

Hidden Information in Solitaire Games

Tom Harley

April 2019

Abstract

An API for converting solvers of perfect-information solitaire games into solvers of solitaire games involving hidden information is presented, and an example implementation is used to calculate solvability percentages for a number of popular solitaire games. The theory of the formal representation of solitaire games is discussed, and optimisations used in the API are described in detail.

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 6,969 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.

Acknowledgements

I would like to thank Charlie Blake for permitting use of *Solvitaire* while it is in private development. I would also like to thank my supervisor, Prof. Ian Gent, for providing me with consistent support throughout the year.

Contents

1	Introduction	1
1.1	Project Aim	1
2	Context Survey	3
2.1	Solitaire	3
2.1.1	Terminology	3
2.1.2	Academic Value	5
2.2	The Question to Answer	5
3	Requirements Specification	7
3.1	DOER	7
3.1.1	Primary Objectives	7
3.1.2	Secondary Objectives	7
3.1.3	Tertiary Objectives	7
4	Software Engineering Process	9
4.1	Dependency Management Through <code>git</code> Submodules	9
4.2	<i>Solvitaire</i> as an API	9
4.2.1	<code>libsol.a</code>	10
5	Ethics	11
6	Design	12
6.1	Making a Decision Using Voters	12
6.1.1	Randomness and Consistency	14
6.1.2	Split Votes	14
6.2	Determining Solvability	14
6.2.1	Alternative Methods	15
7	Implementation	16
7.1	Project Structure	16

7.1.1	The API	16
7.1.2	The Example Implementation	16
7.2	Implementation Language	16
7.2.1	Attempts with Nim	17
7.2.2	Attempts with C++ for the Centre, C for the API	17
7.3	Optimisation	17
7.3.1	Move Buffer	18
7.3.2	Majority Moves	19
7.4	Configuration	19
7.4.1	Specifying Game Rules	19
7.4.2	Game-state Description Files	20
7.4.3	Settings Files	20
8	Evaluation and Critical Appraisal	21
8.1	Result Reproducibility	21
8.1.1	Results	21
8.2	Configurability	21
8.2.1	Results	22
8.3	A C++ API	22
8.4	Reviewing DOER Objectives	24
8.4.1	Primary Objectives	24
8.4.2	Secondary/Tertiary Objectives	25
9	Conclusion	26
A	Testing Summary	27
B	User Manual	28
C	Ethical Approval	33
D	Hardware Information	35
D.1	Computers Used for Section 8.1	35
D.2	Computer Used for Section 8.2	37
E	Formal Representation	38
E.1	For Individual Deals	40
E.2	For Equivalent Deals	42
	Bibliography	45

1. *Introduction*

Advances in game-playing AI are almost exclusively in areas with perfect information [8][12]. This may be useful to aid in the development of expert systems and similar constructs, but without studying the unique strategies of hidden information games there is a risk that key mechanisms with real-world applications will go unnoticed.

Only very recently has attention turned to hidden-information strategy AI, with a significant recent example being *Deep Mind*'s AI agent for the real-time strategy game, *StarCraft II* [3]. While this event shows an exploration of hidden information strategy, it still shares something in common with previous notable AI: it plays a two-player game. Competitive scenarios create unique challenges and prioritise different factors than single-player games — while a chess AI may attempt to evaluate a move based on control of the board, attempting to maximise a score [4], a solver for peg solitaire might look for any solution [16]. Specialisation in single-player games may reveal new ways of finding solutions, since different values are considered important.

What would be a better single-player hidden-information game to explore than solitaire? With its many varieties [18], there is much space to explore, with varying degrees of hidden information. Many enthusiasts have taken to attempting to solve popular variants, such as *Windows 98 FreeCell* [19], and there is also academic exploration of solitaire games [6][14]. With this interest, however, there is still much to explore: the lack of knowledge surrounding solvability percentages has been described as “one of the embarrassments of applied probability” [11].

1.1 Project Aim

Quite a few solvers exist for solitaire games. However, most solvers of note have perfect information: they solve games while knowing the position of every card, including those that are face-down and hidden from a player during regular play. These solvers have their own merits, such as speed and utility as hint systems, but in discarding the hidden-information element of the problem they alter it significantly. The performance of these solvers is admirable, however, so being able to utilise them in some manner would be beneficial.

The aim of the project is to create a system that, given a solver that works with perfect

information, produces a solver that does not use the hidden information. This will be done by using the solver on games that look like the deal to be solved but with the face-down cards shuffled randomly, tallying the moves the solver would make for each of these shuffled variants of the deal, and making the move that occurred the most.

In addition to the above, the project aim includes the creation of an example implementation of the API, so that solvability percentages for games can be calculated.

A more detailed description of the structure of the system can be found in section 7.1.

2. Context Survey

2.1 Solitaire

While many consider solitaire to be a single-player card game, it is in fact a collection of various card games played by oneself. These games take many different forms: some common, such as the *tableau* style of games like *Klondike* [7] and *Canfield* [22]; some more obscure, like the *accordion* format used by *Tower of Babel* [22]. Some solitaire games, such as *Black Hole* [13], have all cards revealed from the beginning of the game; others, such as *Fortunes Favor* [23], use face-down cards to create hidden information. Some games even include entirely unique rules, making solitaire games even more diverse.

Despite their differences, all of these games have in common the aim of manipulating cards to transform one (often randomly-generated) configuration of cards into another. The possible moves may be different, but the goal of reducing some setup of cards into another is a feature shared by all solitaire games. Additionally, solitaire games do not remove or introduce cards during play; for any game, the number of cards is consistent.

This project does not discuss *Peg Solitaire* [5], or indeed any single-player game that does not use cards.

2.1.1 Terminology

This report uses some specific terminology to ensure that explanations are clear. This section addresses the most important aspects.

The word *solitaire* is used in this document to refer to the collection of all tabletop card games that can be played single-player. Games that use additional playing pieces, such as *Chainsaw Warrior* with its use of dice [15], are not considered part of this collection. When discussing general-purpose solitaire solvers, the word *solitaire* may be used to instead refer to games that use specifically cards from one or more standard 52-card decks of playing cards, as these solvers are likely only able to emulate games with these restrictions.

A *game* is a set of rules, describing:

1. a way of dealing out cards to the table,
2. ways to move cards about the table,

3. victory conditions.

In this report, using the word *game* to refer to “application of rules on a layout of cards with the intention of finding a solution” is purposefully avoided — this is instead referred to as a *deal*, since the dealing of cards can be used to uniquely identify it against other *deals* in the same game.

In the previous list, item 2 describes *moves*: a move is an alteration of the positions of the cards on the table. If a move is mentioned in a game’s moves, then it is a *legal* move for that game. Between moves, the layout of the cards can be described as a *game-state*. Moves can be thought of as transitions between game-states. If a game-state satisfies a game’s victory conditions, then it is a *solved* game-state for that game. It follows that a *solution* for a game-state is a sequence of legal moves that, when applied in turn to the game-state, result in a solved game-state. A solution for a deal is similar, where the game-state used is the initial deal¹.

Some solitaire games include face-down cards in their deals, as either literal face-down cards or stacks of cards where only the top card is visible (e.g. the reserve in *Canfield* [22]). In these cases, if a player does not look at these hidden cards, the exact game-state being played cannot be determined. In situations like this, it is useful to consider the collection of deals it could possibly be, based on the face-up cards on the table. *Equivalent* game-states are game-states that have the same face-up cards, and that therefore could both be a game-state if the only information known about the game-state is these face-up cards. In a similar fashion, equivalent deals are deals with equivalent initial game-states. For any game-state, the number of equivalent states is equal to $n!$, including the game-state itself, where n is the number of face-down cards. Evidently, game-states with zero or one face-down cards have no equivalents, other than themselves.

A *strategy* is an association of game-states to moves: for all possible game-states, a strategy will have a move associated. A system that applies a strategy (by using the move associated with a given game-state by the strategy) is referred to as a *solver*. If face-up cards are the only information available about a game-state, then any strategies used must treat all equivalent game-states the same, and as such must associate the same move with each equivalent game-state.

For all solitaire games that make use of face-down cards, there is a *thoughtful* version, in which the face-down cards are known. This is different from dealing all cards face-up, since some rules discriminate between face-up and face-down cards. Thoughtful variants trade the hidden information of the problem for the ability to exhaustively search for a solution. A solver that ‘cheats’ by looking at face-down cards is a *thoughtful solver*.

