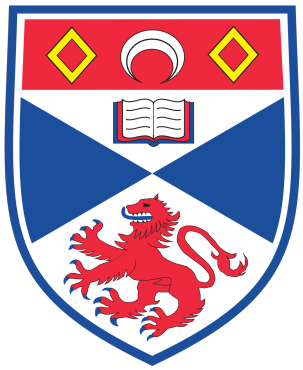
Solvitaire: A Solver for Perfect Information Solitaire Games



Charlie Blake

School of Computer Science

University of St Andrews

Senior Honours Project

9 April 2018

Abstract

This project implements a solver, named *Solvitaire*, for perfect-information solitaire card games. Whereas most solitaire solvers are specific to particular games, this solver is able to solve deals for a wide range of games with different features. The performance of *Solvitaire* is demonstrated to be as strong as other dedicated solvers. I also present a simple JSON format which can be used to describe the rules of a game to the solver. Through its ability to interpret the rules of many different games, and through its strong performance, *Solvitaire* is also able to generate solvability percentages for games. I present solvability percentages for eighteen games where until now the solvability percentage has not been known, which represents a significant addition to the existing body of known solvability percentages. In addition, *Solvitaire* makes it easy to calculate these percentages for many more games.

I demonstrate how various AI techniques can be used to reduce the search required for solving solitaire deals. Primarily this focuses on symmetry between suits, and between piles in the game. I also show how particular moves can be classed as dominances to make search more efficient. Through an evaluation of the techniques implemented here, I show which are the most useful in different circumstances. I also implement two search streamliners, which enable large performance increases for search, but at a small cost in terms of accuracy. Finally, I provide an example of how *Solvitaire* can be used to analyse the solvability percentages for variations of existing games.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 19,877 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

Contents

1 Introduction

1.1 Overview

Some of the most high-profile successes in the field of AI have been the creation of good agents to play strategy games. *Deep Blue*, *Chinook*, and *AlphaGo*—programs to play chess, checkers and Go respectively—have all managed to reach a standard of play exceeding that of the best human players[7][33][34]. These achievements have been widely reported[21][29], and were the culmination of some of the most advanced techniques in artificial intelligence at the time.

Game playing is worth studying from the perspective of AI research for several reasons. Firstly, across various popular games there exists a large number of dedicated players exist who are highly interested in playing strategies. Secondly, games represent a valuable domain for testing AI algorithms that can then be put to work in real-world scenarios. For instance, the technology behind IBM's *Watson* system, which has been able to beat expert humans at the quiz show Jeopardy[36], has now been re-purposed to create clinical decision support systems for medical professionals[14]. Games often capture the essence of real-world challenges in a simplified context, where it is easier for AI practitioners to address the fundamental problems that need solving.

Chess, checkers and Go all have one particular feature in common: players are aware of the entire state of the game at all times. Contrast this with games such as poker, where players do not necessarily know their opponents' hands, and hence some game state is hidden from them. Games where the entire state is open are known as *perfect-information* games, and it is no coincidence that many of the high-profile achievements in AI have been in creating agents for these kinds of games. One of the advantages of perfect information, which makes these games particularly good for experimenting with AI techniques, is that there is no uncertainty introduced by unknown information. In imperfect-information games, a good agent must reason probabilistically about unknown state, but in perfect information games this layer of uncertainty is removed, which restricts the focus of the agent to just the strategic problem of the game itself.

The quality of programs that exist for playing certain perfect-information games is evidence of the depth of interest in this area. However, much of this focus has been specifically on playing *two-player* perfect-information games, where agents are

required to reason about and anticipate the actions of an opponent. Relatively little attention has been given to *single-player* perfect-information games, but these too are of considerable interest. For one, they are highly popular. Single-player perfect-information games—often referred to as just 'puzzles'—include problems such as Sudokus, Rubik's Cubes, solitaire games, and many more. These games are common pastimes, and are very widely known. For instance, the most popular solitaire app on Google's Play Store has over ten million downloads[24]. Although the approaches required to solve single-player perfect-information games tend to be quite different to their multiplayer equivalents, from an AI perspective these problems are still an interesting challenge. The techniques used to solve them have many applications, and the game FreeCell has been used by AI planning programs as a benchmark domain[22].

Yet despite this, when it comes to solitaire—a very popular family of single-player card games—comparatively little work has been done to create good solvers, and good solvers do not yet exist for most solitaire games. This is the focus of my project: to create a program that can not only solve solitaire games efficiently, but can also do so for a *wide* variety of games within the solitaire family. Doing so would provide this popular game with a high-quality yet general solver, and in turn give insight into effective approaches for creating solvers for single-player perfect-information games.

1.2 About Solitaire

Note: for a thorough overview of the solitaire-related terminology used in this report, see Appendix A.

Solitaire (also known as *patience* in the UK) is not a single game, but the term for a genre of card games for one player. There is no single authoritative source for solitaire games, and many have been around for hundreds of years. In the late eighteenth century the first solitaire games were recorded in the German game anthology *Das neue Knigliche L'Hombre-Spiel*[9], and a few decades later during his exile in St Helena, Napoleon Bonaparte was also recorded playing solitaire to keep himself entertained[28].

The most popular and influential source for solitaire games was written in 1870 by Lady Adelaide Cadogan in her *Illustrated Games of Patience*[6]; a book which ran for many editions and is the origin, directly or indirectly, of many of the games explored in this report. Much of the modern popularity of solitaire can be attributed to its inclusion in many of the Windows operating systems[25], where games like FreeCell,

Klondike and Spider were included, often as some of the only games available to users by default.

Solitaire games can differ quite significantly, but there are a few common features which occur in most, if not all of them. Initially, solitaire games begin with a *starting deal*, which is a (typically random) layout of some or all of the cards from a standard shuffled deck (or multiple decks). This layout often organises the cards into stacked piles, and in most games some or all of the initial piles are designated *tableau piles*, which form the main 'playing area' for the game. The objective is then to move from this starting state to some *winning* state, through a series of moves. A move typically consists of taking one or more cards from one pile and placing them upon another, according to some *building policy*. The building policy is a constraint based on card rank or suit, such as 'cards may only be moved onto other cards with the same suit, and a rank one greater', or other similar rules.

The winning objective typically involves moving cards from the tableau piles to another set of four piles known as *foundations*. These piles are each designated a specific suit, and are built up from ace to king within that suit. Removing cards from the foundations is legal in some games and not in others. In addition, many games that do not involve dealing out all of the deck for the starting deal, place the rest of the cards in a pile known as the stock. Cards from the stock can in some games be dealt directly to the tableau piles, and in others to the tableau piles through an intermediate *waste* pile.

There are many more features that occur in solitaire games, and slight variations on existing rules, more of which will be explored later in this report. A huge number of rule variations are possible for solitaire games, and some of them eschew the norms outlined above quite significantly. It is fascinating to see how small changes in the rules for a solitaire can greatly affect both the playing experience and the difficulty in reaching the winning state. For some starting deals of particular games, reaching the winning state is in fact impossible; for many game types it is trivial to construct an starting deal in which no initial moves can be made. However it is seldom obvious to a human player that from a starting deal, reaching the winning state is impossible.

1.3 The Problem to be Solved

The challenge for a good solver therefore, is not only to find solutions for starting deals when they exist, but also to *prove* for unsolvable deals that no solutions exist. Solving solitaire deals thus reduces to a search problem: how can we best traverse all the

reachable states, such that a deal can be categorised as either *solvable* or *unsolvable*. In the case where the search space is too large for us to explore sufficiently in a given time period, we will also categorise deals as part of a third, *intractable* class.

Some games naturally have larger search spaces than others, and more potential paths to solutions. Given a starting deal, a player can expect to be able to reach the winning state for certain types of solitaires much more often than others, sometimes because the deals a player cannot solve actually have no solutions. This leads us to a notion that is central for my application: the idea that each game has its own natural *solvability* percentage. That is to say, that for a given solitaire type, there is a fixed probability that a random starting deal will be solvable. It's worth noting that games with a higher solvability percentage are not necessarily always considered easier than those with lower percentages, as games where there are almost always solutions may require a great deal of effort and ingenuity to find 'good' moves that lead to solutions.

At this stage, one might reasonably ask whether my assertion that exhaustive search is the only approach here, is actually correct. Might there not be some kind of short-cut approach for certain games, which allows us to determine whether deals are solvable without having to traverse the entire search space? Unfortunately, this does not seem likely. Whilst for some games, bespoke instances can be created where we can guarantee that exhaustive searches will invariably 'get stuck' at certain points, in general we know that for several games, solving starting deals is an NP-complete problem (see *Context Survey*, chapter 2 for more details). This indicates that we cannot expect there to be trivial ways of determining the solvability of deals for these games—and therefore their solvability percentages—without having to search through a large number of states.

Because in this project we wish for the solver to be able to prove the unsolvability of deals, the *exhaustive* nature of our search is important. The challenge here, is to find ways in which we can reduce the search space that we must traverse, whilst still being able to claim that if we find no solutions, there must be none. To accomplish this, I make use of the notions of symmetry and dominance used in constraint solving, to eliminate certain states from the search. Additionally, because we are interested only in perfect-information games, for those solitaire games where typically some cards are face-down and hidden from the player, I turn them face up to give our agent full information about the state of the game.

1.4 Objectives

Having outlined the basic features of the problem to be solved, some key objectives for the project can now be laid out. The software artefact to be created in the project is a solitaire solving program, which I have named *Solvitaire*. The precise implementation requirements of this application can be found in section 3, but here will outline the key high-level project aims, in order of importance:

1. To be able to perform a basic search from a starting deal of a simple solitaire game, and (assuming the problem isn't intractable) accurately report if it is solvable or not.

2. To be able to read in some kind of 'game description' that outlines the rules of a particular solitaire game, and search based on those rules.

3. To be able to do this for a *wide* range of games.

4. To implement AI techniques that can be used to reduce the number of states that must be searched in order to solve a game or prove that it is unsolvable.

5. To get good enough performance from my solver that a low percentage of deals are intractable for a range of games.

6. To extend the ability of my solver so that it can calculate the solvability percentage for a game based on its game description. This would be done by attempting to solve a large number of random deals, and then estimating a confidence interval for the overall solvability.

I was able to accomplish all of these goals in the application I have produced; how I approached these objectives specifically will be explored in depth later in this report. Furthermore, I am pleased to report that, using my solver's ability to calculate solvability percentages from simple game descriptions, I have been able to produce a table of solvability percentages for many games where until this point the solvability percentage was unknown (see section 8.1, figure 8.1). For those games where these percentages have been found previously, my solvability results match those found in published papers, which indicates that *Solvitaire*'s results are indeed accurate.

2 Context Survey

Little research has been done on solitaire games in general. That which has been done has tended to focus on a handful of games, and in quite specific areas. Namely, the computational complexity of solving particular games, the solvability percentages of some of those games, search techniques that can be used when it comes to solving deals, and heuristics that can be used to find solutions quickly. We will assess the research done in each of these areas in order:

2.1 Complexity of Solitaire Games

The games that have been focused on most heavily are FreeCell and Klondike (often known, confusingly, as 'Solitaire'), and to a lesser extent the games Spider Solitaire, Black Hole and Canfield. In my introduction (section 1.3), it was stated that there were several solitaire games that are known to be NP-complete. The first proof of the NP-completeness of a solitaire game was for FreeCell[22], in a 2003 study of the complexity of standard benchmark domains for AI planning (of which FreeCell is one). Since then, NP-completeness has been proved for Black Hole[18], Klondike[27], and most recently Spider[35]. The proofs for Klondike and Spider build upon the technique used previously for Black Hole, which encodes the game as an instance of the NP-complete SAT problem. These games are only a tiny fraction of all of the possible solitaire games, but each are quite different to each other, and it would be unsurprising if many other games could also be used to encode instances of the SAT problem. Although this is hardly a proof that all of the games that can be solved by my application are NP-complete, it is reason enough to suggest most of them don't have short-cut methods of finding solutions, and hence my general approach to solving games is the correct one.

2.2 Solvability Percentages

The most advanced objective of this project (section 1.4) was the potential for my application to generate solvability percentages for supplied game types. The inspiration for this idea stems from the work done over many years by the FreeCell community in exploring solutions for various starting deals.

When Microsoft released one of the early versions of FreeCell for its Windows 3.1 operating system, it included 32,000 numbered deals which have since become canonical in the FreeCell community. These deals were played and studied widely and solutions to each one carefully collected. It seemed for a time as though all of these deals were eventually solvable, and by extension that the game was (excepting bespoke deals designed to be obviously unsolvable) to all intents and purposes always solvable. The only problem with this theory was one elusive deal numbered 11982, which no-one could find a solution for. Eventually when the first computerised solvers were produced it became apparent that in fact this deal was unsolvable, despite it looking ostensibly like any other deal.[25]

When more powerful solvers were produced several years later, they were able to attempt many more deals, and out of 100 million random deals it transpired that only 1282 were solvable, giving a solvability percentage of 99.999%. This incredibly high percentage (without actually being 100%) was purely by chance rather than design, and it remains a remarkable fact about the game. A more recent solver[16] managed to prove the solvability percentages for FreeCell with varying numbers of free cells and columns, which can be seen in figure 2.1.

Figure 2.1: Solvability Percentages for FreeCell[16]

Number of Columns	Number of Freecells														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4				0	*	2	24	123	376	706	920	988	999	@	!
5	0	0	*	2	40	248	650	926	994	@					
6	0	*	9	158	615	941	998	@							
7	*	9	244	790	989	!									
8	2	195	795	994	@	@@									
9	59	656	986	@	!										
10	304	931	@	!											
11	635	993	@	!											
12	852	@	!												
13	948	@	!												

White boxes with zeros indicate variants where no winnable random deals are known. Red boxes with asterisks have win rates less than 1 in 1,000. Blue boxes with At Signs have win rates greater than 99.95\%. The lavender box with a double At Sign has a win rate greater than 99.999999\% (there is one known random 8x5 deal which is impossible). Violet boxes with exclamation points indicate that no impossibles have been found.

Knowing that FreeCell is almost always solvable is of great interest to players, because it provides an assurance that their perseverance with a deal will essentially always pay off if enough time is spent. They are not straining themselves to find a solution to an impossible puzzle. For other games which have much lower solvability percentages, this information is interesting for a slightly different reason. For these games players will inevitably give up on a lot of deals, and solve the ones that they deem to be possible. After a while a player will build up a 'win rate', which reflects how often they are able to solve games. If the solvability percentage of the game is also known, this gives players something to compare their win rate too, and see how close they are to being able to win all solvable instances; if their rate differs greatly from the solvability percentage, it indicates that they are giving up on a lot of games that they could have found a solution for.

Because of the interest in this property, solvers have been written for several other games to find out the solvability percentages. It is known that for the game Canfield, the solvability percentage is between 71% and 72%[38], for Black Hole it is between

86% and 89%[18], and for Baker's Game it is 74.3%[15]. Clearly this is something players of solitaire games would like to know, but the challenge here is that writing dedicated solvers for each game is no straightforward task, and techniques that may help search in certain games might not be applicable in others.

2.3 Existing Solvers

Most of the existing solvers written for Solitaire games have been for FreeCell, based on the fact that it has the most active community, and also perhaps because the game became popular through a computerised version. Other solvers have been written, such as those used to generate the solvability percentages mentioned previously, and I've also come across ones for Klondike[10] and the game Seahaven[37]. Many others no doubt exist.

An early influential solver was written for FreeCell in 1994 by Don Woods[25], who was the first to analyse random solitaire deals in bulk. As mentioned previously, a more recent solver by Shlomi Fish[16] was able to solve a number of FreeCell variants, and this solver is probably the most comprehensive solver for FreeCell (and FreeCell variants) to date, which includes a wide range of solving options. A page on his website[17] also includes a much more detailed list of the history of FreeCell solving applications than has been given here.

Generally speaking however, most of the solvers that have been produced have been for games (such as FreeCell) where all or almost all instances can be expected to be eventually solvable. Hence for these games a significant focus of the work done has been to create good heuristics to find solutions quicker. But for my more general solver, which is designed to be run on games where this is not the case (and hence have a solvability percentage that is not 100%!), the concern is not so much finding solutions quickly, as to be able to prove no solutions exist quickly. This means that such heuristics are of little use. In fact, some heuristic measures that can improve times to find winning solutions would greatly slow down exhaustive search. This is one of the key factors that makes my solver different from many other available solvers.

With this in mind, some of the work done in the final area of solitaire research not yet explored here, search techniques, becomes of much less relevance to my project. This is because the focus of many of these papers relates to heuristic search techniques, which are often not of great use to me. Thus some of the research in this area I shall only mention in passing.

2.4 Search Techniques

Paul and Helmert[30] attempt to use search techniques to find minimal solutions (i.e. the fewest moves to reach a winning state) for FreeCell deals. They do this through the use of A* search. The authors state that this is the first technique at time of writing which is able to return minimal solutions for FreeCell. Their choice of A* heuristic is particularly interesting, making use of an estimate of the minimum number of moves to temporary locations that must be made to release 'blocked' cards. I list the ability to find shortened solutions as an tertiary requirement in my requirements specification 3, and my implementation can attempt this too, although I took a slightly different approach. It would be interesting to explore this technique further though.

Another interesting technique is found in Helmstetter and Cazenave's[23] paper on search in the solitaire game Gaps, where their approach relies on finding independent 'blocks' of moves in a search and treating them as an individual move. This approach was again of some interest to me, and I solve the same problem in my implementation, albeit in a different way. The authors also rely on a sampling search algorithm here that does not explore the whole search space, which was less ideal for my purposes.

Moving on to some of the less applicable search techniques, some quite successful research in the domain of solitaire has come from the application of genetic algorithms to create good heuristic-base agents, which can reach winning states with minimal backtracking. Researchers have used this technique to create agents that are able so solve almost all instances in the original Microsoft 32,000 FreeCell deals within a reasonable time limit[12][13]. Other approaches have involved using a rollout search method, which adapts the heuristic policy being used dynamically as the search proceeds[4][39].

All of the solitaire games considered thus far have worked on 'open' versions of games which assume all cards are face up and all games are perfect-information. Bjarnson, Fern and Tadepalli[3] take the original, closed version of Klondike and form a strong policy for winning games without backtracking, by using Monte-Carlo Planning. Dunphy and Heywood attempt to use neural networks to create a heuristic for search in the game FreeCell, although with mixed results[11]. Finally, Chan[8] outlines a number of different AI techniques that could all be applied to FreeCell, in the hope that other researchers might take up the challenge of exploring some of these avenues in more detail.

3 Requirements Specification

The requirements set out here are based on the original requirements produced in my DOER. At the halfway stage of my project, I adjusted, with the input of my supervisor, those original objectives. This was based on the conclusions drawn from the research and implementation done by that stage, by which point it was clear that some of the original objectives were either more or less relevant than originally anticipated.

Please refer to the DOER document for the original objectives. Most of my adjusted objectives still remain the same, but there are a few alterations. The final requirements for the project are as follows:

3.1 Primary Requirements

- An application for solving single-player, perfect information solitaire instances (software artefact).
- Some kind of schema or language for describing a wide range of solitaire games.
- The application must be able to solve valid games expressed using the schema/language.
- Automated testing to demonstrate the correctness of my classifications.
- An implementation of search techniques such as 'meta-moves'/'super moves' and eliminating symmetrical states.
- An implementation of language-level/design optimisations to improve the efficiency of my search.
- An analysis in my report of the effects of the above search techniques and implementation choices on the performance of the solver.

3.2 Secondary Requirements

- Optimise the search algorithm to eliminate searching states in which certain moves/states 'dominate' others.

- Implement and evaluate more than one different type of search (e.g. best first, iterative deepening).

3.3 Tertiary Requirements

- An extension of the algorithm that aims to provide shortened solutions (this may be done in tandem with the 'different type of search' objective).
- Calculating the solvability ratio for some the games my application can solve.

