Modelling and Solving English Peg Solitaire

Christopher Jefferson, Angela Miguel, Ian Miguel[*]
Artificial Intelligence Group, Department of Computer Science,
University of York, Heslington, York, YO10 5DD, UK
{caj,angiem,ianm}@cs.york.ac.uk

S. Armagan Tarim
Department of Management
Hacettepe University, Ankara, Turkey
armagan.tarim@hacettepe.edu.tr

Abstract

Peg Solitaire is a well known puzzle, which can prove difficult despite its simple rules. Pegs are arranged on a board such that at least one 'hole' remains. By making draughts/checkers-like moves, pegs are gradually removed until no further moves are possible or some goal configuration is achieved. This paper considers the English variant, consisting of a board in a cross shape with 33 holes. Modelling Peg Solitaire via constraint or integer programming techniques presents a considerable challenge and is examined in detail. The merits of the resulting models are discussed and they are compared empirically. The sequential nature of the puzzle naturally conforms to a planning problem, hence we also present an experimental comparison with several leading AI planning systems. Other variants of the puzzle, such as 'Fool's Solitaire' and 'Long-hop' Solitaire are also considered.

Keywords: Constraint Programming, Integer Programming, Modelling, Symmetry, Planning.

[*]Corresponding author.

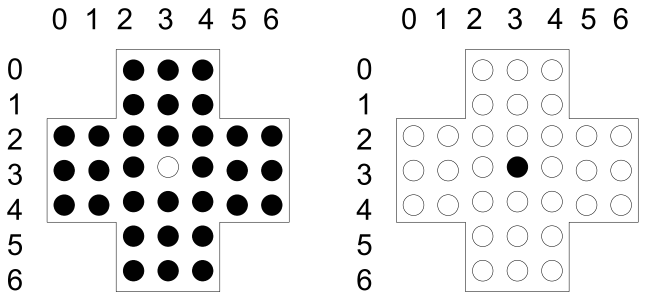

Figure 1: The Solitaire board and first and last states of central Solitaire.

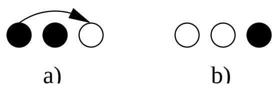

Figure 2: Making a move in Solitaire.

1 Introduction

Peg Solitaire[1] is played on a board with a number of holes. We consider the English version, which uses a cross-shaped board with 33 holes (Figure 1). Pegs are arranged on the board so that at least one hole remains. By making horizontal or vertical draughts/checkers-like moves (Figure 2), the pegs are gradually removed to obtain a goal configuration. Many variants of the game are *reversals*: the objective is to reverse the starting position, replacing pegs with holes and vice-versa. The classic 'central' Solitaire (Figure 1) is one such example, where the objective is to leave a peg in the central hole.

There is a rich literature on this problem, which has been studied for many years. Beasley [3] surveys the mathematical results known at the time of writing and discusses many closely related puzzles. The major results on Peg Solitaire can be found in Uehara and Iwata [25] on the NP-completeness of generalised Solitaire; De Bruijn [9] on the necessary conditions for feasibility, based on a finite field; Berlekamp, Conway and Guy [5] on the necessary conditions for feasibility, using Pagoda functions; Avis and Deza [2] on Solitaire cone and the polyhedral approach; and Moore and Eppstein [22] on the one-dimensional problem. Strategies and symmetries are discussed in Beeler et al. [4].

This paper concerns modelling and solving Peg Solitaire using constraint and mathematical programming techniques. The problem is of interest because it is highly symmetric and because of its sequential nature more usually tackled using AI planning. Interesting optimisation variants, such as 'Fool's' and 'Long-hop' Solitaire, may also be studied in the same framework. Integer and constraint programs are first developed for the standard version of the problem. The constraint model is enhanced through symmetry breaking and both models are improved via a metric enabling the early detection of dead ends. Both models are then compared empirically against leading AI planners on a number of variations of the game. Finally, we consider the modifications necessary to our models in order to apply them to optimisation variants of Solitaire.

[1]Problem 38 at www.csplib.org.

2 Modelling English Peg Solitaire

The possible *transitions* of a peg from one position on the board to another are defined using a coordinate system, as per Figure 1. We adopt the convention that x, y is the xth column and yth row and denote a transition from x, y to x', y' as $x, y \to x', y'$. We define a *move* as a transition at a particular time-step, t.

Solitaire can straightforwardly be characterised as an AI planning problem [1], since it involves transforming an initial state into a goal state using a sequence of moves. It is, however, simpler than a general AI planning problem in two important respects. First, only a single move is allowed at any time-step. Second, it is possible to determine exactly how many moves are necessary to achieve the goal state, as the following lemma shows.

Lemma 1 *Single-peg English Peg Solitaire reversals require 31 moves to solve.*

Proof Each move removes exactly one peg. We begin with 32 pegs. Since the objective is to have one peg remaining, 31 moves are required. QED.

Trivially, similar results hold for more complex initial/goal conditions. These two characteristic enable the construction of simpler integer and constraint programming models than would be possible for a general AI planning problem, as will be discussed.

2.1 Modelling via Integer Programming

Finding a solution to Peg Solitaire may be formulated as an integer programming model (referred to herein as model A, given below). In the terminology of Kautz and Walser [18], our model is an *operator-based* encoding of the problem (also known as the *regular* encoding [27]), since it employs two types of variable: the first records whether an action (in our case, a move) occurs at a particular time-step, and the second records whether or not a *fluent* (time-varying condition, in our case whether a position on the board is occupied) holds. An integer variable, $\mathbf{M}[i, j, t, d]$, is equal to 1 if a move is made from a board position $(i, j) \in B$, where B is the set of 33 holes on the board, in time-step $t = 1, ..., 31$, in direction $d \in \{N, S, E, W\}$. Similarly, an integer variable, $\mathbf{bState}[i, j, t]$, is equal to one if position $(i, j) \in B$ is occupied at time-step t, where $t = 1, ..., 32$.

$\mathbf{M}[]$ and $\mathbf{bState}[]$ are related by *precondition constraints* (Eqs 2–13) in the style of Vossen *et al* [27]. These constraints specify simply that an action implies its preconditions. For example, if transition 0 (i.e. 2,0→4,0) is to be made at time-step t, then the following preconditions must be satisfied:

1. There must be a peg at 2,0 (Eq 2).
2. There must be a peg at 3,0 (Eq 3).
3. There must be a hole at 4,0 (Eq 4).

In AI planning classical *frame axioms* describe which fluents are unchanged by an action. In Peg Solitaire, however, the set of actions that can affect a fluent is small. Hence, we use *explanatory* frame axioms ([14, 18, 27], Eq 14), where instead the set of actions that could have effected a change is specified. As noted in [27], explanatory frame axioms allow multiple parallel actions, necessitating conflict exclusion constraints to prevent contradicting actions from occurring together. Parallel actions are not, however, necessary to model Peg Solitaire, so conflict exclusion constraints are not needed. Instead, we simply state that there is exactly one move per time-step (Eq 15).

When integer programming is applied to AI planning problems, the objective function is often set to minimise the number of actions in the plan [27], or to optimise the use of some set of resources [18]. Neither objective is applicable to Peg Solitaire, so the objective function given in Eq (1) compels the last peg to be moved into the central hole in the final board state. This goal-based objective function is similar in spirit to that employed by Bockmayr and Dimopoulos [7]. Eq (1) is a substantial simplification of the objective function used previously [15], which minimised the sum of every position *except* the centre of the board, exploiting the fact that the problem constraints imply that there is only one peg on the board in the final state.