¹Deals are sometimes referred to as *initial* deals to emphasise that the focus is on the game-state that the deal resides in before any moves are made.

2.1.2 Academic Value

Popular solitaire games are simple to understand. Most popular solitaire games possess few rules, and as such are quite accessible for new players. Additionally, some games share many rules: this overlap makes learning a new solitaire game easy for someone who has already learned a similar game, and further increases accessibility. Solitaire games tend to have all rules apply to most (if not all) play pieces, which is quite different to a game such as chess, where each different type of piece has different rules governing its use. The simplicity and modularity of the rules of popular solitaire games means that solitaire is relatively easy to model in computer systems, and similarly fast to compute with.

While simple at the surface level, solitaire games can be deceptively complex to master. The groups of long-term enthusiasts centred on several of the most popular games is a testament to the complexity and ‘skill ceiling’ these games possess. Yet even with these communities researching them, there is still more to learn about these games. The presence of interested communities shows also that there is demand (albeit recreational or academic) for research into the field.

The complexities of solitaire games prove too intricate for mathematical modelling: the lack of a decent theoretical solvability percentage for popular solitaire game *Klondike* has been described as “one of the embarrassments of applied probability” [11]. This apparent difficulty in mathematically modelling popular solitaire games implies that maybe empirical analysis through simulations is the most effective way to determine approximate solvability percentages.

The variety of solitaire games means that solitaire can be used to study a wide scope of elements in the topic of AI search. Hidden information can be prioritised by using games that deal many face-down cards such as *Canfield* [22]; finding chains in perfect-information scenarios can use *Black Hole* [13]; games with more restricted options for moves, such as *Golf* [1], can be used when gathering many results is important; and so on. For research purposes, creating a brand new solitaire game by piecing together rules from already-established games would be entirely feasible. The configurability of solitaire results in high utility across topics of AI search.

Many solitaire games have the player attempt to transform a deal into a very specific ordering of cards. Across deals, the player attempts to reduce them all into one final game-state. This process mimics real-life use of expert systems, where the system is used to reduce given information into a result or action. This relation to the real-world application of computer systems shows that, if there is reason to further research AI search, solitaire can be used as part of this research.

2.2 The Question to Answer

A common question to hear about solitaire games is that of how many deals for a given game are “winnable by a human being”. This question, however, is rather trivial: if a

human were to make random legal moves in every game-state, then there is a chance that they make the same moves a thoughtful solver would, giving them the same win-rate as an AI that cheats by looking at face-down cards. If a player is extremely lucky, they could be just as successful as a player with access to hidden knowledge. A solver could also be written that always makes a random move in every scenario, and similarly there is a chance that it solves every game that can be solved. The question has a simple answer, and is therefore not particularly interesting.

A similar question asks of the highest possible success rate not of a player, but a strategy. If a strategy does not allow random perturbation of its decisions, then the maximum possible solvability is not immediately apparent. It is this question which is tackled by this project: the building-blocks for solving a variety of games using unchanging strategies are provided and discussed.

It is important to note that even though random generation is used in the project, its use for solving is entirely deterministic [17]. Its use can be thought of as for the *generation* of strategies, rather than their alteration at run-time.

3. *Requirements Specification*

3.1 DOER

The DOER form was submitted early on in the project’s lifespan, and does not accurately outline what was eventually done. For posterity, the original objectives laid out in the form are listed below.

3.1.1 Primary Objectives

1. Create a general solver, capable of solving a wide variety of solitaire games, including those with hidden information.
2. Introduce optimisations to the solver for specific games that involve hidden information (such as *Klondike* or *Canfield*).
3. Analyse the solvability of the chosen games using the solver.

3.1.2 Secondary Objectives

1. Introduce optimisations for more “difficult” games, such as *Spider*, and analyse the solvability of these games.
2. Create a “hint” system using the solver, to suggest moves with the best odds of success in a given scenario.
3. Develop “thoughtless” variants of games (that is, introduce hidden information to games that traditionally have none), to test the solver’s capabilities with a wider variety of games.

3.1.3 Tertiary Objectives

1. Use the hint system in a user-friendly application as a way of helping stuck players, and investigate its utility.

2. Use the solver as a source of training data to develop a second solver through machine learning, and investigate its utility both in the solver and in the hint system.

4. *Software Engineering Process*

Since the project is relatively small-scale with a single developer, minimal-overhead development practices were followed, to afford the most developer time to working on the project.

4.1 Dependency Management Through `git` Submodules

Choosing how to manage dependencies for this project is a decision that must take several issues into account. The project uses a number of source code dependencies: some publicly available, such as `docopt` [2]; some only available privately, such as *Solvitaire* [6]. Additionally, some dependencies have been edited (for example, adding a `.gitignore` to the JSON parser library JSMN [25]). Finally, different dependencies use different build configurations, such as CMake [9] or header-only setups. The system used for managing dependencies should cater to these issues.

Using `git`'s submodule capabilities allows for the nesting of `git` repositories, keeping track of these submodule's checked-out commits, and allowing these submodules to be used as individual repositories for committing and pushing changes. Thanks to `git`'s decentralised nature, submodules can be sourced from anywhere, and as such both publicly- and privately-available dependencies can be managed. Dependencies were branched-off from their main development branch, and minor changes were made to them to allow for their easy building¹; these versions of the dependencies were then included in the project as submodules.

4.2 *Solvitaire* as an API

Solvitaire was not originally developed with being used as an API in mind. With access to its source code, *Solvitaire* has been patched in order to create a static library exposing its solving functionality. The result is stored in the `solvitaire` submodule in the project,

¹An example of a dependency slightly edited can be found at <https://github.com/magnostherobot/jsmn>.

and to make it available at the time of marking, `solvitaire.diff` has been included in the submission.

4.2.1 `libsol.a`

Changing *Solvitaire* to a library format meant writing small helper functions, directly accessing code central to its operation. The files `api.cpp` and `api.h` were added to the *Solvitaire* project, and are used by the CMake build system to create a static library containing the necessary functions to use the solver as a library.

5. *Ethics*

The ethical approval document is included in this report in appendix C. Neither the visualiser nor the user-friendly application were created during the project, and as such no user evaluation was required.

6. *Design*

When given a deal, the system will attempt to solve it. It does this game-state by game-state: at each game-state, it will determine the move to make to leave it. At regular intervals, the system will check whether or not the deal it is playing is still solvable. This description neatly splits the system into two parts: finding a move to make using a thoughtful solver, and determining whether or not the system should continue attempting to solve a deal.

A diagram showing the steps made by the system when attempting to solve a deal is featured in fig. 6.1.

6.1 Making a Decision Using Voters

When considering a set of equivalent game-states, the solver aims to make a move that keeps as many of the equivalent game-states as possible solvable after making the move. This move could be determined by, for each equivalent game-state that can be solved, finding the set of all moves that maintain the game-state's solvability if taken, and choosing the move that features most often in these sets. However, no systems exist that find these required sets, and for some game-states these sets could be very large, and as such this method is infeasible.

Instead of finding the set of all safe moves for a game-state, finding a single safe move would greatly reduce the computational effort required, while still being an indicator of a 'good' move to make in the case that this game-state is indeed the game-state in play. Crucially, and unlike the previous idea, systems that calculate a single move do exist, in the form of thoughtful solvers: if they can produce a solution to a game-state, then the first move in the solution must be a safe move. This is the reason behind using thoughtful solvers to create a non-thoughtful solver: they approximate an oracle capable of showing a safe move for a given game-state.

When dealing with many face-down cards in a game-state, possible equivalent game-states become too numerous to calculate solutions for. Instead, a randomly-chosen subset of these equivalent deals is generated, solutions are found for these, and the move that was made most often as the next move to solve these deals is made as part of the attempted solution proper.

