Constraint Models for Relaxed Klondike Variants

Nguyen Dang
School of Computer Science, University of St Andrews, UK

Ian P. Gent
School of Computer Science, University of St Andrews, UK

Peter Nightingale
Department of Computer Science, University of York, UK

Felix Ulrich-Oltean
Department of Computer Science, University of York, UK

Jack Waller
School of Computer Science, University of St Andrews, UK

Abstract

Klondike is the most famous single-player card game, and remains a challenging search problem even in the 'thoughtful' variant where all card locations are known. We create constraint models to solve relaxed variants of Klondike, which have the property that the full game is unwinnable when they are unwinnable. Our models place bounds on when a card can move, which can propagate to yield an impossibility proof when a blocking set of cards exists. If we can show that an instance cannot be solved in a relaxed variant, we therefore know that we are unable to solve the instance in the main game. Our various relaxations give a tradeoff between model complexity and the proportion of layouts which are unsolvable. For the least relaxed variant, our model proves more than 70% of all impossible layouts to be unsolvable despite not doing a full search for the game. We then show that we can combine these constraint models with a Klondike-solving program to outperform either on its own. We use constraint modelling to produce schedules for running algorithms which outperform any single solver across a range of time limits.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases AI Planning, Modelling, Constraint Programming, Solitaire and Patience Games

Supplementary Material https://github.com/turingfan/Modref2024-Patience/

Funding *Peter Nightingale*: EPSRC grant EP/W001977/1
Felix Ulrich-Oltean: EPSRC grant EP/W001977/1

Acknowledgements Experiments were carried out on the Viking and Cirrus clusters. The Viking cluster is a high performance computing facility provided by the University of York, and we are grateful for computational support from the University of York IT Services and the Research IT team. Cirrus is a UK National Tier-2 HPC Service at EPCC (`http://www.cirrus.ac.uk`) funded by the University of Edinburgh and EPSRC (EP/P020267/1). Thanks to reviewers for their careful comments.

1 Introduction

'Klondike' is the formal name for the most popular solitaire or patience game, getting 100 million plays per day in 2020 just in Windows Solitaire [7]. We show that constraint models can perform extremely well in Klondike on impossible layouts, despite the inherent complexity of the game and its arbitrary rules. As with most previous research, we focus on the thoughtful variant, where the positions of all cards are known at the start of the game. AI

methods had not been generally effective at solving patience games until the introduction of Solvitaire [3], which performs a very fast depth-fast search with optimisations such as use of dominances and transposition tables, but nothing akin to constraint propagation. Solvitaire then performs poorly when there are multiple near-solutions but none can be converted to a full solution [3]. Constraint solving has previously been used for different single player card games [6, 13], but not successfully for the full game of Klondike.

Figure 2 gives a sample layout of Klondike, except that some cards which would normally be hidden have been revealed. We provide a glossary of terms in Appendix A, as some may not be familiar, for example 'worrying back' which allows a card to be moved back to the back to the tableau. We quote the standard rules of Klondike from [3]: "A single standard deck is used and the goal is to build all cards on foundations in suit from A to K. The game begins with a tableau of 28 cards in a triangular form with piles from 1 to 7 cards, with all but the top card face-down. Face-up cards on the tableau may be built in alternating colour, and built groups may be moved. Face-down cards may not be moved. Spaces may be filled only by a K. Cards may be worried back from foundations to tableau. A stock of 24 cards may be drawn in groups of three, and redeals are allowed without limit." Note that this is 'deal-3' Klondike, with an important variant being 'deal-1' where cards are moved one at a time.

Our constraint models exploit a key concept that has already been used in proving Klondike layouts unwinnable: that of a *blocking set* [4]. A blocking set is a set of cards in a given layout with the property that none of them are able to move until at least one of the others has. Since this is impossible, none of the cards in the blocking set can ever be moved. The presence of one does guarantee the layout must be unwinnable. The absence of a blocking set does not guarantee winnability, however. In this paper, we relax the rules of the game so that the presence of a blocking set does correspond exactly to winnability in the relaxed game. Our constraint models determine winnability in the relaxed variant, thus proving unwinnability in the original game when the relaxed game is unwinnable. We provide three increasingly strict relaxations: each is the same except in how it deals with the stock, and increasing strictness correlates with increased accuracy but also increased runtime.

We report experimental results showing that our constraint models can prove a high percentage of games unwinnable despite working on a relaxed version of the game. We then show that we can combine these constraint models with the state-of-the-art patience solving program Solvitaire. We use constraint modelling to produce schedules for running algorithms which outperform any single solver across a range of time limits.

2 Relaxed Klondike Variants

In this section we describe our three relaxed variants of Klondike. The key change we make in all variants is to relax the rules which determine the availability of the tableau and foundation cards. The three variants differ only on how we treat cards in the stock: it can be required to obey the normal rules of Klondike; or the stock rules can be partially relaxed but still enforced in part; or the stock rules can be totally relaxed. The key point of all our relaxations is that any winning game of Klondike gives a winning sequence of moves for the relaxed variant. Thus, if we prove that a relaxed game is unwinnable, we have proved that the same layout would be unwinnable in regular Klondike.

We will say that a card is 'foundation-built' when it is put on the foundation pile of its suit which already contained the card one rank less and of the same suit. We will say that a

card is 'tableau-built' when it is put on the tableau on a card one rank higher and of the opposite colour. Finally we will say that a King is 'space-built' when it is moved into a free space on the tableau. We can now specify the non-stock aspects of all relaxed variants of Klondike in this paper:

- Once a card is in the foundation, the next card in the same suit can be foundation-built onto it at any time. This is a relaxation because in the real game we are allowed to 'worry-back' the foundation card by tableau-building it from foundation to tableau. Thus in the real game, after being worried back, a card cannot be foundation-built on until it is moved to the foundation a second time.
- Similarly, once a card is available to be tableau-built upon, it remains available for the rest of the game with one key exception. The exception is that we disallow tableau-building two cards onto the same card at the same time. However, the relaxation still allows cards to be tableau-built upon even after they have been moved to foundation, without the need to be worried-back first.