As we will see, model A performs well. In future, however, we will examine alternative encodings, such as the state-based encoding [16], which eliminates action variables.

$$\max_{\substack{M\in\{0,1\}\\ \mathbf{bState}\in\{0,1\}}} \mathbf{bState}[3,3,32] \tag{1}$$

s.t.

$$\begin{aligned}
&\mathbf{M}[i,j,t,E] \leq \mathbf{bState}[i,j,t] && (2) \qquad && \mathbf{M}[i,j,t,S] \leq \mathbf{bState}[i,j,t] && (8)\\
&\mathbf{M}[i,j,t,E] \leq \mathbf{bState}[i+1,j,t] && (3) && \mathbf{M}[i,j,t,S] \leq \mathbf{bState}[i,j+1,t] && (9)\\
&\mathbf{M}[i,j,t,E] \leq 1-\mathbf{bState}[i+2,j,t] && (4) && \mathbf{M}[i,j,t,S] \leq 1-\mathbf{bState}[i,j+2,t] && (10)\\
&\mathbf{M}[i,j,t,W] \leq \mathbf{bState}[i,j,t] && (5) && \mathbf{M}[i,j,t,N] \leq \mathbf{bState}[i,j,t] && (11)\\
&\mathbf{M}[i,j,t,W] \leq \mathbf{bState}[i-1,j,t] && (6) && \mathbf{M}[i,j,t,N] \leq \mathbf{bState}[i,j-1,t] && (12)\\
&\mathbf{M}[i,j,t,W] \leq 1-\mathbf{bState}[i-2,j,t] && (7) && \mathbf{M}[i,j,t,N] \leq 1-\mathbf{bState}[i,j-2,t] && (13)
\end{aligned}$$

$$\begin{aligned}
\mathbf{bState}[i,j,t]-\mathbf{bState}[i,j,t+1] = &\sum_{d\in\{E,W,S,N\}} \mathbf{M}[i,j,t,d]\\
&+\mathbf{M}[i-1,j,t,E]-\mathbf{M}[i-2,j,t,E]+\mathbf{M}[i+1,j,t,W]-\mathbf{M}[i+2,j,t,W]\\
&+\mathbf{M}[i,j-1,t,S]-\mathbf{M}[i,j-2,t,S]+\mathbf{M}[i,j+1,t,N]-\mathbf{M}[i,j+2,t,N] \qquad (14)
\end{aligned}$$

where $(i,j)\in B$, $\quad t=1,...,31$ in (2) - (14)

$$\sum_{(i,j)\in B} (\mathbf{M}[i,j,t,E]+\mathbf{M}[i,j,t,W]+\mathbf{M}[i,j,t,S]+\mathbf{M}[i,j,t,N]) = 1 \tag{15}$$

where $t=1,...,31$

2.2 Modelling via Constraint Programming

Our first constraint model (model B) is similar to model A in that it employs variables to record the moves taken. Rather than the **M**[] matrix, however, model B uses an matrix, **moves**[], which has 31 elements, one per move required to solve the problem. Like model A, this approach takes advantage of Lemma 1, but it also immediately captures the fact that only one move is allowed per time-step without the need for further constraints. The domain of each variable in **moves**[] is one of the 76 possible transitions, as enumerated in Table 1. Note how individual pegs are not distinguished: otherwise, there would be a much greater number of transitions to consider, many of which would be symmetrical since different pegs can be moved to the same coordinate position.

A further difference from model A is that model B specifies the problem constraints on **moves**[] alone to produce a novel, purely operator-based encoding. Recall the example of transition 0 above. Its preconditions can be checked by examining the sequence of moves before move t to ensure that the *most recent* moves made involving these three

No.	Trans.	No.	Trans.	No.	Trans.	No.	Trans.	No.	Trans.
0	2,0→4,0	16	2,2→2,4	32	2,3→0,3	48	1,4→3,4	64	6,4→6,2
1	2,0→2,2	17	2,2→0,2	33	2,3→2,1	49	1,4→1,2	65	6,4→4,4
2	3,0→3,2	18	3,2→3,0	34	2,3→2,5	50	2,4→0,4	66	2,5→4,5
3	4,0→2,0	19	3,2→5,2	35	2,3→4,3	51	2,4→2,2	67	2,5→2,3
4	4,0→4,2	20	3,2→3,4	36	3,3→1,3	52	2,4→4,4	68	3,5→3,3
5	2,1→4,1	21	3,2→1,2	37	3,3→3,1	53	2,4→2,6	69	4,5→2,5
6	2,1→2,3	22	4,2→4,0	38	3,3→3,5	54	3,4→1,4	70	4,5→4,3
7	3,1→3,3	23	4,2→6,2	39	3,3→5,3	55	3,4→3,2	71	2,6→4,6
8	4,1→2,1	24	4,2→4,4	40	4,3→2,3	56	3,4→5,4	72	2,6→2,4
9	4,1→4,3	25	4,2→2,2	41	4,3→4,1	57	3,4→3,6	73	3,6→3,4
10	0,2→0,4	26	5,2→3,2	42	4,3→4,5	58	4,4→2,4	74	4,6→2,6
11	0,2→2,2	27	5,2→5,4	43	4,3→6,3	59	4,4→4,2	75	4,6→4,4
12	1,2→3,2	28	6,2→6,4	44	5,3→3,3	60	4,4→6,4		
13	1,2→1,4	29	6,2→4,2	45	6,3→4,3	61	4,4→4,6		
14	2,2→2,0	30	0,3→2,3	46	0,4→0,2	62	5,4→3,4		
15	2,2→4,2	31	1,3→3,3	47	0,4→2,4	63	5,4→5,2		

Table 1: The set, T, of Solitaire transitions.

coordinates leave the board in the required state. Every transition, τ, involves three positions (and so preconditions, Pre(τ)). Therefore, each move is constrained as follows:

$$\forall t \in \{1,..,31\}\ \forall \tau \in T\ \forall p \in \text{Pre}(\tau)\ \forall \tau^+ \in \text{Support}(\tau,p)\ \forall \tau^- \in \text{Conflict}(\tau,p) :$$
$$\mathbf{moves}[t] = \tau \rightarrow \exists g : \mathbf{moves}[g]\tau^+ \wedge \neg\exists h : g < h < i \wedge \mathbf{moves}[h] = \tau^-$$

where Support() and Conflict() map onto a set of transitions that support and conflict with precondition p respectively. There is also the special case where the initial state establishes a precondition of τ.

Given 31 moves and 76 transitions for three preconditions, this leads to 7,068 constraints. However, each constraint can have a substantial size. In a format more suitable for input to a constraint solver each constraint looks like:

$$\begin{array}{rl} \mathbf{moves}[t] = \tau & \rightarrow \\ (\vee\{\mathbf{moves}[t-1] = \tau^+ | \tau^+ \in \text{Support}(\tau,p)\}) & \vee \\ ((\vee\{\mathbf{moves}[t-2] = \tau^+ | \tau^+ \in \text{Support}(\tau,p)\}) \wedge & \\ (\wedge\{\mathbf{moves}[t-1] \neq \tau^- | \tau^- \in \text{Conflict}(\tau,p)\})) & \vee \quad ... \end{array}$$

The worst case occurs when $t = 31$. By inspection, the largest size of Support(τ, p) is 4 and the largest size of Conflict(τ, p) is 8 when p requires a peg to be present, and symmetrically 8 and 4 when p requires an empty space. Since the Conflict() set appears more frequently, the right hand side of the above implication constraint can contain up to: $30 \times 4 + \frac{29(29+1)}{2} \times 8 = 3{,}600$ sub-terms.

Given the substantial size of Model B, it is not surprising to find that the constraint solver (Ilog Solver 5.3) exhausts available memory (256Mb) before all constraints in the model are even posted. Therefore, it is not considered further. It does, however, form the basis for our second constraint model (model C), which takes elements from both models A and B to produce an effective constraint model.

Given both **bState**[] and **moves**[], the constraint model is much simplified. Action pre- and post-conditions, along with explanatory frame axioms, are captured in a single set of constraints by exploiting the structure of the problem. There are 4 mappings of each board position from each time-step to the next: it could start empty or full, and end empty or full. The two cases where the state of the position remains the same can be combined, since these are caused by exactly the same set of moves. However the cases where the state of a position changes cannot be combined, since different transitions lead to the position changing from empty to full, and from full to empty.

We generate and post 3 constraints for each position at each time period, giving $1,023 * 3 = 3069$ constraints to specify the problem. For position i, j at time-step t:

1. $(\mathbf{bState}[i,j,t] = \mathbf{bState}[i,j,t+1]) \leftrightarrow (\wedge\{\mathbf{moves}[t] \neq \tau \mid \tau \in \text{Changes}(i,j)\})$
2. $(\mathbf{bState}[i,j,t] = 0 \wedge \mathbf{bState}[i,j,t{+}1] = 1) \leftrightarrow (\vee\{\mathbf{moves}[t] = \tau \mid \tau \in \text{PegIn}(i,j)\})$
3. $(\mathbf{bState}[i,j,t] = 1 \wedge \mathbf{bState}[i,j,t{+}1] = 0) \leftrightarrow (\vee\{\mathbf{moves}[t] = \tau \mid \tau \in \text{PegOut}(i,j)\})$

where Changes(i,j) is the set of transitions which change the state of i,j, and PegIn(i,j) and PegOut(i,j) are the sets of transitions which place a peg at and remove a peg from i,j respectively. At most 12 transitions can affect any one position, so the right-hand side of the first constraint type can have a maximum of 12 conjuncts. Similarly, a maximum of 4 and 8 transitions can place a peg at or remove a peg from a particular position.

Like the CPlan constraint-based planner [26], model C relies on a domain-specific encoding. A more general approach [10] uses the *planning graph* constructed by the Graphplan planner [6]. A planning graph is divided into levels (each represents a step in the plan) containing a set of proposition nodes and a set of action nodes. Level 0 contains only propositions describing the initial state. The graph is incrementally extended to a new level by adding a node for every action whose preconditions exist (and are mutually consistent) in the current level. Each such action node asserts its effects via new proposition nodes. Mutual exclusion constraints stipulate that two actions cannot appear together in the same step of a valid plan. A valid plan is a consistent sub-graph connecting proposition nodes expressing the goal conditions to the initial conditions. Identifying a consistent sub-graph can be modelled as a constraint program in a variety of ways [10, 21], which we will explore in future. This general approach, however, may limit our ability to exploit the structure of Peg Solitaire and hence give a weak model.

