the solver should be able to assess the current state of the game and find all possible, legal moves.
2.04	Functional	Using the possible moves from requirement 2.03, the solver will be able to apply a randomly selected move from the possible moves.
2.05	Functional	The solver will continue to apply randomly selected moves until either the game is lost or the game is won (see requirement 1.17 in Table **??**).
2.06	Functional	The game is lost when there are no possible moves, or the same moves have been repeated multiple times.
~~1.18~~	~~Non-Functional~~	~~A text based user interface will be output after every move, to provide visual feedback for each move.~~
~~1.19~~	~~Non-Functional~~	~~The user interface will resemble a real solitaire layout.~~
~~1.23~~	~~Non-Functional~~	~~The game will be controlled by keyboard, the user will select a choice of possible moves from a list displayed with the UI.~~
~~1.24~~	~~Non-Functional~~	~~If the player selects a move that is not possible, the game will not crash and the user will be able to make the choice again.~~
		Iteration 3: Greedy heuristic solver

Continued on next page

3.01	Functional	Each move in the possible moves list created by the solver will be assigned a heuristic value.
3.02	Functional	The heuristic values will aim to prioritise moves that reveal hidden cards and move cards to the foundations.
3.03	Functional	The heuristic values will aim to prevent moves that do not progress the game forward or result in hidden cards being blocked.
3.04	Functional	If multiple moves have the same heuristic, a priority will be assigned and used to select the move.
		Iteration 4: Monte Carlo solver
4.01	Functional	A solver will be created that uses Monte Carlo rollouts to determine the best move at each point of the game.
4.02	Functional	The rollouts will simulate the rest of the game for each move. During simulations an R-greedy heuristic algorithm will be used to determine which move to take (See Section **??** for definition of this algorithm).
4.03	Functional	Each simulation will affect a 'Monte Carlo score' of the move being simulated.
4.04	Functional	The move with the best Monte Carlo score at the end of all the simulations will be selected.
		Iteration 5: Parallelised solver
5.01	Functional	The same configurations of cards should be able to be ran through solvers multiple times to reduce the chance of false negatives.
5.02	Functional	Requirement 5.01 should be able to be executed in parallel to prevent drastic reduction in throughput for the solvers.