4 Software Engineering Processes

4.1 Software Development Methodology

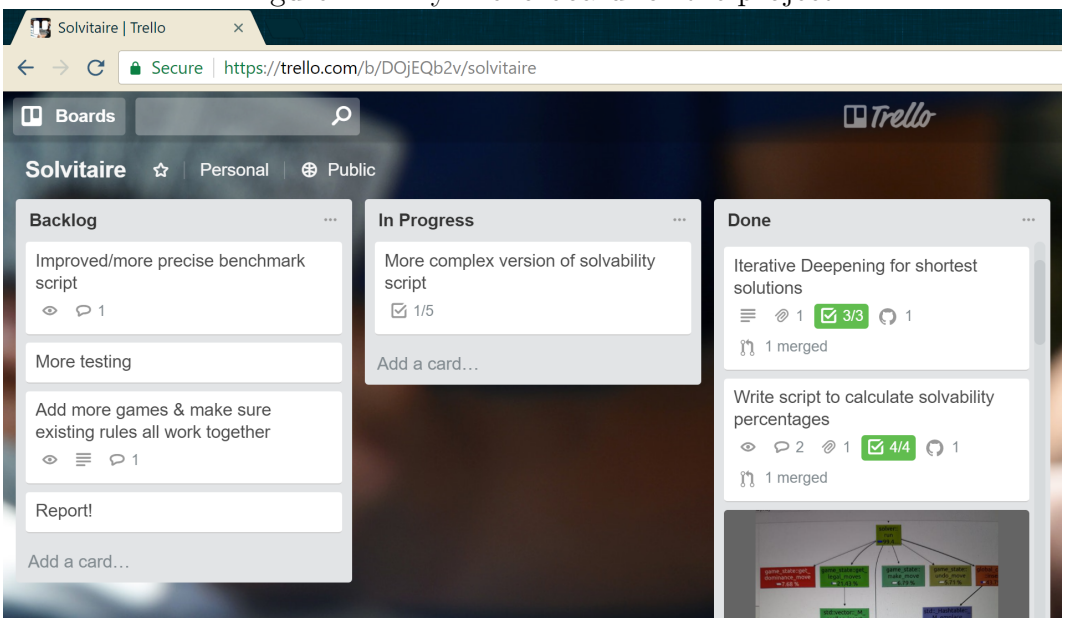
Working on a fairly substantial software project such as this one, I reasoned that although I would not be developing my solution as part of a team, it was still important to have some kind of structured development methodology to keep my work focused, on schedule, and well-organised. With this in mind, I tried to adhere as closely as possible to the *Agile*[2] software development methodology.

This approach is based around breaking objectives down into small tasks that are flexible to change if the requirements of the project alter. Although the main objective of the program was unchanged throughout, because I approached the project with relatively little initial understanding of this area, it was inevitable that some of my assumptions about what was required would need revising. Hence, rather than coming up with a rigid plan starting out, this approach gave me the flexibility to change the direction or emphasis of tasks, if and when this became necessary.

To implement this approach, I decided to try and stick to a version of the Kanban development framework. This approach emphasises the use of visualising the workload for a project, and organising the 'flow' of tasks from conception to completion. For my project I used the website Trello to implement my Kanban board, categorising tasks within ordered *Backlog*, *In Progress* and *Done* columns (see figure 4.1).

The intended timeframe for each task was set at roughly a week's work for each, and every task corresponds to a branch in my version control system (for which I used Git). To give me an extra layer of oversight into my own development workflow, I also connected my GitHub repository to my Trello board such that each pull request merging a development branch into the main branch, is linked to its corresponding task on the Kanban board. Thus, at a click I can see exactly which changes were made for each task in my development cycle. Finally, because the granularity of each task was still a little large, I made use of the checklists and comments within a task provided by Trello, so that I could plan out the exact work that needed to be done at each stage. More often than not, items on the task checklists correspond directly to commits into my version control.

Figure 4.1: My Trello board for the project



The link for my Trello board is: https://trello.com/b/DOjEQb2v/solvitaire. (*Note: I have made this board publicly available for the sake of ease. My GitHub repository, as is required for all undergraduate coursework, is still private, and so the links between tasks and their git pull requests mentioned previously will not be visible to the examiner*)

Another important part of Agile software development relates to automated testing. It is important to ensure that every task that is completed is not only *demonstrably* working, but that it also hasn't broken existing functionality in the project. For this reason, I made sure that where applicable, I wrote the relevant tests for each task to make sure that my implementation was correct.

My weekly meetings with my supervisor were another important part of my development process. We tried to schedule these meetings regularly each week, and during our meetings we would review the progress made last week, discuss areas of interest emerging from the work done, and then set objectives for the upcoming week. This was a useful part of my development cycle, as it gave me a regular opportunity to re-evaluate my priorities and set the short-term objectives for myself necessary to keep me on track. The advice, insight and direction gained during these meetings was often invaluable, and led me down many rewarding avenues that otherwise would have been closed off to me.

I tried each week to do a write-up of the discussion had during our meeting, which I kept a record of and can be found in the `docs/supervisor-meetings` directory in my submission. These write-ups aimed to record firstly the work done since the last meeting, then the main discussion points of the meeting, and finally the objectives set out for the next meeting, to be done in the coming week. An example of one of these write-ups can be seen in figure 4.2.

Figure 4.2: Weekly meeting write-up

Supervisor Meeting: 2017-11-07

Done since last meeting

Add States Searched

When my solver prints out a solution, it now also outputs the number of states searched.

Add Free Cell

I have added the game Free Cell to the solver, which involved implementing the "cell" feature.

Almost Completed Canfield

I have almost finished adding the game Canfield to the solver, which involves implementing the "stock/waste" feature, and the "reserve" feature.

Discussion Points

We spoke about the usefulness of having some kind of script to evaluate at what "level" my solver can currently solve a range of basic game types. To do this it may be necessary to implement the feature whereby game rules are interpreted according to a user-supplied JSON schema.

We also discussed the fact that it might be worth quickly implementing some kind of hash table to store states seen so far, just because the current naive implementation has performance so bad it makes the solver painfully slow.

It was also discussed that in the longer term, it would be worth thinking about some heuristics that might help in getting to correct solutions quicker.

Future Work

- Finish implementing Canfield
- Add another two games: one to specifically target the stock, and one for the reserve
- Add some basic unit testing for each "core" game type
- Clean up and refactor the code a little
- Add reading in the game rules from JSON
- Create a performance benchmark script

4.2 Development Tools

The following is a brief overview of the tools I used to create my solver and a short justification as to why each was necessary:

Programming Language: C++

There were quite a few good options here, but in the end I opted to use C++ for my project. My reasoning here was firstly based around the fact that well-optimised C++ can be made to run very fast indeed. Secondly, the C++ ecosystem contains some very good tools and libraries. The profiling tools I used were extremely helpful in making my implementation more efficient, and the peer-reviewed, open-source Boost[5] libraries which I made use of are built to be both reliable and highly efficient. Finally, C++ provides facilities for low-level memory manipulation, which I felt might come in use when implementing my solver, and aren't necessarily available in other languages.

Build System: CMake

Strictly speaking, CMake isn't actually a build system but rather a generator of build systems. This enables it to be more flexible than, say, using regular Makefiles, and enables me to easily make multiple builds for different purposes (which I do!).

Automated Testing: GTest

I found that unless some more advanced testing features are required (which isn't the case here), there is little to choose between testing frameworks for C++. In the end, I chose the GTest[19] because of its clear documentation and simple usage.

Performance Profiling: Gperftools

Profiling my code was a very important step in developing my project. It let me to identify implementation decisions that were slowing down my code unnecessarily. My choice here was between Gperftools[20] and the Valgrind framework. The latter is a well established tool, but adds a quite substantial performance overhead, whereas the former (which I chose) is a little less extensive but a lot more lightweight. I also connect the output of the profiler to the Kcachegrind profile visualiser, which produces clear and informative visualisations of profiling results. Examples of this can be seen as images for tasks in my Trello board.

IDE: CLion

Version Control: GitHub

I chose git for my version control system, hosted on the service GitHub. This was primarily based on my own familiarity with GitHub, and also the integration it has with Trello.

5 Ethics

There were no ethical considerations that needed to be taken into account for this project to be completed.

6 Design

In this section we will explore the high-level design of *Solvitaire*, with a particular focus on the novel features in my project and how I approach them. This begins with an overview in section 6.1 of the outward features of the solver, including how *Solvitaire* handles game rules and starting deals. Then in section 6.2 we shall examine how the solver is structured internally and the design of my search algorithm. Section 6.3 gives an analysis of the different techniques I use to reduce the size of the space the solver needs to search. And finally section 6.4 contains an explanation of the rigorous (and possibly novel) approach I take to calculating and outputting solvability percentages.

6.1 Solver Features

The following section explores the key outward-facing features of my application. To understand how my solver is used and the command-line options that can be supplied to it, see the User Manual in Appendix C.

6.1.1 Describing Game Rules

One of the core objectives for this project was to make my solver applicable to a *wide* range of games. It made sense therefore to have a general method of allowing the user to specify the rules of a game for my program. This would ideally take the form of some kind of language or structured representation of the rules which could be easily parsed by *Solvitaire*. This representation should be easy to modify, to read, and be flexible enough to describe many different game rules succinctly.

My solution to this challenge was to use the JSON representation format to define a schema that allows users to express the rules of a particular solitaire game. This format lets one easily specify things like the number of tableau piles, whether a game has a stock/waste, which tableau moves are legal, and so on. The schema I created checks against the JSON created by a user to see whether it is consistent. Example rule descriptions for the games Black Hole and FreeCell can be seen in listings 6.1 and 6.2. One can see the focus here on a clear and simple representation.

One will notice however that these JSON examples don't seem to be comprehensive; there is much that they miss out. For instance, FreeCell doesn't provide a

```
{
  "tableau piles": {
    "count": 17,
    "build policy": "no-build"
  },
  "hole": true,
  "foundations": false
}
```

Listing 6.1: JSON describing Black Hole rules

```
{
  "tableau piles": {
    "build policy": "red-black"
  },
  "cells": 4
}
```

Listing 6.2: JSON describing FreeCell rules

tableau pile count, and Black Hole doesn't specify that is desires no cells. They are able to do so, because I also define a 'default' rules JSON file (seen in listing 6.3), which is comprehensive and covers every possible JSON feature, setting out the states of each if the user doesn't specify otherwise. This is what fills in the 'missing' values in listings 6.1 and 6.2.

I tried to set these defaults to sensible values which reflect standard assumptions about the rules of a game. For instance by default 'built group' moves (see Appendix A) are not allowed, unless the user enables them for the specified game, based on the fact that this type of move is disallowed in more game types than it is allowed in. Not having to specify the rules for a game unless they deviate from the expected norm, lets my JSON rules files focus specifically on the combination of features which makes games unique.

My schema is not actually quite enough by itself to fully define what is and isn't allowed in my JSON rules format, because there are some subtle and complex combinations of rules which do not make sense together. For example, what happens if a user tries to specify a hole pile and foundations? Or when the user forbids building on tableau piles, but specifies that the waste pile deals directly to the tableau piles? Trying to encode checking for these problematic combinations of rules in a JSON schema proved unwieldy and excessively complex, and led to incomprehensible error messages for the user. Hence in my code I have an extra layer of rule-checking to ensure that some of these unusual combinations are forbidden from being supplied to

```
{
  "tableau piles": {
    "count": 8,
    "build policy": "any-suit",
    "spaces policy": "any",
    "diagonal deal": false,
    "move built group": false,
    "move built group policy": "same-as-build"
  },
  "max rank": 13,
  "two decks": false,
  "hole": false,
  "foundations": true,
  "foundations initial card": false,
  "foundations removable": false,
  "foundations complete piles": false,
  "cells": 0,
  "stock size": 0,
  "stock deal type": "waste",
  "stock deal count": 1,
  "stock redeal": false,
  "reserve size": 0,
  "reserve stacked": false
}
```

Listing 6.3: JSON describing the 'default' rules for games

the solver. The user manual describes exactly what is or isn't allowed in the supplied JSON.

Despite the usefulness of this format, users might not wish to have to manually input JSON description files for the most common games. Thus it made sense (and helped my testing) to embed a list of preset JSON game descriptions into my code. And so, when a user wants to solve instances of one of these common games, now all they have to do is supply the name of the game to my program on the command line and *Solvitaire* will generate the rules of the game from its own internal table.

6.1.2 Describing Game Deals

The specification of the game type isn't the only information a user need supply to the solver. They also need to tell it which deal(s) they wish for it to solve. *Solvitaire* provides two options here: firstly, a user can ask it to generate a random deal based on a seed and attempt to solve it, or alternatively, they may provide one or more deals to the solver themselves and ask it to solve them.

The random deal generation takes a deck of cards and shuffles (randomly orders) them, dealing them out to the game piles as necessary for the specified game type.

```
{
  "tableau piles": [
    ["AH","3D","KD","JC","6C","JD","KC"],
    ["AS","3H","6H","5D","2C","7D","8D"],
    ["4H","QS","5S","5C","10H","8H","2S"],
    ["AC","QC","4D","8C","QH","9C","3S"],
    ["2D","8S","9H","9D","6D","2H"],
    ["6S","7H","JH","10D","10C","QD"],
    ["10S","AD","9S","KH","4S","4C"],
    ["JS","KS","3C","7C","7S","5H"]
  ]
}
```

Listing 6.4: JSON for impossible FreeCell deal 11982

The random seed supplied by the user is used when shuffling the deck, meaning that each seed has a completely different corresponding deal.

The alternative, whereby the user supplies their own deal, is intended for scenarios where a user is stuck on a difficult deal for a particular game type and wishes to find a solution, or verify that there is none. The design for this feature is very similar to the one used for describing the game rules: the user supplies a JSON file to *Solvitaire*, specifying which cards are currently on which pile. The program will then run the search based on the user's supplied deal. They may also supply multiple deals to be solved should they wish.

Again I define a schema for this simple JSON representation. The user manual specifies what is permitted in this format, although it is fairly intuitive. Listing 6.4 shows the example input for the impossible FreeCell deal 11982 (see section 2.2 for more details). Enabling the solver to be run on user supplied instances is also extremely useful for testing purposes. It enables me to create bespoke solvable and unsolvable deals, to verify whether my solver does indeed report the correct solvability.

6.2 The Search Algorithm

The core part of my solver is the algorithm that explores potential moves. Once the game rules have been read in and the random/supplied deal has been generated, the next stage is to take the starting deal and attempt to traverse all of the possible states available by making all legal moves in each state. This is a search problem, and thus needs an appropriate search algorithm. For this, I decided to model my solution around a depth-first search.

I chose depth-first search (DFS) versus breadth-first search (BFS), because its space complexity is better, and it seemed at the initial stage as though memory might be an important factor given the aim here is to explore all (non-symmetrical or dominated) states in the search space. I began the project by implementing a simple, naive DFS for the game Black Hole. I built up the JSON game rules that *Solvitaire* can interpret incrementally with each preset game added, so at this stage it could only be used for Black Hole.

Black Hole involves moving cards from the tableau piles to a 'hole' pile, but only if they are of adjacent rank to the card already on top of the hole. One is also forbidden from moving cards between tableau piles. This gives the game a particularly useful property for depth-first search: it has a fixed-length solution. Thus the search cannot enter into loops by moving back to a state it has seen before. This meant my initial DFS didn't have to worry about loop checking, and I could just implement a standard non-recursive DFS algorithm[32].

The performance of this naive solver was reasonable (although it would not be competitive with my current solution) and solved a simplified version of the game using half a full deck within a short time period. This indicated that my general approach was probably the correct one, but this method could not be expanded to most solitaire games due to the loop-checking problem. To solve this, I first had to implement a game (and its corresponding rules) which didn't have a fixed-length solution, for which I chose Spanish Patience. This game is about as close as one can get to a 'default' solitaire game: it simply consists of tableau piles (which one can now move cards between), and the four standard foundation piles. Nevertheless, the ability to move back-and-forth between piles introduces loops, and hence needs more than just my basic DFS algorithm.

My first solution to this problem was again to implement a naive algorithm. This involved at each node in the search tree, traversing its parents right back to the root node and checking for equality with the current state to find loops. This approach, as suspected, worked poorly. For Spanish Patience, when I modified the rules to have a deck size of anything greater than 16 cards (i.e. with cards only going up to rank 4) it found solving deals intractable. This is because traversing the depth of the tree at every step of the search, gives complexity d^2, where d is the depth (or $log^2(n)$ where n is the total number of nodes in the search tree).

So any approach based on this kind of search was out of the question. The obvious alternative was to implement my loop-checking by using some sort of state cache. My initial design for this was a 'local' cache, which keeps track of just the parent states

of the current node, going back to the start state. This eliminates loops and does so in constant time. Given that I was going to implement the local cache, I reasoned at this stage that it would also be easy to also implement some kind of 'global' cache that keeps track of all of the states seen in the search thus far across all branches. In fact, it would be even easier to implement this cache, as the solver would never have to remove states from the cache. The downside to using the global cache was that its space usage would be linear in the size of the search space. This was one of the reasons I initially avoided breadth-first search, so this seemed at the time like an obvious problem. But the advantage of the global cache was that there was a good chance that different branches in my search would come across the same states, and so having this would get rid of potentially a huge amount of unneeded search. I decided to implement this global cache next and see at what stage it filled up, at which point I could implement a feature to revert back to just the local (loop-checking) cache which uses significantly less space.

Implementing this global cache first turned out to be both hugely effective in terms of the performance increase, and also didn't suffer from the memory problems that I anticipated it would. From a position where a rank-4 game was the upper limit for Spanish Patience, this feature now enabled it to solve games of up to rank 10. And although the space complexity of the algorithm is poor, I hadn't appreciated how little space the game state representation actually takes up. Consider the following back-of-the-envelope calculation: if a basic representation of the game state (my final implementation is much more compact than this) uses a byte to represent each of the 52 cards, a linked list of cards to represent each pile, and another dozen pointers to reference each pile, this uses 64 8-byte pointers in total, plus another 52 bytes for each card, totalling 564 bytes. Therefore, using a gigabyte of RAM, one can store just under two million game states before the cache fills up. With a little more memory and a much more efficient state representation, one can store orders of magnitude more states. In fact searching through this many states is so time consuming that even after running searches for hours, I never came across an instance where my solver ran out of memory.

The next step in my search algorithm design was to move from a model where each node in the search tree contains a representation of the game state at that point in the search, to a model where each node in the search tree contains the *move* made at that point in the search. With this new approach, there is a single representation of

the game state, which is modified as the search nodes/moves are explored, applying the move for the current search node to this single mutable 'search' state at each step.

This type of DFS is known as *backtracking search*[32], and its advantages here are quite subtle. For searches without natural loops, representing only the changes between states can save a large amount of memory. However, for my search each state needs to be copied into the cache, so there is no avoiding copying the entire game state at every step in the search. Yet crucially, if the only copy of the entire state is the one done into the cache, we are afforded a certain luxury: this copied state need only satisfy the property that it fully and uniquely represents the state of the game at that point in time. It does not have to be copied in such a way that the copied version is easy to manipulate for search purposes. Whereas the 'search' game state (the one modified at each step) contains a number of space-consuming data structures to makes moving cards fast and easy, the one we copy into the cache can be much simpler as it will never need to be modified.

A more concise representation means that less data has to be copied into memory at each step in the search. Not only does this help slow the rate at which memory is consumed, more importantly, because this copy is an expensive operation, this change makes the search run much faster. The precise implementation of this technique can be seen in section 7.2.3, and an evaluation of its performance can be seen in section 8.3.2.