3 Pagoda Functions

Pagoda functions are useful to prove that some problems in Peg Solitaire are insoluble [5]. A value is assigned to each board position, so that for three positions a, b, c adjacent in a line, their corresponding values satisfy $a + b >= c$. The *Pagoda value* of a state for a specific Pagoda function is the sum of the values the Pagoda function takes at each occupied board position. Moves in Solitaire take a line a, b, c of positions, remove pegs from a and b, and place one in c. Hence, as moves are made the Pagoda value cannot increase. The *Pagoda condition* is that the Pagoda value for an intermediate position must never be strictly less than that of the goal position. Otherwise, it is not possible to extend the current partial solution to a full solution.

There are infinitely many Pagoda functions. However, for any particular problem some Pagoda functions are more useful than others. In an intermediate position it must be possible for the Pagoda function to take a smaller value than that of the goal state or the Pagoda condition will never be violated. It is beneficial for this to be possible for as many intermediate positions as possible to detect dead ends early. We consider three

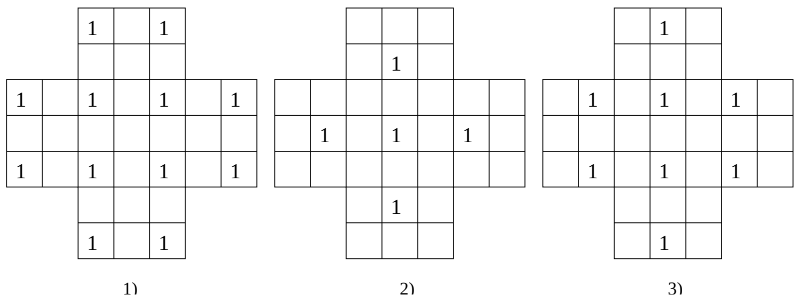

Figure 3: Three Pagoda Functions from [5].

Pagoda functions from [5] (Figure 3). Each is sparsely populated with non-zero values, increasing the likelihood of breaking the Pagoda condition and so pruning the search.

Adding a Pagoda function, represented by $pagvals(i, j)$, to models A or C, which explicitly record the state of the board, is straightforward. A new matrix of variables, **Pagoda**[] with an element per board state, is added, and the following are imposed:

1. $\mathbf{pagoda}[t] = \sum_{i,j} pagvals(i, j) \times \mathbf{bState}[i, j, t]$
2. $\mathbf{pagoda}[t] \geq \mathbf{pagoda}[F]$, where F is the final state

4 Symmetry in Peg Solitaire

Peg Solitaire is highly symmetric, which is a significant factor in the difficulty of the problem. This section discusses these symmetries and methods used to break them.

4.1 Board Symmetries

The English Solitaire board has rotational and reflectional symmetry. When composed, these two types of symmetry produce eight symmetries including the identity. That is, a rotation about 0, 90, 180 or 270 degrees, with or without a reflection across the horizontal or vertical axes. If one board state can be transformed into another via the application of one of the eight symmetries then the two states are symmetrical. There are often multiple ways of arriving at symmetrical board states, as presented in Figure 4 (an identical state) and Figure 5 (symmetrical states).

Given a pair of transition sequences like those shown in the figure, the search space beyond the point where they reach the same position is symmetrical (identical in this case, since the same state is reached by the two sequences). This can lead to a large amount of wasted effort, especially as the number of such symmetrical sequences increases.

Breaking this symmetry involves forming equivalence classes from the symmetric sequences and adding constraints to permit just one representative from each class. Given multiple sequences leading to an identical board state b after move i, and a representative s, the symmetry breaking constraint may be informally stated:

$$\mathbf{bState}[i + 1] = b \rightarrow \mathbf{moves}[1 \ldots i] = s$$

Given a representative s_1 leading to board state b_1 after move i, and sequences $s_2, \ldots, s_n$ leading to board states $b_2, \ldots, b_n$ that are symmetrical (but not identical) to b, the

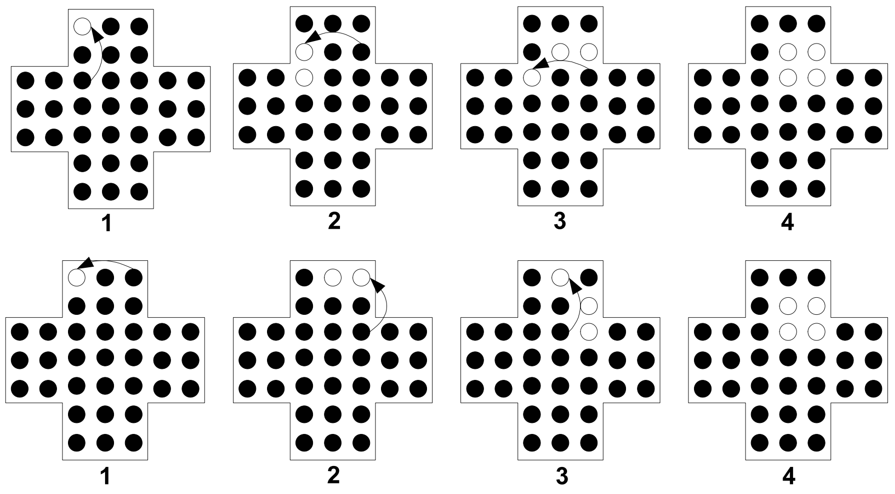

Figure 4: Identical positions reached by symmetric sequences.

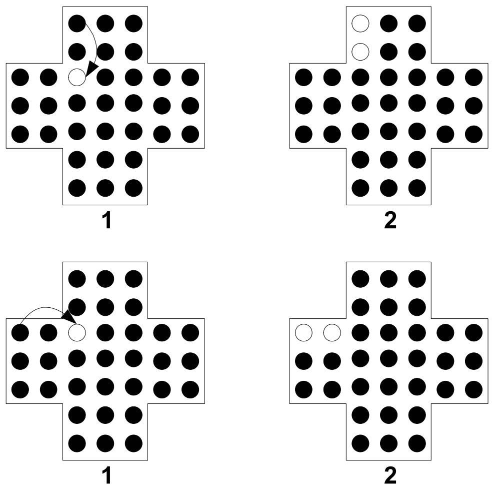

Figure 5: Symmetrical positions (rotation+reflection) reached by symmetric sequences.

symmetry breaking constraints are slightly simpler:

$$\forall j \in \{2 \ldots n\} : \mathbf{bState}[i+1] \neq b_j$$

Adding all such constraints is prohibitively expensive: it involves exploring the entire search tree to form the equivalence classes. Symmetry breaking to a limited depth is explored in Section 4.4. Two simpler, and less expensive, methods of breaking a substantial amount of the symmetry among sequences are discussed in the following sub-sections.

4.2 Symmetries of Independent Moves

Many pairs of moves can be applied in any order without affecting the rest of the solution. A move inverts the states of the three pegs upon which it operates, so two moves are *independent* iff the sets of pegs they involve are disjoint. Naive backtracking thrashes on all orderings of each pair of independent moves. To break this symmetry the transitions are ordered (via the ordering in Table 1) and constraints imposed such that adjacent, independent pairs of moves must be ordered smallest to largest under this ordering.

Theorem 1 *Given a Boolean function independent*(t, t')*, which returns "true" if t and t' are independent transitions, then if a solution to Solitaire is represented as a matrix x of ordered elements, then placing the additional constraint independent*$(x[i], x[i+1]) \rightarrow x[i] \leq x[i+1]$ *does not modify the set of unique solutions.*

Proof: Adding constraints cannot increase the set of unique solutions. It remains to show that a solution represented by a matrix x can always be transformed into a solution represented by a matrix x' which satisfies the ordering constraints. From the definition of independence, exchanging two adjacent moves $x[i]$ and $x[i+1]$ which do not satisfy the ordering constraints is solution-preserving if *independent*$(x[i], x[i+1])$. Having made this transformation, and since $x[i+1] < x[i]$, the matrix x' is now lexicographically less than x. Since both x and x' are finite, a finite number of transformations of this type are possible before reaching a matrix which satisfies the ordering constraints. QED.

Symmetry-breaking constraints of this type, which are like the *action-choice* constraints introduced by van Beek and Chen [26], can be added straightforwardly to model C via the **moves**[] matrix. Unfortunately there are other ways in which a sequence of transitions can be interchanged. For example, if transitions 2 and 11 are independent of transition 10 then the above set of ordering constraints would accept both the sequence $\langle 10, 11, 2\rangle$ and $\langle 11, 2, 10\rangle$ although they are equivalent. Attempting to break these larger symmetries quickly becomes very complex, both in terms of the number of constraints involved and ensuring that the sets of constraints do not conflict.