A number of key rules remain which will help prove impossibility.

1. No card which is face-down in the tableau may be moved at all until the card immediately covering it is moved (by being either tableau-built or foundation-built).
2. Cards cannot be tableau-built on before they are face-up in the tableau. For face-down tableau cards, this is when the card immediately covering them is either tableau-built or foundation-built. For non-tableau cards, this is when the card is tableau-built onto a face-up card in the tableau, or (for Kings) when it is moved into a free space.
3. Except for Aces, cards cannot be foundation-built until the card one lower in rank and of the same suit has been foundation-built.
4. If two cards are tableau-built to the same card, the second one cannot be built there until after the first one has moved to foundation. Note the statement that the first card moves to foundation, not within the tableau. The validity of this restriction follows from the dominance proved by Blake and Gent [3], that if a card is tableau-built for the second time, the card it moves from should immediately be built to foundation.
5. Kings cannot move onto the tableau except to a space. A King can only be space-built when the card previously in that space has been tableau- or foundation-built. An exclusion principle applies, that no two Kings can occupy the same space at the same time.

There is a notable difference between the first three rules and the last two. Rules 1 to 3 bound values in a way that constraint solvers can propagate effectively. In contrast, rules 4 and 5 state that two events must be strictly ordered, and search might be necessary to choose the order. Our results will show that search is necessary, but is effective.

Stock Relaxations

All the above rules apply to every Klondike variant we consider in this paper. However, we do provide three different variants, depending on how we treat the order cards are played from the stock. We vary this to produce three variant versions of Klondike with increasingly relaxed stock rules:

1. The strictest rule is that we have to play cards according to the standard stock rules of deal-n Klondike. This means that the next card played has to be one of: immediately under the last card played; or a multiple of n further in the stock from the last card; or a multiple of n from the start of the stock; or the last card in the stock. All of the above conditions apply to the stock at the state the stock is in at the time the move is made.

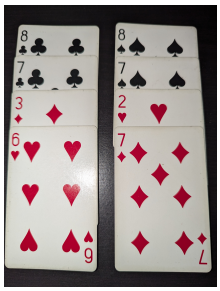

Figure 1 Handcrafted small example of a blocking set. Two stacks from the tableau are shown. All cards except the fully visible ones should be face-down but are shown here to explain the impossibility of completing the game.

2. A considerable relaxation is the following. Initially all cards at a multiple of n from the start of the stock, and the last card, are available to play. Other cards become available when the card directly after them in the stock has been moved, or any card in an earlier multiple of n in the stock has been moved.
3. A total relaxation is that any card can be played from stock at any time if there is somewhere legal (in the relaxed version of the game) to move it. This is exactly equivalent to playing deal-1 Klondike.

3 Blocking Sets

A *blocking set* [4] is a set of tableau cards, none of which can make its first move until at least one of the others has before it. The cards underneath the blocking set are *blocked* and can neither be played or built on. A very simple example of a blocking set would be the single card 7♠ in a pile of hidden cards 8♡, 8♢, 6♠, 7♠. The 7♠ cannot be moved anywhere until strictly after it has itself moved, as it can only move to one of the cards it is hiding. Thus this layout can never be won. In the rest of this section we first discuss an example blocking set involving two different tableau piles, and then a very complex one involving all seven tableau piles. All blocking set examples in this section apply whichever stock relaxation is used, however more complex blocking sets exist which rely on the stock rules.

Figure 1 shows a small example of a *blocking set* where the relaxed Klondike rules described above can be used to prove impossibility. The stock is not relevant in this example. The blocking set consists of 6♡ and 7♢. The 6♡ cannot be moved, for the following reasons: it cannot be foundation-built, because the 2♡ would need to be foundation-built earlier and 2♡ is blocked by 7♢; and it cannot be tableau-built because both black 7 cards are blocked. The 7♢ cannot be moved, as follows: it cannot be foundation-built because the 3♢ is blocked; and it cannot be tableau-built because both black 8 cards are blocked. The 6♡ and 7♢ together are hiding all the cards that would be needed to allow them to move; hence they are a blocking set.

Example of a Complex Blocking Set

Figure 2 shows an example where these rules combine to prove a layout unwinnable (including deal-1) even though the A♣ can be played immediately. We will explain the way that the rules combine to illustrate the rules, the complexity of their interactions, and the concept of a blocking set. We will show that the cards 5♢, 4♠, 8♢, 10♢, 8♡ are a blocking set, so none

Figure 2 Handcrafted complex layout that our model proves unwinnable. All cards in the tableau except the fully visible ones should be face-down but are shown here to explain the impossibility. Cards which are not explicitly shown are unnecessary to prove unwinnability.

can be moved though they can be tableau-built on. The full proof is provided in Appendix B; here we sketch the proof and link it to the five rules.

First, note that the 8♢ and 8♡ cannot be foundation-built, because they are blocking the red 7s which would need to be foundation-built first (**rules 1 and 3**). Red 8s also cannot be tableau-built because black 9s are blocked and rule 2 prevents them being built on.

Second, the 4♠ cannot be foundation-built because the 3♠ is blocked (**rules 1 and 3** again). The 4♠ cannot be tableau-built for a complex chain of reasons that eventually makes use of **rule 5**: we need 3 kings (from stock) to be in the tableau at the same time but only two spaces can be used because the other five tableau piles contain a blocking card.

Third, the 10♢ cannot be foundation-built because the 8♢ cannot be moved. It cannot be tableau-built because the only card it can be tableau-built onto is the J♣, and the 10♡ would have already been built on it (**rule 4**).

Finally, the 5♢ cannot be foundation-built because the 4♢ is blocked, and a potential move within the tableau is prevented by the red 7s being blocked (**rule 2**).

Since no card in the blocking set can be moved until at least one of the other cards has been before it, none can ever be moved. Notice that all five rules were used in this (exceptionally complex, handcrafted) example.