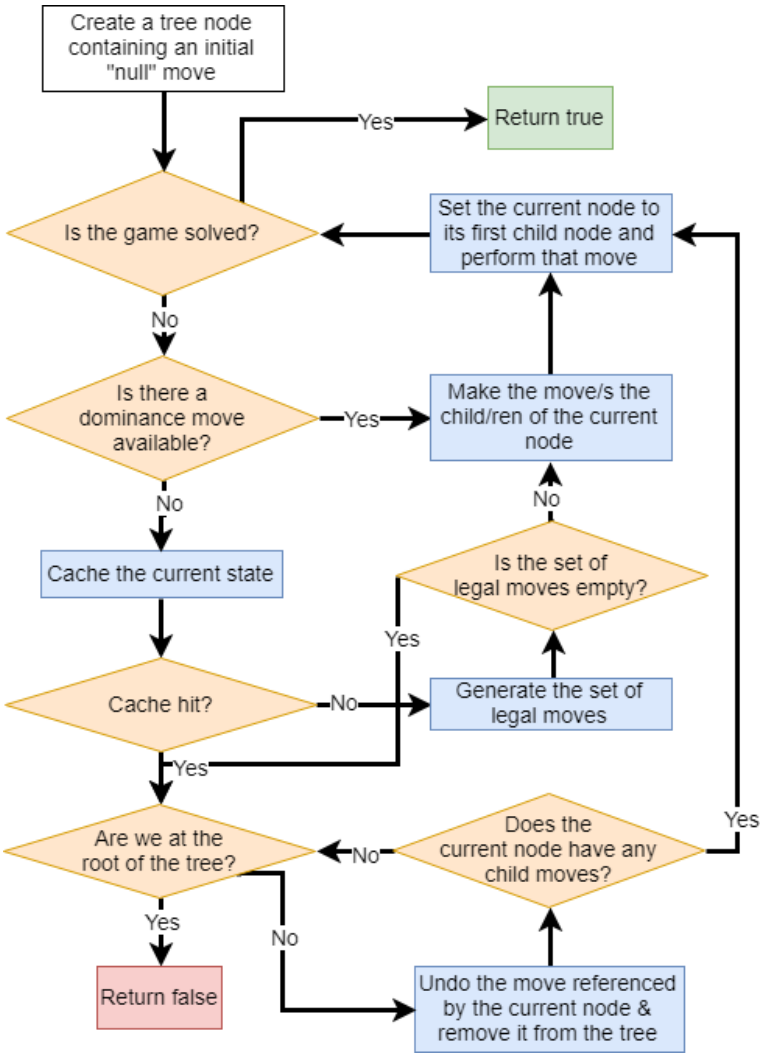
One side-effect of backtracking search is that the ordering of the search changes slightly. Before, when each node was an entire game state, the frontier was based on a stack data structure where nodes were popped and their children then pushed on to the top of the stack, giving a standard preorder traversal. To visit a node's child nodes, I did not have to re-visit the parent. However, now that each node represents the change *between* states, this 'jumping' between child states is not possible without revisiting the parent, so moving between states in the search tree now has to happen strictly between adjacent nodes. Revisiting the parents involves implementing a mechanism for 'undoing' moves to get back to the previous state. Fortunately, because my moves implementation is very simple, undoing a move is just as simple as doing it in the first place.

The key features of my search algorithm have now mostly been covered. One small final feature, worth mentioning briefly here before we look at the algorithm as a whole, relates to *dominances*. This is an idea taken from constraint programming, and refers to moves which we can prove are always preferable to certain other moves, meaning

that we don't need to explore the other 'dominated' moves. We will return to this idea in more detail in section 6.3.2, but for now one simply needs to appreciate that *Solvitaire* can identify a dominance move if it exists in a state, and if this is the case, doesn't need to consider alternatives.

Combining all of the search logic explored thus far, we are left with the search algorithm outlined by the flowchart in figure 6.1.

Figure 6.1: The outline of my DFS algorithm



We see how the backtracking depth first search progresses, caching those states it needs to and making dominance moves where possible. Note that states which have an available dominance move don't need their legal moves to be generated, and are also left out of the cache. These states don't need caching as they always lead to

the same state from the dominance move, so caching the next state resulting from a non-dominance move suffices. Search stops for branches where cache hits indicate we have (globally) seen a state before, and if the search returns to the root state with no children left to explore and without ever having seen a state that represents a solution, then it can report a deal as unsolvable.

The only additional feature of my search not captured by this diagram relates to my technique for finding shorter solutions. Although DFS has some useful properties, by its nature it tends towards finding deep solutions in the search tree. And because the depth here is equal to the number of moves made, this leads to long solutions. But particularly if humans wish to understand the solutions for particular deals, this is not ideal. It would be nice to find shorter solutions, even if this is a little more expensive.

There are several possible approaches to this, ranging from switching to a breadth-first search, to using A* search, but both would have involved very significant modifications to my search algorithm and logic, and as finding shorter solutions was not the main focus of the project, this seemed too complex an approach for the time being. The alternative I opted for to find short solutions, is a variation of my depth first search known as iterative-deepening depth-first search. The idea behind this algorithm, is that it runs a regular DFS search, but with a bound for the depth of the search. Then, if this bounded search does not return a solution, the bound is increased and the search run again, until a solution is found. This guarantees that shallow (i.e. minimal) solutions are found first in the tree, but still enables me to use my existing DFS algorithm. This sounds computationally expensive, but in reality has the same theoretical time complexity as DFS[26], as in theory most nodes in search tree tend to be at lower levels, so it makes little difference visiting upper levels multiple times.

Solvitaire provides a runtime option to enable this search to be run on deals, outputting the current depth bound at each step of the iterative search. In practice, as we shall see, this search technique was not nearly as effective as its theoretical properties suggested it might be. An evaluation of its performance can be seen in section 8.2.

6.3 Reducing Search

Although the solver must guarantee that its deal classifications (solvable/unsolvable) are correct, there are ways of doing this without naively searching through every

possible move in every possible state. There are several techniques that *Solvitaire* employs to reduce the size of the search space without sacrificing its accuracy, greatly improving its performance. These are centred around the notions of symmetries and dominances, which identify areas where we can guarantee a safe pruning of the search tree. A more advanced technique known as streamliners enables even more effective search, still guaranteeing that unsolvable instances will never be reported as solvable, but occasionally classifying solvable instances as unsolvable.

6.3.1 Symmetry

There are several forms of symmetry that exist within solitaire games. We have seen already that during search, if a state is reached that is equal to one seen before, we can safely reduce the size of the search space (and avoid loops) by not exploring its children. The same applies to states that are *symmetrically* equal to ones seen before. As long as we can guarantee that the two states really are symmetrical–in other words, they can be swapped with each other and still essentially behave in the same way–then this will not harm the accuracy of the solver. Systematically identifying many symmetrical states during a search can often reduce the search space drastically, making intractable problems tractable after all.

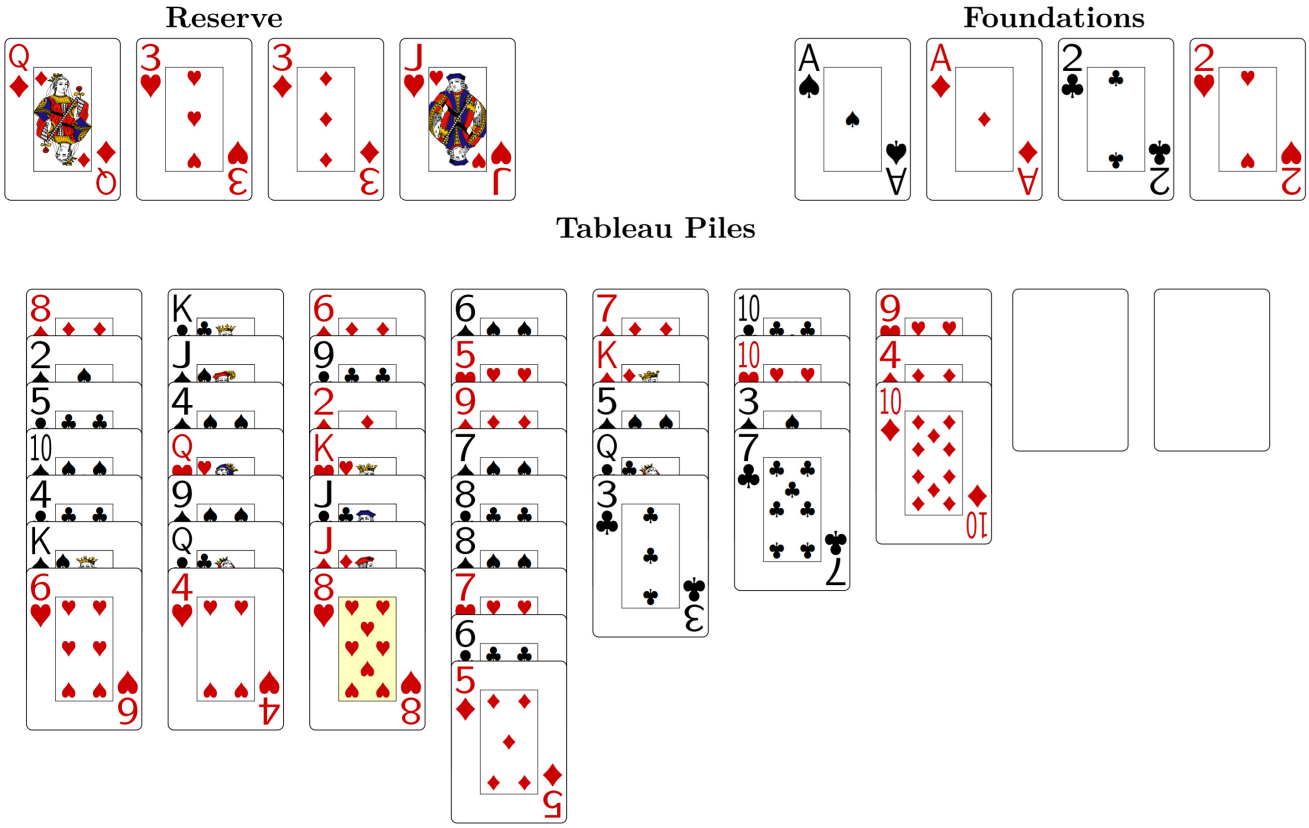
6.3.1.1 Pile Symmetry

The most obvious symmetry in solitaire games relates to pile ordering. Very few solitaire games place any semantic significance on which order the tableau piles are in, and hence permuting them makes no difference. However, in a basic solver one might well hard-code these pile orderings such that permuting them *does* make a difference in the eyes of the solver. In the context of *Solvitaire*, the program would generate different hashes for the same tableau piles in different orders. This may seem at first like a fairly rare symmetry. After all, especially with single-deck games, are we often going to reach a game state identical to one already encountered but with one or more piles swapped? The answer to this is actually yes, due to the fact that empty columns are inherently identical to one another. Certain games are particularly prone to empty or single-card piles, whose location is often arbitrary.

Consider figure 6.2 which shows an early game state in the game Raglan. For this example, the rules are largely unimportant for those who are unfamiliar with the game. Let's say the player wishes to remove cards from the third column (perhaps to get to the 2♦ underneath). They have to choose whether they wish to move the 8♥ into

the first empty pile or the second. To a human player, this choice makes no difference, but to a solver which doesn't take into account symmetry, these are entirely different moves. This would create two separate search branches, each of which would need to be explored fully, doubling the amount of work that the solver needs to do. However, because *Solvitaire* can identify pile symmetry, once it has visited one of these two states, upon visiting the second it identifies that a symmetry exists between the two and knows it doesn't need to explore any resulting moves (for precise details of how I implemented this symmetry feature, see section 7.3.2).

Figure 6.2: Pile Symmetry Example for the game Raglan



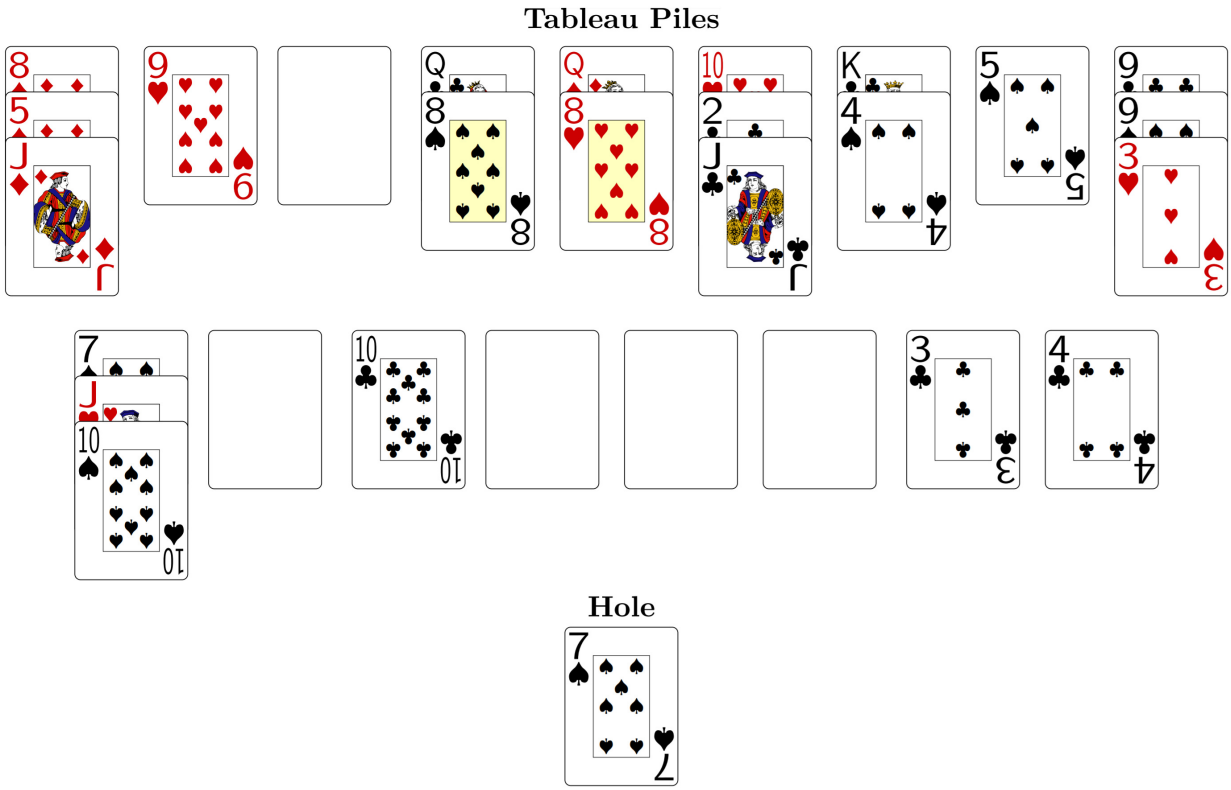
It is not just tableau piles that exhibit this symmetry; I also implement the same logic for reserve piles and free cells, which again gives me a large performance boost for games using these features. The other symmetry that *Solvitaire* exploits, suit symmetry, which we will now examine, also combines with pile symmetry particularly well to reduce the size of the search space.

6.3.1.2 Suit Symmetry

Certain games have the property where cards of different suits but the same rank function in exactly the same way as each other in all circumstances. I term this property *suit symmetry*. We see this in the game Black Hole, where cards can only be moved from the tableau piles to the hole, and moves are constrained only by rank. Here all suits are symmetrical. This enables *Solvitaire* to avoid a great deal of search, by essentially treating cards with the same rank as being identical.

In figure 6.3 we see an example of this. If *Solvitaire* were attempting to solve the game from this sate, it might first move the 8♠ into the hole. It would then explore the rest of the resulting search space and either solve the game, or get stuck. Assuming the latter, the solver would then backtrack to the state pictured, and might now attempt to move the 8♥ into the hole. Having done this, (in accordance with the flowchart in figure 6.1) it would cache the state.

Figure 6.3: Suit Symmetry Example for the game Black Hole



Here is where the symmetry kicks in. The solver at this stage realises it has seen this state before because–due to a combination of pile and suit symmetry–it is logically identical to the state reached when the other eight was moved. This causes

the current branch of the search to stop, a backtrack takes place, and the search continues. This combination of the two symmetries is key to making Black Hole solvable, and many other games also benefit from this property. Although in Black Hole the symmetry spans across all the suits, other symmetries exist that only apply to pairs of suits. For instance, if we add the constraint to Black Hole that cards put into the hole must be of alternating colour, then there would not be a single suit-wide symmetry, but a pair of symmetries for hearts-diamonds and clubs-spades. *Solvitaire* is able to detect the appropriate symmetries based on which types of cards can be moved to the hole, and runs its search accordingly.

Black Hole is quite a restrictive solitaire game in terms of the types of moves allowed, but even if cards were allowed to be moved between tableau piles in the game, suit symmetry would still apply so long as the 'build policy' were the same for the hole pile and the tableau piles. The build policy is my term for which suits are allowed to be placed on a pile based on the suit of its top card; for instance, 'any suit', 'red-on-black' or 'same suit'. 'any suit' gives full symmetry, 'red-on-black' gives a pair of symmetrical suits, and 'same suit' gives no symmetry. If the hole and tableau pile build policies differ, then the more restrictive of the two corresponding symmetries applies to the entire game.

Unfortunately, when we expand this technique to games which use foundation piles instead of a hole as the 'target' pile, we encounter a problem. Because each suit has its own dedicated foundation pile, each suit behaves uniquely, destroying any suit symmetry. Even if the tableau piles have an 'any suit' build policy, we cannot treat cards of identical rank in the same way, because at some point they will have to be moved to their own suit-specific foundation pile. This means that suit-symmetry is not applicable in foundation-based games, which is a large percentage of solitaire games. This is a shame, as suit symmetry can be a powerful tool. Fortunately though, there is still some advantage to be gained in search for foundation games through the use of a suit-symmetry streamliner. This is examined in section 6.3.3.1.

6.3.2 Dominances

A very brief overview of dominances had to be given when explaining the search algorithm, but we will revisit this topic fully now. To reiterate, in the context of solitaire, dominances are moves which we can prove are always preferable to certain other moves, which we therefore don't have to explore. The dominance that *Solvitaire* exploits relates to moving cards onto the foundation piles. For games with foundation piles, filling them is the goal of the game, so in most circumstances if a player can

put a card on its corresponding foundation they do so. However, many games also stipulate that cards cannot be removed from foundation piles once placed there, so if a card may still be needed in the tableau piles, for whatever reason, then one should avoid putting it 'up' (i.e. onto a foundation pile) until they can be certain it is 'safe' to do so (i.e. the card is definitely no longer needed in the tableau piles). Under the conditions where moving a card up to a foundation pile is demonstrably safe however, such a move can be considered a dominance.

There are two rules for when foundation moves are safe to move up in FreeCell. The first states that a card can always be moved up to the foundations if it's at most one greater in rank than the lowest rank foundation card of the opposite colour. The second rule is that a card can also always be moved up if it is two greater in rank than the lowest rank foundation card of the opposite colour, *and* the other foundation card of the same colour is at most three ranks below it. These facts are stated here without proof or analysis. To confirm these rules myself, I had to expend a lot of time and effort thinking about the logic that enables such moves to be made safely, and my reasoning is outlined in depth in Appendix D, for those who don't wish to take these rules on faith.

The logic I use in Appendix D is based around the game FreeCell, but will work for all games with a 'red-black' build policy where cards can't be removed from the foundations. As for games with the 'same suit' build policy, because the reliant set (see Appendix D for an explanation of what I mean by 'reliant set') for each card only contains the card it would be placed on in the foundations, all foundation moves are considered dominances. For the 'any suit' build policy, the rule is simply that a card can be moved to the foundations as long as its rank minus two, is less than or equal to all of the other foundation piles' rank.

Whilst studying this problem and considering the implications of this logic across different build policies, I formulated a general algorithm for determining if a move onto a foundation pile could be considered a dominance. All this algorithm needs from a build policy is that it define a reliant set, so although the solitaires considered in this project only use the three build policies outlined above, this algorithm could also be used for more complex policies. I outline its workings in Algorithm 1.

It works using a generalised form of the inductive reasoning we have already seen. For each tableau pile it uses a boolean to represent whether moving up the next card is a dominance, and every time the foundations are altered, re-assesses these variables. The algorithm that does this, loops through each foundation and turns false booleans

true if their next card has its reliant set in the correct state. If one false boolean is turned true, the algorithm restarts because this change might enable other new auto moves for other foundations.

Algorithm 1 Determining Auto-Foundation Dominance Moves

```
// Indicates the foundations whose next card can be moved up automatically
F ← (FALSE, FALSE, FALSE, FALSE)
F ← UPDATEAUTOFS(F) // If a card is moved to the foundations, call this...

procedure UPDATEAUTOFS(F)
    F' ← FILTER(FALSE, F)
    for f in F' do
        f ← ASSESSFOUNDATION(f, F)

        // If foundation f's auto-move boolean is now true, this may enable other
        // auto-moves for other foundations, so we must restart
        if f is TRUE then
            return UPDATEAUTOFS(F)
    return F

// Returns true if the supplied foundation can automatically move the next card up
procedure ASSESSFOUNDATION(f, F)
    n ← NEXTLEGALFOUNDATIONCARD(f) // The next card to be placed on f
    rs ← NEXTLEGALTABLEAUCARDS(n) // n's reliant set

    // All cards in the reliant set must either be already in the foundations, or must
    // automatically go up to the foundations when they are uncovered
    for r in rs do
        c1 ← CONTAINS(foundations, r)
        c2 ← ISAUTOFOUNDATIONCARD(r)
        if ¬c1 ∧ ¬c2 then
            return FALSE
    return TRUE
```

Applying this algorithm is what gives us the rules for FreeCell dominance moves previously stated. I initially used this general algorithm directly in my code to find dominances, but it turned out to be rather slow, and it was much easier and more efficient to simply hard-code the specific rules for each build policy into *Solvitaire*.