Breaking pairwise independent move symmetry is a compact method of breaking symmetry among sequences. This method does not break all symmetry, however. Reconsider Figure 4: the two sequences are composed of entirely disjoint sets of moves.

4.3 Self-symmetric Board States

Rotating or reflecting certain board states produces no change in the state of the board. The initial state of central Solitaire is an example where the board can be rotated or reflected without being changed (Figure 1). When self-symmetry arises, equivalence classes can be formed from the moves possible and constraints added to disallow all but a single representative from each class. In central Solitaire, the rotational symmetry of the initial state can be broken by insisting that the first move is transition 7: 3,1→3,3. Notice that reflection symmetry persists after the first move, which can be broken by pruning transition 26: 5,2→3,2 from the domain of **moves**[2] in models B and C.

Self-symmetry is broken when symmetric sequences are removed, as described in Section 4.1. Returning to central Solitaire, the states reached following initial moves consisting of transition 7 and transition 26 are symmetric. However, recognising self-symmetry can lead to more compact symmetry-breaking constraints. Given a self-symmetric state reached by sequence s after move i, and an equivalence class of transitions $\tau_1, \ldots, \tau_n$ with τ_1 chosen as the representative, the symmetry breaking constraint can be expressed:

$$\mathbf{moves}[1 \ldots i] = s \rightarrow \forall j \in \{2 \ldots n\} : \mathbf{moves}[i+1] \; \neq \tau_j$$

This constraint replaces $n-1$ sequence symmetry-breaking constraints as described in Section 4.1: each unary constraint added to **moves**$[i+1]$ prevents the search from exploring a board state symmetrical to that obtained by appending τ_1 to sequence s. Given multiple equivalence classes of transitions, the consequent of the implication constraint is extended to disallow all but the representative of each class.

Breaking all such symmetry statically is expensive, requiring the exploration of the whole search tree. The next sub-section follows this approach to a limited depth.

4.4 Generating Symmetry Breaking Constraints

Table 2 presents the results of generating sequence and self-symmetry breaking constraints for a subset of the Solitaire reversals. This subset was chosen because it contains each unique board position, up to symmetry. The symmetry-breaking constraints are generated incrementally: at each depth constraints generated at all previous depths are used to reduce the search space and therefore the number of other symmetry-breaking constraints that must be generated. Depth 0 (initial state) to depth 2 are omitted for brevity. In addition, the pairwise independent move symmetry-breaking constraints described in Section 4.2 are used throughout to reduce the search space further.

The size of the search space varies widely among the different reversals, depending on the branching factor. In some cases, such as the central reversal (3, 3), it is possible to use symmetry-breaking to reduce the branching factor effectively, whereas in other cases (e.g. (2, 1) or (2, 2)) symmetry breaking does not have as great an effect. Even with the incremental approach, a substantial set of new constraints is necessary to break the sequence symmetry by depth 7. Discovering and adding constraints deeper in the search tree is unfortunately not currently feasible. However, assuming a variable ordering that tries the moves in order, the symmetry-breaking constraints generated here involve the top of the search tree and so prune a significant proportion of the search space. These constraints will be exploited in the following sections.

4.5 Symmetry-breaking and the Goal State

Care must be taken with symmetry-breaking if the goal state is fixed: symmetry-breaking prunes members of the equivalence class of the goal state, and we must ensure that the fixed goal is not one of the elements pruned. If the final state has self-symmetry, say rotational symmetry, then it is consistent to apply rotation symmetry-breaking throughout. Breaking independent move symmetry has the advantage that it can be applied without considering the goal state, since it applies to move sequences to identical states.

	Depth	Choices	Time(s)
(2,0)	3	30	0.82
	4	157	2.12
	5	830	9.47
	6	4,118	46.5
	7	18,675	232.0
(2,1)	3	47	1.02
	4	251	3.19
	5	1,204	13.53
	6	5,675	63.7
	7	24,701	313.0
(2,2)	3	47	0.99
	4	256	3.19
	5	1,250	14.0
	6	5,774	64.7
	7	24,900	314.0

	Depth	Choices	Time(s)
(3,0)	3	7	0.65
	4	41	0.99
	5	204	2.79
	6	922	10.8
	7	3,926	47.0
(3,1)	3	14	0.72
	4	75	1.42
	5	379	4.82
	6	1,820	21.5
	7	7,907	97.8
(3,2)	3	40	0.98
	4	208	2.77
	5	1,006	11.6
	6	4,668	53.9
	7	19,961	262.0
(3,3)	3	7	0.62
	4	41	0.98
	5	204	2.74
	6	922	10.7
	7	3,926	46.9

Table 2: Incrementally generating symmetry-breaking constraints for Solitaire reversals. Times are to 3 significant figures. Hardware: 750Mhz Pentium III, 256Mb RAM.

5 Single-peg Solitaire Reversals

The first set of experiments concerned single-peg Solitaire reversals. This variant of the puzzle is solved as soon as the first solution is found. Hence, the most successful solution techniques will quickly select the 'right' area of the search space to focus on.

5.1 Variants of Model A

Model A can be used to solve Solitaire reversals by expressing the objective in terms of the required board position. As well as model A in its basic form, we experimented with introducing Pagoda functions in an attempt to improve efficiency. For each reversal tested, an appropriate Pagoda function from the three presented in Figure 3 was chosen.

5.2 Variants of Model C

Similarly, model C is modified to solve each reversal by changing the constraints on **bState**[] that specify the initial and goal positions. To gauge their relative efficiency, Pagoda functions and symmetry-breaking were added both individually and together.

5.2.1 Heuristics

The variable/value ordering heuristic is crucial to the efficient solution of a constraint model. Preliminary experimentation revealed that branching on the **moves**[] matrix, instantiating the variables in ascending order, performed well. Hence, this variable ordering was adopted throughout. Our default value ordering was simply a static, ascending ordering following that given in Table 1. To evaluate model C thoroughly, we also experimented with several more complex dynamic value orderings, as follows. In all cases, if no 'preferred' move can be made the heuristic reverts to the default value ordering.

Corner Bias. Prefers to move corner pegs towards the middle (e.g. $2, 0 \rightarrow 2, 2$).

Minimise Isolated Pegs. Prefers to reduce the number of pegs with no other pegs in neighbouring holes.

Maximise/Minimise Possible Moves. Prefers to maximise/minimise the number of possibilities for the next move.

Generate Packages. Prefers to generate one of the predefined *packages*.

Follow Packages. Prefers to follow move sequences to resolve *packages*.

Maximise/Minimise Pagoda. Prefers to maximise/minimise the Pagoda value.

where a package [5] is a frequently occurring pattern of pegs for which there is a known series of moves to reduce the package to a single peg.

5.3 Comparison with AI Planning Systems

Models A and C are compared against four leading domain-independent AI planning systems. Blackbox 4.2[2] [17] is a Graphplan-based [6] planner that transforms the planning graph into a large propositional satisfiability (SAT) problem. The solution to this problem, which is equivalent to a valid plan, is obtained by using a dedicated SAT solver. We used the Chaff solver [23]. Fast-forward 2.3[3] [19] is a forward-chaining heuristic state space planner that generates heuristics by relaxing the planning problem and solves using a Graphplan-style algorithm. HSP 2.0[4] [8] maps planning instances into state-space search problems that are solved using a variant of the A* search algorithm, with heuristics extracted from the representation. STAN 4[5] [20] is a Graphplan-based planner with a powerful representational structure for the planning graph and a mechanism to reduce search and graph construction costs once the graph is stable.

5.3.1 Encoding

A simple STRIPS [12] encoding is sufficient to describe Peg Solitaire. The plan objects are the 33 board positions, which may be full or empty. Both empty and full predicates are required since some STRIPS planners do not support negated preconditions or goals. Transitions are stated in terms of the three board positions which they involve, e.g. (*transition loc1 loc2 loc3*). Move operators can use these transitions in either direction, depending on the state of the relevant board positions, as presented in Figure 6.