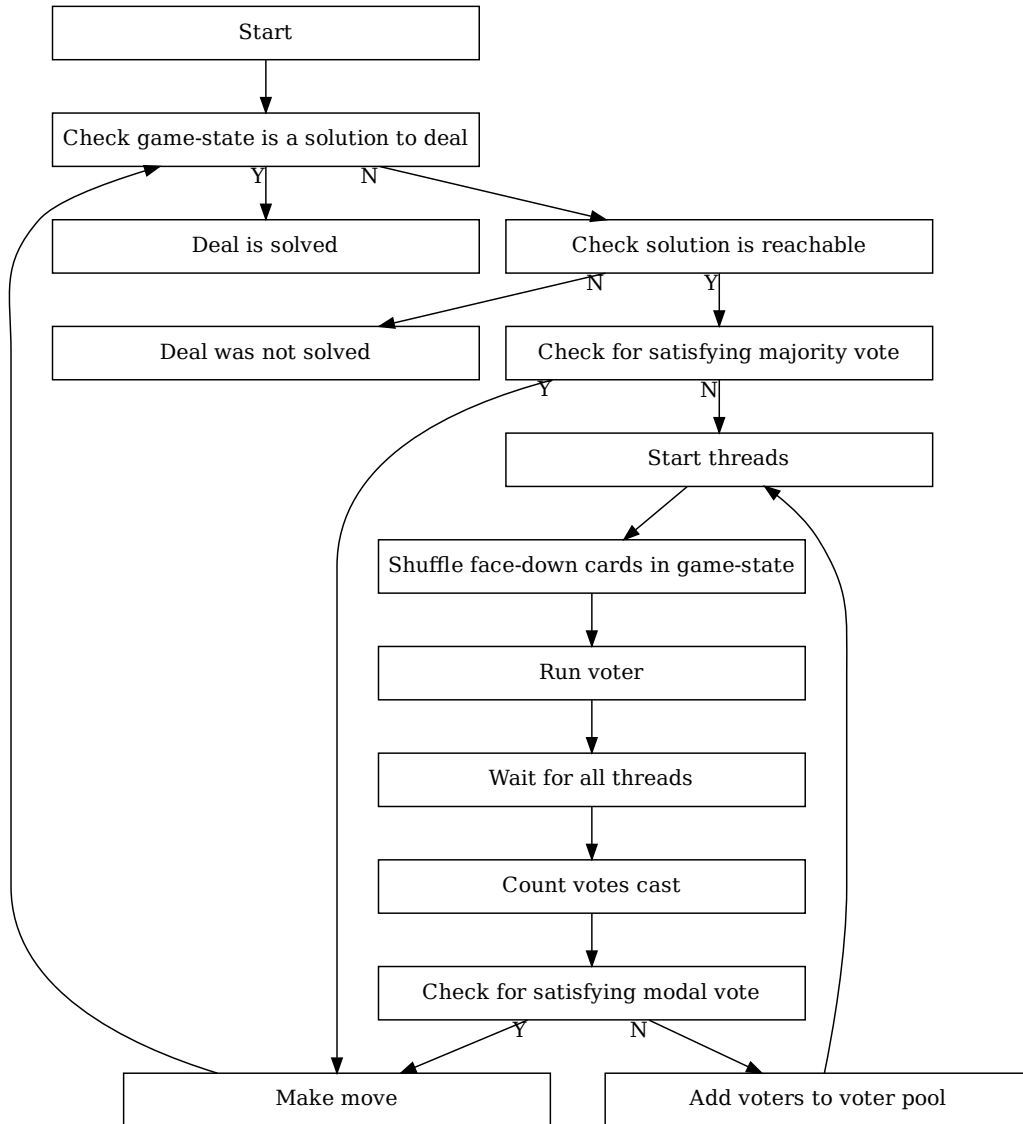


Figure 6.1: The steps taken by the system when attempting to solve a deal.

6.1.1 Randomness and Consistency

The purpose of the random shuffling of face-down cards to form different equivalent game-states is two-fold:

1. It produces a variety of game-states, with the aim to cover the variety of possible equivalent game-states;
2. It obfuscates the original game-state from the logic used when determining the next move, keeping hidden information hidden.

With randomness comes issues with inconsistency: being able to reproduce behaviour is useful when debugging, and additionally the reproducibility of results is crucial for scientific validity. The *Mersenne Twister* engine [17] implementation available in C++ produces output consistent across uses, and consistent across different machines [10], and so has been used as a random source in the system.

6.1.2 Split Votes

In the case that there is not a clear majority of votes agreeing on one move, the number of voters is increased temporarily, and the new voters are given new randomly-generated equivalent game-states to make a decision with. The results from the new voters are combined with those of the older voters, and the votes are checked for a clear majority a second time. In the case of another split vote, this process can repeat, until eventually the system will choose the most popular move, even if there is no majority.

6.2 Determining Solvability

Using only revealed information, the time required to show that a deal is not solvable (or, in the case that a bad move is made, is *no longer* solvable) can be extensive, due to the large number of ‘incidental’ moves possible in many games — this is apparent in games such as *will-o-the-wisp*, where the movement rules and number of face-up cards on the tableau allow for lengthy sequences of moves without loops.

Rather than exhaust possible moves in this fashion, the oracle solver is instead passed the current game-state, without randomisation of the face-down cards. If a solution is possible, play continues, whereas if no solved game-state can be reached, play is halted. If the oracle cannot find a solution from the current game-state, then it is impossible for further play to find one, rendering further play pointless. Similarly, if there is fewer than two face-down cards remaining, the game-state being played is known, so if a thoughtful solver can find a solution play is cut short, and the deal is considered solved.

This optimisation significantly shortens the length of time required by the system when attempting to solve a deal it will fail. However, checking the game-state’s solvability

between moves will definitely slow the system on a per-move basis. Currently in the system, solvability is checked after every move, except majority moves (described in section 7.3.2, since these moves can be processed very quickly. Another method employed to mitigate the time used by determining solvability this way is determining solvability in parallel with calculating the next move to take: if it is revealed that the game is no longer solvable, the parallel calculations still running can be terminated; otherwise, very little time has been wasted by running the oracle once more.

6.2.1 Alternative Methods

Since using the oracle solver with perfect information to determine solvability is quite effective, no alternatives have been implemented. Other methods are described here and compared against the currently-used method.

One advantage that other methods have in common when compared to the method implemented is that they do not rely on hidden information. When using a system for solving games, it might be preferred that the system does not possess the hidden information, for two reasons:

1. To prove that the system does not use the hidden information to inform its decisions (i.e. to prove that the system does not *cheat*);
2. To be able to use the system when no agent has access to the hidden information (e.g. when playing solitaire with real cards on a table).

User Input

While ineffective for solving many games in parallel, asking a user to determine whether or not the solver should continue play at intervals could be effective, as enthusiasts of the particular game being played may have an intuitive understanding of what game-states hold promise with regards to solvability. This, however, would heavily depend on the skill of the person in question.

Machine Learning

Solvability could be determined heuristically by the use of a machine learner. This may have the consequences of inaccurate decisions: it could allow an unsolvable game to run longer, or more importantly could terminate a game that would otherwise be solved. It may have a performance improvement over the implemented method, however as mentioned earlier in this section the time taken is heavily mitigated by running the calculation in parallel with other tasks.

7. *Implementation*

7.1 Project Structure

Early on, it was decided that the project would be split into two parts: an API that could be used by a third-party (named `metasol`), and an example use of that API (named `centre`). Throughout this report, the two are referred to together as “the project” or “the system”.

7.1.1 The API

The API functions as a wrapper around a solver that uses hidden information, producing a solver that does not use hidden information. A developer that wishes to wrap a solver must pass callback functions to `metasol`, for such tasks as running the solver on a game-state.

A reference manual of all functions exposed by the API is available, provided with this report.

7.1.2 The Example Implementation

An example of how to use the API, `centre` is an example implementation of a wrapper around generic solitaire solver *Solvitaire* [6].

Since both *Solvitaire* and `metasol` can accept descriptions of solitaire games, `centre` has also been made generic. This means that `centre` is able to solve for a wide variety of games, similar to *Solvitaire*.

A user guide on operating `centre` is provided with this report. Additionally, immediate help can be summoned with the command `./centre --help`.

7.2 Implementation Language

For a system that intends to be used by others as an API, it is important to choose a language that can be easily integrated into other systems — in particular, using a language that can interface with *Solvitaire*’s C++ code was considered a priority. However, for maintainability purposes, the language chosen should be fit for the purpose of implementing

the system itself. A few languages were considered, and eventually C++ was chosen as the language with which to write the system.