All of my analysis has been done for games where it is forbidden to remove cards from the foundation piles, but what about those where this is legal? For these games, *Solvitiare* still makes the same dominance moves, but adds the constraint that those cards moved up to the tableau piles as a result, need never be moved back down to the tableau piles during our search. It's worth pointing out here that the search algorithm

still explores moves where cards are put up to the foundations 'unsafely', but does so using regular game moves, rather than as dominances where alternatives are not explored. For an evaluation of how these dominance moves affect the performance of the solver, see section 8.3.6.

6.3.3 Streamliners

The final search technique that *Solvitaire* implements are two *Streamliners*. This is a search technique used to find valid solutions for problems that would otherwise be hard to solve, by reducing the size of the search space in a way that gets rid of some (but ideally as few as possible) valid search paths. It's important to realise that this technique is different from those seen before, insofar as it does not guarantee 100% accuracy for its solvable/unsolvable classifications. We are effectively sacrificing accuracy for performance. The challenge is to try and trade away as little accuracy as possible for large performance gains. Although streamliners will sometimes report solvable instances as unsolvable, they do guarantee that unsolvable instances will never be reported as solvable. This means that if this technique returns a solution, it is guaranteed to be a valid solution; but if we are conversely told that an instance has no solution, there is a small possibility that it may actually have one.

6.3.3.1 Suit Reduction Streamliner

We saw in section 6.3.1.2, that for hole games we can use suit symmetry to reduce the search space quite significantly. Unfortunately, having foundation piles in a game breaks this symmetry. This is a shame, as the symmetry is powerful and *almost* seems to work - it is just the occasional move to the tableau piles which stops it. This makes this technique a good candidate for a streamliner: we suspect that it will improve our search greatly, whilst only occasionally changing the classification of a deal.

Hence I provide *Solvitaire* with an option to be run using what I call a *Suit Reduction Streamliner*, which applies suit symmetry to games based on the tableau pile build policy, even if there are foundations present. Consider what is happening to the search space here: this symmetry identification is pruning branches every time it sees one of these symmetries. Some of these branches may lead to solutions, but hopefully not many. If we find solutions they will certainly be valid, as we are not adding any new branches to the search tree, but some valid solutions may be missed. We only hope that not too many solvable games have all their solutions removed. For an evaluation of the accuracy and performance of this streamliner, see section 8.3.7.

6.3.3.2 Auto Foundation Streamliner

My other streamliner relates to the auto-foundation moves outlined in section 6.3.2. In this section we took great care to identify the specific scenarios in which we could guarantee that moving cards up to the foundations wouldn't eliminate solutions. However, many novice solitaire players of foundation-based games, seeing that the objective is to get cards into the foundation piles, will simply move cards up as soon as they can. This tactic often works perfectly well, and is another good example of a strategy that sacrifices accuracy to make search easier. Hence, I also make this a streamliner. Users of *Solvitaire* can specify the *Auto Foundation Streamliner*, which treats all potential moves from the tableau piles to the foundations as dominances. Again, to see how accurate and powerful this streamliner is, see section 8.3.8.

6.4 Generating Solvability Percentages

One of the tertiary objectives for this project was to calculate solvability percentages for any game whose rules can be described to *Solvitaire*. Given that we have an easy way to generate and solve random deals based on seeds, and can do so for a wide variety of games, we are most of the way there. But how do we go from here to a solvability percentage? Firstly, it's worth considering that (unless we were to explore all 52! starting deals) these solvability percentages are always going to be *approximate* measures. Yet we can build up an estimate by attempting to solve a large number of random deals/seeds and looking at how many we are solvable/unsolvable. More formally, we are trying to estimate a confidence interval for the mean of the solvability of a given game type, which we can model as a binomial distribution. There are a number of different methods that can be used to calculate the confidence interval of a binomial distribution. *Solvitaire* makes use of the Agresti-Coull interval, which is a simple approximate method that has been shown to give accurate estimates[1]. The pseudocode implementation for this method can be seen in Algorithm 2.

However, in reality this model is not all that useful in its current form, as we don't really have a boolean outcome (solvable/unsolvable). This is due to a third possible state for a solution, which is 'intractable' (timed out). Were we not to set a timeout, for a solitaire where the vast majority of deals are easily solvable, one particularly hard instance might grind our solver to a halt. We may be better off letting this instance time-out, and solving other instances instead. But how do we deal with the instances we've classified as intractable when estimating the solvability percentage? We can make no assumptions about the distribution of intractable deals,

Algorithm 2 Calculating The Agresti-Coull Interval

procedure AGRESTICOULL(X, n) // *Given X successes in n trials*
$z \leftarrow 1.96$ // *For 95% conf. interval*
$n' \leftarrow n + z^2$
$p \leftarrow \frac{1}{n}\left(X + \frac{z^2}{2}\right)$
$i \leftarrow z\sqrt{\frac{p}{n'}(1-p)}$
return $(p - i, p + i)$

and it's certainly not safe to assume that they have the same distribution as the ones we've already solved. The best we can do is calculate two confidence intervals: one assuming they are all unsolvable, and another assuming they are all solvable. We then take the lower bound from the former, and the upper bound from the latter, giving us a new (wider) overall confidence interval. This approach is outlined in Algorithm 3. For this method there are now two factors that can cause our confidence interval to be undesirably wide: firstly, not having run enough deals, but now also, having too many intractable instances.

Algorithm 3 The Intractable-Adjusted Agresti-Coull Interval

procedure INTRACTAGRESTICOULL(s, u, i) // *solvable, unsolvable & intractable*
$lb, \_\_ \leftarrow$ AGRESTICOULL($s, u + i$)
$\_, ub \leftarrow$ AGRESTICOULL($s + i, u$)
return (lb, ub)

This gets us most of the way towards a method for calculating solvability percentages, but one problem remains: how long do we set our timeout for? Too short and we end up with too many intractable instances, and too long and we don't get through enough deals in a reasonable period of time. We have seen from Algorithm 3 that both of these factors can stop our confidence interval from narrowing, so we must find a balance. We could use trial and error to come up with a reasonable timeout, but this method is both imprecise, and has to be re-done for every different game we wish to generate a solvability percentage for. To solve this problem, I have created a method for scaling the timeout to balance the rate at which deals are attempted with the number of intractable instances. I believe this to be a novel approach; I'm not aware of any existing method which does this, and this algorithm could be used for many kinds of AI problems beyond solitaire.

The basic premise of the algorithm is as follows: the solver attempts to solve deals

starting at seed zero, using some (small) timeout value. As it attempts to solve deals, it builds up lists of those that are/are not solvable, and those that are intractable. After each deal attempted it then estimates the following:

1. If the timeout was doubled and we went back over the instances previously found to be intractable, how many might we now expect to find solvable/unsolvable? And following this, if we add those to the current counts of solvable/unsolvable/intractable seeds, what would we expect the resulting overall confidence interval to be?

2. If we were to continue solving new seeds using the current timeout, for the length of time we expect it would take to do step 1 (above), what would the expected overall solvable/unsolvable/intractable counts be by the end? And again, what would we expect the resulting overall confidence interval to be?

The size of these two intervals is then compared, and whichever is smaller dictates what course of action the solver takes. What we're essentially asking here, is: do we expect to narrow the confidence interval more if we double the timeout and go back over the intractable seeds, or if we continue with the current timeout for the same period of time? What tends to happen in practice, is that initially when the timeout is low, many deals are found to be intractable and doubling the timeout is the best way of narrowing the confidence interval. However, the timeout eventually becomes high enough that a large number of deals can be solved, and doubling it makes less and less difference, meaning that the solver is much more inclined towards continuing, rather than doubling the timeout. Thus, the timeout converges towards the optimum, and finds the ideal balance between attempting many deals, and having few intractable deals.

But we have not addressed the question as to how we estimate the number of deals we expect to find solvable/unsolvable/intractable if we either continue or re-evaluate. The simplest of these two cases is the one where we continue with same timeout. This can be seen in Algorithm 4. Assume for the moment that we have some estimate for the time taken to do the re-evaluation step. We can use the time taken to solve the tractable seeds so far, to estimate how many seeds we would be able to solve in the same time period. Having done this, we can then use our counts of how many deals thus far have been solvable/unsolvable/intractable, to estimate how many deals in this time window will be solvable/unsolvable/intractable. This is again, a problem of estimating means, so we can use the Agresti-Coull method

here. Adding these estimated values for the time window onto our values thus far, we form an estimate for the overall number of solvable/unsolvable/intractable were we to continue, rather than re-evaluate. These values are then used to form our estimated confidence interval, again using the Agresti-Coull method used for the main confidence interval.

Algorithm 4 Expected Confidence Interval For Continuing

procedure ContinueCI($s, u, i, c_t, t_{out}, t_{total}$) // *solvable, unsolvable, intractable, time coefficient, timeout, total time taken*

$d_{total} \leftarrow s + u + i$
// *Agresti-Coull estimates for solvable/unsolvable/intractable so far*
$\mu_s \leftarrow$ AgrestiCoullMean(s, d_{total})
$\mu_u \leftarrow$ AgrestiCoullMean(u, d_{total})
$\mu_i \leftarrow$ AgrestiCoullMean(i, d_{total})

$t_{reeval} \leftarrow i \cdot 2t_{out} \cdot c_t$ // *Expected time taken to re-evaluate intractable seeds*
$d_{continue} \leftarrow d_{total} \left(1 + \frac{t_{reeval}}{t_{total}}\right)$ // *Total deals evaluated after time period*

// *Total number of solvable/unsolvable/intractable after time period*
$s_{continue} \leftarrow \mu_s \cdot d_{continue}$
$u_{continue} \leftarrow \mu_u \cdot d_{continue}$
$i_{continue} \leftarrow \mu_i \cdot d_{continue}$

return IntractAgrestiCoull($s_{continue}, u_{continue}, i_{continue}$)

Judging what will happen if we double the timeout and re-evaluate the intractable seeds is trickier. We wish to estimate how many intractable deals will become solvable/unsolvable, but this is problematic because any assumption about this distribution will be an extrapolation. I reason that the best heuristic for estimating this is what happened on previous occasions we doubled the timeout and re-assessed. Each time this procedure takes place, I record the number of solvable/unsolvable/intractable deals. I then use these values to update a group of coefficients for each. These coefficients are calculated using an exponentially-weighted moving average, and can be used to estimate what will happen during future re-evaluations of intractable seeds. It is quite possible that the distribution of of solvable/unsolvable/intractable deals changes as the timeouts increase, which is why weighting more recent values is likely to give more accurate estimations. I also keep track of the time taken per deal for this re-evaluation procedure (as a percentage of the size of the timeout). I calculate an exponentially-weighted moving average for this too, and use it in Algo-

rithm 4 to calculate the time window for how long we would expect re-evaluation to take. My approach to estimating the re-evaluation confidence interval can be seen in Algorithm 5, and the way I update the coefficients can be seen in Algorithm 6.

Algorithm 5 Expected Confidence Interval For Doubling Timeout & Re-evaluating

procedure REEVALCI(s, u, i, c_s, c_u, c_i) // *solvable, unsolvable, intractable, solvable coefficient, unsolvable coefficient, intractable coefficient*

// Expected total sol/unsol/intract after re-evaluation of intract. seeds
$s_{reeval} \leftarrow s + c_s \cdot i$
$u_{reeval} \leftarrow u + c_u \cdot i$
$i_{reeval} \leftarrow c_i \cdot i$

return INTRACTAGRESTICOULL($s_{reeval}, u_{reeval}, i_{reeval}$)

Algorithm 6 Updating Re-evaluation Coefficients

procedure UPDATECOEFFICIENTS($s, u, i, w, c_s, c_u, c_i, c_t$) // *(Out of the re-evaluated seeds:) solvable, unsolvable, intractable, EWMA weight, solvable/unsolvable/intract coefficients, time coefficient*

$d_{total} \leftarrow s + u + i$
// Estimates for solvable/unsolvable/intractable mean for the re-evaluated seeds
$\mu_s \leftarrow$ AGRESTICOULLMEAN(s, d_{total})
$\mu_u \leftarrow$ AGRESTICOULLMEAN(u, d_{total})
$\mu_i \leftarrow$ AGRESTICOULLMEAN(i, d_{total})
// These values are then normalised to make sure they all add to 1
// ...

$c_s \leftarrow w \cdot \mu_s + (1 - w) \cdot c_s$
$c_u \leftarrow w \cdot \mu_u + (1 - w) \cdot c_u$
$c_i \leftarrow w \cdot \mu_i + (1 - w) \cdot c_i$

// Something similar is then done for c_t

return (c_s, c_u, c_i, c_t)

Bringing all this together we end up with a reliable method for automatically scaling the timeout to find the optimum for any game type. This especially useful for *Solvitaire* users who may not understand how generating solvability percentages works, and just want to enter the rules for a game and press 'go', without having to tweak parameters.

A typical output from *Solvitaire* when generating solvability percentages can be seen in listing 6.5 (the timeout begins higher than it normally would here, for demon-

```
[Lower Bound, Upper Bound] | Solvable/Unsolvable/Intractable |
   Current seed | (Est. Confidence Interval: Continue/Re-eval)
--- Timeout = 500 milliseconds ---
[0.000, 100.000] | 0/0/0 | 0 | (N/A)
[0.000, 100.000] | 0/0/1 | 0 | (88.745, 83.251)
--- Timeout doubled to 1000 ms. Re-evaluating intractable seeds ---
[-- re-eval --] | 1/0/0 | 0
--- Re-evaluation complete ---
[16.749, 100.000] | 1/0/0 | 1 | (N/A)
[29.022, 100.000] | 2/0/0 | 1 | (N/A)
[38.252, 100.000] | 3/0/0 | 2 | (N/A)
[28.913, 96.592] | 3/1/0 | 3 | (N/A)
[35.962, 97.968] | 4/1/0 | 4 | (N/A)
[41.780, 98.864] | 5/1/0 | 5 | (N/A)
[35.235, 92.437] | 5/2/0 | 6 | (N/A)
[40.085, 93.694] | 6/2/0 | 7 | (N/A)
[35.090, 94.657] | 6/2/1 | 8 | (68.028, 53.610)
--- Timeout doubled to 2000 ms. Re-evaluating intractable seeds ---
[-- re-eval --] | 0/0/1 | 8
--- Re-evaluation complete ---
[35.090, 94.657] | 6/2/1 | 9 | (56.844, 53.610)
--- Timeout doubled to 4000 ms. Re-evaluating intractable seeds ---
[-- re-eval --] | 0/0/1 | 8
--- Re-evaluation complete ---
[35.090, 94.657] | 6/2/1 | 9 | (56.844, 53.610)
--- Timeout doubled to 8000 ms. Re-evaluating intractable seeds ---
[-- re-eval --] | 1/0/0 | 8
--- Re-evaluation complete ---
[44.279, 94.657] | 7/2/0 | 9 | (N/A)
[47.936, 95.411] | 8/2/0 | 9 | (N/A)

...

[79.463, 92.928] | 85/12/0 | 96 | (N/A)
[78.481, 93.003] | 85/12/1 | 97 | (17.209, 13.465)
--- Timeout doubled to 16000 ms. Re-evaluating intractable seeds ---
[-- re-eval --] | 0/0/1 | 97
--- Re-evaluation complete ---
[78.481, 93.003] | 85/12/1 | 98 | (13.014, 13.465)
[78.685, 93.076] | 86/12/1 | 98 | (12.894, 13.344)
[78.884, 93.148] | 87/12/1 | 99 | (12.776, 13.224)
...
```

Listing 6.5: Output when generating solvability percentage for Alpha Star

strational purposes). Initially the timeout is too low and the first thing *Solvitaire* does is double it. Then it finds the next few seeds tractable, but soon finds one that isn't. At that point it is able to project confidence interval sizes for continuing versus re-evaluating, and it decides re-evaluation is a better tactic. This continues until the timeout reaches 8 seconds, at which point it finds most deals tractable. The next seed (97) that it times out on causes a re-evaluation, which is not successful in making the deal tractable. At this point the algorithm projects that it's not worth doubling the timeout again, and so the solver is willing to let this intractable deal remain so, as it thinks it can close the confidence interval more by continuing with other seeds. Thus we see how a balance is found between the rate at which deals are attempted, and the number of intractable instances remaining.

7 Implementation

This section will briefly cover some of the interesting implementation details for various areas outlined in my design section. For an overview of how I tested this functionality, please refer to Appendix B.

7.1 Solver Features

7.1.1 Rule & Deal Representation

To parse in users' JSON descriptions for the rules of solitaire games, I make use of the C++ RapidJSON[31] library. This not only allows me to read JSON into C++ objects, but also to check the validity of the JSON against the schemas I have defined for rule and deal descriptions. The object into which the rules are parsed is a simple fixed class, which just contains the booleans, integers and enums that correspond to the supplied JSON. The second JSON description for the starting deal is then used to populate the initial setup for my game state representation, which is manipulated during the search.

7.2 Solver Design

7.2.1 Search Implementation

My search implementation can be found in my `Solver` class. There is not much particularly noteworthy about my search implementation that hasn't already been mentioned in the Design section, so I won't go into great detail here. Each node in the search stores a reference to its parent node, its child nodes, and the move that node represents. For my implementation of iterative-deepening search, I also pass a depth-limit to the solving algorithm, meaning that not only does the search terminate when a solution is found or the search is exhausted, it also terminates when we reach a certain depth in the search tree.

7.2.2 Game State Implementation

As mentioned in the Design section, my game state implementation consists of a single mutable state, to which 'moves' are applied throughout the search. A move, in

my implementation, is essentially just a reference to the columns to and from which cards are to be moved (plus an extra flag to indicate things like 'built-group' moves). The game state itself is comprised of three classes:

7.2.2.1 Cards

This is the most basic object in my game state representation. Each card consists of a rank, and a type, which I represent as 8-bit `uint8_t` objects. This is the smallest size primitive in C++ (except booleans), but is actually larger than I need. Thus to save more space within the card class, I use the 'bit field' language feature to compress each down to 4 bits, making each card take up just a byte. This level of compression makes a significant difference when it comes to the solver's performance, as the most expensive part of the search algorithm is copying data into the cache, and this compact representation requires less data to be copied.

7.2.2.2 Piles

My piles representation groups cards as part of a vector (dynamic array), to which cards can be added and removed. The consequence of this implementation is that moving cards involves only reading and writing a few bytes at each step. In general, moving cards is a very cheap operation.

7.2.2.3 Game State

My game state class simply contains a vector of these piles, and stores pointers to them to identify which are tableau piles, which are foundation piles and so on. It also stores a few other variables which are used for some of my other advanced search features, which we will see in later sections.

7.2.3 State Caching

The way in which state is cached in my implementation is what enables some of my more powerful search optimisations: chiefly, suit symmetry and state-reduction. We will begin with the latter, which is the term I use for my method of compressing the game state when I put it in the cache. This is an optimisation that can be toggled on and off, so that in the evaluation is section 8.3.2 we can explore the effect it has on the performance of the solver. I have already justified why this optimisation is useful in the Design section of this report.

The idea behind state-reduction is that all we need to put in the cache is the minimal-size unique representation of the game state. Were the solver to put the full search game state representation into the cache, it would also be copying across all of the pointers used to manage the pile vectors, which take up a considerable amount of memory relative to the size of a single card (three 8-byte pointers per vector, versus one byte per card). If we simply copy each card from the game state in some fixed order, into a plain array, then we compress the game state representation significantly. However, this is not quite enough, as we now no longer have a way of differentiating the top of the first pile from the bottom of the second in this array. Thus to complete this compressed representation, the solver also adds some blank card 'markers' of rank zero to delimit the piles.

The caching itself makes use of the C++ `unordered_set` class, which implements a standard hashset. However, because I'm caching classes which are user-defined, there is no default way to hash them. This means I have to define my own hash function for each of them. For this, I take my lead again from the source of the C++ Boost libraries. They define a much wider family of hash functions for a range of STL classes, which I use myself to define how my piles and the overall game state build up hashes from individual cards.

The other notable feature of my caching implementation is how I handle suit symmetry, which we will come to in the next section.

7.3 Reducing Search

7.3.1 Suit Symmetry Caching

My implementation of suit symmetry is based upon the way *Solvitaire* caches the game state. Rather than encoding symmetry directly into the game state representation, all we really need do is make sure that symmetrical states hash to the same value. This will mean that symmetrical states are *treated* by the search algorithm as though they are identical, which is precisely what we need. Thus when the game state is put into the cache, when each card is copied, *Solvitaire* checks the game's suit symmetry. If the game has red-black symmetry, then it hashes hearts-diamonds to the same value, and the same is done for clubs-spades. With full suit symmetry, all suits of the same rank are hashed to the same value.