[2]http://www.cs.washington.edu/homes/kautz/blackbox/
[3]http://www.informatik.uni-freiburg.de/~hoffmann/ff.html
[4]http://www.cs.ucla.edu/~bonet/
[5]http://www.dur.ac.uk/computer.science/research/stanstuff/html/dpgstan.html

```
(:action MOVE-FORWARDS
    :parameters
      (?loc1 ?loc2 ?loc3)
    :precondition
      (and (transition ?loc1 ?loc2 ?loc3)
           (full ?loc1)
           (full ?loc2)
           (empty ?loc3))
    :effect
      (and (empty ?loc1) (not (full ?loc1))
           (empty ?loc2) (not (full ?loc2))
           (full ?loc3) (not (empty ?loc3))))
```

```
(:action MOVE-BACKWARDS
    :parameters
      (?loc1 ?loc2 ?loc3)
    :precondition
      (and (transition ?loc1 ?loc2 ?loc3)
           (empty ?loc1)
           (full ?loc2)
           (full ?loc3))
    :effect
      (and (full ?loc1) (not (empty ?loc1))
           (empty ?loc2) (not (full ?loc2))
           (empty ?loc3) (not (full ?loc3))))
```

Figure 6: STRIPS operators for Solitaire.

5.4 Results

Throughout, all experiments were performed on a Pentium III 750Mhz with 256 Mb RAM. Models A and C were solved with Ilog Solver 5.3 and CPLEX 8.0 respectively. The experiments concerned the full set of single-peg Solitaire reversals. From [5] it is known that all possible single-peg reversals are solvable. We set each of our systems the task of solving all such reversals. This may seem unnecessary — after all if a reversal for position (3,0) can be solved, the reversals for (6,3), (3, 6) and (0,3) can be obtained easily by symmetry. However, this symmetry is broken by our default value ordering. Nor does the symmetry hold for the planners, which are also given the transitions in a particular order. Hence, we experimented with the full set for a fair comparison.

Table 3 summarises the results for models A (rows 5 and 6) and C (rows 1 – 4) with the default value ordering. The performance of the planners is given in Table 4. We begin by considering model A. On this set of problems model A performs relatively better than model C with the default value ordering, solving 14 and 17 instances without and with Pagoda functions, respectively. However, the Pagoda functions do not provide a uniform improvement, sometimes leading Cplex away from a solution. By considering the number of instances solved by the AI Planners (BBox=27, FF=22, HSP=15, Stan=1) one may say that integer programming performs as well as an average AI planner here.

In [15] we reported that Model A with an objective function that incurs a penalty if the last peg is not in the centre hole failed to return a solution in 12 hours using Cplex 8.0 with its default settings (the branching variable is automatically selected). In view of this failure, a number of variable selection strategies were tried. We were able to solve model A in 388 seconds (10 nodes visited) using the variable selection strategy based on "pseudo reduced costs". Employing Eq (1) instead and using the default settings, a solution is obtained in 521 seconds (28 nodes visited). Hence, Cplex successfully selected a branching strategy befitting the new objective function. This variable selection strategy is "branch on variables with maximum infeasibility".

As noted, Peg Solitaire does not involve optimisation, and neither function represents a genuine objective in that both can be written simply as constraints. In fact, Cplex determines the optimal objective function value, and hence the best bound possible, during preprocessing. Hence, the role of the objective function is not to provide strong bounds, but to select a branching variable. Therefore, the discrepancy in performance of these two objective functions is due to choosing the right variable selection strategy.

We now turn our attention to model C. The pairwise symmetry-breaking constraints

		(2,0)	(4,0)	(2,1)	(3,1)	(4,1)	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(2,3)	(3,3)	(4,3)	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(2,5)	(3,5)	(4,5)	(2,6)	(4,6)
Lex.Val.Ord Basic Model	(1)				460,387		46,423				654			1,599	1,398	264,412			9,051									
Lex.Val.Ord Pairwise Sym-Break	(2)				15,551		4,991				173			348	415	13,785		130,686	1,315	24,243								
Lex.Val.Ord Pagoda	(3)				340,292		46,423				654			1,599	951	264,412			9,001									
Lex.Val.Ord Pagoda+ Sym-Break	(4)				4,510		4,991				173			348	415	13,785		64,905	1,315	24,243								
IP	(5)		250	30					138	154		240	43	321	28	10	201	356			180	187				62		
IP+Pagoda	(6)	213	67	276		220		178				92	284	77		29	86	209		200		90	167		400	12	168	

		–/– –/– –/1,270	–/– –/– –/–	–/– –/– 1,820/600		
		–/– –/– 451/2,210	2,900/222 2,730/55.0 –/–	–/– –/– –/1,860		
439/61.0 443/65.0 –/–	–/– –/– –/1,170	–/– –/– 1,150/–	–/– –/– 1,240/–	7.00/2.70 8.00/2.70 –/–	–/– –/– 1,950/777	–/– –/– 562/2,210
–/– –/– –/–	–/– –/– –/–	17.0/3.50 18.0/4.90 2,590/1,080	16.0/4.10 7.90/5.00 521/–	1,700/197 1,710/200 375/444	–/– –/– –/–	–/– –/– –/–
–/– –/– 1,530/860	–/1,890 –/1,040 2,560/1,230	116/19.0 102/22.0 –/–	–/338 –/350 –/2,000	–/– –/– 1,410/–	–/– –/– 1,690/763	–/– –/– –/1,240
		–/– –/– –/–	–/– –/– –/2,960	–/– –/– 820/264		
		–/– –/– –/1,340	–/– –/– –/–	–/– –/– –/–		(1)/(2) (3)/(4) (5)/(6)

Table 3: Solitaire reversals solved via CP and IP models. Choices (CP) and nodes (IP) are presented in the top table. Times (to three significant figures) are presented in the bottom table. A dash indicates no solution within 1 hour.

		13.0/49.0 */*	*/* 148/*	14.0/622 */*		
		25.0/121 */–	543/* */*	17.0/1.00 */–		
19.0/0.20 30.0/*	25.0/0.70 */1,130	47.0/1.0 */*	48.0/3,540 86.0/*	38.0/0.60 27.0/*	14.0/273 48.0/–	21.0/0.60 32.0/*
/ */*	862/– */–	28.0/0.10 125/*	*/* 57.0/*	30.0/48.0 */*	620/– 574/–	*/– */*
19.0/276 */*	14.0/0.05 97.0/–	42.0/0.15 */*	44.0/0.05 313/*	49.0/0.05 60.0/*	16.0/1,560 */–	19.0/* 125/*
		16.0/553 */–	*/* */–	27.0/1,520 */–		
		18.0/– 298/*	*/* */*	21.0/9.80 154/*		Bbox/FF HSP/Stan

Table 4: Solitaire reversals solved via AI Planners. Times given to three significant figures. An asterisk indicates memory resources exhausted, and a dash indicates time exhausted (1 hour allowed).

(Section 4.2) effectively reduce search effort, allowing some problems to be solved that were not previously solvable within one hour. However, the symmetry-breaking constraints based on self-symmetry and symmetric sequences were not useful (results omitted for brevity). These constraints were generated only to a limited depth (Section 4.4) and, due to the size of the search space, this area of the search tree is not visited often enough for them to make an impact. In subsequent sections, when the whole space must be explored to prove a solution optimal, we expect that these constraints will be much more useful. As for model A, using Pagoda functions to guide the search also significantly improves model C. This is despite the fact that the single-peg goal state has a Pagoda value of one, hence only states with value zero are pruned. Also, Pagoda functions and symmetry-breaking are complementary and can be combined for further improvements.

Overall, however, model C with the default value ordering gave the weakest coverage on this set of problems of all the systems tested, except Stan. We conjectured that this was due to the naive value ordering heuristic, which is crucial when finding a single solution, as noted above. The heuristics given in Section 5.2.1 are designed to guide the search in a more informed way. The results of combining these heuristics with model C are presented in Table 5. Note that symmetry-breaking was not used in these experiments, since it tended to conflict with the heuristics. Furthermore, preliminary experimentation revealed that the value heuristics either guide the search towards the solution relatively quickly or towards a fruitless area of the search tree such that a solution cannot be found in a reasonable time. Hence, these experiments were run with a 15-minute time limit.