A.3 Class Diagram

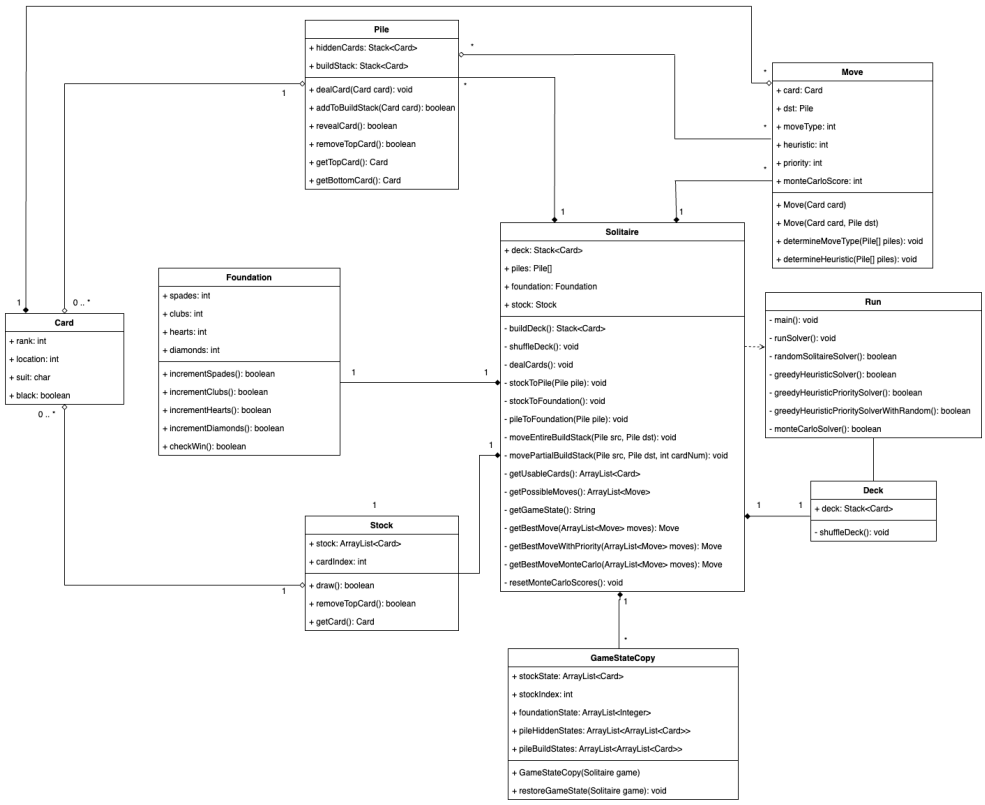

Figure .1: Full class diagram of the Java program

A.4 determineHeuristic()

```
case 0:
    for (Pile pile : piles) {
        for (Card card : pile.getHiddenCards()) {
            if (card.isBlack() != this.getCard().isBlack()) {
                if (card.getRank() == this.getCard().getRank() - 1) {
                    this.updateHeuristic(-5);
                    break;
    } } } } // Stock to foundation move may block a card of an opposite colour and rank-1 from being moved
    if (card.getLocation() != 7) {
        if ((!piles[card.getLocation()].getHiddenCards().isEmpty()) &&
                (piles[card.getLocation()].getBuildStack().size() == 1)) {
            this.updateHeuristic(10);
    } } // Pile to foundation move will reveal a hidden card
    if (card.getRank() == 1) {
        this.updateHeuristic(10);
        // Aces should always be moved to foundation
    }
    this.updateHeuristic(10);
    break;
case 1:
    this.updateHeuristic(5);
    this.setPriority(1);
    if (card.getRank() == 13) {
        for (Pile pile : piles) {
            for (Card card : pile.getHiddenCards()) {
                if (card.getRank() == 12) {
                    if (card.isBlack() != this.getCard().isBlack()) {
                        this.setPriority(-1);
    } } } } }
    break;
case 2:
    if (!piles[card.getLocation()].getHiddenCards().isEmpty()) {
        this.updateHeuristic(10);
        this.setPriority(1 + piles[card.getLocation()].getHiddenCards().size());
    } // Full build stack move will reveal a hidden card
    else {
        this.updateHeuristic(5);
        this.setPriority(1);
    } // Full build stack move will create an empty pile

    if ((card.getRank() == 13) && (dst.getPile().isEmpty())) {
        if (piles[card.getLocation()].getHiddenCards().isEmpty()) {
            this.setHeuristic(0);
```

```
44          } // Moving a king to empty pile creating another
    empty pile achieves nothing
45          else {
46              this.updateHeuristic(5);
47      } } // Moving a King to an empty pile
48      break;
49  case 3:
50      Pile srcPile = piles[card.getLocation()];
51      Card nextCard = srcPile.getCardAtIndex(srcPile.
    getCardIndex(card) - 1);
52      for (Pile pile : piles) {
53          if (!pile.getPile().isEmpty()) {
54              if (pile.getTopCard().getRank() == nextCard.
    getRank() + 1) {
55                  if (pile.getTopCard().isBlack() != nextCard.
    isBlack()) {
56                      this.updateHeuristic(5);
57                      break;
58          } } } // Partial build stack move could reveal
    another card on the next turn.
59          this.setHeuristic(0);
60      } // Partial build stack move may not reveal a hidden
    card so has no benefit
```

Listing 1: determineHeuristic() method from Move.java

A.5 R-greedyHeuristic Algorithm

```
ArrayList<String> gameStates = new ArrayList<>();
gameStates.add(game.getGameState());
ArrayList<Move> possibleMoves;

boolean end = false;
while (!end) {
    possibleMoves = game.getPossibleMoves();
    if (possibleMoves.isEmpty()) {
        return 52 - game.getHiddenCardsCount();
    }

    int randInt = (int) (Math.random() * 100);
    if ((0 <= randInt) && (randInt < RANDOMNESS_PERCENTAGE))
    {
        int randomInt = (int) (Math.random() * possibleMoves.
    size());
        game.makeMove(possibleMoves.get(randomInt));
    } else {
        Move bestMove = game.getBestMoveWithPriority(
    possibleMoves);
        game.makeMove(bestMove);
    }

    String currentState = game.getGameState();

    if (Collections.frequency(gameStates, currentState) > 3)
    {
        return 52 - game.getHiddenCardsCount();
    }
    if (game.getFoundation().checkWin()) {
        return 52;
    } else {
        gameStates.add(currentState);
    }
}
return -1;
```