7.3.2 Pile Symmetry Ordering

My first attempt at implementing pile symmetry also involved using just the cache. The idea behind this was that whenever the state was cached, the solver would look through the tableau piles (and other commutative piles such as the reserve and cells) in the game state, and order them somehow. It would then copy the piles from the game state into the cache based on this ordering. In theory, this works well, but in practice it proved difficult to implement in a way that didn't add some noticeable time overhead to the search. This was due to the fact that the ordering had to be recomputed entirely every time the state was cached.

After profiling my code and identifying the nature of the problem, I came up with an alternative implementation. What I now do, rather than sorting the piles every time I hash the state, is maintain a constant data structure identifying the pile orderings within the game state object, which I update incrementally. Thus every time I alter the game state, this ordering has to be updated, but this is usually just a small change, rather than having to go through all the piles to find the ordering at every step. The ordering I use is a lexicographical ordering starting at the base of the piles, which also minimises the extent to which the pile ordering changes. It's worth noting here that when computing these pile orderings I never actually move the piles around in memory, but simply swap around pointers to the piles, which I then follow when selecting the order in which I copy piles into the cache. Hence my implementation adds no significant overhead, and allows me to implement symmetry for all groups of commutative piles.

7.3.3 Implementing Streamliners

In my build script I define different builds for the various optimised versions of *Solvitaire* that I have described. This creates different executables where certain optimisations are enabled/disabled (for more information see the user manual in section C). I create different builds which implement my streamliners too. I could have set this as a command line option, but having all of my optimisations implemented in this way made it easier to test and compare them.

When the compile-flag is turned on for the suit-reduction streamliner, the search runs in exactly the same way as it would do otherwise, except when it comes to hashing suits, in which case the constraint that symmetry can only be applied to hole games rather than foundation games is ignored. Similarly, when the auto-foundations streamliner compile-flag is enabled, the complex logic for when cards can safely be

put up as dominances is ignored, and the algorithm simply indicates that all moves to the foundations are dominances.

8 Evaluation & Critical Appraisal

The aim of this project was not simply to try and get the solver to run as quickly as possible. Much of my focus was on exploring and assessing different techniques involved in creating a solitaire solver, particularly one that can get strong performance across a range of games. It was not known beforehand how many different games it was actually feasible to perform exhaustive search on, so 'strong performance' did not necessarily mean that the solver should be able to solve a large number of games, but simply that it implement a range of techniques as effectively as possible. Fortunately, I have found many games where solving starting deals *was* possible in a reasonable time period (as well as quite a few games where it wasn't).

8.1 Solvability Percentages

Because of the work I did to enable my solver to easily generate solvability percentages, I have been able to calculate solvability percentages for a large number of games. This includes 18 games where until now (as best I can tell) the solvability percentage has not been known, and three games (plus variations on a game) where solvability percentages have been published, and my results confirm those previous findings. All of these games are all built in to *Solvataire* as presets. Another 13 of the preset game types added to the solver were not tractable, but for 6 of these I was able to generate a lower-bound for the solvability percentage using the streamliners I have implemented. As my experimental results will show, these lower bounds are also likely to be close to to the true solvability percentages of these games. The solvability percentages for these 18 games can be seen in figure 8.1. These findings represent the culmination of my project.

Figure 8.1: Generated Solvability Percentages For A Range Of Common Games

Game Type	**95% Confidence Interval for Solvability Percentage**	**Solvable/Unsolvable/Intractable Instances**
Previously Unknown Percentages		
Alpha Star	[46.5%, 47.0%]	128490/146445/3
Delta Star	[33.9%, 35.3%]	6470/12247/2
Eight-off	[98.5%, 100.0%]	313/0/0
Eight-off (any card in space)	[98.0%, 100.0%]	229/0/0
Fan	[45.0%, 52.0%]	564/600/17
ForeCell	[91.3%, 100.0%]	99/0/3
Fortune's Favour	[100.0%, 100.0%]	10000/0/0
King Albert	[66.2%, 69.5%]	4591/2153/75
Martha	[87.7%, 95.2%]	638/45/25
One Cell	[87.8%, 100.0%]	183/0/15
Penguin	[97.8%, 100.0%]	211/0/0
Raglan	[76.3%, 81.4%]	1215/318/16
Spanish Patience	[99.4%, 100.0%]	1076/1/0
Sea Towers	[95.9%, 100.0%]	3136/0/111
Somerset	[50.8%, 54.4%]	2765/2487/49
Two Cell	[91.9%, 100.0%]	1290/0/92
Percentages Confirming Published Results [15][18][25][16]		
Baker's Game	[71.6%, 75.4%]	1528/551/0
Black Hole	[86.0%, 88.6%]	2184/316/0
FreeCell	[96.7%, 100.0%]	3170/0/86
FreeCell (0 cells)	[0.1%, 0.9%]	3/998/0
FreeCell (1 cell)	[14.1%, 20.7%]	125/617/9
FreeCell (2 cells)	[75.2%, 81.1%]	1008/273/19
FreeCell (3 cells)	[91.6%, 99.7%]	206/3/7
FreeCell (4 piles)	[0.0%, 0.0%]	0/10041/0
FreeCell (5 piles)	[3.1%, 5.1%]	77/1857/3
FreeCell (6 piles)	[56.7%, 63.1%]	837/557/19
FreeCell (7 piles)	[95.4%, 99.9%]	162/2/0
Upper Bounds Using Streamliners		
Castles Of Spain	>81.1%	179/11/17
Blind Alleys	>79.1%	382/54/25
East Haven	>73.9%	482/112/29
Flower Garden	>78.8%	612/125/12
Spiderette	>98.8%	1004/4/1
Will-O-The-Wisp	>98.5%	499/2/0

I have tried to implement the rules for these games as closely as possible to what I considered to be the 'standard' versions of each, with the only difference being that all cards are face-up. These standard versions was based on game rules described in three main sources: Morehead's *The Complete Book of Solitaire and Patience Games*[28], the Android App *250+ Solitaire Collection*, and the *Wikipedia* pages for those games which have them. Where the standard version were ambiguous (e.g. *Eight-Off*), I have implemented multiple versions. *Solvitaire*'s `--describe-game-rules` option outputs the rule implementations I have chosen for each preset game type (see User Manual in Appendix C for more details, and a full list of available games).

To generate these results I used the `--solvability` option which *Solvitaire* provides, which implements the algorithm discussed in section 6.4. I attempted to run this for all of the preset games implemented by my solver, and those listed in figure 8.1 were the ones found to be tractable. The raw data for all of the confidence intervals in the table can be found in my `results/solvability` directory. Verifying these results should be as simple as supplying the solvability option and the name of the game to `Solvitaire`, and waiting.

For those games in the 'Confirming Published Results' section, I have cited the papers/websites in which the original results were published, and for the FreeCell variants, these results also align with the graphic displayed in my context survey (figure 2.1). My confidence intervals for all of these games are consistent with those previous findings.

For those games where I state lower-bounds for the solvability of games, the games in question were found to be intractable using my regular approach to generating solvability percentages. Hence, I reran these games with the `--solvability` option supplied, but using the version of the *Solvitaire* executable which uses both of my streamliners in combination. This made many of the intractable games tractable, although no longer guarantees an accurate upper bound (some 'unsolvable' games may actually be solvable). The results for these runs can be found in the `results/lower-bounds` directory. For the game *Spanish Patience*, I originally generated a lower bound, but found that the streamliners solved all but 1 deal out of 1076, which was deemed unsolvable. Although the non-streamliner approach finds many *Spanish Patience* deals intractable, running it on this single deal, it was able to *prove* that it was unsolvable. Because the streamliners guarantee that 'solvable' deals will never turn out to be unsolvable, I am therefore able to assert that my results for Spanish Patience are all of guaranteed accuracy. Hence, I present the full confidence interval for this game, rather than a lower bound.

Overall, my research here into the solvability percentages for these games adds significantly to the body of games for which we know the solvability percentage, and in addition provides a tool with which solvability percentages can be generated easily for many more games.

8.2 Assessing Accuracy & Performance

I took a number of steps to verify the accuracy of *Solvitaire*'s solutions and solvability percentages. One of the most robust and reassuring measures involved the calculation of solvability percentages for *Baker's Game*, *Black Hole*, *FreeCell* and *FreeCell* variants. Given that for all of these games, each with their own different set of features, the confidence intervals for my results align very closely with those gained elsewhere, it suggests that my implementation is correct and my results reliable. I also implemented a large number of integration tests and unit tests, to further ensure its correctness (see Testing Summary in Appendix B for more details).

To verify the accuracy of my solver when my various optimisations are turned on/off, I also created a python script (`optimisation-test.py`), which runs the solver against a large number of deals of different game types with different optimisations enabled/disabled, and compares whether the solvability of each deal is consistent across the optimisations. I can confirm that running *Solvitaire*'s different optimisations across 9 quite different games, and 10,000 random deals for each, the different optimisations always returned the same solvable/unsolvable classification as each other.

The satisfactory nature of *Solvitaire*'s performance is evident in the solvability percentages it has been able to calculate. Figure 8.1 also includes the number of deals attempted when generating each solvability percentage. Most of these percentages took around an hour to generate; some significantly less, some several hours more. One can see from this that strong performance across a wide range of games has been achieved.

There is not much to compare *Solvitaire* to in terms of performance across a *range* of games, but it is demonstrably competitive with other, game-specific, solvers. Gent et al.[18] use a number of different AI approaches to create solvers for Black Hole, and run each solver across 2500 deals, recording the time taken for each deal. *Solvitaire* provides a `--benchmark` option which can be supplied a preset game type, whereupon it continuously attempts to solve deals for that game, recording the median/mean time taken, the mean/median states searched, and the solvability ratio over all the deals

attempted (this is quite similar, but not identical to the `--solvability` option for calculating solvability percentages). Figure 8.2 shows a comparison of the results of my solver over 2500 random Black Hole deals, versus the three most promising solvers in Gent et al.'s paper The purpose of this comparison is not to see which solver is faster/better, but simply to demonstrate that *Solvitaire*'s performance is indeed of a good standard and is broadly competitive with other solvers (indeed, my results have been generated on different hardware so a strict comparison would not be accurate). Over these 2500 deals my solvability percentage was also very similar.

Figure 8.2: Comparison of *Solvitaire* with other Black Hole solvers

	Gent et al.			Solvitaire
	FF Planner	SAT Solver	FF Planner	
Median sol.	0.88s	1.75s	0.03s	0.046s
Mean sol.	6.83s	3.47s	1.97s	0.043s
(results taken over 2500 random deals)				

I could generate many more results to quantify and assess the performance of *Solvitaire* on specific games here, but beyond what has already been discussed, there is little interest value in this exercise. What is much more relevant is the effects that my different optimisations have on the performance of the solver. This can tell us which are the most important in creating a powerful solver, and in which circumstances different optimisations are most effective. we will see an analysis of this in the next section.

The only feature of *Solvitaire* which doesn't attain strong performance is its method for finding shortest solutions. This particular area turned out to be less important to the project than the ability to generate good solvability percentages, so my efforts in this area were more limited and I was inclined to stick to a solution that made use of my existing DFS implementation. This resulted in my use of iterative-deepening DFS, which as we have already discussed, has some useful properties which make it able to find the shortest solutions, and in theory should not take much longer than regular DSF.

In practice, this was unfortunately not the case. Only on more simple deals of one or two games (*Alpha Star* and *Delta Star*) is it able to find shorter solutions in a reasonable time period. Running this feature over the simplified versions of full games that I define as presets demonstrates that this approach does succeed in finding

shorter solutions, but because in general it performs poorly, it was hard to generate any sort of meaningful results using this feature for full-size games. I must further conclude that it is not a suitable search algorithm for finding minimal solutions for solitaire games.

How could an algorithm with such good properties in theory, be so bad in practice? The performance of iterative-deepening DFS is based on a model of a search tree which expands in a constant manner. This makes the lowest level of the tree by far the largest, and relatively speaking, searching the upper levels is inexpensive and can be done repeatedly with little downside. But for solitaire games this is almost always not the case. Although the search space increases greatly initially, it begins to taper quite quickly as many branches will lead to dead-ends early on in the search. This means that repeating early levels is expensive, and the cost of iterative-deepening DFS is *significantly* more than dfs. Watching the logging output of my iterative-deepening implementation shows exactly this. Running the solver using regular DFS for seed 4 of the game *Delta Star*, it finds a solution at depth 87, and does so in a few seconds. However, with the `--shortest-sols` flag supplied, although it finds a solution at a depth of only 79 moves, *each step* of the iterative search seems to take several seconds, and hence the overall time taken was around 15 minutes. Strong methods certainly exist for finding shorter solutions to deals, but unfortunately for solitaire, iterative-deepening DFS is not one of them.

8.3 Evaluating Optimisations & Streamliners

My solver was implemented with the idea of comparing its different optimisations in mind, which enables me to assess and evaluate them here. The different *Solvitaire* executables generated by my build system, with different optimisations enabled for each, make this particularly easy. Apart from my main, regular *Solvitaire* executable, the various other versions of the solver provided are builds whereby each of the solver's different optimisations are disabled. This allows me to assess how much difference each of them makes towards my solver performing efficiently. I also have two builds where my two streamliners are enabled, and one build where both are enabled. By comparing and assessing these optimisations, others attempting to design solvers might be able to identify which are most important for them to implement; perhaps even for games and applications beyond the domain of solitaire.

8.3.1 All Optimisations Enabled

Firstly, we will begin with an evaluation of the solver with all optimisations (but no streamliners) enabled. These results are not particularly noteworthy in themselves, but provide a point of comparison for later experiments where certain optimisations will be disabled. To perform this evaluation, I again make use of the 'benchmark' mode that *Solvitaire* provides, to assess its performance over a range of deals for particular games. The results of this basic evaluation can be seen in figure 8.3.

Figure 8.3: Benchmark results with all optimisations

Game Type	**Solution Time(μs)**		**States Searched**		**Instances Attempted**	
	Median	**Mean**	**Median**	**Mean**	**Solvable**	**Unsolvable**
Test Black Hole	1067	3181	233	699	480	520
Test Any Suit	185	1238	44	235	401	599
Test Red-Black	252	6240	34	711	297	703
Test Same Suit	1245	6216	155	688	630	370
(results taken over 1000 runs)						

Initially I began my evaluation using a suite of different regular games that my regular solver was able to solve instances of without too much difficulty. However, there was a serious problem with this approach: most of these games are only tractable because of the effects of all my optimisations working together to make search more efficient. Disabling any optimisations meant that in some circumstances it was almost impossible to find deals that could be solved in any reasonable length of time, let alone run the solver oven hundreds of deals to generate averages. Unfortunately, this means comparing the effects of my optimisations on a group of real games is not feasible. Consequently, I decided to create four simple, bespoke games, designed to be simple enough that even without optimisations my solver is able to solve deals in a reasonable length of time. The JSON for the rules which define these four games, can be seen in listing 8.1.

These four games between them capture several different properties, which will be useful in assessing how enabling/disabling my optimisations affects different games. The Black Hole-based test game is really just a scaled-down version of the original game. The fact that it has a hole rather than foundations is significant because of its effect on symmetry. The other three games are all quite similar, but firstly have different build policies. This again allows us to see how different build policies interact with different optimisations through symmetry. Additionally, I wanted to have all of

```
"test-game-black-hole": {
  "tableau piles": {
    "count": 7,
    "build policy": "no-build"
  },
  "max rank": 8,
  "hole": true,
  "foundations": false
}

"test-game-red-black": {
  "tableau piles": {
    "count": 6,
    "build policy": "red-black"
   suit"
  },
  "max rank": 7
}
```

```
"test-game-any-suit": {
  "tableau piles": {
    "count": 4,
    "build policy": "any-suit"
  },
  "max rank": 5
}

 "test-game-same-suit": {
   "tableau piles": {
     "count": 10,
     "build policy": "same-

   },
   "max rank": 9
 }
```

Listing 8.1: Rules for bespoke evaluation games

my games take around the same amount of time to run (in the order of roughly a few milliseconds per deal), and have roughly the same solvability percentage (all somewhere around 50/50). But because different build policies make games easier or harder, to achieve these goals I had to alter the max-rank and number of piles in the different games. Hence to achieve the same solve-time and solvability for each game, I had to make those with the more restrictive build policies larger (more piles and a larger max-rank), and those with less-restrictive policies smaller. This gives me the additional advantage of being able to compare how the different optimisations interact with games of different max-rank and pile count.

For my evaluations, all of my results are generated over the same 1000 random deals, and the mean and median time and states searched are recorded. Tracking the number of states lets me see if different optimisations change the nature of the search itself, and tracking the time indicates both the effect on the search algorithm, and the efficiency of my implementation. Looking at both the median and the mean is particularly interesting, as the former measure is not affected by particularly hard 'outlier' deals, while the latter very much is. Thus if the two values are very different it suggests that the particular game/optimisation is prone to getting stuck on very hard deals, even if its average-case performance is still good. Finally, I also take note of the number of deals out of the 1000 each time that are solvable/unsolvable. For all of my optimisations these values are consistent across games, which further indicates

their validity. When we come to looking at streamliner performance, this will also indicate the extent to which the they sacrifice accuracy for performance.

8.3.2 Without 'Reduced Game-State' Optimisation

Figure 8.4 shows us what happens when the 'reduced game-state' optimisation is disabled. The results shown in brackets are the change relative to the results for the regular solver, and I highlight those results which are changed significantly. This optimisation was explored in section 6.3, and involves shrinking the size of the game state into a compact version, based on the fact that the cached version of the state doesn't need to contain all of the search-based data structures that the main one does. This is the only of my optimisations which doesn't affect the *nature* of the solver's search, but rather is an implementation efficiency. This efficiency affects the performance whenever we cache a state, and there are no games to which it cannot be applied.

Figure 8.4: Benchmark results without the 'reduced game-state' optimisation

Game Type	**Solution Time(μs)**		**States Searched**		**Instances Attempted**	
	Median	**Mean**	**Median**	**Mean**	**Solvable**	**Unsolvable**
Test Black Hole	3241 (+2174)	14855 (+11674)	233	699	480	520
Test Any Suit	343 (+158)	2040 (+802)	44	235	401	599
Test Red-Black	459 (+207)	8104 (+1864)	34	711	297	703
Test Same Suit	2250 (+1005)	10765 (+4549)	155	688	630	370
(results taken over 1000 runs)						

Hence, we see that all of the games suffer an increase in the times taken to reach solutions. The first and last games in the table are the worst affected, and it is no coincidence that in the original results these games had the most average search states. This means that the amount of copying done into the cache done for these games in the same time period as the others is greater, and thus its no surprise that these games slow down the most. However, it is worth bearing in mind that the performance hit these games take without this optimisation should scale linearly with the size of the game (we will see later ones that scale much worse), which means that although not having this optimisation seems to double or triple the time taken, this effect is not significantly worse for full-size games. It is also worth noting here that the rest of the table contains identical results. In other words, the states searched

and the classifications of deals are identical, which is what we would expect as we are just altering the sate representation.

In terms of how much this optimisation alters the size of the game state, we can quantify this exactly. Debugging the solver and inspecting the internal representation of the optimised and non-optimised versions shows that the compressed state uses three eight-byte pointers to index the main data vector (this cannot be a fixed-size array as we do not know the size of the game to be solved until runtime), and I reserve an initial 70 bytes within this vector for cards and pile separators (which both take up a byte each). This is enough for 52 cards and 18 piles, which is sufficient for almost all games. In total then, this compressed representation involves copying 94 bytes into memory at each step for caching purposes.

In comparison, the non-optimised state representation uses the same 24 bytes to index the main data vector, but then another 24 bytes for each of the pile vectors within the main data vector. If we assume a game of FreeCell, with 8 piles, 4 cells and 4 foundations, this gives a total of 12 piles, meaning another 384 bytes to be copied. Plus another 52 bytes for the cards, and we end up having to copy 460 bytes at each step, which is almost four times as much work for the solver to do throughout. For the games in figure 8.4 there are slightly fewer piles, which is why the increase in time taken is of a slightly lower magnitude, but still in line with what we would expect given how this optimisation alters the implementation.

8.3.3 Without 'Suit Symmetry' Optimisation

We explored suit symmetry in section 6.3.1.2, and saw how it could be used to eliminate searching branches in a number of situations in which cards of the same rank but different suit behave in identical ways. I have also produced evaluation results for the solver without using this symmetry, which can be seen in figure 8.5.