4 Constraint Models for Relaxed Variants of Klondike

Our first constraint models sought to find blocking sets, and therefore had the property that when a solution was found the underlying Klondike game was unwinnable. In principle one could negate this model to get a model which would be satisfiable when no blocking existed: this would have the advantage that it could be used as an adjunct to a model intended to find winning sequences of moves. Unfortunately, a simple negation leads to stating 'all possible sets of cards are not blocking sets' which would result in a model with an impossibly large number of variables. Instead of this, we adopt the approach of seeking an *unblocked artifact* such that when it exists, no blocking set can occur. The artifact we construct is to assign each card a stage when its first move occurs. We add constraints that if one move depends

on an earlier move, then it must occur at a later stage. Any solution to this problem proves the nonexistence of a blocking set, as it would necessarily involve a card that was required to be played before itself. We outline our models below but also have provided them in full online together with experimental results.[1]

4.1 Viewpoint

Rather than having to model each move in the full game, we model an abstraction of the original moves. Our model works by constraining when the *first* move of each card occurs. Instead of constraining the exact time that cards move, we allow multiple unrelated moves to happen at the same stage. Therefore, independent moves can occur at the same stage, whilst dependent moves must occur at different stages. The number of stages can be limited to twice the number of cards, as we only care about the stages when a card can be played to tableau or foundation, giving at most two relevant stages for each card.

Our model uses a number of domains: for simplicity we assume here the use of a standard deck of 52 cards. First, `CARDS` is an integer encoding of playing cards. Cards are ordered by complete suits in the order ♠, ♡, ♣, ♢. Within a suit cards are ordered in rank order from Ace to King. For example, the A♠ = 0, 2♠ = 1, A♡ = 13, with card 51 being K♢. We have additional domains: `SUITS` from 1 to 4 for spades to diamonds; `PILES` from 1 to 7 for the seven tableau piles; and `STAGES` for the possible stages a card can be played at. We sometimes need a dummy value of each type to represent a nonexistent card, pile, or stage: we introduce domains including these dummy values when necessary. For example, `DUMMY_STAGES` appends the dummy stage value to the STAGES domain.

The first decision variables in our model are a vector of variables `first_moved_stage`, giving the stage that each card's first move is made, together with a number of other vectors of variables which determine when certain moves become possible. The stage at which it is possible to tableau-build onto each card is given by `tableau_face_up_stage`, i.e. when the card is face-up in the tableau and therefore can be built on. The stage at which a card is on foundation is given by `foundation_stage`, which is also the stage that the next card in rank can be played to foundation. The first stage when a stock card can be played is given by `stock_available_stage`, irrespective of the availability of a place to build to. For Kings' move to spaces: `stage_first_space` gives the stage at which a pile can act as a space to receive a King; `king_pile` gives the pile a given king is moved to a space in, or 0 if none; and `king_played_space_stage` gives the stage at which a king is moved to a space. The snippet shows the decision variables of the model.

```
find first_moved_stage : matrix indexed by [ CARDS ] of STAGES
find tableau_face_up_stage : matrix indexed by [ CARDS ] of DUMMY_STAGES
find foundation_stage : matrix indexed by [ CARDS ] of STAGES
find stock_available_stage : matrix indexed by [ CARDS ] of STAGES
find stage_first_space: matrix indexed by [ PILES ] of STAGES
find king_pile: matrix indexed by [ SUITS ] of int(0..numPiles)
find king_played_space_stage : matrix indexed by [ SUITS ] of DUMMY_STAGES
```

4.2 Constraints

In the following, `tableau[pile,i]` represents the initial tableau; the constant matrix `tableau_parents[card,j]` (where `j` is 1 or 2) represents the two cards that `card` can

[1] https://github.com/turingfan/Modref2024-Patience/

be tableau-built onto; and `tableau_twin[card]` indicates for each card which 'twin' card has the same colour and rank but different suit to `card`.

The constraint below states that a card's first move has to be after either the previous card in its suit is available for foundation building, or after one of its tableau parents is available for tableau building. There are two cases, depending on whether the card is ever played to tableau or not. Kings played to the tableau, and all Aces, are not included in these constraints.

```
forAll card : CARDS .
    ... $ if card is not ace                                    (detailed constraints omitted)
    /\ tableau_face_up_stage[card] = dummyStage  $ and card not played to tableau
    ->  first_moved_stage[card] > foundation_stage[card - 1],

forAll card : CARDS .
    ... $ if card is not king or ace                            (detailed constraints omitted)
    /\ tableau_face_up_stage[card] != dummyStage $ and card was played to tableau
    -> first_moved_stage[card] > foundation_stage[card - 1]
          \/ first_moved_stage[card] > min(tableau_face_up_stage[tableau_parents[card, 1]],
                                           tableau_face_up_stage[tableau_parents[card, 2]]),
```

The following constraint enforces Rule 1, that face-down cards can only be played or built on after their covering card is first moved.

```
forAll pile : PILES . forAll i : int(0..numCards) .
    ... $ if card tableau[pile,i-1] is face-down                (detailed constraints omitted)
    -> first_moved_stage[tableau[pile, i - 1]] > first_moved_stage[tableau[pile, i]]
       /\ foundation_stage[tableau[pile, i - 1]] > first_moved_stage[tableau[pile, i]]
       /\ tableau_face_up_stage[tableau[pile, i - 1]] > first_moved_stage[tableau[pile, i]],
```

Rule 2 is that a card cannot be tableau-built until after at least one of the cards of opposite colour and one higher rank is face-up in the tableau. It is expressed as follows:

```
forAll card : CARDS .
    ... $ if card is not already in tableau and is not king     (detailed constraints omitted)
    ->  tableau_face_up_stage[card] > min(tableau_face_up_stage[tableau_parents[card, 1]],
                                          tableau_face_up_stage[tableau_parents[card, 2]]),
```