Two of the heuristics, *Minimise Pagoda* and *Maximise Possible Moves*, were unable to solve any of the reversals in 15 minutes, and were both unable to solve the central reversal given 10 hours. Investigating the search trees generated by these two heuristics suggested that the problem is that early in the search they lead to board states with many distinct holes and make no attempt to remove pegs from the outmost corners (which are typically the most difficult pegs to remove). The most successful heuristics were *Corner Bias* and *Minimise Isolated*. However, the default value ordering performs relatively well

		(2,0)	(4,1)	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(6,2)	(2,3)	(3,3)	(2,4)	(3,4)	(4,4)	(5,4)	(2,5)	(3,6)	(4,6)
Corner Bias	(1)							30		64	77	135		28		45		446
Follow Packages	(2)									1,599			2,526				6,364	
Find Packages	(3)			46,399	25,744			46,324							119			
Minimise Isolated	(4)		351				73,898		60,341			14,524		49				7,505
Minimise Moves	(5)				63,780	175						13,708						
Maximise Pagoda	(6)	71,495					6,816		5,303									

		–/– –/– –/887	–/– –/– –/–	–/– –/– –/–		
		–/– –/– –/–	–/– –/– –/–	–/– –/3.52 –/–		
–/– 522/– –/–	–/– 348/– 811/–	–/– –/– 2.52/–	–/– –/850 –/832	0.72/– 640/– –/–	–/– –/– –/–	–/– –/802 –/54.5
–/– –/– –/–	–/– –/– –/–	1.21/16.7 –/– –/–	1.30/– –/– –/–	–/– –/– –/–	–/– –/– –/–	–/– –/– –/–
–/– –/– –/–	–/– –/– –/–	1.90/– –/104 181/–	–/30.1 –/– –/–	0.70/– –/1.62 –/–	–/– 2.43/– –/–	–/– –/– –/–
		1.20/– –/– –/–	–/– –/– –/–	–/– –/– –/–		
		–/– –/– –/–	–/60.1 –/– –/–	5.54/– –/104 –/–		(1)/(2) (3)/(4) (5)/(6)

Table 5: Solitaire reversals solved via model C and a number of value heuristsics. Choices are presented in the top table. Times (to three significant figures) are presented in the bottom table. A dash indicates no solution within 15 minutes.

when compared with the other heuristics. This appears to be because it tends to remove all pegs from one area of the board before moving on to another, which reduces the chance of pegs becoming isolated. The *Minimise Moves* and *Maximise Pagoda* heuristics are at first glance counter-intuitive as they are the opposite of what may be expected. Their advantage however comes from the fact that like *Minimise Isolated* they encourage clearing one area of the board before performing moves in another.

The *Follow Packages* and *Find Packages* heuristics performed somewhat disappointingly. This may be because they were implemented "statelessly", where at each search node the current packages and partly-removed packages were found and a move chosen accordingly. As a board typically contains many overlapping packages, the heuristic did not remove complete packages as intended. Developing a heuristic able to resolve individual packages is a more complex task, and forms an important piece of future work.

Finally, we consider the performance of the four AI planners. The most successful planning systems, Blackbox and FF, achieved a high percentage of coverage in terms of number of problems solved. The less successful systems appeared to suffer as much from lack of resources as lack of time. The most difficult reversals for the planners to solve were the central reversal (which is one of the reversals solved most easily using model C) and reversals at positions 2 and 3 spaces distant from the centre in a straight line. In particular, the reversal 3 spaces distant from the centre, namely at (3, 0) and symmetric equivalents, is well known as the 'notorious' reversal because of its difficulty [5].

6 Solitaire Patterns

The second set of experiments concerned Solitaire patterns: goal configurations with a number of pegs in some specific arrangement. The instances tested are a representative selection from those given in [3] (Figure 7). Throughout, the initial state is the same as that of central Solitaire. Again, this variant is solved as soon as the first solution is found, so success depends on examining the right areas of the search space early.

Prior to running the experiments it was expected that the Pagoda functions would prove more useful, since goal states containing a number of pegs tend to have higher Pagoda values than single-peg reversals. Hence, it is more likely that the Pagoda value of an intermediate state will be less than that of the goal, increasing pruning.

6.1 Results

The results are presented in Table 6. On this occasion, integer programming is the most robust approach, solving all of the instances studied. The objective function used is a straightforward modification of that used for the Solitaire reversals: in the final state, the sum of the positions where there must be a peg is maximised. This is more constraining than the single-peg final states in the reversals problems and is exploited well by integer programming. The relatively good performance of integer programming on the Solitaire patterns can be attributed to a better search guidance, which is a direct consequence of the use of linear relaxations. In the absence of such guidance, the CP and AI Planning approaches sometimes proceed down dead-ends from which they cannot recover.

As expected, the Pagoda functions can provide a substantial improvement for Model A on these instances. Consider, for example, the results for pattern 4. However, they can

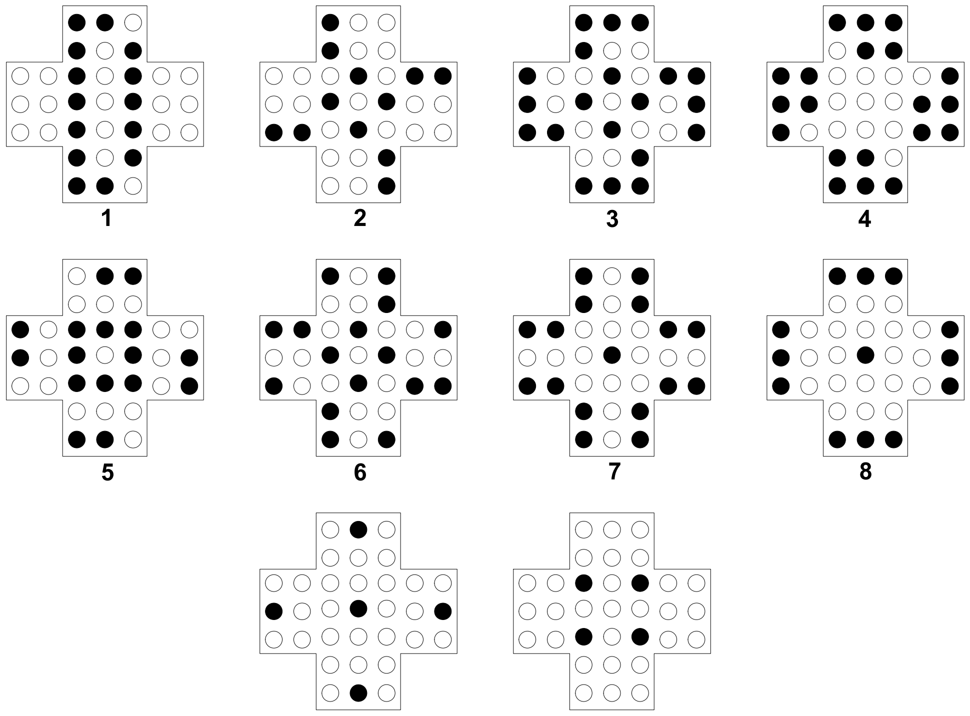

Figure 7: Selected Solitaire Patterns Problems. Goal states shown.

	CP		CP+P		CP+sy m		CP+sym+P	
No.	Time	Choices	Time	Choices	Time	Choices	Time	Choices
1	-		-		-		-	
2	-		-		-		-	
3	194	26,888	201	26,876	181	22,766	181	22,754
4	-		3,140	442,348	410	57,202	419	57,202
5	-		-		-		-	
6	-		-		-		-	
7	301	38,256	294	37,475	294	35,749	269	34,990
8	-		-		-		-	
9	3,090	314,222	1,010	127,305	304	34,937	196	25,383
10	1	1	1	1	601	38,143	610	38,135

	IP		IP+P		BBox	FF	HSP	Stan
No.	Time	Nodes	Time	Nodes	Time			
1	102	40	89.0	14	16.0	-	*	*
2	338	109	-		*	-	*	*
3	12.0	7	25.0	29	0.2	0.10	25.0	0.30
4	2,280	5,472	146	390	24.0	-	*	*
5	114	81	71.0	36	0.5	-	*	0.60
6	3,550	3,121	1,000	1,049	14.0	0.10	*	*
7	314	249	-		17.8	-	*	*
8	76.0	31	2,710	1,200	25.0	-	*	*
9	2,940	521	-		*	130	253	*
10	943	157	1,270	177	45.0	94	*	*

Table 6: Results: Solitaire Patterns. Times in seconds to 3 significant figures. An asterisk indicates memory resources exhausted, and dash indicates time exhausted (1 hour allowed).

also have detrimental effects. On these occasions, the additional constraints are unable to reduce the search and so become a burden to the solver.

Model C was used with the default value ordering, since it proved reasonably successful on the single-peg reversals, and because the more informed heuristics were tailored towards a goal with a single peg. The performance of model C is comparable to that of the planners FF, HSP and Stan in terms of coverage. In contrast to these systems time rather than memory resources are exhausted when performance is poor. Pagoda functions generally help but sometimes this is outweighed by their overheard, as per the IP models. Unlike the IP models, the search space is monotonically reduced by adding the Pagoda functions (given a static heuristic, adding constraints always has this effect) but the overhead of maintaining these extra constraints increases the time taken overall.

Symmetry breaking is almost uniformly successful at reducing search. On the one occasion where the performance is affected badly this is due to a conflict with the default heuristic. As observed in the reversals results, Pagoda functions and symmetry breaking are complementary, leading to an improved performance overall.

With the exception of Blackbox, the planners performed relatively poorly. This is partially due to the significantly more complicated goal state, which for some systems led to memory exhaustion — despite the fact that the search space is smaller than that of the Solitaire reversals. Blackbox itself performs strongly, as it did on the Solitaire reversals. The strategy of converting Solitaire to a propositional satisfiability problem therefore appears to be a good one, possibly because of the simplicity of the game.