Note that removing this optimisation only seems to affect the Black Hole-type games. For the other games, the only changes are small natural variations in solution times. We saw previously how the presence of foundations means that this optimisation isn't quite valid, so my implementation doesn't allow suit symmetry for these games regularly, which explains why the other three games are unchanged. However, for the Black Hole-type game the time changes *are* significant, and we also see an associated increase in the number of states searched.

We've already analysed how the *combination* of suit and pile symmetry can eliminate search states, but now without suit symmetry there is no search that can be

Figure 8.5: Benchmark results without the 'suit symmetry' optimisation

Game Type	Solution Time(μs)		States Searched		Instances Attempted	
	Median	Mean	Median	Mean	Solvable	Unsolvable
Test Black Hole	1397 (+330)	4711 (+1530)	276 (+43)	955 (+256)	480	520
Test Any Suit	177 (+8)	1283 (+45)	44	235	401	599
Test Red-Black	248 (-4)	4291 (+51)	34	711	297	703
Test Same Suit	1218 (-27)	6099 (+117)	155	688	630	370
(results taken over 1000 runs)						

eliminated. The pile symmetry that remains enabled here makes no difference, as in Black Hole, cards can't move between tableau piles, and without suit symmetry these piles are guaranteed to be unique. Hence no symmetry exists whatsoever. This is what results in the increase in search times/states that we see in this table.

8.3.4 Without 'Pile Symmetry' Optimisation

It has been suggested thus far that symmetry-breaking in Black Hole relies upon the combination of suit and pile symmetry. This is largely true, but my evaluation of *Solvitaire* without using pile symmetry revealed an interesting type of suit symmetry that I had not previously considered in the game. The results for this evaluation can be seen in figure 8.6.

Figure 8.6: Benchmark results without 'pile symmetry' optimisation

Game Type	Solution Time(μs)		States Searched		Instances Attempted	
	Median	Mean	Median	Mean	Solvable	Unsolvable
Test Black Hole	902 (-165)	2893 (-288)	244 (+11)	772 (+73)	480	520
Test Any Suit	187 (+2)	1121 (-117)	49 (+5)	259 (+24)	401	599
Test Red-Black	234 (-18)	55271 (+49031)	35 (+1)	9774 (+9063)	297	703
Test Same Suit	54726	2102786	4245	194884	35	(intract.) 12
(results taken over 1000 runs)						

The results for Black Hole are equivocal: it has taken less time, but searched more states than the regular solver. The reduction in time is due to the small overhead in implementing pile symmetry that has been removed. As for the number of states searched, we might expect the increase here to be the same as when suit symmetry is disabled. This would suggest that there is no symmetry for either, as is both cases

we can't combine suit and pile symmetries to eliminate states. Yet this is not the case. What we actually see is a smaller increase in the number of states we have to search when pile symmetry is disabled. This indicates that in fact, using only suit symmetry, there still exists kind of symmetry that the solver can use to eliminate searching states, without having to rely on pile symmetry.

It took some consideration to identify why this is the case. After all, it shouldn't matter if suit-symmetrical piles are given same hash, if (without pile symmetry) they are still considered independent and in a fixed order. However, this does not take into account suit-symmetry that takes place within the *hole* pile. This is best seen in an example. Consider a game of black hole, where there is an A♠ in the hole, and a 2♥, a 3♦, and a 2♣ available on the tableau piles, which are then moved into the hole. Regardless of which two is moved onto the hole first, the tableau piles are left in the same state after these moves. If the twos hash to the same value, then a search with suit symmetry will consider these two branches identical and will only have to search one. Here we have an instance of suit symmetry that doesn't rely at all on pile symmetry. Hence, we can conclude that for hole-based games, the best results can be gained from considering both symmetries in combination, but on their own suit symmetry is able to eliminates states, whereas pile symmetry is not.

As for the other games here, a lack of pile symmetry seems to affect the red-black game to a reasonable extent, and makes the same-suit game completely intractable (even at this small scale). I was only able to run 47 deals for it within a reasonable time window, and the values I recorded in that period were very high indeed. This is perhaps unsurprising, as these games have the most piles, and it is likely that in this games there is more potential for pile symmetry. Interestingly, for the red-black games we see the median time / states searched largely unaffected, but the mean for both jumps very significantly. This indicates the presence of very difficult deals–which are what skew the mean–that are no longer made easy to solve through the use of pile symmetry. If we scale these games up to their full-size equivalents, one finds that without pile symmetry, the solver is very poor and can find few tractable instances for most games, which is just as these evaluation results suggest. Hence we can say that although for Black Hole suit symmetry may be more useful, for most other games pile symmetry is an extremely important consideration, whereas suit symmetry usually doesn't apply.

8.3.5 Without Either Symmetry Optimisation

I have also evaluated the solver when both optimisations are disabled. Results for this can be seen in figure 8.7.

Figure 8.7: Benchmark results without either symmetry optimisation

Game Type	Solution Time(μs)		States Searched		Instances Attempted	
	Median	Mean	Median	Mean	Solvable	Unsolvable
Test Black Hole	1160 (+93)	4023 (+842)	276 (+43)	955 (+265)	480	520
Test Any Suit	183 (−2)	1025 (−213)	49 (+5)	259 (+24)	401	599
Test Red-Black	215 (−37)	52967 (+46727)	35 (+1)	9774 (+9063)	297	703
Test Same Suit	544321	2101722	4245	194884	35	(intract.) 12
(results taken over 1000 runs)						

This confirms much of what we already suspect. We see that for the Black Hole-type game the results are essentially the same as when there is no suit symmetry, proving my assertion that without suit symmetry, turning pile symmetry on and off makes no difference. And as for the other game types, the results seen here are essentially the same as when there is no pile symmetry, which is to be expected as suit symmetry does not apply to these games.

8.3.6 Without 'Auto-Foundations' Optimisation

The final optimisation to be considered is the 'auto-foundations' optimisation presented in section 6.3.2, which involves moving cards up to the foundations as dominance moves, when it can be proved that they are no longer needed in the tableau piles. The results for this evaluation can be seen in figure 8.8.

Figure 8.8: Benchmark results without 'auto-foundations' optimisation

Game Type	Solution Time(μs)		States Searched		Instances Attempted	
	Median	Mean	Median	Mean	Solvable	Unsolvable
Test Black Hole	1166 (+99)	3311 (+130)	233	699	480	520
Test Any Suit	298 (+114)	2752 (+1514)	66 (+22)	516 (+281)	401	599
Test Red-Black	525 (+273)	9349 (+3109)	82 (+48)	1649 (+938)	297	703
Test Same Suit	4892 (+3627)	9424 (+3208)	659 (+504)	1298 (+610)	630	370
(results taken over 1000 runs)						

These results need less interpretation than the previous optimisations. Nothing changes for the Black Hole-type game, which does not have foundations, and for all other games there is a reduction in performance without this. This performance decrease is quite significant, and in practice, this optimisation helped make several full games tractable that would not have been otherwise. This was an optimisation that could easily have removed solutions if implemented wrongly, so it is reassuring to see that in the final column, my solvable/unsolvable counts are the same with and without it.

8.3.7 'Suit Reduction' Streamliner

My 'suit reduction' streamliner, as explained in section 6.3.3.1, implements the suit symmetry optimisation we have already evaluated. However, it does so for games with foundations, where strictly speaking this symmetry doesn't apply, but we suspect ignoring this fact could give a considerable performance boost without greatly damaging the solver's accuracy. The results for my evaluation of this streamliner can be seen in figure 8.9.

Figure 8.9: Benchmark results using 'suit reduction' streamliner

Game Type	**Solution Time(μs)**		**States Searched**		**Instances Attempted**	
	Median	**Mean**	**Median**	**Mean**	**Solvable**	**Unsolvable**
Test Black Hole	1153 (+86)	3410 (+229)	233	699	480	520
Test Any Suit	168 (−17)	281 (−957)	40 (−4)	61 (−174)	396 (−5)	604 (+5)
Test Red-Black	275 (+13)	1850 (−4390)	34	204 (−507)	295 (−2)	705 (+2)
Test Same Suit	1276 (+31)	6465 (+249)	155	688	630	370
(results taken over 1000 runs)						

There is no difference here for the Black Hole-type and same-suit games. For the former, this symmetry already applies, so this has no effect, and for the latter, a 'same-suit' build policy means that there can never be any kind of suit symmetry. For the any-suit game, the streamliner enables symmetry across all suits, and consequently we see it is able to give a very large performance boost. The downside here is that there are five deals that have been misclassified. Yet out of 1000 in total, this give us 0.5%, which is fairly insignificant. For the red-black game, there is less possibility for symmetry, and in fact this streamliner does affect the median time/states searched significantly. However, it does reduce the mean by a surprising amount,

which indicates that it still has a significant effect in helping with the most difficult instances, and does so again with a tiny trade-off in accuracy.

8.3.8 'Auto-Foundations' Streamliner

We have already evaluated the auto-foundations optimisation, and we can now do the same for the streamliner version of this feature, as examined in section 6.3.3.2. Here, we assume that cards that can go up, always should do, by means of a dominance move. The results for my evaluation of this streamliner can be seen in figure 8.10.

Figure 8.10: Benchmark results using 'auto-foundations' streamliner

Game Type	**Solution Time(μs)**		**States Searched**		**Instances Attempted**	
	Median	**Mean**	**Median**	**Mean**	**Solvable**	**Unsolvable**
Test Black Hole	1039 (+28)	3091 (−90)	233	699	480	520
Test Any Suit	114 (−71)	693 (−545)	37 (−7)	146 (−255)	382 (−19)	618 (+19)
Test Red-Black	200 (−52)	2643 (−3597)	28 (−6)	353 (−358)	261 (−36)	739 (+36)
Test Same Suit	1235 (−10)	5962 (−54)	155	688	630	370
(results taken over 1000 runs)						

Again, the Black Hole-type and same-suit games are not affected. This is because the former doesn't have foundations at all, and the latter's same-suit policy means that it already puts all available cards up automatically. We see for this streamliner too, a significant boost in the performance of the solver. This is based on the fact that removing these cards from the tableau piles permanently when we can, reduces the number of cards in play and hence the size of the search space.

Versus the 'suit-reduction' streamliner, the decrease in median solution time/states searched is more significant, the decrease in the mean is less. The accuracy is also slightly worse here, with 1.9% and 3.6% of any-suit and same-suit games misclassified respectively, versus 0.5% and 0.2% for the 'suit-reduction' streamliner. Comparing the two, I would argue that the 'suit-reduction' streamliner is generally more effective, as reducing the mean search time is more important than the median, as this indicates that it will help more with intractable deals, which are what really give us problems when generating solvability percentages. Moreover, its accuracy is significantly better. Nevertheless, both have proven to be powerful streamliners, and both are largely independent of one-another. Thus we can combiner the two of them, which as we shall see, can be done to great effect.

8.3.9 Both Streamliners

The results for using both streamliners at once can be seen in figure 8.11.

Figure 8.11: Benchmark results using both streamliners

Game Type	Solution Time(μs)		States Searched		Instances Attempted	
	Median	Mean	Median	Mean	Solvable	Unsolvable
Test Black Hole	1075 (+8)	3191 (+10)	233	699	480	520
Test Any Suit	101 (-84)	190 (-1048)	35 (-9)	49 (-186)	376 (-25)	624 (+25)
Test Red-Black	208 (-44)	1334 (-4906)	28 (-6)	148 (-563)	256 (-41)	744 (+41)
Test Same Suit	1216 (-29)	5988 (-228)	155	688	630	370
(results taken over 1000 runs)						

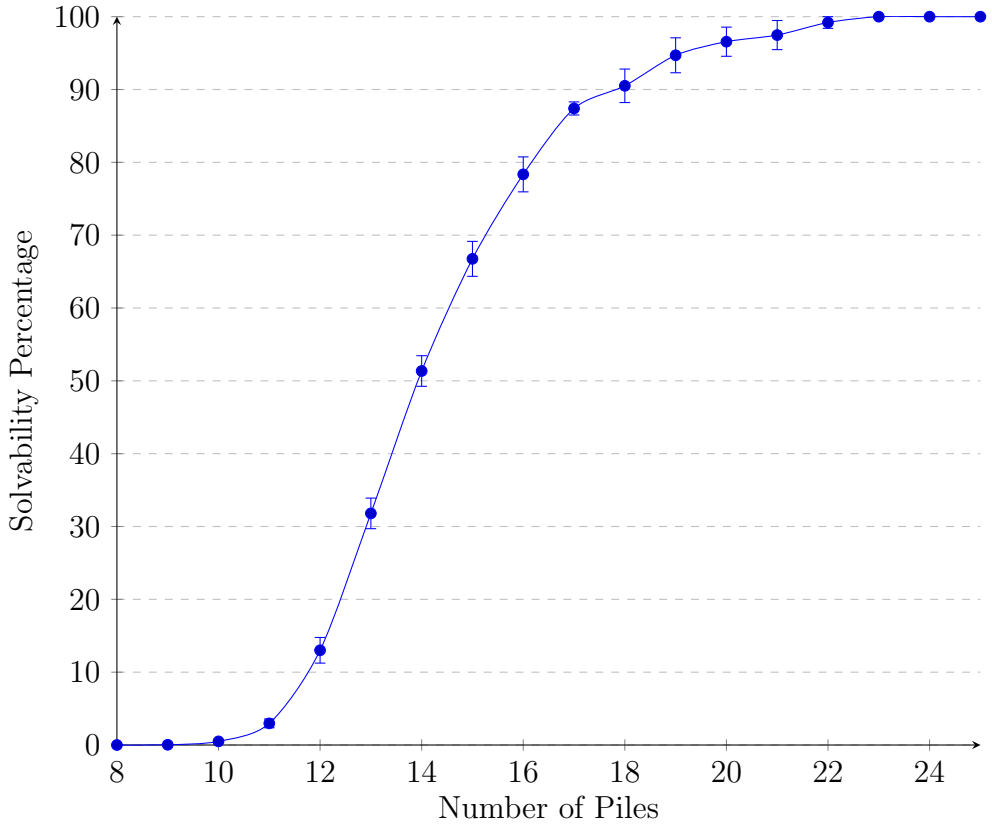
Combining these streamliners gives us a significant reduction in the time taken and states visited for the games where these techniques are applicable. Our accuracy is certainly not ideal, but there are still many scenarios where this trade-off is probably worthwhile. This was certainly the case when it came to calculating solvability percentages for games that my solver usually finds intractable. Using these streamliners was the only way these games could be approached. Although one can only rely on the lower bounds of streamliner-generated confidence intervals (due to the unreliability of streamliners' 'unsolvable' classifications), we can see the from the accuracy of this combined streamliner here that it is probably not unfair to assert that the lower bounds I have calculated, aren't too far off the actual solvability percentages for these games.

8.4 Analysing Game Variants

Finally, we will turn our attention briefly to some of the applications having such a flexible way of generating solvability percentages provides. Because these percentages can be generated for anything legal defined in a rules JSON file, it is trivial to assess what happens to the solvability of 'regular' games, if certain parameters are tweaked. This is of great interest to players. Often beginners will find the basic variants of games too challenging, and expert players may also find the same game too easy. They may therefore desire to play a version of the game more suited to their ability level. However, until now, there has been no way of players ascertaining (except through imprecise and laborious trial and error) what happens if they alter the rules of a game.

We saw an example for the game FreeCell, where experimental results have shown how changing the number of cells and piles affects the solvability (figure 2.1), and I also verified these results myself (figure 8.1). But now we have a much more general way of doing this, which can be applied to any game that *Solvitaire* can solve in reasonable time. To demonstrate this–and because the results themselves are rather interesting–I have calculated how the solvability percentage changes for the game Black Hole, when its number of tableau piles is altered. This can be seen in figure 8.12.

Figure 8.12: Difficulty of variants of the game Black Hole



These results assume that where piles are added/removed from the regular (17-pile) game, the cards are spread as evenly as possible over the remaining piles. The raw data for this experiment can be found in my `results/black-hole-variants` directory. For each I display the 95% confidence interval for the solvability percentage. Note that for the 17-pile game, this graph also makes use of the solvability percentage already calculated for regular Black Hole.

We can conclude from these results that for beginners, increasing the number of piles to 20 or more, should guarantee that almost all deals are actually solvable,

meaning that persevering with difficult deals will almost never be in vain. And for expert players, removing three of the piles should make the game around 50% solvable, which would make a good target for a player's own win rate. No solvable deals have been found thus far (with 300,000 attempted) for the 8-pile variant of the game or fewer, although some have been found for the 9-pile. And no unsolvable deals have been found for the 22-pile game or above either (although only a few hundred deals have been searched for this variant, as it is generally more expensive).

The same information generated here for variants of Black Hole, could be done for any feature of any game type my solver finds tractable. To do so is simple using *Solvitaire*, and for games which are particularly difficult, lower bounds for the solvability of variants could at least be calculated, using the same technique but with streamliners applied.

8.5 Fulfilment of Project Requirements

Having seen the full details of my design and implementation, and an evaluation of *Solvitaire*'s performance and features, we can now assess the extent to which this project fulfils its stated requirements.

8.5.1 Primary Requirements

An application for solving single-player, perfect information solitaire instances: This has certainly been achieved. The solver is able to fully solve games, and for reasons we have explored, we can be confident of its accuracy.

Some kind of schema or language for describing a wide range of solitaire games: The JSON rules feature that I have implemented indeed enables a wide range of games with many different features to be solved, with many games also built in as presets.

The application must be able to solve valid games expressed using the schema/language: That *Solvitaire* is able to solve the games expressed using this JSON is evident in the solvability percentages I have calculated using many different JSON rule descriptions.

Automated testing to demonstrate the correctness of my classifications: I have mentioned in passing the large amount of testing that I have done for my

project, using both unit tests and integration tests. A full overview of these can be found in my testing summary in Appendix B.

An implementation of search techniques such as 'meta-moves'/'super moves' and eliminating symmetrical states: The idea of implementing 'meta-moves' was abandoned as it is not useful for exhaustive search, and is really only applicable to heuristic solvers. However, as has been demonstrated in this report, I have done much work and analysis for both suit symmetry, and pile symmetry.

An implementation of language-level/design optimisations to improve the efficiency of my search: This requirement is fulfilled in my 'state reduction' optimisation (as well as several other smaller optimisations), which I have shown to have improved the performance of my search significantly, and have evaluated in depth.

An analysis in my report of the effects of the above search techniques and implementation choices on the performance of the solver: The extensive evaluation and comparison in this section of all my optimisations and streamliners on the same set of benchmark games/deals, hopefully satisfies this requirement fully.

8.5.2 Secondary & Tertiary Requirements

Optimise the search algorithm to eliminate searching states in which certain moves/states 'dominate' others: I have fulfilled this objective through the use of my 'auto-foundation' moves, which make use of dominances to eliminate search.

Implement and evaluate more than one different type of search and **An extension of the algorithm that aims to provide shortened solutions:** Although the my project focussed less thoroughly on these requirements, I did use a different kind of search in the form of iterative-deepening, to enable the finding of minimal solutions. While the performance of this technique was not satisfactory, I was able to indicate how and why this technique is unlikely to be of use for those wishing to find short solutions to solitaire deals.

Calculating the solvability ratio for some the games my application can solve: This was a particular focus of mine once the basic implementation of the solver was complete. I aimed to satisfy this particularly thoroughly, and through my calculation of a large number of solvability percentages and my design for my timeout-scaling algorithm, I believe this tertiary requirement has been met in full.

In addition, although it was not a stated requirement, I also implemented streamliners for my search, which enabled me to find lower bounds for the solvability of games that would otherwise have been intractable.

9 Conclusions

Through developing *Solvitaire* I have created a powerful solitaire solver which, unlike most other solvers, is capable of solving a *wide* range of games with much success. The feature that enables users to describe the rules of a game to the solver is both simple and expansive. My solution is well tested, demonstrably accurate, and makes the most of a range of powerful search techniques to enable it to solve deals. Its performance is similar to that of other good, game-specific solvers, and it can solve both user supplied deals, and randomly generated ones.

The most significant feature of my project has been the ability of my solver, because of its efficiency and generality, to produce solvability percentages for eighteen games where until now the solvability percentage has not been known. In addition, I was able to use an implementation of streamliners to find lower bounds for the solvability percentages of particularly difficult games. Not only have I been able to calculate these new percentages, but in the form of *Solvitaire*, I have created a tool which will make doing this a simple exercise for many more games. My calculated solvability percentages also make use of a novel timeout-scaling algorithm which balances reducing the number of intractable deals against getting through deals quickly.