Rule 3 prevents cards being foundation-built until the preceding card of the same suit is.

```
forAll card : CARDS .
    ... $ if card is not ace                                    (detailed constraints omitted)
    ->  foundation_stage[card] > foundation_stage[card - 1],
```

Rule 4 is that two different cards cannot be tableau-built to the same card at the same time. This is clearly necessary in the original game but it is less clear that it will have an important effect in the relaxation. In fact it changes many layouts from being apparently possible to provably impossible: in informal experiments this increased by 15-20% the number of layouts confirmed impossible. We express this with a constraint which applies when two twin cards (defined earlier) are both tableau-built and they have to be played to the same parent because the other parent is not face-up in the tableau until both twins have been played to foundation. In that situation adds a disjunction that one of the twins has to leave the tableau parent to foundation before the second twin is played to the parent. We quantify over `HALF_CARDS` which is the set of cards covering just one suit of each colour instead of both.

```
$ two cards cannot be played to the same tableau parent at the same time
forAll card : HALF_CARDS .
    ... $ if card is not king                                    (detailed constraints omitted)
        $ and both the card and its twin are played to tableau before foundation
    /\ (exists i : int(1..2) . $ exists a tableau parent which isn't dummy
        tableau_parents[card, i] != dummyCard
        /\ tableau_face_up_stage[tableau_parents[card, i]] >
                  max(first_moved_stage[card],first_moved_stage[tableau_twin[card]]))
            $ can only act as a tableau parent after the card twins have been played
     -> (first_moved_stage[card] > foundation_stage[tableau_twin[card]]
         \/ first_moved_stage[tableau_twin[card]] > foundation_stage[card]),
```

Rule 5 concerns playing Kings to spaces. These constraints are not particularly complicated but do involve a number of interrelated constraints which link the various variable vectors involving Kings, spaces and the tableau together. Note that the last quoted constraint enforces the exclusion principle on kings being in the same place at the same time.

```
forAll pile : PILES . $ non-empty piles act as a space after the stage the bottomost card has been moved
    firstFaceUpCards[pile] > 0 -> (stage_first_space[pile] = 1 + first_moved_stage[tableau[pile, 1]]),
forAll suit : SUITS . $ Enforcing "dummy" pile and stages for king variables
    (king_played_space_stage[suit] = dummyStage) = (king_pile[suit] = 0),
$ A king must be played to a space or to the foundation
forAll suit : SUITS .  king_pile[suit] = 0 .
    ... $ if the king hasn't been moved to a space    (detailed constraints omitted)
    -> first_moved_stage[(numRanks * suit) - 1] > foundation_stage[(numRanks * suit) - 2],
forAll suit : SUITS .  king_pile[suit] > 0 .
    ... $ if the king is  moved to a space    (detailed constraints omitted)
    ->  first_moved_stage[(numRanks * suit) - 1] = king_played_space_stage[suit]
        \/ first_moved_stage[(numRanks * suit) - 1] > foundation_stage[(numRanks * suit) - 2],
forAll suit : SUITS . $ A king not played to a space must be played to foundation
    king_pile[suit]=0 -> first_moved_stage[(numRanks*suit)-1] > foundation_stage[(numRanks*suit)-2],
forAll suit : SUITS . $ A king must be played to a pile after it is available
    king_pile[suit] > 0 -> (king_played_space_stage[suit] >= stage_first_space[king_pile[suit]]),
$ a bottommost tableau king never needs to be played to a different space
forAll pile : PILES .  $ it would not be incorrect to do so but would never be useful
    firstFaceUpCards[pile] > 0 /\ tableau[pile, 1] % numRanks = numRanks - 1 ->
        (king_pile[(tableau[pile, 1] / numRanks) + 1] = 0),


forAll suit : SUITS . $ non-tableau kings act as a tableau parent after they are played to the tableau
    ... $ king of this suit is not in the tableau originally    (detailed constraints omitted)
    -> tableau_face_up_stage[(numRanks * suit) - 1] = king_played_space_stage[suit],

forAll suit, otherSuit : SUITS . $ Two kings can't be in the same space at the same time
    ... $ if kings of two suits are played to the space in the same pile    (detailed constraints omitted)
    -> ( (king_played_space_stage[suit] > foundation_stage[(numRanks * otherSuit) - 2])
            \/  (king_played_space_stage[otherSuit] > foundation_stage[(numRanks * suit) - 2])
```

Finally, we consider the constraints on the order cards that are played from the stock. For the total relaxation, equivalent to deal-1, we do not need any constraints at all as all stock cards can be played at any time as long as they have somewhere to be built to. For the partial relaxation, we need to ensure that any constraints on the order stock cards are played also feed in to the stages they can be built to foundation or tableau.

```
forAll card : CARDS . $ Linking stock_available_stage with other stage vectors
    stock_set[card] ->  first_moved_stage[card] > stock_available_stage[card]
                        /\ foundation_stage[card] >= stock_available_stage[card]
                        /\ tableau_face_up_stage[card] >= stock_available_stage[card],
forAll i : int(1..max_stock_index) .
    ...  $ indexes i and i+1 are in the same stock group of size n (detailed constraints omitted)
    ->  stock_available_stage[stock[i]] > first_moved_stage[stock[i + 1]])
    \/ $ or a card has been played in the stock below the current group
    exists k : int(1..i) .  ... $ k is the largest index below the stock group (details omitted)
       /\ stock_available_stage[stock[i]] > min([first_moved_stage[stock[j]] | j : int(1..k)]),
```

For the strict version of stock, we include the above relaxed stock constraints, but also introduce new vectors of decision variables. Two vectors are channelled together to give the order stock cards are played in and when each stock card is played. First, there are four different conditions which can each make it legal to play a stock card, leading to a four-way disjunction. Second, they each apply to the state of the stock at the time the card is played. For this, we therefore have to compute the number of cards still in the stock at the relevant time, leading to numerous calculations on reified inequalities.

```
find stock_order: matrix indexed by [ STOCK_INDICES ] of STOCK_INDICES
find stock_index: matrix indexed by [ STOCK_INDICES ] of STOCK_INDICES
allDiff(stock_order),
forAll i : STOCK_INDICES . stock_index[stock_order[i]] = i,
forAll order : int(2..max_stock_index) .
    stock_available_stage[stock[stock_index[order]]] > stage_played[stock[stock_index[order - 1]]],
forAll stock_card : STOCK_INDICES . $ must obey one of four constraints
      $ Stock card  is played when is now last card in stock
```