7.2.1 Attempts with Nim

Nim is a “high-performance garbage-collected language”, intended for general-purpose application design [21]. The initial skeleton for the project was originally written in Nim, but this was quickly discarded as interfacing with C++ code was extremely difficult, and would be necessary if the system was to interact with other *solitaire* solvers.

Using Nim would have increased the rate at which the system would have been developed — its high-level treatment of memory (including garbage collection) would remove the chance to mismanage memory, and its distinct typing mechanism would allow for compile-time checks on elements of design such as if hidden information is accidentally being leaked to the solving system. Its Python-like syntax promotes maintainability, which is a positive for any system. While not as performant as C, the system implemented ultimately does not need to run quickly itself, since the oracle solver will take up the vast majority of run-time. Nim would indeed be a good language to use for projects similar to this in the future.

Using Nim would have its drawbacks, however. Using an obscure language inherently reduces maintainability, since fewer people are comfortable with the language. Additionally, since few projects depend on Nim, the developers of the language could cease development. Nim’s obscurity could also dissuade potential API users. If Nim becomes more popular over time, it may be a decent choice as a language, however for now its obscurity would be too hampering for an API that aims to be useful.

7.2.2 Attempts with C++ for the Centre, C for the API

Making the API as generic as possible was a priority, as the API should be as accessible as is feasible in order to be useful to the most people. It was decided for this reason that the API should be written in plain C, while the example implementation should be written in C++ in order to interface with C++ *solitaire* solvers. However, during development, it was found that use of C++-exclusive libraries (such as the thread pool library CTPL [24]) would speed up production, and so ultimately both the API and the example implementation have been written in C++.

7.3 Optimisation

During development, it became apparent that the vast majority of run-time was spent waiting on *Solvitaire* methods, and that the main source of optimisation would be in reducing the number of times the external solver would be invoked.

7.3.1 Move Buffer

Some solvers are only capable of generating moves by seeing the game to its end — this is the case when using *Solvitaire*. With these solvers, a whole list of moves leading from the given game-state to a solved state are generated every time the solver is used. If such a solver generates a move, and that move is taken, then the same solver given the same arrangement of face-down cards in the new game-state should generate the same sequence of moves, less the move taken¹. Rather than re-run a voter with the new game-state to obtain this information, the sequence of moves generated last run is reused, removing the first move. Only move sequences that accurately represent the current game-state are preserved: if a move is taken that contradicts that of a sequence, the sequence is replaced by re-running the solver.

Buffering moves this way reduces the total number of times a voter is used when attempting to find a solution to a deal. This is at little expense: storing moves requires very little memory, especially when compared with the memory used by state-searching solvers.

Some solitaire games allow for infinitely-long sequences of legal moves. These sequences inevitably form loops, as the number of possible game-states given a finite number of cards is finite. Solvers for such games must employ strategies to avoid generating infinitely-long move sequences. Such strategies often involve looking at previous game-states or previous moves, to detect or avoid repeated cycles in which the same game-state is visited twice in one solution. When given the game-state of a solution-in-progress as a voter, this information of previous states/moves is unavailable to the solver, and so when requesting individual moves, cycle-detecting countermeasures are rendered ineffective. Requesting several moves in sequence remedies this, as it allows the cycle-detection of the voter to work for whole sequences of moves at a time. Using a max sequence size of n prevents cycles of lengths up to n . By default, the system requests and stores entire sequences.

Some games enable the creation of very long loops, especially when including use of the stock as a move, however long loops are prone to being broken eventually since their large numbers of game-states provides many different opportunities for the random shuffling of face-down cards to result in the voting for a move not part of the loop. The maximum length of sequence preserved should depend also on the hardware resources available: storing extremely long sequences of moves could result in the use of large quantities of memory, and copying their data from the voter to local memory could require significant lengths of time².

There are a number of events that would warrant the reset of a move sequence:

¹This assumes that the solver being used is deterministic, however even if this is not the case, the fact a solver *could* generate the same move sequence is enough.

²Copying move sequences is done by the thread dispatched to run the voter rather than the main thread, so time is realistically likely not an issue.

1. A face-down card is turned face-up, revealing new information³.
2. The move taken (as dictated by the voters) is not the next move in the sequence.
3. The sequence has run out of moves (and the game is not complete).

However these events are infrequent enough (especially when nearing the end of a solution) that storing entire sequences is worth the extra overhead.

7.3.2 Majority Moves

There are situations in which a sequence of moves is the only possible sequence of legal moves in a game — this becomes much more likely when loops are avoided within the sequence, and when only sequences that do not prevent victory are considered. In these situations, most (if not all) of the voters will produce identical sequences of moves: these moves are called *majority moves* in the system. In the case that a majority of voters agree on the next move to take straight after making a move, no voters need to be run before the next move is taken. This reduces the total number of voter runs further, and also reduces the total number of rounds of voting that take place. Reducing the number of voter rounds is valuable, since the more rounds there are, the more chances there are of a particularly difficult permutation of face-down cards being generated and causing the whole system to wait for one voter.

7.4 Configuration

Both `metasol` and `centre` require a variety of options to be specified: for the general operation of the system, to the rules for playing a game, to a description of a specific game to solve.

A detailed description of configuring `centre` and `metasol` are available in `centre`'s user manual and `metasol`'s API reference manual, respectively.

7.4.1 Specifying Game Rules

Similarly to *Solvitaire* [6], `centre` can accept a description of a game's rules to solve deals from that game. Since *Solvitaire* is used by `centre`, the rules accepted are very similar — `metasol` also requires some information about the rules of the game, but the rules it requires are a subset of those required by *Solvitaire*.

³ To reduce the number of sequences being reset in this event, only voters that used game-states with the newly-face-up card in a different location need to be reset. For anything more than a couple of face-down cards, it is quite unlikely that this will occur, unless many voters are used.

7.4.2 Game-state Description Files

A JSON file describing a specific game-state can be passed to `centre`, to have `centre` attempt to solve this game-state. This is rarely useful outside of testing

7.4.3 Settings Files

In addition to game rule files, `centre` can also accept some settings through configuration files, such as the maximum number of threads to use simultaneously. This was done in an attempt to aid reproducibility across uses: each set of experiments featured in this report used a settings file, which is provided with this report, to be used to verify the data given herein.

8. *Evaluation and Critical Appraisal*

8.1 Result Reproducibility

A primary goal of the project was to create a system capable of calculating the solvability of solitaire games, but this is valuable to the academic community if results found are reproducible. By providing configuration through settings files, and by using sources of consistent random generation, any results produced by the system should be easy to reproduce.

To test the consistency of the system, deals of *Golf* solitaire were played on two different very hardware configurations:

1. A 64-core server reserved for use for heavy-duty constraints-related calculations;
2. Over 100 2-core desktop computers.

Detailed information on the hardware configurations used can be found in appendix D.

Golf solitaire was chosen because of its relatively small state-space: being able to run many deals made using the 2-core computers far more tolerable.

8.1.1 Results

Both configurations produced exactly the same results for all 894 deals tested, showing that results are indeed reproducible on entirely different hardware.

As a side note, of the 894 deals run, 143 were solved.

8.2 Configurability

In order to be of use as a general-purpose tool, **centre** must be a highly configurable API: with the wide varieties of solitaire games that are possible, creating the API to be customisable is the only way to ensure as many games are catered to as possible.

One subject in which customisability is important is the time/accuracy trade-off. Allowing a solver to run for longer before terminating it means that it is more likely to find a solution, leading to better-informed decisions and therefore a higher chance of solving

the game; but the longer a solver is allowed to run, the longer the whole process takes, meaning fewer deals can be run. In particular, calculating solvability percentages is a task that is particularly sensitive to both the solver’s accuracy, and the time it takes.

To investigate the importance of **centre**’s flexibility, eight *Klondike* deals were run many times each, changing the limit on the number of nodes *Solvitaire* may use each vote. The nodes were limited rather than *Solvitaire*’s operating time, since the nodes *Solvitaire* travels are deterministically chosen and so can be reproduced, unlike operating time restrictions. The results of the experiment can be seen in fig. 8.1.

8.2.1 Results

Increasing *Solvitaire*’s node limit does not seem to have an effect on whether or not the system can find a solution. The only seeds affected are seed #4867 (which temporarily cannot find a solution) and seed #22703 (which unfortunately *can* find a solution before the node limit is increased).