I have also shown that when attempting to find minimal-length solutions to deals, iterative-deepening depth-first search is not a suitable method. One possible area for future work would be to implement other search algorithms for *Solvitaire* to find short solutions faster. However, this project has demonstrated that doing so requires a quite different approach to search than that which is needed to calculate solvability percentages.

Much of my design and evaluation for this project focused on potential optimisations, specifically symmetries and dominances. I have shown how suit symmetry and pile symmetry can be implemented for a solitaire solver, and also how foundation moves can implemented as dominances. My evaluation of the solver with these optimisations turned on and off, has shown that each, for the right game type, is a powerful tool for reducing search.

Suit symmetry has been shown to be a effective for hole-based games, working particularly well in combination with pile symmetry, but also in some circumstances without it. Pile symmetry is a more generally applicable symmetry, and is key to

making a large number of game types tractable. The auto-foundations dominance gives good performance generally across games that have foundations, and I have shown in depth the conditions in which it is safe to be used. I have also evaluated the performance of streamliners based on some of these optimisations, and shown how they can be used to gain a large increase in performance, with relatively little trade-off in accuracy. Between the 'auto-foundations' streamliner and the 'suit reduction' streamliner, the latter has been shown to be slightly more effective. I also demonstrate a trick to reducing the amount of data copied for caching at each step in the algorithm, which makes a significant improvement for the solver's performance across all games.

Finally I provide an additional example application of *Solvitaire* for analysing variations of existing games. Taking the game Black Hole, I show how its solvability percentage changes as the number of piles is increased/decreases. Because of the simplicity of supplying custom-games to *Solvitaire*, this type of analysis is easy to do for any game which the solver finds tractable.

This would be another good avenue for future work. Were I to continue my development of *Solvitaire*, I would be very interested to see how altering different game features affects solvability percentages. The outcome of this would be to use results generated by *Solvitaire* to try and build up a more precise model for how the presence and size of certain game features, interact with other features in terms of their effect on the solvability percentage. One would then perhaps be able to design games with a good idea of their solvability percentage, without even having to run a solver to generate experimental results. Achieving this goal however, requires a great deal more research to be done.

A Solitaire Terminology

On the following page is an overview of the terminology used to refer to features of solitaire games used in this report, and used in the schema which defines the rules which can be supplied to *Solvitaire* (see Appendix C). This terminology varies widely in different publications and solvers, so one can expect to find different terms used elsewhere. Additionally, solitaire itself is sometimes known (particularly in the UK) by the name 'Patience'.

Feature	Description
Tableau	The main 'playing area' of the game where cards typically start. The tableau consists of a number of piles, which cards are usually moved between.
Pile	A stack of overlapped cards, from which only the top one can typically be taken to move onto other piles. Cards can also be moved onto a pile depending on its 'build policy'.
Build Policy	Determines which cards can be moved onto a pile. Options are 'no build', where building is forbidden, 'same suit', 'red-black' or 'any suit', where cards can move onto a pile of rank one greater if its suit corresponds with the card it is moving on top of, according to the policy.
Built Pile Moves	Games which allow these moves enable players to move several cards from a pile in a single 'block', and put them on another pile if they are all of a rank and suit that satisfies the build policy.
Foundations	Four piles with specific suits designated to each. The aim of a game with foundation piles is to move all cards onto the foundations. Some games allow cards to be removed from the foundations, while others do not.
Complete-pile Foundations	A special type of foundation where cards can only be moved to it in a single 'block' from king to ace of the same suit.
Hole	An alternative to foundations, where the aim of the game is to move all cards onto the hole. Cards can be moved onto the hole if they are of rank one greater or one less than the hole, regardless of suit. This wraps from ace to king.
Cell	A type of pile where only one card is allowed at a time. In other words, cards cannot be stacked on a cell. Cells typically start empty.
Reserve	Similar to a cell, but a reserve typically starts with a card in it. However with a reserve, once a card is removed, none can be placed back on it.
Stacked Reserve	A type of reserve pile which contains multiple stacked cards, where only the top one can be moved. Cards cannot be added still.
Stock	A pile which typically begins with a large number of cards, and onto which no cards can be placed. Similar to a stacked reserve, but with some special features.
Stock Deal Type	Specifies where cards from the stock can be moved. Either to a special 'waste' pile, or to the top of all tableau piles at once
Waste	Dealt onto by the stock. Cards can be moved from the waste to other piles, but it cannot be built on.
Stock Deal Count	The number of cards dealt from the stock to the waste at-a-time. The standard value is typically one or three.
Stock Redeal	If the stock deals onto the waste, a redeal means that one the stock is 'out', the waste is picked up, turned over, and becomes the stock again.

B Testing Summary

B.1 Testing

I mention in the main report the general approach I take to software development, which is based around the Agile methodology, and thus is heavily reliant on testing to ensure a working product at all times. Here I will give a more thorough overview of the specific testing that I have done to ensure the correctness of my code. I make good use of the Google Test framework here, which enables me to write clear and simple unit tests and integration tests.

B.1.1 'Canonical' Games

Before we look at these tests in depth, we must first examine a group of games which I describe as 'canonical' games for *Solvitaire*. These were the first games I implemented, and they are as follows: alpha-star, bakers-dozen, black-hole, canfield, flower-garden, fortunes-favor, free-cell, somerset, spanish-patience, spider. Each of these games was chosen because they all introduce new and different features to one another, and few have features in common (besides the most basic features). For example, alpha-star allows built-group moves, whereas FreeCell doesn't, but it implements cells instead.

Hence, focusing my testing on these games in particular, enables me to isolate particular features. Hence, if my FreeCell test are the only ones failing, it suggests that there may be a problem with my cells implementation. Testing on full-size games was problematic, as solving deals for these can often take some time. To solve this problem, I added preset games to my solver which implement 'simple' scaled-down versions of my canonical games, that are all easily tractable. I was then able to implement tests using these versions of the games.

At various stages, particularly early on, it was also useful to see how well the solver performed on some of these canonical games, as their features were scaled up to become full-size. For this purpose, I designed the `benchmark.py` script, which scales up the size of these canonical games types gradually for each game until the solver begins to find deals intractable, and then outputs the results.

B.1.2 Unit/Integration Testing

For my actual tests themselves, I put a lot of my focus on creating good integration tests. These are based around the 'simple' versions of the aforementioned canonical game types. For each one of these ten game types, I wrote four types of integration tests based around bespoke starting deals. These were as follows:

- Simple solvable instances
- Simple unsolvable instances
- Complex solvable instances
- Complex unsolvable instances

These starting deals I came up with by hand, carefully formulating deals that were both obviously solvable/unsolvable, and also deals which took a little more effort to prove were solvable/unsolvable. I then wrote a testing function that enables me to create unit tests from these custom deals, which checks to see if the solver agrees with my classifications.

Here is an example for the game FreeCell, of my two complex instances:

```
{
  "tableau piles": [
    ["AC","2C","4C","3C"],
    ["AD","2D","4D","3D"],
    ["AH","2H","3H","4H"],
    ["AS","2S","3S","4S"]
  ]
}
```

Listing B.1: Complex solvable test deal for Simple FreeCell

```
{
  "tableau piles": [
    ["2S","2C","4D","3C"],
    ["2H","2D","4C","3D"],
    ["AH","AD","3S","4H"],
    ["AS","AC","3H","4S"]
  ]
}
```

Listing B.2: Complex unsolvable test deal for Simple FreeCell

The two are very similar, but if one steps through the logic of a solution for both (assuming a single free cell in the game), one will see that the former is solvable, whereas the latter is not. I spent a lot of time figuring out these integration test deals,

and came up with 40 in total across the 10 canonical game types, each one targeting a different feature. These test deals can be seen in my `src/test/resources` directory.

As for my unit testing, all of my important classes have their own range of unit tests, to ensure that they provide the required functionality. These can be found in my `src/test/unit_tests` directory. Firstly, I have test classes designed to ensure the correctness of my JSON schemas. These exist for both my rules schema, and my deals schema. For each of these, I have unit tests to target specific features made available by the JSON schemas, ensuring that they are accepted by `Solvitaire` correctly.

I then have a range of testing for the my core game representation classes, ensuring that cards and the piles within which they are manipulated, all provide the correct operations. I also test the functions used to construct these classes, and to move cards from one pile to another.

In addition, I have a class which defines the set of legal moves between different piles, depending on different game rules. I have a set of unit tests for this class too, which looks at a large set of the different options for game types that *Solvitaire* can handle, and ensures that the corresponding logic for which cards can move where is implemented correctly (e.g. checking build policies work as expected, built group moves, etc.). This set of tests in particular caught many implementation bugs as I went along.

I also have unit tests for my cache, upon which a lot of my symmetry is implemented. I wrote these tests before implementing any symmetry, then filling in the implementation to make my tests pass. Finally, I have a set of tests for my foundation dominance moves. These test that, for a simple solitaire game, the auto-foundation moves for different build policies work in different ways, according to the particular set of allowed foundation moves for each build policy. Overall I have more than a hundred different unit tests that can be run to ensure the correctness of my implementation.

I also mention in the main report that I have written a python script (`optimisation-test.py`), which runs the solver against a large number of deals of different game types with different optimisations enabled/disabled, and compares whether the solvability of each deal is consistent across the optimisations. Running *Solvitaire*'s different optimisations across 9 quite different games, and 10,000 random deals for each, the different optimisations always returned the same solvable/unsolvable classification as each

other. This demonstrates the correctness of my optimisation implementations. Moreover, comparing my generated solvability percentages to those of published results, I get the same values as they do, which further indicates the accuracy of *Solvitaire*.

These unit-tests must be run from within my `src/tests` directory, where the user should simply enter `./run_unit_tests`.

C User Manual

C.1 *Solvitaire* Usage

The usage of my program and its runtime options are as follows;

```
Usage: solvitaire [options] input-file1 input-file2 ...
options:
  --help                      produce help message

  --type arg                  specify the type of the solitaire game
                              to be solved from the list of preset
                              games. Must supply either this 'type'
                              option, or the 'custom-rules' option.

  --available-game-types      outputs a list of the different preset
                              game types that can be solved.

  --describe-game-rules arg outputs the JSON that describes the
                              rules of the supplied preset game type.

  --custom-rules arg          the path to a JSON file describing the
                              rules of the solitaire to be solved.
                              Must supply either 'type' or
                              'custom-rules' option.

  --random arg                create and solve a random solitaire
                              deal based on a seed. Must supply
                              either 'random','solvability',
                              'benchmark' or list of deals to be
                              solved.

  --classify                  outputs a simple 'solvable/not solvable'
                              classification.

  --shortest-sols             for each instance returns the shortest
                              possible solution. cannot be supplied
                              alongside '--solvability'.

  --solvability               calculates the solvability percentage of
                              the supplied solitaire game. Must supply
                              either 'random', 'benchmark',
                              'solvability' or list of deals to be
                              solved.

  --benchmark                 outputs performance statistics for the
                              solver on the supplied solitaire game.
                              Must supply either 'random',
                              'benchmark', 'solvability' or list of
```

```
                                deals to be solved.
```

Most of these options should be self explanatory, and several are referenced specifically in the body of the report if one wishes for a more thorough overview of why a particular option exists. The following game types are available to be used with the `--type` option:

```
alina
alpha-star
bakers-dozen
bakers-game
black-hole
blind-alleys
canfield
castles-of-spain
chameleon
default
delta-star
duchess
east-haven
eight-off
eight-off-any-card-spaces
fan
flower-garden
fore-cell
fortunes-favor
free-cell
free-cell-0-cell
free-cell-1-cell
free-cell-2-cell
free-cell-3-cell
free-cell-4-pile
free-cell-5-pile
free-cell-6-pile
free-cell-7-pile
king-albert
klondike
klondike-deal-1
martha
northwest-territory
one-cell
penguin
raglan
scotch-patience
sea-towers
simple-alpha-star
simple-bakers-dozen
simple-black-hole
simple-canfield
simple-flower-garden
simple-fortunes-favor
simple-free-cell
simple-somerset
```

```
simple-spanish-patience
simple-spider
somerset
spanish-patience
spider
spiderette
test-game-any-suit
test-game-black-hole
test-game-red-black
test-game-same-suit
two-cell
will-o-the-wisp
```

C.2 Schemas for rule and deal JSON

Looking at some of the json descriptions for these preset game types using the `--describe-game-rules` option (particularly the 'deafult' game type which uses all possible features), one can get a fairly good overview of the different options available when supplying a custom game to *Solvitaire*.

For a thorough overview of the different options that one can supply, see the following schema:

```
{
  "\$schema": "http://json-schema.org/draft-04/schema#",
  "description": "JSON Schema representing a solitaire game",

  "type": "object", "properties": {

    "tableau piles": {
      "type": "object", "properties": {
        "count": {"type": "integer", "minimum": 0},
        "build policy": {"type": "string", "enum":
                ["any-suit", "red-black", "same-suit", "no-build"]
        },
        "spaces policy": {"type": "string", "enum":
                ["any", "no-build", "kings"]
        },
        "diagonal deal": {"type": "boolean"},
        "move built group": {"type": "boolean"},
        "move built group policy": {"type": "string", "enum":
                ["same-as-build", "any-suit", "red-black",
                 "same-suit", "no-build"]
        }
      }, "additionalProperties": false
    },
    "max rank": {"type": "integer", "minimum": 1, "maximum": 13},
    "two decks": {"type": "boolean"},
    "hole": {"type": "boolean"},
    "foundations": {"type": "boolean"},
    "foundations initial card": {"type": "boolean"},
```

```
    "foundations removable": {"type": "boolean"},
    "foundations complete piles": {"type": "boolean"},
    "cells": {"type": "integer", "minimum": 0},
    "stock size": {"type": "integer", "minimum": 0},
    "stock deal type": {"type": "string", "enum":
            ["waste", "tableau piles"]
    },
    "stock deal count": {"type": "integer", "minimum": 1},
    "stock redeal": {"type": "boolean"},
    "reserve size": {"type": "integer", "minimum": 0},
    "reserve stacked": {"type": "boolean"}

  }, "additionalProperties": false
}
```

Even for those not familiar with the syntax used to define JSON schemas, comparing this schema to some of the existing rules files, it should not be difficult to interpret the options available and how to construct a rules file. Note that the 'two decks' option is experimental, and has only been proven to work for the combination of features that exist in my implementation of the game 'spider'. If the meaning of any of these options is not obvious, it may help to consult Appendix A, which gives an overview of the terminology used by *Solvitaire*.

Finally, the more simple schema which defines what starting deals can be supplied to the solver, is as follows:

```
{
  "\$schema": "http://json-schema.org/draft-04/schema#",
  "description": "JSON Schema representing a solitaire deal",

  "definitions": {
    "card": {"type": "string", "pattern":
            "^(([0-9]|1[0-3]|a|A|j|J|q|Q|k|K)(c|C|d|D|h|H|s|S))\$"
    },
    "cardarray": {"type": "array", "items": {"\$ref": "#/definitions
   /card"}}
  },

  "type": "object", "properties": {

    "tableau piles": {
      "type": "array", "items": {"\$ref": "#/definitions/cardarray"}
    },
    "hole": {"\$ref": "#/definitions/card"},
    "stock": {"\$ref": "#/definitions/cardarray"},
    "waste": {"\$ref": "#/definitions/cardarray"},
    "reserve": {"\$ref": "#/definitions/cardarray"}

  }, "additionalProperties": false
}
```

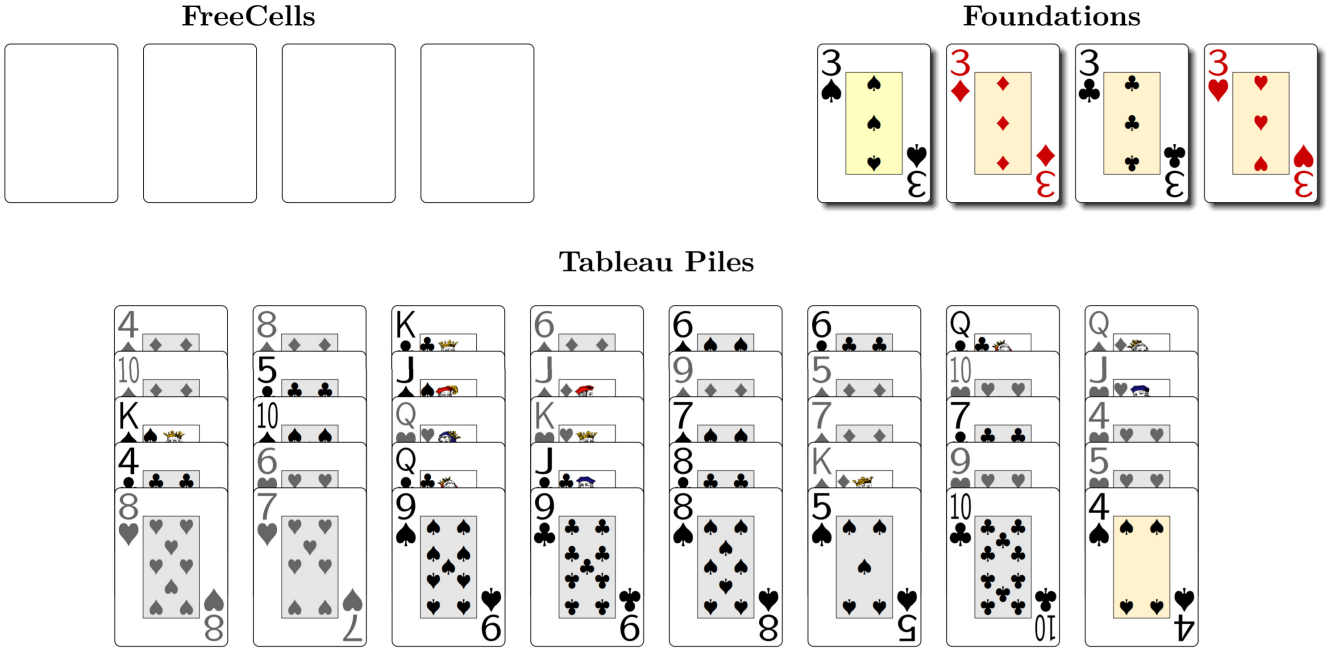
C.3 Supplied Files/Directories

The following is an overview of the files and directories supplied in my submission:

File	Description
solvitaire	The main *Solvitaire* executable.
bin/	Contains a number of other executables generated, which represent different versions of *Solvitaire*, with different optimisations enabled/disabled.
src/	The main source files for my project.
cmake-build-release/	The main build directory for the project. Executables when built are generated within this directory's 'bin' folder, which I have copied into the above directories for the sake of ease.
cmake-build-debug/	A version of the above directory but for debug versions of my implementation.
CMakeLists.txt	My main build file.
clean-build.sh	A simple convenience script which enables the user to run a clean build of my system.
build-notes.md	Some notes for me to remind myself about features of the build setup. Only left in here in case the marker wishes to build the system themselves and is having problems.
optimisation-test.py	A script used to test whether different optimisation builds return the same results across a range of seeds (mentioned in the main report)
benchmark.py	A script used during development to test the performance of the solver against different games.
docs/	Contains some documents relating to the general project.
resources/	Contains some useful JSON files for testing purposes, including a copy of rules JSON files for a group of games used in 'benchmark.py'
lib/	Contains external libraries used in the project
results/	Contains raw data for the results stated in the main report

D ‘Safe’ Foundation Moves In FreeCell

Figure D.1: A Safe Foundation Auto-Move



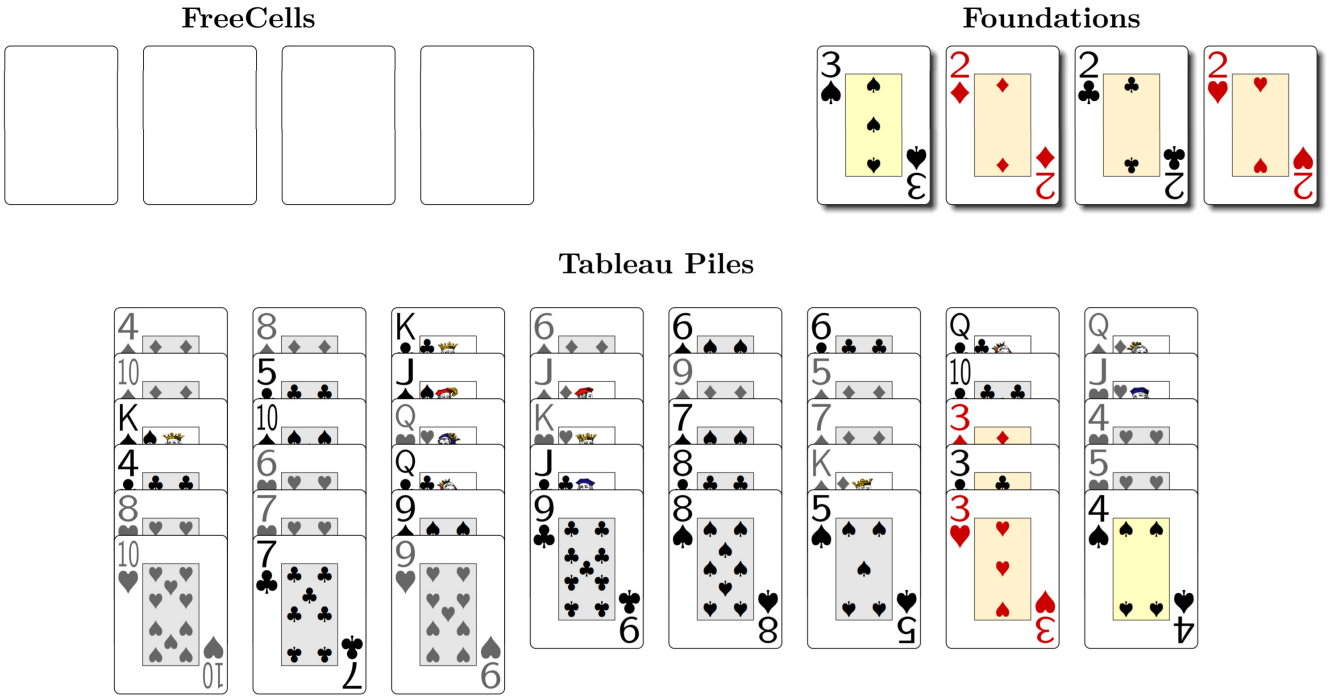
To begin this explanation we will consider the game FreeCell. The tableau pile build policy for FreeCell is ‘red-on-black’, which, it turns out, creates a rather complex set of conditions under which this dominance–which I call the ‘auto-foundations’ dominance–applies. FreeCell also stipulates that cards cannot be removed from the foundations. It is easy to see that under certain conditions, it is safe to move cards to the foundations, and hence this dominance applies. Consider the FreeCell position in figure D.1: there is nothing left in the tableau piles that can be placed on the 4♠, as the 3♥ and 3♦ are already in the foundations (I will refer to this set of cards that can be placed on a given card as the ‘reliant set’, as they may rely on being placed on that card to move around the tableau piles. In this case the 3♥ and 3♦ are the 4♠’s reliant set). We know that the 4♠ has none of its reliant set in the tableau piles and

therefore there is no downside to moving it up. All it can now do in the tableau piles is block other cards.