Figure .2: Full R-greedyHeuristic algorithm

Bibliography

[1] X. Yan, P. Diaconis, P. Rusmevichientong and B. Roy, 'Solitaire: Man versus machine,' *Advances in Neural Information Processing Systems*, vol. 17, 2004.

[2] R. Bjarnason, P. Tadepalli and A. Fern, 'Searching solitaire in real time,' *ICGA Journal*, vol. 30, no. 3, pp. 131–142, 2007.

[3] M. Dummett, 'The history of card games,' *European Review*, vol. 1, no. 2, pp. 125–135, 1993.

[4] A. H. Morehead, *The Complete Book of Solitaire and Patience Games*. Read Books Ltd, 2015.

[5] M. Voima, 'Klondike solitaire solvability,' 2021.

[6] A. Cadogan, *Illustrated Games of Solitaire or Patience NEW REVISED EDITION INCLUDING American Games*. PHILADELPHIA DAVID McKAY COMPANY, 1914.

[7] D. Parlett. 'Historic card games by david parlett.' (), [Online]. Available: https://www.parlettgames.uk/histocs/patience.html (visited on 03/02/2025).

[8] P. Jensen. 'Celebrating 30 years of microsoft solitaire with those oh-so-familiar bouncing cards.' (2020), [Online]. Available: https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/ (visited on 03/02/2025).

[9] D. Platt. 'Don't get me started - 100 years of solitaire.' (2015), [Online]. Available: https://learn.microsoft.com/en-us/archive/msdn-magazine/2015/july/don-t-get-me-started-100-years-of-solitaire (visited on 03/02/2025).

[10] J. de Ruiter, 'Counting classes of klondike solitaire configurations,' *Bachelor thesis, LIACS, Leiden University*, 2012.

[11] M. Kortsmit, 'Strategies for klondike solitaire,' 2014.

[12] S. Raychaudhuri, 'Introduction to monte carlo simulation,' pp. 91–100, 2008.

[13] 'Frequently asked questions.' (), [Online]. Available: https://www.solitairecardgames.com/frequently-asked-questions#:~:text=A%3A%20If%20we%20calculate%20how,50%20percent%20of%20these%20games. (visited on 13/02/2025).

[14] G. Tesauro and G. Galperin, 'On-line policy improvement using monte-carlo search,' *Advances in neural information processing systems*, vol. 9, 1996.

[15] C. Atkinson and O. Hummel, 'Iterative and incremental development of component-based software architectures,' in *Proceedings of the 15th ACM SIGSOFT symposium on component based software engineering*, 2012, pp. 77–82.

[16] M. L. Despa, 'Comparative study on software development methodologies.,' *Database systems journal*, vol. 5, no. 3, 2014.

[17] M. Carr and J. Verner, 'Prototyping and software development approaches,' *Department of Information Systems, City University of Hong Kong, Hong Kong*, pp. 319–338, 1997.

[18] Java™. 'Package java.util.' (2025), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html (visited on 20/04/2025).

[19] oracle.com. 'Package java.util.concurrent.' (2025), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html (visited on 22/04/2025).

[20] R. del Nero. 'Shallow and deep copy: Two ways to copy objects in java.' (2024), [Online]. Available: https://www.infoworld.com/article/2173838/how-to-copy-objects-in-java-shallow-copy-and-deep-copy.html (visited on 22/04/2025).

[21] I. Jovanović, 'Software testing methods and techniques,' *The IPSI BgD transactions on internet research*, vol. 30, 2006.

[22] york.ac.uk. 'Viking.' (2024), [Online]. Available: https://www.york.ac.uk/it-services/tools/viking/ (visited on 22/04/2025).