While finding a solution is mostly unaffected, increasing the node limit does seem to increase performance. In particular, seeds #22703 and #29837, which cannot/can find solutions respectively, steadily decline for the most part — #22703 shortens its runtime by almost 400%. This performance improvement contradicts what was anticipated: since *Solvitaire* is given more nodes, it was predicted that run-time would increase due to the opportunity to explore more nodes. For the case where a solution is found, it may be that raising the limit allows the system to decide on moves that move directly to a solved game-state when it would otherwise make an extra move and restart voting. For the case where no solution is found, something similar may be happening: the solver may be making bolder moves since the solutions that require uncommon moves will indeed be the “harder” ones that require the extra time to see to the end. The other seeds seem to not change in performance nearly as much, with the exception of seed #4867, which has a very strange signature. Perhaps a very closely-split and crucial move is tipped over the edge several times? Regardless, even #4867 converges slightly as the limit on nodes is raised: generally, performance increases as the number of nodes increases.

By increasing *Solvitaire*’s node limit through **centre**, performance increases have been found. This is a strong argument for a configurable API. A corollary to these results is that running *Klondike* solitaire games with a higher node limit will reduce the time taken to produce results.

8.3 A C++ API

As discussed in section 7.2, C++ was chosen partly for of its ability to call *Solvitaire* functions with little overhead. This was considered very important during decision-making since it was expected that the solver would be called extremely frequently. However, since the majority of optimisations implemented in this project reduced the number of calls to

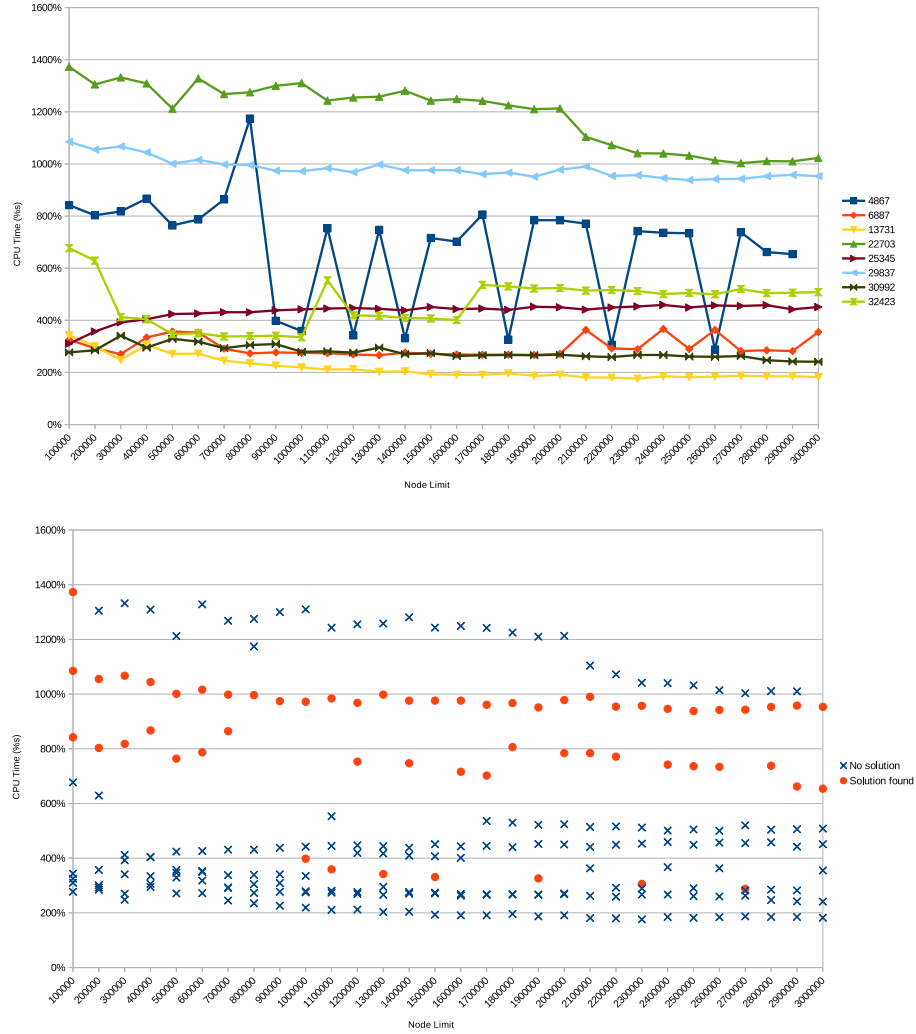


Figure 8.1: Deals of *Klondike* solitaire played with varying node limits. In the upper graph, the results for each seed have been connected, to show the trend of CPU time taken as the node limit increases; in the lower graph, the results have been grouped by whether or not a solution was found.

solver code, this became less of an issue than was originally feared. With this knowledge, the set of languages to choose from would not be restricted by those that can natively call C++ functions.

Another language would be chosen over C++ for a number of reasons. C++ is low-level enough that memory errors are commonplace, yet the power gained from this level of abstraction is not required for this project. C++ is not considered a “user-friendly” language, and as such developers may decide against contributing to this project if it were ever made open-source — this is possibly particularly important when considering that a large portion of those interested in this subject are enthusiasts. Personally, I had no C++ experience before starting this project; I probably would have chosen another language if I thought I had the choice.

A language such as *Python* would be a great fit for a project such as this:

- It is lauded as user-friendly and so would be accommodating to committed enthusiasts;
- It has support for communicating with external applications;
- Personally, I have experience with it;
- It is very popular, aiding maintainability.

8.4 Reviewing DOER Objectives

8.4.1 Primary Objectives

A general solver was created, capable of solving the same games as *Solvitaire*. Unfortunately, quite a few of the varieties supported by *Solvitaire* use only face-up cards, which are trivially supported by `centre`.

Soon into development, it was decided that `metasol` would manipulate cards in play as little as possible, since any optimisations introduced at this level may make the wrapped solver’s optimisations redundant, or may even directly conflict with them. Therefore, no optimisations were added to the system for specific games.

Despite not optimising *specific* games, significant optimisations were introduced, that reduce run-times down from days to minutes per game.

Due to the immense number of parameters provided through both *Solvitaire* and `metasol`, analysing the solvability of an array of games proved to be time-infeasible. Instead of focussing on this front, then, effort was committed to showing that the flexibility of the API would allow for further work to calculate solvability statistics for games.

8.4.2 Secondary/Tertiary Objectives

As mentioned earlier, many of the games *Solitaire* implements lack face-down cards. For some of these games, such as *Black Hole*, flipping some of the cards face-down is as simple as adding a line to the rule file. For these “thoughtless” variants, however, the chance of solving a deal are very slim: of all of the games played during testing *Thoughtless Black Hole* (an estimated 50–60 deals), only one was solved.

For the remainder of the games lacking face-down cards, it would be impossible to implement thoughtless versions while still using a thoughtful solver. An example of this is the accordion solitaire game *Tower of Babel*: in the non-thoughtful variant, cards are dealt to the table one at a time, and only these dealt-out cards can be moved; in the thoughtful variant, all cards are played at once, and any can be moved. This difference in what cards can be moved is an insurmountable problem if the thoughtful solver is to be used as-is.

Because the aim of the project split from the original goals, no other secondary or tertiary objectives were met.

9. *Conclusion*

Provided as artefact is an API capable of taking in a thoughtful solver and producing a non-thoughtful solver, for a wide range of solitaire variations.

I have not provided solvability percentages for games. Instead, I have provided thorough documentation and evidence for the system I have created. When making the decision between producing percentages and building more infrastructure around the system, I chose the option that would help others in using the system I have created. I feel that, after working with a fellow student's code base in part, it is only right that I should prepare my project for similar interaction from a future Senior Honours student. Having access to *Solvitaire* has allowed me to work on a truly unique project — one that hasn't been possible until now. I believe strongly in the successes of teamwork over individual work, so having the opportunity to build upon another student's work is an honour twice. The few tweaks that I made to the code of *Solvitaire* were difficult due to the complexity of the system, and since the solver wasn't designed to be an API, it lacked internal documentation. This has inspired me to ensure the system I have created is thoroughly documented.

I regret choosing to use C++. It overcomplicated many otherwise trivial tasks. I feel that this decision, made very early on, severely hindered the development of the system. In the future I will certainly think twice when choosing a language for a new system. As a silver lining, choosing C++ means I now have a working understanding of the language, which is a highly-sought-after skill in parts of the industry.