Stock Rule	Count	Nodes	SR Time	Solver	Total	Max total
Total relaxation	**10000**	**13311**	**1.078**	**0.081**	**1.159**	**11.214**
Unwinnable	565	8890	0.761	0.146	0.907	11.214
Possibly Winnable	9435	13576	1.097	0.077	1.174	2.498
Partial relaxation	**10000**	**24303**	**1.177**	**0.204**	**1.381**	**21.762**
Unwinnable	1042	54884	0.966	1.029	1.995	21.762
Possibly Winnable	8958	20746	1.202	0.108	1.309	2.837
Strict	**10000**	**885253**	**4.666**	**30.264**	**34.931**	**1373.541**
Unwinnable	1328	881784	3.921	37.886	41.807	1373.541
Possibly Winnable	8672	885784	4.781	29.097	33.878	424.635

Table 1 Experimental results on Klondike using our models with different stock relaxations. The first line for each Stock Rule gives the total, with breakdowns on result in the following two lines. The nodes is the mean nodes reported by Kissat. Mean times in seconds are given for Savile Row, Kissat solving, and their sum. The final column gives the maximum of total time for any layout.

```
    (forAll higher_index : int(stock_card + 1 .. max_stock_index) .
        stock_order[higher_index] < stock_order[stock_card])
    \/ $ Or card is now immediately underneath previously played card
    (exists higher_index : int(stock_card + 1 .. max_stock_index) .
        stock_order[higher_index] = stock_order[stock_card] - 1
        /\ (sum i : int(stock_card+1 .. higher_index-1) . stock_order[i] > stock_order[stock_card]) = 0)
    \/ $ Or card is now at a multiple of deal n from the start of the stock
    ((sum i : int(1..stock_card - 1) . stock_order[i] > stock_order[stock_card]) % dealN = dealN - 1)
    \/ $ Or card is now multiple of deal n later in the stock than previous stock card
    (stock_card > stock_index[stock_order[stock_card]-1] $ after card just played
         /\ (sum i : int(1..stock_card - 1) .
             stock_order[i] > stock_order[stock_card]) % dealN = dealN - 1
             /\ i > stock_index[stock_order[stock_card] - 1]),
```

We make some general points about our models. They all allow correctly for worrying-back, allowing the foundation build to be before the tableau build. As well as the search choices discussed earlier, the stock order requires search, so we do not capture impossibility using only propagation. Finally, we do not attempt to constrain exactly the stages played, leading to redundant solutions. We do this because we are interested in proving impossibility, which should be detected by constraints leading to failure, and extra solutions on solvable instances often allow solvers to work quicker.

5 Experimental Results

The model and parameter files are written in Essence Prime [10] and solved with Savile Row [11] version 1.10.0 with some minor changes, built using the Graal JDK (`https://graalvm.org`) version 22.0.1. The backend solver was Kissat 3.1.1 (`https://github.com/arminbiere/kissat`). Experiments were run on the Cirrus HPC system. CPU Nodes contain 2xIntel Xeon "Broadwell" 18-core CPUs, 2.1 Ghz, and 256 GB RAM.

Solvitaire's code and experimental results on a line-by-line basis are openly available [2, 5]. This allows us to compare our results in detail with Solvitaire on identical instances: in doing so it is important to point out that our models can *only* prove unwinnability, so can never demonstrate winnability. Table 1 gives our experimental results on the first 10,000 seeds in the Solvitaire experimental data set [5]. Most importantly, none of the instances any model reported as impossible were shown to be winnable by Solvitaire. While not a guarantee of correctness, it is a great reassurance especially considering how many layouts were claimed

to be impossible. Our result of 13% of layouts unwinnable in Table 1 greatly improves on any previous approach for proving games unwinnable without exhaustive state-based search. Previous results have been 2.24% of layouts proved unwinnable [14], 3.33% [4] and 8.56% [1]. Of the 10,000 seeds, Solvitaire reports that 8,131 are winnable, 1,868 unwinnable, and one unknown [5]. This means that when using the full stock rules, our model was able to prove that 71% of the impossible layouts were unsolvable in the relaxed version of Klondike.

Using the relaxed stock rule, more than 55% of unwinnable layouts were proved using an average of just over 1s CPU time. For the total relaxation, recall that this corresponds precisely to deal-1 Klondike, for which Solvitaire showed exactly 1000 as unwinnable with 11 unknown. The 565 therefore again represents 55% of unwinnable layouts. The mean runtime of Solvitaire on the 10,000 instances (as reported in [5]) was 44s but on different hardware so a direct comparison is not appropriate. But our results do suggest that it would be very beneficial to use the two solvers together and likely to result in greatly reduced total runtime. We report on our investigations into this in Section 6

We also experimented on instances which Solvitaire was unable to resolve either way among the 1,000,000 it experimented on [5]. Solvitaire left 157 layouts unresolved for deal-3 and 1145 for deal-1 Klondike. Our models were able to prove 63 were unwinnable for regular Klondike and as many as 522 for deal-1. The longest CPU time for any of these instances was less than 4s for deal-1 and less than 213s for deal-3. This compares extremely favourably with the fact that Solvitaire failed on all these instances after hours of CPU time each. Solvitaire gave the 95% confidence interval of the winnability of deal-3 Klondike as $81.945 \pm 0.084\%$ and of deal-1 as 90.480 ± 0.116 [3]. Our resolution of many unknown layouts allow us to improve those estimates to 81.942 ± 0.081 for deal-3 and 90.454 ± 0.090 for deal-1, using the same calculation method [3]. Though a small improvement, this is the first time deal-1 Klondike has been estimated with a 95% confidence interval under 0.1%.