7 Fool's Solitaire

An optimisation variation of Solitaire (named Fool's Solitaire by Berlekamp, Conway and Guy [5]) is to reach a position where no further moves are possible in the shortest sequence of moves. This section describes methods to solve this problem. This is an optimisation problem, so success now depends on strong pruning of the search space — it is no longer sufficient simply to find a solution, it must be proven optimal. When searching the entire space, value ordering is less important (although finding a good solution early aids pruning). This is borne out experimentally, hence we focus on an instance of Fool's Solitaire for each of the unique (up to symmetry) board positions.

There does not appear to be a straightforward way to state this optimisation problem as a STRIPS model without negated preconditions. Hence, we focus on developing IP and CP models. These models will be used to solve the problem incrementally. A 'dead-end' position is sought after one move. If one cannot be found, the search is extended to two moves and so on. Trivially, this guarantees optimality. Hence, a series of small problems are solved instead of one large problem. When a dead end can be found in a relatively short number of moves, this approach reduces the overall workload considerably.

7.1 Modelling via Integer Programming

An extension of IP Model A is used in an incremental manner to determine the minimum number of moves required to reach a dead-end. To serve this purpose, a binary decision variable, $C[i, j, t]$, $t = 1, ..., I$, is introduced which equals 1 iff there is a peg in hole (i, j)

with a legal move. I denotes the current depth to which the search extends.

$$C[i,j,I] \geq \mathbf{bState}[i,j,I] + \mathbf{bState}[i+1,j,I] - \mathbf{bState}[i+2,j,I] - 1 \tag{16}$$
$$C[i,j,I] \geq \mathbf{bState}[i,j,I] + \mathbf{bState}[i-1,j,I] - \mathbf{bState}[i-2,j,I] - 1 \tag{17}$$
$$C[i,j,I] \geq \mathbf{bState}[i,j,I] + \mathbf{bState}[i,j+1,I] - \mathbf{bState}[i,j+2,I] - 1 \tag{18}$$
$$C[i,j,I] \geq \mathbf{bState}[i,j,I] + \mathbf{bState}[i,j-1,I] - \mathbf{bState}[i,j-2,I] - 1 \tag{19}$$

The new constraints, Eqs. 16–19, in conjunction with the original constraints Eqs. 2–15, set $C[i,j,I] = 1$ if there is a legal move in time-step I.

The objective is to minimise the total penalty incurred by having legal moves. Therefore, the following simple objective function will suffice:

$$\min \sum_{(i,j) \in B} C[i,j,I] \tag{20}$$

It is clear that the objective function with value zero indicates a dead-end.

7.2 Modelling via Constraint Programming

Model C is used with a few small modifications. A 77th transition is added, denoted 'DeadEnd', representing the fact that the search has reached such a dead-end. The DeadEnd transition must only be allowed if no other move is possible. A simple way to achieve this is to use an implication constraint based on **bState**[]. If any of the preconditions hold for any other transition, the DeadEnd transition is not allowed:

$$\begin{aligned} & \forall t \in \{1, \ldots, 31\} : \\ (\mathbf{bState}[2,0,t] = 1 \wedge \mathbf{bState}[3,0,t] = 1 \wedge \mathbf{bState}[4,0,t] = 0) & \quad \vee \ldots \\ \vee(\mathbf{bState}[4,6,t] = 1 \wedge \mathbf{bState}[4,5,t] = 1 \wedge \mathbf{bState}[4,4,t] = 0) & \\ \rightarrow \mathbf{moves}[t] \neq \text{DeadEnd} & \end{aligned}$$

If the incremental search is seeking a dead end at time-step t, **moves**[t] is set to DeadEnd. Constraint propagation then forces a sequence of moves to be chosen that lead to a dead end, if such a sequence exists.

7.3 Results

The results are presented in Table 7 and, as an example, the optimal route to a dead end in central Solitaire is presented in Figure 8. This optimisation problem is significantly more difficult than the single-solution problems studied in the preceding sections, hence a longer period (15 hours) is allowed to solve each instance.

The CP model performs strongly on this problem and is able to solve all instances when symmetry-breaking is employed. The large beneficial effect of symmetry-breaking is as expected, since it reduces the search space (which must be explored fully to ensure optimality) considerably. The symmetry-breaking version of the CP model makes use of the constraints generated in Section 4.4. Note that, since a specific dead-end state is not sought, it is safe to use the full set of symmetry-breaking constraints generated.

	# of Moves	Choices	Time
(2,0)	9	1,464,461 41,917 NA	7,820.0 362.0 242.0
(2,1)	13	– 7,393,851 –	– 61,300.0 –
(2,2)	13	– 6,910,648 –	– 57,600.0 –

	# of Moves	Choices	Time
(3,0)	6	391 96 NA	2.90 1.70 6.90
(3,1)	10	1,285,889 20,698 NA	7,130.0 175.0 3,900.0
(3,2)	9	399,145 11,556 NA	2,230.0 135.0 1,680.0
(3,3)	6	552 96 NA	4.10 1.70 26.8

Table 7: Fool's Solitaire (without sym-breaking, with sym-breaking, IP). Cumulative times given to three significant figures. A dash indicates no result in 15 hours.

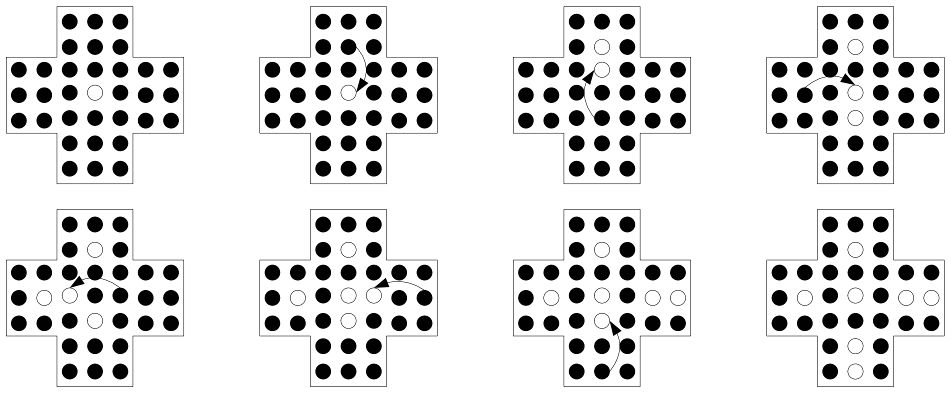

Figure 8: An optimal route to a dead-end in central Solitaire.

8 Long-hop Solitaire

This section describes an attempt to solve an optimisation variant of Peg Solitaire, known as Long-hop Solitaire ([3], chapter 8). Pegs are removed in the usual way, but like draughts/checkers a move starting from the end position of the previous move is counted as a single move (hereafter referred to as a *hop*). A hop may contain several moves. The objective is now to solve the puzzle using as few hops as possible.

8.1 Modelling via Constraint Programming

There does not appear to be a straightforward encoding of this problem for either the AI Planners or integer programming. Hence, we focus on a CP model. We maintain the **moves**[] and **bState**[] matrices used to model the standard version of the puzzle. In addition, we use another 0/1 matrix, **hops**[] to count the number of hops used in a solution. Informally, the following constraints link **moves**[] and **hops**[]:

$$\forall i \in \{1, \ldots, 30\} : \texttt{end}(\mathbf{moves}[i]) = \texttt{start}(\mathbf{moves}[i+1]) \leftrightarrow \mathbf{hops}[i+1] = 1$$

Clearly, **hops**[1] is always 1. These constraints can easily be specified in extensional or logical form. The extensional form is used since it provides stronger pruning due to the way in which Solver treats logical formulae, and because it does not appear to require too much overhead. The objective is now to minimise the sum of **hops**[].

8.2 Symmetry Breaking

Care must be taken when breaking symmetry in Long-hop Solitaire. Consider the symmetry of independent moves [15]. As an example, transitions 15 ($2, 2 \rightarrow 4, 2$) and 35 ($2, 3 \rightarrow 4, 3$) are independent. In the standard version of the puzzle, it is necessary simply to disallow ($\mathbf{moves}[i] = 35 \wedge \mathbf{moves}[i+1] = 15$) for i in $\{1 \ldots 30\}$ to break this symmetry. However, adding these constraints in Long-hop Solitaire may prune solutions. If, for example, **moves**[] contains the sequence ⟨30, 35, 15⟩, where transition 30 is $0, 3 \rightarrow 2, 3$, exchanging the 'independent' moves to give the sequence ⟨30, 15, 35⟩ does *not* produce an equivalent solution. This is because ⟨30, 35⟩ is a single hop, whereas ⟨30, 15⟩ is not. A similar problem arises with the move immediately following a pair of independent moves.