This project certainly has changed how I think about card games. Never again will I be able to sit in front of dealt cards, without thinking of all of the near-countless possibilities the face-downs provide. As my friend describes a game they found online, all I'll be able to focus on is the fragments of JSON assembling in my head to form a valid rule file. I really hope I don't seg-fault if I put a card in the wrong pile, but I don't really want to find out. In short, this project has been fun, but a cursed sort of fun: I'll never be able to look at a pack of playing cards the same way again.

A. *Testing Summary*

The nature of the system being developed would make formal testing difficult. The use of random generation results in a system tht acts unpredictably, and so cannot be tested using integration testing; yet sections of the system run for so long that unit testing would prove infeasible.

Because of the reasons stated above, the system was tested verifying answers generated during solvability calculations.

B. *User Manual*

N.B. *Documentation for the `metasol` API is also included in this submission, but not in this document.*

centre - A General Solitaire Solver that Respects Hidden Information

centre has been developed to showcase the abilities of **metasol**: an API for wrapping thoughtful solvers (ones that look at face-down cards) to create solvers that do not require any hidden information. The thoughtful solver used by **centre** is *Solvitaire*, which can solve many different solitaire variants, and as such **centre** is also highly flexible.

Getting centre onto Your System

Dependencies

centre contains all of its dependencies, however *Solvitaire* requires the C++ Boost libraries installed on the system.

centre's dependencies can be built by running `make init-submodules`.

Building

Building should be as simple as running `make`.

Usage

The most basic usage of **centre** is to run it like so:

```
centre rules/klondike
```

This will run a randomly-generated game of *Klondike* solitaire. Options can be passed after the rule file is specified - the list of possible options is viewable using `centre --help`.

Rule Files

The rule files accepted by **centre** are simple JSON files that accept the following options:

Option	Meaning
<code>tableau size</code>	The number of piles in the tableau.
<code>deck count</code>	The number of decks used.
<code>max rank</code>	The highest rank of card used in the game. The usual is 13 (up to kings).
<code>hole present</code>	Whether or not a hole is featured.

Option	Meaning
hole build loops	Whether or not aces can be placed on kings in the hole, and vice versa.
foundations present	Whether or not foundations are featured. The number of foundations, if enabled, is always equal to $4 * \text{deck count}$.
foundations removable	Once a card has been placed in the foundations, whether or not it can be moved again.
foundations accept only complete piles	Whether or not complete piles are the only way to move cards to the foundations.
diagonal deal	Whether or not a diagonal deal (i.e. the deal used in <i>Klondike</i>) is used when laying out the tableau.
number of cells	The number of cells used in the game.
cells pre-filled	The number of cells that start play containing a card.
cards in stock	The number of cards in the stock at the start of play.
stock deal count	The number of cards removed from the stock when dealing from it.
stock redeal	Whether or not the stock can be reused once it has been entirely dealt.
cards in reserve	The number of cards placed in the reserve.
reserve stacked	Whether or not the reserve is treated as a pile.
cards in sequence	The number of cards in the sequence.
sequence fixed suit	Whether or not a sequence uses a specific suit.
accordion size	The number of cards in the accordion.
build policy	The rule governing the movement of single cards across tableau piles.
spaces policy	The rule governing the movement of single cards into empty tableau piles.
move built group	The conditions for moving a group of cards.
group build policy	The rule governing the movement of a group of cards across tableau piles.
foundations initialised	Which foundations, if any, start play containing a card.
stock deal type	Where cards from the stock go.

Option	Meaning
face up cards	What cards are face-up when play begins.
sequence direction	The direction in which cards are placed on the sequence.
sequence build policy	The rule governing how cards are moved within the sequence.
accordion moves	An array of legal accordion moves.
accordion policies	An array of policies for the above moves.

Settings

Some settings can be passed to **centre** via the command line. Additionally, settings can be specified in a JSON file.

Name in File	Name on CLI	Meaning
max concurrent threads		The maximum number of threads centre will have working at once.
max concurrent games		The maximum number of games centre will try to run in parallel.
solveseed	--solveseed	The seed used by centre when making decisions.
	--dealseed	The seed used by centre when generating a game-state.
run forever	--forever	Runs centre indefinitely, on as many deals as possible.

Name in File	Name on CLI	Meaning
run timeout	--limittime	The maximum time a <i>Solvitaire</i> instance may run.
run node limit	--limitnodes	The maximum number of nodes a <i>Solvitaire</i> instance may use.
initial vote count		The number of voters centre usually uses.
agree ratio		The percentage of votes that must agree for the move to be made.
vote increase step		The number of voters to add in the case of a split vote.
max vote count		The maximum number of voters centre will use.

C. *Ethical Approval*

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
ARTIFACT EVALUATION FORM

Title of project

HIDDEN INFORMATION IN SOLITAIRE GAMES

Name of researcher(s)

TOM HARLEY

Name of supervisor

IAN GENT

Self audit has been conducted YES ☒ NO ☐

This project is covered by the ethical application CS12476

Signature Student or Researcher




Print Name

TOM HARLEY

Date

28/09/2018

Signature Lead Researcher or Supervisor



Print Name

IAN GENT

Date

28/9/2018

D. *Hardware Information*

D.1 Computers Used for Section 8.1

116 of the following were remotely connected to by a 117th unit via `ssh`, and their results gathered through a virtual filesystem:

```
Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):            4
On-line CPU(s) list: 0-3
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s):         1
NUMA node(s):      1
Vendor ID:         GenuineIntel
CPU family:        6
Model:             58
Model name:        Intel(R) Core(TM) i5-3470S CPU @ 2.90GHz
Stepping:          9
CPU MHz:           1596.456
CPU max MHz:       3600.0000
CPU min MHz:       1600.0000
BogoMIPS:          5786.88
Virtualization:    VT-x
L1d cache:         32K
L1i cache:         32K
L2 cache:          256K
L3 cache:          6144K
NUMA node0 CPU(s): 0-3
Flags:             fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                  pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm
                  constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
                  aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr
                  pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c
                  rdrand lahf_lm cpuid_fault epb pti ssbd ibrs ibpb stibp tpr_shadow vnmi
```

```
flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln pts
flush_l1d
```

Only one of the following was used:

```
Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):            64
On-line CPU(s) list: 0-63
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s):         4
NUMA node(s):      8
Vendor ID:         AuthenticAMD
CPU family:        21
Model:             2
Model name:        AMD Opteron(tm) Processor 6376
Stepping:          0
CPU MHz:           1400.000
CPU max MHz:       2300.0000
CPU min MHz:       1400.0000
BogoMIPS:          4600.20
Virtualization:    AMD-V
L1d cache:         16K
L1i cache:         64K
L2 cache:          2048K
L3 cache:          6144K
NUMA node0 CPU(s): 0-7
NUMA node1 CPU(s): 8-15
NUMA node2 CPU(s): 16-23
NUMA node3 CPU(s): 24-31
NUMA node4 CPU(s): 32-39
NUMA node5 CPU(s): 40-47
NUMA node6 CPU(s): 48-55
NUMA node7 CPU(s): 56-63
Flags:             fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                  pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
                  rdtscp lm constant_tsc art rep_good nopl nonstop_tsc extd_apicid amd_dcm
                  aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 popcnt aes xsave
                  avx f16c lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3
                  dnowprefetch osvw ibs xop skinit wdt fma4 tce nodeid_msr tlbm topoext
                  perfctr_core perfctr_nb cpb hw_pstate retpoline_amd ssbd ibpb vmcall bmi1
                  arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid
                  decodeassists pausefilter pfthreshold
```

D.2 Computer Used for Section 8.2

```

Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):            24
On-line CPU(s) list: 0-23
Thread(s) per core: 2
Core(s) per socket: 6
Socket(s):         2
NUMA node(s):      2
Vendor ID:         GenuineIntel
CPU family:        6
Model:            45
Model name:        Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz
Stepping:          7
CPU MHz:           2897.197
CPU max MHz:       3000.0000
CPU min MHz:       1200.0000
BogoMIPS:          5000.10
Virtualization:    VT-x
L1d cache:         32K
L1i cache:         32K
L2 cache:          256K
L3 cache:          15360K
NUMA node0 CPU(s): 0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
Flags:             fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                  pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
                  rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
                  nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
                  tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt
                  tsc_deadline_timer aes xsave avx lahf_lm pti ssbd ibrs ibpb stibp tpr_shadow
                  vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts flush_l1d

```

E. Formal Representation

N.B. *The following is a hastily-assembled collection of claims that has not been proof-read, but has been included to help the reader visualise aspects of solitaire play.*

The model of game-states and moves is similar to a finite state automaton: states represent game-states and transitions represent moves. Accepting states are solved game-states (which generally have no equivalent game-states since all cards are face-up); rejecting states are states for which there is no reachable solution¹. It is beneficial to think of solitaire games in such a way since:

- The moves made by a strategy can be visualised as a path across the transitions of the FSA;
- Collections of similar unsolvable game-states can be grouped together visually, and tend to form regions in graph visualisations;
- The visual element makes it apparent whether or not a solved game-state (an accepting game-state) can be reached from any state.