6 Scheduling Solvers

Although our proposed models cannot solve winnable instances, our experimental results in Section 5 show that they can prove unwinnability very quickly for a large number of cases. This observation suggests that it can be beneficial to build a *schedule* that combines the complementary strengths of our relaxed models and the Solvitaire solver. In this section, we consider various options for building such schedules and empirically evaluate them against the individual solving approaches.

We adopt four individual solving approaches in our study, three of which are expected to run quickly but are *incomplete*, i.e., they can prove unwinnability but cannot demonstrate winnability or vice versa, while the last one, Solvitaire, aims at covering the instances not determined by the former ones. The three incomplete solving approaches include:

- Two unwinnable-targeting solvers: these include the relaxed stock and full stock rule models (shown as `Partial relaxation` and `Strict` in Table 1). We omit the `Total relaxation` model since the `Partial relaxation` model proves almost twice as many layouts unwinnable for a comparatively small extra cost.
- One winnable-targeting solver: this is a *streamlined* version of the Solvitaire solver [2]. “Streamlined" means that it makes simplifying assumptions about the form of the solution which often greatly speed up the finding of a solution. However, the assumptions may be false so the solver can be effective at demonstrating the winnability of a layout but it cannot prove unwinnability. It therefore has a dual nature compared to the unwinnable-targeting constraint models.

Naive schedules. Since the incomplete approaches often finish in a short amount of time, to utilise the complementary strengths of both solving approaches, an intuitive approach is to build a schedule that runs a chosen incomplete solver first, followed by Solvitaire if the former one does not return a guaranteed answer. We consider three schedules of this type in our evaluation: `relaxed -> Solvitaire`, `strict -> Solvitaire`, and `streamliner -> Solvitaire`, which corresponds to running the `Partial relaxation`, `Strict`, or streamlined Solvitaire approach, respectively, to completion, followed by Solvitaire.

Constraint-based schedules. A more generic approach to building an effective algorithm schedule is to model the scheduling task as an optimisation problem [12, 8]. Concretely, given a *training* instance set $\mathcal{I}$, a portfolio of n algorithms $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ with *complementary* strengths, and a cutoff time T that specifies the time limit we can spend on solving each instance. Assuming that we can collect information about the performance of all algorithms in the portfolio $\mathcal{A}$ on the instance set $\mathcal{I}$, an algorithm schedule $\int$ is defined as a sequence of algorithms chosen from $\mathcal{A}$ and the maximum amount of solving time assigned to each algorithm in the sequence (the total amount of time must not exceed the cutoff time T). The scheduling problem aims at finding a schedule that can solve as many instances in $\mathcal{I}$ as possible within the shortest amount of time. In this work, we model this optimisation problem as a constraint model and solve it using the chuffed constraint solver [2].

Note that the application of a schedule in the *test* phase (i.e., on instances unseen during the schedule building process) is slightly different. Given that an incomplete algorithm may finish before its maximum assigned runtime without producing a conclusive answer, we allocate the total *leftover time* (the positive difference between the maximum execution time of all incomplete algorithms in the schedule and the actual execution time of those algorithms when inconclusive) to the last complete algorithm in the schedule (not surprisingly, in our experiments, Solvitaire is always the last algorithm in the optimal schedules returned by our scheduling constraint model). This approach ensures that we can utilise the whole cutoff time during the solving process on unseen instances.

Current algorithm scheduling techniques only make use of each solver once in the schedule, however, in a small preliminary study, we observe that it can be beneficial to allow a solver to be repeated in the schedule. For example, on a 40 second timeout, the optimal schedule suggests running `Solvitaire` first with a cutoff of 0.3 seconds, then `relaxed`, then `streamliner`, and finally repeating `Solvitaire` for the remaining time. This might seem counterintuitive but is explained by the fact that `Solvitaire` is able to prove some instances winnable and others unwinnable in the 0.3s timeout.

Performance metric (PAR10). Note that the objective of our scheduling task consists of two components: (i) the number of instances solved by the schedule, and (ii) the average solving time on the solved instances. A typical approach to aggregate them into a single objective is to use the *Penalised Average Runtime (PAR10)* [9], where the runtime of each unsolved instance is counted as ten times the cutoff time. We will adopt this metric as the objective function when building our schedule and in our evaluation.

The constraint model for this scheduling task together with a detailed explanation on the model are available at `https://github.com/turingfan/Modref2024-Patience/`. It is important to note that our scheduling constraint model allows an arbitrary number of repetitions for each individual approach, however, for practical reasons (due to limited computational resources), in our experiments, we limit the number of repetitions of each solving approach in the schedule to two.