In light of this observation, breaking independent move symmetry is more complicated. Symmetry breaking constraints must now take into account the context of a pair of independent moves, in terms of the immediately preceding and following moves:

$$\begin{aligned} &\forall i \in \{1 \ldots 28\}, \{\tau, \tau'\} \subseteq T : \\ &\texttt{indep}(\tau, \tau') \wedge \mathbf{moves}[i+1] = \tau \wedge \mathbf{moves}[i+2] = \tau' \wedge \\ &\neg\texttt{hop}(\mathbf{moves}[i], \mathbf{moves}[i+1]) \wedge \neg\texttt{hop}(\mathbf{moves}[i+2], \mathbf{moves}[i+3]) \quad \rightarrow \\ &\mathbf{moves}[i+1] \leq \mathbf{moves}[i+2] \end{aligned}$$

where $\{\tau, \tau'\}$ is a two-element subset of the set of transitions, T, $\texttt{indep}(\tau, \tau')$ returns true if τ, τ' are independent, and $\texttt{hop}(a, b)$ returns true if the sequence $\langle a, b \rangle$ is a hop. These constraints produce an infeasibly large number of tuples in extensional form, therefore they are posted as shown here.

Fixed	Optimal	No Symmetry-breaking		Symmetry-breaking	
		Time(s)	Choices	Time(s)	Choices
21	27	10.5	818	6.60	336
20	27	29.7	2,423	13.6	882
19	27	83.3	6,627	30.9	2,040
18	27	736	59,319	162	11,248
17	26	3,920	290,513	412	28,819
16	26			717	49,014
15	25			2,200	147,777
14	25			4,260	270,570
13	25			67,500	4,139,194
12	24			428,000	24,813,519

Table 8: Experimental Results: Long-hop Solitaire. Times given to three significant figures. Experiments with less than 17 moves fixed from the standard reversal solution only performed with symmetry-breaking.

8.3 Experiments

We experimented with the central Solitaire reversal. As reported in [3], the optimal number of hops in this case is 18. This optimisation problem is significantly more difficult than determining a single solution. Our current models are unable to solve the complete problem. Instead we provide the 'head' of the solution found for the standard reversal, and attempt to optimise the remainder. The results are presented in Table 8, where 'Fixed' denotes the number of elements of **moves**[] fixed (starting from **moves**[1]) using assignments from standard central Solitaire.

The standard solution, when viewed as a sequence of hops, takes 27 hops. Until only 16 moves are used from the standard solution, it is not possible to optimise the tail of the solution to produce fewer hops than this. The independent moves symmetry-breaking constraints prove to be very effective in pruning the search tree, despite the extra complication of checking 4 moves at once rather than just pairs. It should be possible to add symmetry-breaking constraints based on board and sequence symmetries in a similar manner to reduce search further.

9 Conclusions and Future Work

This paper has concerned the modelling and solving of certain variations of English Peg Solitaire. It was demonstrated how CP, IP and AI Planning systems can be used to solve basic variations of the puzzle, where the aim is to find a single solution. On these problems, success depended on focusing on the 'right' area of the search space early. The best of the AI planning systems were able to do this most successfully, giving the most 'robust' performance across the set of experiments. The guidance the IP approach received from relaxations of the problem was also sufficient to allow it to solve a significant proportion of the problems in a reasonable time. Similarly, by experimenting with a number of heuristics, a reasonable coverage of this set of problems was obtained via the CP approach.

The flexibility of the CP and IP approaches was demonstrated when experimenting with two optimisation variants of Solitaire, Fool's Solitaire and Long-hop Solitaire, for which we could find no formulation suitable for the planners. Indeed, for the latter variation of the game we could only formulate the problem as a constraint program. These optimisation problems require the whole search space to be explored in order to prove a solution optimal. Both CP (especially when enhanced with symmetry-breaking) and IP proved to be well suited to this task.

The lessons learned from modelling Solitaire should generalise to other sequential 'planning-style' problems. Using channelling constraints to specify action pre- and post-conditions and the techniques used to break symmetries of independent moves and to remove symmetrical sequences should prove particularly useful. In future, we will explore the application of dynamic symmetry-breaking methods, such as SBDS [13] or SBDD [11] to Peg Solitaire. These systems could break some symmetry by applying the board and independent move symmetries to transform the board states explored by the current move sequence, and thereby generate constraints/perform a dominance check to prevent the exploration of a symmetric sequence. One possible obstacle to this approach is that generating a move sequence from a set of transformed board states may prove expensive.

This approach would not break all symmetry. For example, the first sequence of Figure 4 cannot be transformed into the second via the application of board or move symmetries. In order to break this type of sequence symmetry dynamically, it would be necessary to maintain a record of each unique state visited for comparison against the current state. This may also prove to be prohibitively expensive.

Acknowledgements. Ian Miguel is supported by a Royal Academy of Engineering/EPSRC Post-doctoral Research Fellowship. We thank Chris Beck, Alan Frisch, Steve Linton and members of the APES research group, especially Ian Gent, Barbara Smith and Toby Walsh. We are also very grateful to our anonymous referees for their insightful comments.

References

[1] J. Allen, J. Hendler, A. Tate. Readings in Planning. Morgan Kaufmann, 1990.

[2] D. Avis and A Deza. On the boolean solitaire cone. Technical Report, McGill University and Tokyo Institute of Technology, 1999.

[3] J. D. Beasley. The ins and outs of peg solitaire. Oxford University Press, Oxford, 1992.

[4] M. Beeler, R. W. Gosper and R. Schroeppel. HAKMEM. Technical Report, MIT, Artificial Intelligence Laboratory, Memo AIM-239, 1972.

[5] E.R. Berlekamp, J.H. Conway, R.K. Guy. Winning ways for your mathematical plays, vol.2: games in particular. Academic Press, London, 1982. p. 729-730.

[6] A. Blum and M. Furst. Fast planning through planning graph analysis. Artificial Intelligence 1997; 90:281-300.

[7] A. Bockmayr and Y. Dimopoulos. Mixed integer programming models for planning problems. Proceedings of the CP98 Workshop on Constraint Problem Reformulation, 1998.

[8] B. Bonet and H. Geffner. Planning as heuristic search. Artificial Intelligence 2001; 129:5-33.

[9] N. G. de Bruijn. A solitaire game and its relation to a finite field. Journal of Recreational Mathematics 1972; 5:133-137.

[10] M. B. Do and S. Kambhampati. Planning as constraint satisfaction: solving the planning graph by compiling it into CSP. Artificial Intelligence 2001; 132:151-182.

[11] T. Fahle, S. Schamberger and M. Sellman. Symmetry breaking. Proceedings Principles and Practice of Constraint Programming, LNCS 2239, 2001. p. 93-107.

[12] R. Fikes and N. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. Artificial Intelligence 1971; 5(2):189-208.

[13] I. Gent and B. Smith. Symmetry breaking in constraint programming. Proceedings European Conference on Artificial Intelligence 2000. p. 599-603.

[14] A. Haas. The case for domain-specific frame axioms. Proceedings of the Workshop on the Frame Problem in Artificial Intellgence, 1987.

[15] C. Jefferson, A. Miguel, I. Miguel, A. Tarim. Modelling and solving english peg solitaire. Proceedings 5th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), 2003. p. 261-275.

[16] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. Proceedings 13th National Conference on Artificial Intelligence, 1996. p. 1194-1201.

[17] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. Proceedings 16th International Joint Conference on Artificial Intelligence, 1999. p. 318-325.

[18] H. Kautz and J. P. Walser. State-space planning by integer optimization. Proceedings 16th National Conference on Artificial Intelligence, 1999. p. 526-533.

[19] J. Koffman and B. Nebel. The FF planning system: fast plan generation through heuristic search. Journal of Artificial Intelligence Research 2001; 14:253-302.

[20] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. Journal of Artificial Intelligence Research, 1999; 10:87-115.

[21] I. Miguel. Dynamic flexible constraint satisfaction and its application to AI planning, Springer distinguished dissertation series, 2004.

[22] C. Moore and D. Eppstein. One-dimensional peg solitaire, and duotaire. In, R.J. Nowalski (ed), More Games of No Chance, Cambridge University Press, 2000. p. 341-350.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: engineering an efficient SAT solver. Proceedings 38th Design Automation Conference (DAC'01), 2001.

[24] S. Skiena. Implementing discrete mathematics: combinatorics and graph theory with mathematica, Addison-Wesley, 1990.

[25] R. Uehara and S. Iwata. Generalized hi-Q is np-complete. Trans IEICE, 1990; 73:270-273.

[26] P. van Beek and X. Chen. CPlan: a constraint programming approach to planning. Proceedings 16th National Conference on Artificial Intelligence, 1999 p. 585-590.

[27] T. Vossen, M. Ball, A. Lotem and D. Nau. applying integer programming to AI planning. Knowledge Engineering Review, 2001; 16:85–100.