For formal models of solitaire games to be useful, formal definitions of game representation must be considered. For determining the solvability of solitaire games, it is important that the representations used aid in finding solving sequences.

Definition 1. *A model represents a deal if and only if the model can be used to accurately determine whether or not the deal is solvable.*

Definition 2. *A model represents a game if and only if the model represents all possible deals of the game.*

Lemma 3. *For any game, there is an FSA that represents it.*

Proof. Let Σ_m be the alphabet of all possible moves in any solitaire game, and Σ_c be the alphabet of all playing cards used in any solitaire game². A move describes the pile to

¹These states need not be rejecting states explicitly, since by definition they have no path to an accepting state and so any input that leads to them cannot possibly be accepted.

² Σ_c contains at least 52 elements (all of the cards in a standard pack of playing cards), but may contain more (e.g. joker cards) if a solitaire game uses these.

move the card(s) from, the destination pile, and the number of cards to move. The number of piles in any game is finite, as is the number of cards in any one pile, and as such Σ_m is finite. The different card types used in any solitaire game is finite, and so therefore is Σ_c . M , an FSA, decides on $\Sigma = \Sigma_m \cup \Sigma_c$.

Definition of the various components of FSA M :

$$M = (Q, \Sigma, \delta, q_0, Q_A)$$

$$\delta: Q \times \Sigma \rightarrow Q$$

$$q_0 \in Q$$

$$Q_A \subset Q$$

For any game there are a finite number of game-states, since there are a finite number of cards and therefore a finite number of arrangements of these cards. Let $Q_s \subseteq Q$ be the set of states that each represent a single game-state. If the game-state is a solution for the game, the state that represents it is also in the set of accepting states Q_A .

Let $Q_d \subset Q$ be the set of states that represent a partial deal of the cards to the table — in these states, some of the cards required for the game are yet to be placed on the table. Q_d is finite for similar reasons as Q_s . If some cards are placed in specific places for the game, this placement is featured in all states in Q_d : they are treated as already placed. M 's initial state $q_0 \in Q$ represents a partial deal if at least some cards are dealt randomly; if the positions of all cards are preset, it is an initial deal.

Possible deals of cards while in a partial deal state are represented as a transition from the state to the state representing the table after the card has been dealt. A deal of a card that cannot be dealt in a given partial deal (either because the card type is not used by the given game, or because the total number of copies of the card have already been dealt) will be rejected. The dealing of the final card will transform a partial deal into an initial deal: in this case, the transition representing the card deal leads to a state in Q_s . The position of the next card to be placed is predetermined, and no two states represent the same partial deal — this ensures that there is exactly one path from q_0 to an initial deal.

Legal moves for a specific game-state are represented as a transition from its state to the state of the game-state reached by taking the move. Sequences of moves containing illegal moves are rejected.

M accepts words of the form $u \circ v$, where:

1. u is a substring representing a sequence of card placements that are possible in the represented game,
2. v is a substring representing a sequence of moves that are legal under the represented game's rules, and
3. the move sequence represented by v is solution to the deal described by u .

This means that M accepts exactly the words that describe the setup and solution of a deal within the game.

A deal can be uniquely identified by the order of the placement of cards to form it. Likewise, after M has taken every character of u as input, the occupied state will be one that represents an initial deal — specifically, the initial deal which is identified by the unique placement of cards represented by u . If M accepts a string $u \circ v$, then the deal formed by the card placements in u is solvable, since there must exist a sequence of legal moves that transform the deal into a solved game-state (these moves are represented by v). If, for a constant u , M does not accept any string of the form $u \circ v$, then the deal formed by the card placements in u is unsolvable, since if it were solvable then there would be a sequence of legal moves that could be represented in v and that would be accepted by M . To determine if a deal of the game is solvable, one can determine if any accepting states are reachable from the state representing the deal (reached by parsing u). Therefore, the constructed FSA M represents the given game. \square

E.1 For Individual Deals

From the proof of theorem 3, a number of further conclusions can be made:

Corollary 3.1. *For any deal, there is an FSA that represents it.*

A trivial proof to this corollary is to reuse the FSA M from the proof of Lemma 3, however another, more substantial proof is provided here.

Proof. FSA m will be constructed to represent a given deal d .

Using the method described in the proof of Lemma 3, construct an FSA M to represent the game of d . Within M , the state representing the initial game-state of d (q_d) is reached by following the transitions that represent the placement of cards that form d 's initial state. Let q_d be the initial state of m . The set of states in m contains all states reachable from q_d in M , and the set of accepting states in m contains all accepting states reachable from q_d in M . m now accepts any string v for which M would accept $u \circ v$, where u is the string representing the placement of cards to form d ; m rejects all strings that do not satisfy this condition. The solvability of d can be determined by the presence of an accepting state in m : if there is at least one accepting state, then at least one accepting state is reachable, meaning at least one solved game-state is reachable from deal d . Therefore, d is represented by the FSA m . \square

The previous corollary constructs an FSA that resembles the form a deal takes when modelled in many solvers of solitaire games. However, if the constructed FSA were to be used as-is, infinite loops would be possible in the solver logic, and as such it is pertinent to show that deals can be modelled without the formation of loops.

Lemma 4. *For any deal, there is a labelled tree that represents it.*

Proof. This will be a proof by construction, describing tree T . The solvability of a deal can be ascertained through T by checking for an uninterrupted path from the root node to a node that represents a solved game-state.

There are a finite number of possible game-states in any game, as described in the proof of theorem 3. Therefore, the number of game-states for a particular deal must also be finite, as well as the number of possible game-states reachable from a given initial game-state. Let each of these reachable game-states be represented by a node in T , with the root node representing the deal.

Data: Unconnected graph G with nodes representing possible game-states

Result: G is connected as a tree

nodesToConnect = [];

newNodes = [];

add r , the root node of G , to nodesToConnect;

while nodesToConnect is not empty **do**

for node n in nodesToConnect **do**

for move m in legal moves from n **do**

if m 's target node t has in-degree 0 **and** $t \neq r$ **then**

 add edge e representing m to G ;

 add t to newNodes;

end

end

end

 nodesToConnect = newNodes;

 newNodes = [];

end

Algorithm 1: Connecting a tree for a deal

For all legal moves at the initial game-state, an edge is made in T between the root node and the node representing the game-state that results from making the move. This is repeated for the nodes now connected to the root, and repeated with newly-connected nodes until no unconnected nodes remain. Edges are not formed if they would increase a node's in-degree over 1 (or increase the root node's in-degree over 0). This method ensures that all nodes can be reached from the root node. A pseudocode description of the described method can be found as algorithm 1.

T should now possess the features of a tree. algorithm 1 ensures that all non-root nodes have an in-degree of exactly 1 so long as these nodes are reachable from at least one other node — this is guaranteed by including only nodes representing reachable game-states in T .