[2] `https://github.com/chuffed/chuffed`

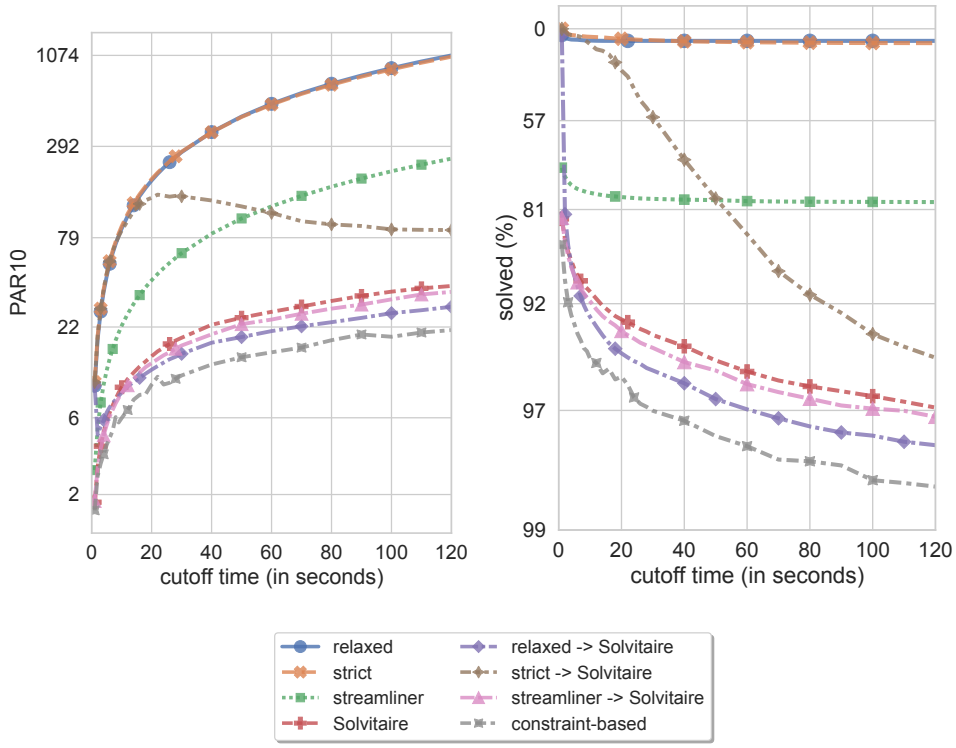

Figure 3 Performance of four individual solving approaches (`relaxed`, `strict`, `streamliner`, and `Solvitaire`), three naive schedules (`relaxed -> Solvitaire`, `strict -> Solvitaire`, and `streamliner -> Solvitaire`), and the constraint-based schedule (`constraint-based`), on different cutoff times (ranging from 1 second to 120 seconds). The performance is measured on a test set that includes 9000 instances: i) using PAR10 score (left plot, smaller is better); and ii) the percentage of instances solved by each approach (right plot, larger is better, note that the y-axis is presented in a reversed direction for visual effect). The values on the y-axis in both plots are shown in log scale (with base 10).

We compare performance of the three naive schedules and the constraint-based one described above against the four individual solving approaches on different cutoff times. The PAR10 score and the percentage of instances solved by each approach are presented in Figure 3. Intuitively, the larger the cutoff time, the less advantage a scheduling of solvers could offer compared to just using Solvitaire alone. Therefore, in this study, we choose to limit the cutoff time up to 120 seconds. Among the 10,000 instances available, we take 10% of them as the training set to build the constraint-based schedule. The results reported in Figure 3 are based on the remaining 9,000 instances.

Figure 3 clearly indicates the advantage of building an optimal schedule using our scheduling constraint model, as it is the winner for all cutoff times in the considered range. The two individual unwinnable-targeting solvers (`relaxed`, `strict`) perform equally badly, followed by the winnable-targeting solver (`streamliner`). Interestingly, the naive schedule `strict -> Solvitaire` performs worse than the `streamliner` solver when the cutoff time is less than one minute, but after that, it starts catching up and beating the `streamliner`. This can be explained by the fact that our `strict` constraint model often takes more than 30s to run (as shown in Table 1) compared to the `relaxed` model, and therefore its effectiveness

is only shown when the schedules are allowed a sufficient amount of time. Nevertheless, this naive schedule is the worse among the four schedules evaluated. The three other schedules (`relaxed -> Solvitaire`, `streamliner -> Solvitaire`, and `constraint-based`) do offer improvement over using the Solvitaire solver alone in all cases, confirming our hypothesis about the advantage of combining the individual approaches in a solving schedule for our case study.

7 Conclusions

We have described constraint models which can prove layouts of Klondike unwinnable. The models are based on the idea of finding an 'unblocked artifact', a solution to a relaxed variant of the game which proves that no blocking set can exist for this relaxation. Our models have passed extensive testing by never stating a winnable layout of Klondike as impossible. Our constraint models are able to prove many layouts unwinnable that the state-of-the-art solver Solvitaire is unable to resolve.

As a result we are able to slightly improve the best-known estimate of winnability of thoughtful Klondike. On many other layouts our models prove them unwinnable much faster than Solvitaire, suggesting that the two would be very appropriate to use together, for example in a portfolio solver. We therefore investigate the potential of building a schedule of solvers by modelling the scheduling task itself as a constraint model. Our results on a range of cutoff time limits up to 120 seconds reveal that it is indeed beneficial to combine the proposed relaxed models with the state-of-the-art solver Solvitaire: the obtained schedules perform better than both Solvitaire and the individual models on the whole range of cutoff time considered.

In future work, we would like to build up the models to solve the full game of Klondike. We would also like to extend this work to other patience and solitaire games, as well as investigating the applicability of our modelling approach to other domains.

References

1 Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. Searching solitaire in real time. *ICGA Journal*, 30(3):131–142, 2007. `doi:10.3233/ICG-2007-30302`.

2 Charlie Blake and Ian Gent. thecharlesblake/Solvitaire: Release for Zenodo DOI-issuing (v0.10.2), November 2019. Zenodo. `doi:10.5281/zenodo.3529524`.

3 Charlie Blake and Ian P. Gent. The winnability of klondike solitaire and many other patience games, 2023. `arXiv:1906.12314`.

4 Johan de Ruiter. Counting classes of klondike solitaire configurations. Technical Report Internal Report 2012-9, Leiden Institute of Advanced Computer Science, 2012. URL: `https://theses.liacs.nl/pdf/2012-09JohandeRuiter.pdf`.

5 Ian P. Gent and Charles Blake. Patience Experimental Results (Version 5), 1 2023. URL: `https://figshare.com/articles/Patience_Experimental_Results/8311070/5`, `doi:10.6084/m9.figshare.8311070.v5`.

6 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 Communications*, 20(3):211–226, 2007.

7 Paul Jensen. Celebrating 30 years of Microsoft Solitaire with those oh-so-familiar bouncing cards, May 2020. Xbox.com. URL: `https://web.archive.org/web/20200522210053/https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/`.

8 Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–*

CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17, pages 454–469. Springer, 2011.