This leads us to our first rule for 'auto-foundations' moves in FreeCell: a card can be moved safely to the foundations if its rank minus one, is less than or equal to the lowest rank of the cards on top of its opposite colour foundations. This is just the same as saying that a card's reliant set must all be in the foundation piles.

However, we can go further than this rule and make foundation moves safely under a wider set of conditions. Consider figure D.2. We wish to make the same foundation move, but the other foundation piles now contain only the twos of their respective suits. Ignoring the state of the tableau piles, can we still safely move the 4♠ up? The answer here is again, yes. This is because although the 4♠ does not have all of its reliant set in the foundations, just looking at the foundations we can see that, using the logic of the rule examined previously, the cards in its reliant set can be automatically moved up as soon as they are uncovered. Therefore there can be no harm in putting the 4♠ up, as nothing relies on it.

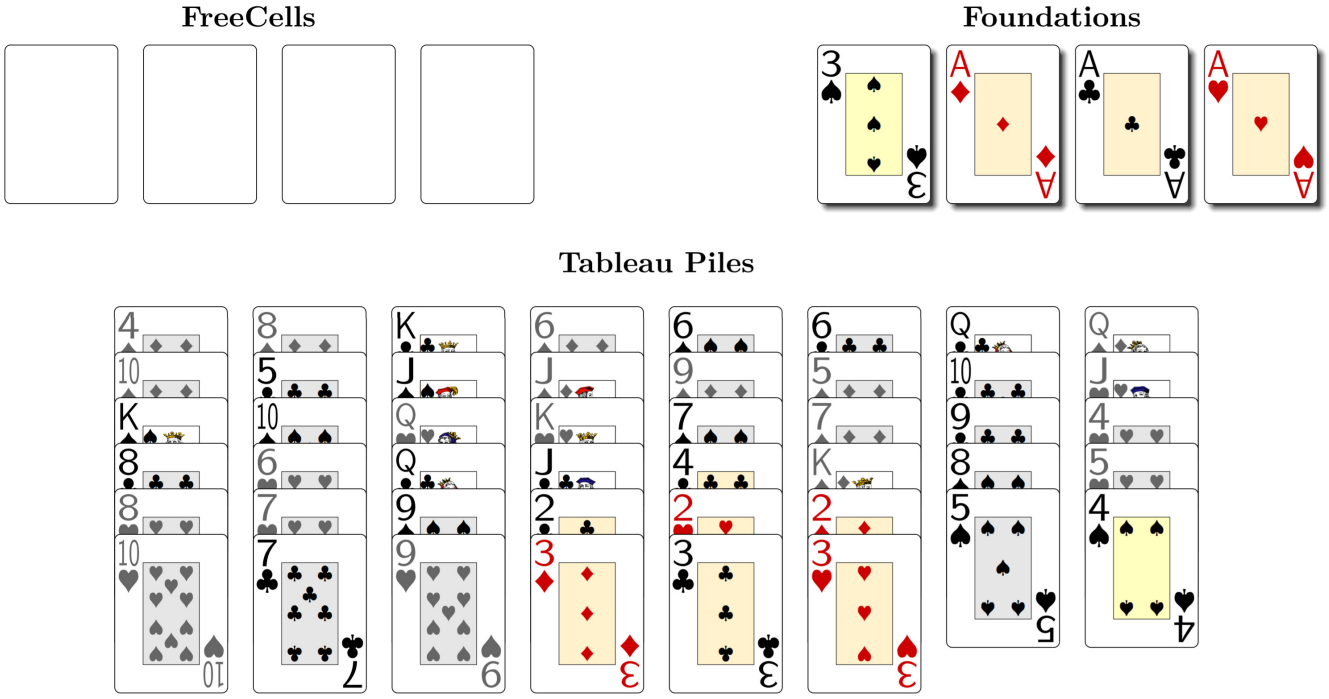
Figure D.2: Another Safe Foundation Auto-Move



However, we can't safely make this foundation move if the other foundations only contain the ones of their respective suits. This can be seen in figure D.3. Again, just considering the foundation piles for the moment, the 4♠ which we would like to move

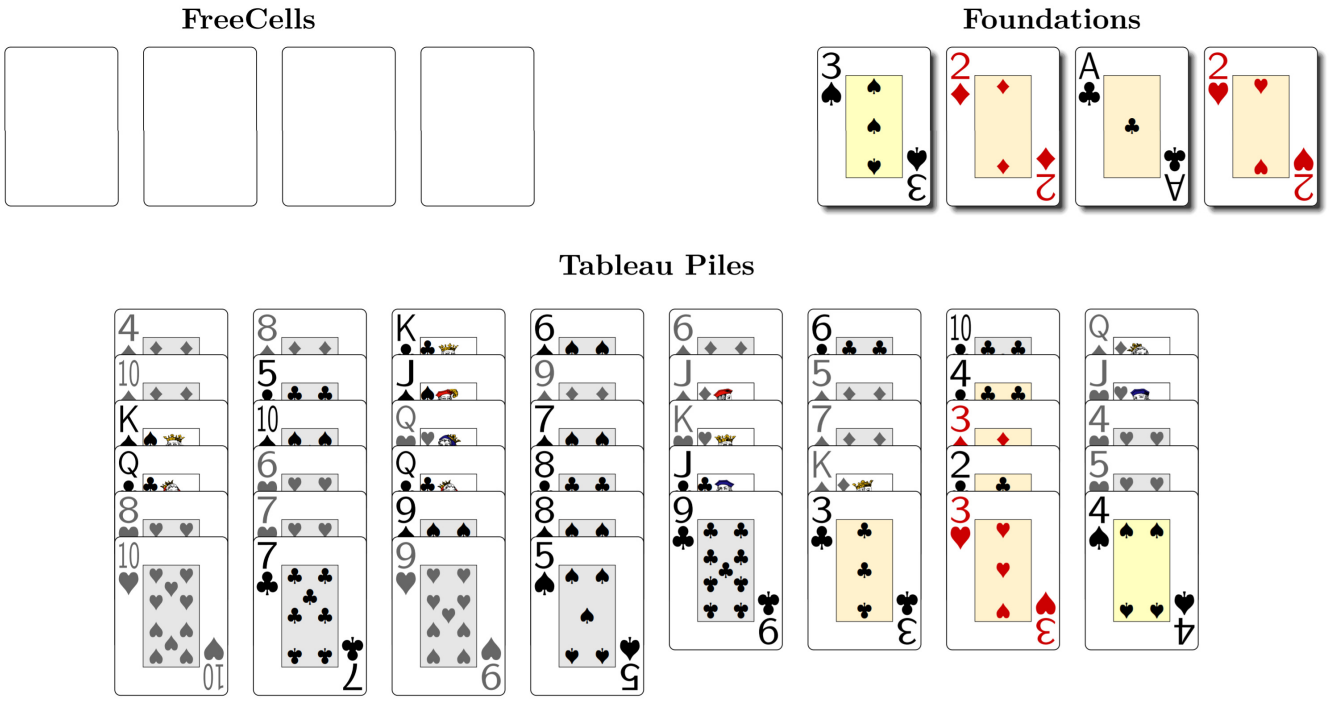
up does not have its reliant set in the foundations, and this time the foundations aren't the right rank to move up the reliant set automatically either. Figure D.3 demonstrates the worst-case scenario if we were to move the 4♠ up. Making this move leaves the 3♥ and 3♦ stuck, and the only other card they could move to, the 4♣, can only be reached by moving one of these red threes, which we now cannot do. This deal then cannot be solved. Hence, under these conditions we can see that this auto-foundation dominance does not apply.

Figure D.3: An Unsafe Foundation Auto-Move



Let us revisit figure D.2 again briefly. We used an inductive step based on the fact that the 3♥ and 3♦ could safely be moved up, to enable the 4♠ to then be moved up too. But what makes the 3♥ and 3♦ safe to move up exactly? The answer is based on the rank of their foundations, and whether the reliant set of those cards, the 2♠ and the 2♣, are in the foundations. In this example, both of them are in the foundations so we know that the threes are safe to move up. Given we are considering the foundation move for the 4♠, clearly the 2♠ is necessarily in the foundations, but what if the 2♣ is not? This scenario can be seen in figure D.4, and in fact the move is still always safe for this foundation layout. The reason for this is because the 2♣'s reliant set is in the foundation piles (the red aces), and so it can automatically be moved up.

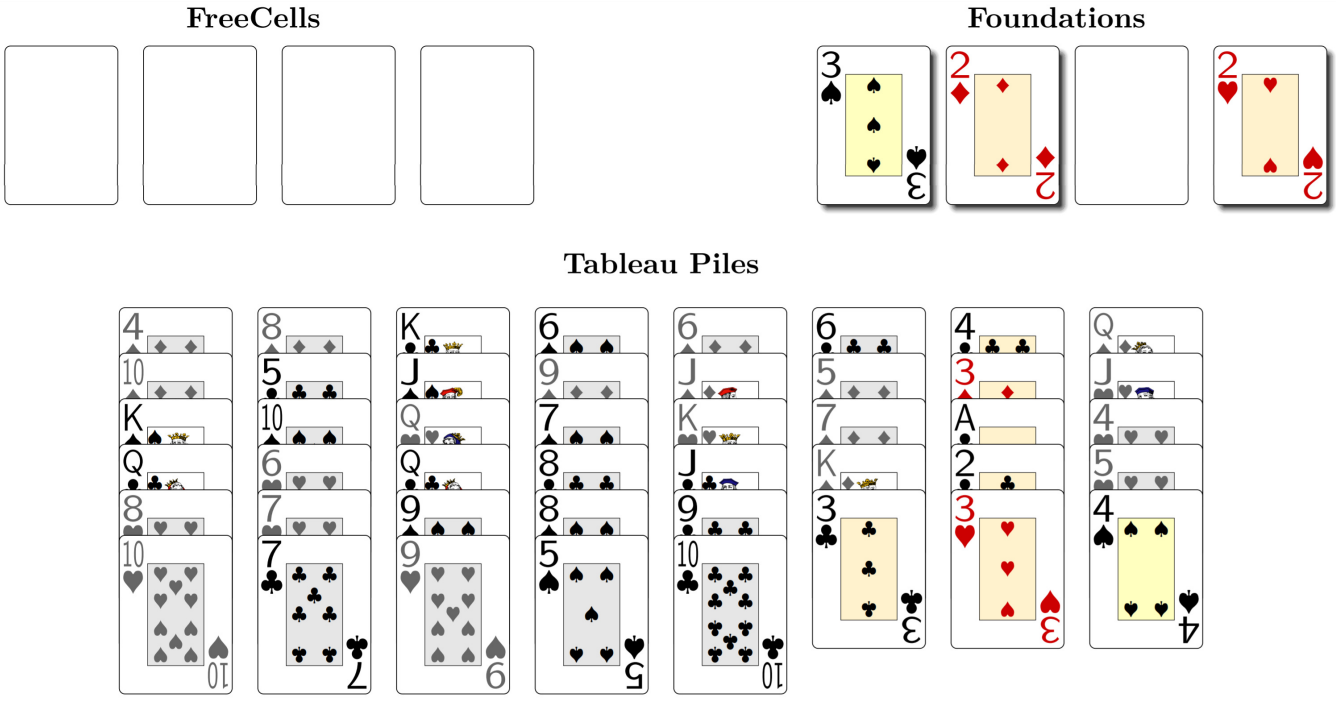
Figure D.4: A More Complex Safe Foundation Auto-Move



It's tempting to think at this stage that perhaps this reliant set-based induction spreads down through the ranks to make all foundation moves dominances. However, figure D.5 demonstrates why this is not the case. It is the same example as figure D.4, except that the A♣ is also not in the foundations. This removes the base case for our induction (that the 2♣ can safely go up), and hence we can no longer prove that the chain of steps leading to moving the 4♠ up is valid. In fact, I have constructed this pathological case specifically so that moving it up makes the deal unsolvable. Once the 4♠ goes up, the 3♥ cannot move anywhere in the tableau piles, and if we move the 3♥ up, then the 2♣ cannot move and we are stuck.

Combining all of this logic, we are left with two rules for when foundation moves are dominances in FreeCell. The first is the rule previously stated, which says that a card can always be moved up if it's at most one greater in rank than the lowest rank foundation card of the opposite colour. The second rule we can now deduce (based on the inductive reasoning we have seen), is that a card can also always be moved up if it is two greater in rank than the lowest rank foundation card of the opposite colour, *and* the other foundation card of the same colour is at most three ranks below it. Revisiting examples explored in this section, one can see this rule in action.

Figure D.5: A More Complex Unsafe Foundation Auto-Move



Bibliography

[1] Alan Agresti and Brent A Coull. "Approximate is better than 'exact' for interval estimation of binomial proportions". In: *The American Statistician* 52.2 (1998), pp. 119–126.

[2] Kent Beck et al. "Manifesto for agile software development". In: (2001).

[3] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. "Lower Bounding Klondike Solitaire with Monte-Carlo Planning." In: *ICAPS*. 2009.

[4] Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. "Searching Solitaire in Real Time". In: *ICGA Journal* 30.3 (2007), pp. 131–142.

[5] *Boost C++ Libraries*. URL: http://www.boost.org/. (accessed 25/03/2018).

[6] Lady Adelaide Cadogan. *Illustrated Games of Patience*. S. Low, Marston, Searle and Rivington, 1876.

[7] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. "Deep blue". In: *Artificial intelligence* 134.1-2 (2002), pp. 57–83.

[8] Chris Chan. "Helping Human Play Freecell". In: (). University of Kent.

[9] Wikipedia contributors. *Patience (game) — Wikipedia, The Free Encyclopedia*. (accessed 23/03/2018). 2018. URL: https://en.wikipedia.org/w/index.php?title=Patience_(game)&oldid=831130115.

[10] DevilSquirrel. *Klondike-Solver*. 2017. URL: https://github.com/ShootMe/Klondike-Solver. (accessed 23/03/2018).

[11] A Dunphy and MI Heywood. "" Freecell" neural network heuristics". In: *Neural Networks, 2003. Proceedings of the International Joint Conference on*. Vol. 3. IEEE. 2003, pp. 2288–2293.

[12] Achiya Elyasaf, Ami Hauptman, and Moshe Sipper. "Evolutionary design of freecell solvers". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.4 (2012), pp. 270–281.

[13] Achiya Elyasaf et al. "Evolving solvers for FreeCell and the sliding-tile puzzle". In: *Fourth Annual Symposium on Combinatorial Search*. 2011.

[14] David Ferrucci et al. "Watson: beyond jeopardy!" In: *Artificial Intelligence* 199 (2013), pp. 93–105.

[15] Shlomi Fish. *Freecell / Patience Solving Discussion*. 2011. URL: https://groups.yahoo.com/neo/groups/fc-solve-discuss/conversations/topics/1084. (accessed 23/03/2018).

[16] Shlomi Fish. *Freecell Solver*. URL: http://fc-solve.shlomifish.org/. (accessed 23/03/2018).

[17] Shlomi Fish. *Freecell Solver Links*. URL: http://fc-solve.shlomifish.org/links.html#other_solvers. (accessed 23/03/2018).

[18] Ian P Gent et al. "Search in the patience game 'black hole'". In: *AI Communications* 20.3 (2007), pp. 211–226.

[19] *Google Test*. URL: https://github.com/google/googletest. (accessed 25/03/2018).

[20] *gperftools*. URL: https://github.com/gperftools/gperftools. (accessed 25/03/2018).

[21] Luke Harding and Leonard Barden. *Deep Blue win a giant step for computerkind*. May 1997. URL: https://www.theguardian.com/theguardian/2011/may/12/deep-blue-beats-kasparov-1997. (accessed 23/03/2018).

[22] Malte Helmert. "Complexity results for standard benchmark domains in planning". In: *Artificial Intelligence* 143.2 (2003), pp. 219–262.

[23] Bernard Helmstetter and Tristan Cazenave. "Searching with analysis of dependencies in a solitaire card game". In: *Advances in Computer Games*. Springer, 2004, pp. 343–360.

[24] Lemon Games Inc. *Solitaire*. Google Play Store. 2018.

[25] Michael Keller. *FreeCell – Frequently Asked Questions (FAQ)*. 2015. URL: http://solitairelaboratory.com/fcfaq.html. (accessed 23/03/2018).

[26] Richard E Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial intelligence* 27.1 (1985), pp. 97–109.

[27] Luc Longpre and Pierre McKenzie. "The complexity of Solitaire". In: *Theoretical Computer Science* 410.50 (2009), pp. 5252–5260.

[28] Albert H Morehead. *The Complete Book of Solitaire and Patience Games*. Read Books Ltd, 2014.

[29] Paul Mozur. *Google's AlphaGo Defeats Chinese Go Master in Win for A.I.* May 2017. URL: https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html. (accessed 23/03/2018).

[30] Gerald Paul and Malte Helmert. "Optimal solitaire game solutions using A* search and deadlock analysis". In: *Ninth Annual Symposium on Combinatorial Search*. 2016.

[31] *RapidJSON Library*. URL: http://www.rapidjson.org/. (accessed 25/03/2018).

[32] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016, pp. 85–88.

[33] Jonathan Schaeffer et al. "Checkers is solved". In: *science* 317.5844 (2007), pp. 1518–1522.

[34] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[35] Jesse Stern. "Spider Solitaire is NP-Complete". In: *arXiv preprint arXiv:1110.1052* (2011).

[36] *The IBM Challenge Day 1.* Jeopardy. Season 27. Episode 23. Feb. 2011.

[37] Terry Weissman and Charles Haynes. *seahaven.* 1991. URL: http://seahaven.sourceforge.net/. (accessed 23/03/2018).

[38] Jan Wolter. *Experimental Analysis of Canfield Solitaire.* Apr. 2013. URL: http://politaire.com/article/canfield.html. (accessed 23/03/2018).

[39] Xiang Yan et al. "Solitaire: Man versus machine". In: *Advances in Neural Information Processing Systems.* 2005, pp. 1553–1560.