If the deal was unsolvable, then no game-state reachable from the deal would be a solution, meaning no nodes in T would represent a solution, meaning in turn that no

path from the root node would reach a node representing a solved game-state, making the representation of unsolvable deals accurate. In the case of a solvable deal, at least one path from the root node to a node representing a solved game-state. A solvable deal must have a reachable solved game-state, meaning a node representing it must have been included in T . Since all nodes in a tree are reachable from the root node, this solved game-state's node is reachable from T 's root. This means that solvable deals are also accurately represented, and therefore that all deals can be represented in this manner. \square

Inherent to the structure of a tree is an absence of cycles [20], and as such a tree created with algorithm 1 will have no cycles. An FSA can be formed from a tree, using its nodes as states; its labelled, directed edges as transitions; its leaf nodes as accepting states; and its root node as an initial state. This tree will represent the same deal as the tree that formed it, in the same fashion that an FSA created through the method in corollary 3.1 will represent a deal. The differences between the two FSA produced are the number of transitions present, and the presence or absence of loops. The models of a deal used by solvers that use hidden information tend to consider all possible moves that do not revisit an already-seen game-state, meaning that neither FSA is equivalent to this model. An FSA that more closely resembles the model could be formed by adding transitions to the tree's FSA that form undirected cycles but not directed cycles. Removing transitions that are part of a cycle from corollary 3.1's FSA is also an option, but removing these transitions without affecting the initial state's reachability to all other states may prove difficult.

Corollary 4.1. *Determining the solvability of a deal is a decidable problem.*

Proof. Given a deal d , a tree T that represents it can be created, and from T , d 's solvability can be determined by checking for the presence of a node representing a solved game-state. The generation of T takes a finite length of time, since algorithm 1 only loops when nodes are connected, and there is a finite number of nodes in total. Similarly, checking T for a solve-state node takes a finite length of time due to the finite number of nodes. The whole process must therefore terminate. \square

E.2 For Equivalent Deals

The propositions made and proved in appendix E.1 focus on the solvability of single deals. For solving games without access to hidden information, it is unknown exactly what deal is being solved, meaning that moves must be made without the knowledge of whether or not the deal will still be solvable afterwards. Instead, considering all the deals that could possibly be the deal being attempted (i.e. the deals that have the same configuration of face-up cards as the face-up cards on the table) is a method that can be used to evaluate the quality of a move. A new model that considers a group of equivalent deals should be constructed.

Definition 5. *A model represents a set of equivalent deals if and only if it represents each of the deals in the set.*

Theorem 6. *Any set of equivalent deals can be represented by a labelled tree.*

Proof. Instead of representing moves as a simple transition/directed edge as in previous proofs, moves between game-states are described as the following:

1. A transition representing the movement of one or more cards;
2. An *intermediate* game-state, representing the state of the table after the card move but before any cards are revealed;
3. A transition representing the reveal of one or more cards.

Data: Unconnected graph G with nodes representing sets of equivalent game-states (including intermediate states)

Result: G is connected as a tree

nodesToConnect = [];

newNodes = [];

add r , the root node representing the equivalent deals, to nodesToConnect;

while nodesToConnect is not empty **do**

```

  for node  $n$  in nodesToConnect do
    for move  $m$  in legal moves from  $n$  do
      if  $m$ 's target intermediate node  $i$  has in-degree 0 then
        add edge representing  $m$  to  $G$ ;
        for reveal  $\varepsilon$  in possible reveals of  $i$  do
          if  $\varepsilon$ 's target node  $t$  has in-degree 0 and  $t \neq r$  then
            add edge representing  $\varepsilon$  to  $G$ ;
            add  $t$  to newNodes;
          end
        end
      end
    end
  end
  nodesToConnect = newNodes;
  newNodes = [];
end
```

Algorithm 2: Connecting a tree for a set of equivalent deals

Algorithm 1 from theorem 4 is adapted for this representation of moves, to create algorithm 2. Whenever a face-down card (except the final face-down card) is revealed, the

equivalent intermediate game-states representing the table before the reveal will reveal the card to be a different rank and/or suit from one another. The ε -labelled edges in trees produced by algorithm 2 represent this ‘splitting-up’ of equivalent game-states, into several smaller equivalent sets.

As is shown in the proof of theorem 3, there are a finite number of states in any solitaire game. When considering a set of equivalent deals, the number of reachable game-states is at most the number of states in the entire game, and therefore is also finite. When all equivalent game-states are represented by a single node, the number of these nodes is also bounded and therefore finite. There must also be a finite number of equivalent intermediate game-states, since they are by definition connected to two non-intermediate equivalent game-states. Therefore a graph constructed by algorithm 2 has a finite number of nodes.

A graph T output by algorithm 2 represents a deal d in a set of equivalent deals as follows:

- If d can be solved, then there is a path from the root node of T to a node representing a solved game-state, in which each taken ε -labelled edge represents the reveal of a card in the position on the table that the same card occupies in d .
- If d cannot be solved, then no such path exists. □

For any game-state, the position of all face-down cards is known, as is which cards are face-down. It is not known, however, which face-down card is placed in which position. For any game-state with multiple face-down cards, there are other game-states with the same face-up card positions, but different face-down card positions. To a player, these game-states are indistinguishable from one another, and so any strategy used by a player must make the same move for all of these states. This is emulated in the trees generated by algorithm 2, since a move made for one game-state (by following a directed edge) must also be made for all equivalent game-states.

Similar to the trees created by algorithm 1, those made using algorithm 2 can have moves added to more closely resemble the collection of decisions made by a player during play.

Bibliography

- [1] <https://www.miniclip.com/games/golf-solitaire/en/>[Accessed: 12/04/19].
- [2] nov 2013. <https://github.com/docopt/docopt.cpp>[Accessed: 12/04/19].
- [3] Kai Arulkumaran, Antoine Cully, and Julian Togelius. AlphaStar: An evolutionary computation perspective. *CoRR*, abs/1902.01724, 2019.
- [4] D.F. Beal. Benefits of minimax search. In M.R.B. CLARKE, editor, *Advances in Computer Chess*, Pergamon Chess Series, pages 17 – 24. Pergamon, 1982.
- [5] John D. Beasley. *The Ins & Outs of Peg Solitaire*. Oxford University Press, 1985.
- [6] Charlie Blake. *Solvitaire*: A solver for perfect information solitaire games. University of St Andrews School of Computer Science, 2018.
- [7] Douglas Brown. *150 Solitaire Games*. Barnes & Noble, 1985.
- [8] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57 – 83, 2002.
- [9] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorf, 2000. <https://cmake.org/>[Accessed: 12/04/19].
- [10] cplusplus.com. mt19937 — C++ reference, 2000. <http://www.cplusplus.com/reference/random/mt19937/>[Accessed: 05/12/19].
- [11] Persi Diaconis. The problem of solitaire. Stanford University Department of Statistics, november 2004.
- [12] David B. Fogel and Kumar Chellapilla. Verifying anaconda’s expert rating by competing against chinook: experiments in co-evolving a neural checkers player. *Neuro-computing*, 42(1):69–86, 2002. Evolutionary neural systems.
- [13] Ian P. Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M. Smith, and S. Armagan Tarim. Search in the patience game ‘black hole’. *AI Commun.*, 20(3):211–226, August 2007.

- [14] Ian P Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, volume 141, pages 98–102, 2006.
- [15] Stephen Hand, 1987. <http://www.chainsawwarrior.net/gameplay-help>[Accessed: 12/04/19].
- [16] Chris Jefferson, Angela Miguel, Ian Miguel, and Armagan Tarim. CSPLib problem 037: Peg solitaire. <http://www.csplib.org/Problems/prob037>.
- [17] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.
- [18] Albert H. Morehead and Geoffrey Mott-Smith. *The Complete Book of Solitaire and Patience Games: The Most Comprehensive Book of Its Kind: Over 225 Games*. Bantam, august 1983.
- [19] Tim O’Reilly, Troy Mott, and Walter J. Glenn. *Windows 98 in a Nutshell*. O’Reilly Media, Inc., september 1999.
- [20] Ondřej Pavlata. Ruby object model: Data structure in detail, March 2011. <http://www.atalon.cz/rb-om/ruby-object-model/#algebraic-tree> [Accessed: 04/12/19].
- [21] Andreas Rumpf. Nim programming language, 2019. <https://nim-lang.org/>[Accessed: 05/12/19].
- [22] Alfred Sheinwold. *101 Best Family Card Games*. Sterling Publishing Company, Inc., 1992.
- [23] Various Contributors. https://en.wikipedia.org/wiki/Fortune%27s_Favor[Accessed: 12/04/19].
- [24] vit-vit. CTPL: Modern and efficient C++ thread pool library, August 2014. <https://github.com/vit-vit/ctpl>[Accessed: 05/12/19].
- [25] Serge Zaitsev, nov 2010. <https://zserge.com/jsmn.html>[Accessed: 12/04/19].