9 Marius Lindauer, Jan N van Rijn, and Lars Kotthoff. The algorithm selection competitions 2015 and 2017. *Artificial Intelligence*, 272:86–100, 2019.

10 Peter Nightingale. Savile Row manual, 2021. URL: https://arxiv.org/abs/2201.03472, doi:10.48550/arXiv.2201.03472.

11 Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

12 Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

13 Barbara M. Smith. Caching search states in permutation problems. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 637–651, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11564751_47.

14 M Voima. Klondike solitaire solvability. Technical Report Internal Report 2012-9, Tampere University of Applied Sciences, 2021. URL: https://urn.fi/URN:NBN:fi:amk-2021060213520.

A Appendix: Glossary of Patience/Solitaire Terms

While extremely widely known, both terminology and precise rules of Klondike can be unclear and/or vary. We have given the rules we use in the main text, but for precision and reviewers' convenience we define the key terms used in those rules, following the "Terminology of Patience Games" section by Blake and Gent [3]. We describe these in terms of the standard western deck of 52 cards.

Colour: Hearts and Diamonds are red while Spades and Clubs are black.

Deal-n: The number of cards that are moved from the stock to waste in a single movement. Standard Klondike is deal-3. A major variant is deal-1: with infinite redeals allowed, this in fact allows us to play the stock in any order.

Deck: A set of 52 cards, comprising one copy of each rank/suit combination.

Face-down: A tableau card placed with its reverse facing up, not allowed to be moved or played to.

Foundation: Initially empty piles in which to build up from A to K in suit. Only Aces can be placed in an empty pile. Cards can be moved to the foundation pile, provided that the card of the same suit and one lower in value is already in the same pile.

Group: A sequence of cards in a tableau pile which are all face-up and in valid build sequence. All the cards may be moved together in the tableau if the highest card in the sequence is legal to be moved.

Hidden: A card is hidden when it is face-down in the tableau.

Layout: A particular placing down of the deck onto the tableau and stock, typically random.

Pile: One of 7 locations in the tableau that we can build cards down in alternating colours. If all face-up cards are moved from a pile, the topmost face-down card on that pile is turned face-up

Rank: One of (in order) Ace (valued as 1), 2 to 10, Jack (11), Queen (12), King (13).

Redeal: Once the stock is exhausted we are allowed to start from the beginning again, keeping the stock cards in the same order.

Space: A tableau pile that currently has no cards in it.

Stock: The 24 cards of the deck not in the tableau are initially placed in an ordered pile called the 'stock'. In deal-n Klondike, we can draw n cards from the stock and place them in a separate pile named 'waste', maintaining the order from the stock.

Suit: One of 'Spades' (written ♠), 'Hearts' (♡), 'Clubs' (♣), and 'Diamonds' (♢)

Tableau: The 7 piles which initially range from 1 to 7 cards so contain 28 in total. Only the top card of each pile is originally face-up. Face-up cards can be moved from one pile on the tableau to another tableau pile, provided that the card they are being placed on is of a different colour and that the value of the card being placed is one less than the card being placed on.

Thoughtful: The variant of Klondike we study in this paper, where we know the location of all cards at the start of the game but the rules about cards being unplayable when face-down still apply.

Waste: The pile we move cards from the stock to. The topmost card from the waste can be placed on a tableau pile or a foundation pile, if it is legal to do so. When no more stock cards exist we can take the waste as the stock again and redeal.

Win: A layout is considered won when each Foundation contains the relevant K.

Worrying-back: Building a card from the foundation to a legal place on the tableau.

B Appendix: Full Explanation Of The Sample Blocking Set

We will show that the cards 5♢, 4♠, 8♢, 10♢, 8♡ collectively form a *blocking set* for the layout in Figure 2: i.e. none of these cards can make its first move until at least one of the others has before it. The cards underneath the blocking set are *blocked* and can neither be played or built on.

Red 8s cannot be foundation-built. Start by noting that both 8♢, 8♡ cannot be foundation-built until after the 7 of the same suit has been (rule 3). But each red 7 is hidden by the same suit's 8 so cannot be built on the foundation until after the 8 has moved (rule 1).

Red 8s cannot be tableau-built if the 4♠ cannot move. The 9♠ is blocked and cannot be built on until the 8♡ has moved (rule 2), so one of the red 8s would have to be tableau-built on 9♣. The 9♣ is also blocked by the 4♠ from the blocking set, so we have to show that the 4♠ cannot move.

4♠ cannot be foundation-built if 10♢ cannot move. It cannot be foundation-built until the 3♠ is available, but that is blocked by the 10♢.

10♢ cannot move. We can certainly tableau-build the J♣ onto Q♡ and then 10♡ onto the same J♣. But the J♠ is blocked by the 8♢ so the 10♢ cannot be tableau-built until the 10♡ is foundation-built (rule 4). Also, neither 10♡ nor 10♢ can be foundation-built until their matching 8 is.

4♠ cannot be tableau-built. The 4♠ cannot be built on either red 5. The 5♡ is blocked, but as part of our blocking set, the 5♢ is potentially available to be tableau-built on. However, to do so both Q♠ and Q♡ would have to be moved. Their foundation-builds depend on the blocked cards 9♠, 5♡ and they can only be tableau-built on Kings. These are all in stock so have to be moved to spaces first (rule 2). The blocking set only allows spaces in piles 1 and 2, but making both spaces would depend on moving the Q♣ which in turn requires a third K as its foundation-build depends on the blocked 5♣. So to play both Q♠ and Q♡, we need three spaces when only piles 1 and 2 can create spaces. But we can't have three Kings occupying two spaces at the same time (rule 5).

5♢ cannot move. The remaining card in the blocking set is 5♢ so we have to show this cannot move. Its foundation move is impossible because of the 4♢. But it can only be tableau-built to either 6♠ or 6♣. These are in the stock so have in turn to be built to a red 7, but both the 7♡ and the 7♢ are blocked and not playable to.

As promised, we have shown that each card in the blocking set cannot be moved until at least one of the other cards has been before it, so none can ever be moved.