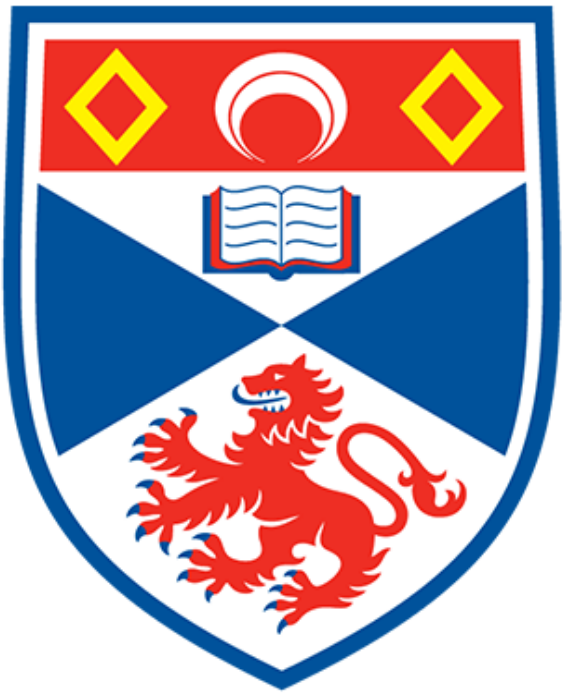
University of St Andrews

School of Computer Science



Solving European Peg Solitaire with Constraint Programming

Niall Colfer

130018883

MSc in Computing and Information Technology

Declaration

I hereby certify that this dissertation, which is approximately 9,863 words in length, has been composed by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree. This project was conducted by me at the University of St Andrews from 06/14 to 08/14 towards fulfilment of the requirements of the University of St Andrews for the degree of MSc under the supervision of Prof Ian Miguel.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Date: 22/08/2014 Signature: 

Acknowledgements

I would like to thank my supervisor Prof. Ian Miguel for the guidance and assistance he provided me throughout this project, and all other members of faculty of the School of Computer Science for their help and support over the course of the year.

I'd also like to thank my family, friends and classmates who were always on hand to provide moral support, both over the course of this project, and for the duration of my stay at St Andrews.

Table of contents

Abstract
1. Introduction
2. Context survey
3. Methodology
    3.1 Constraint satisfaction problems
    3.2 Software and language
    3.3 Peg solitaire
    3.4 Modelling as a problem class
4. Modelling Peg Solitaire
    4.1 State representation
    4.2 Extending the model to multiple boards
    4.3 Enforcing legal moves
    4.4 Complete model
5. Testing the model
    5.1 Arrow configuration
    5.2 Direction of solving
    5.3 Results
6. Refining the Model
    6.1 Symmetry in constraint programming
    6.2 Reflectional and rotational symmetry
    6.3 Independent move symmetry
    6.4 Identifying dead ends
7. Evaluating the model
    7.1 Square configuration
    7.2 Reflectional and rotational symmetry
    7.3 Independent move symmetry
    7.4 Identifying dead ends
    7.5 Solver control
8. Solving European peg solitaire
9. Conclusion and further work
10. References
11. Appendices
    11.1 Ethics pre-assessment form
    11.2 Ethics supervisor checklist
    11.3 Arrow configuration model in full
    11.4 Square configuration model in full
    11.5 European board model in full

Abstract

This project looks at the game of peg solitaire as a constraint satisfaction problem, and attempts to solve the European version of the game. A model is devised to describe the problem class of peg solitaire, with a generic board arrangement. Symmetry, both in reflections and rotations of the board and in choosing independent moves, is broken, and the effectiveness of this is evaluated. Spotting dead ends earlier in the search is also addressed. The most effective model is then applied to the European board, towards finding a solution to the game.

1. Introduction

Peg solitaire is a popular, one-player board game that is played worldwide in many variations, which are stipulated by the arrangement of the pegs to form the board. One square is left empty to facilitate the opening move. Pegs are then removed by draughts style 'jumps', until no legal moves remain. The goal is to reduce the board to a single peg, or some other, predefined goal state.

The two traditional variations of the game are standard, or English, peg solitaire, and European peg solitaire, which are displayed below in Figures 1 and 2 respectively. The goal of this dissertation is to model the European variation using a constraint programming model.

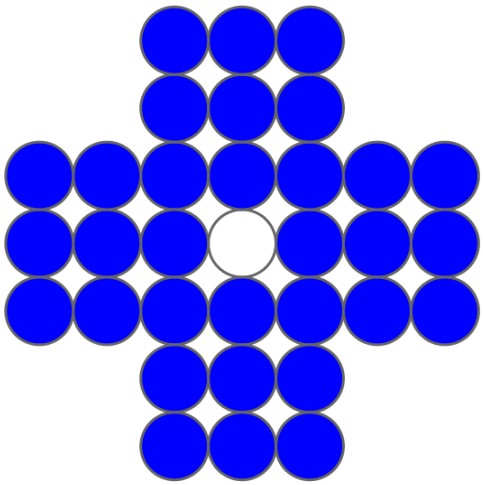


Figure 1

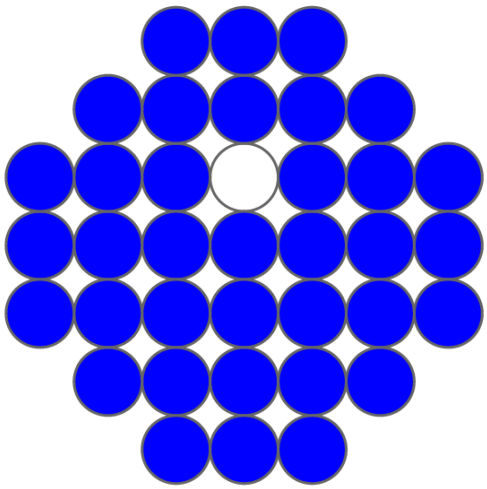


Figure 2

Legal moves within the game follow the same principles as 'jumps' made in draughts, as above, and this is illustrated in Figure 3 below. In the English and European games, and in all games considered for this dissertation, legal moves can only be made in vertical or horizontal movements, i.e. diagonal moves are not permitted. In alternate solitaire games there are triangular or hexagonal board structures with different options for valid moves; these are not considered.

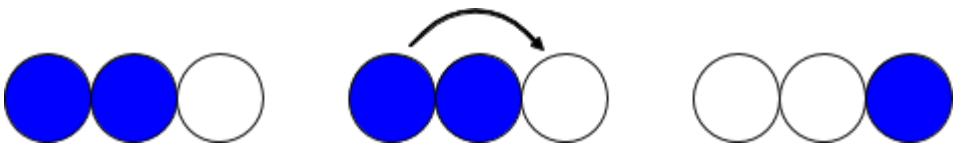


Figure 3

This project begins by reviewing some the literature that has already been published regarding peg solitaire, both in the field of computer science and mathematics. This project shares common themes with 'Modelling and solving English Peg Solitaire' [1], and the similarities and differences are compared and contrasted.

The next section looks at the methodology involved in designing a constraint model for a problem, and illustrates this with an example. The software and programming language that will be used in this project are outlined, and the application of a constraint model to peg solitaire is discussed. In particular, the choice of designing a model that describes the problem class of all peg solitaire games of this type is mentioned.

The following chapter discuss how a model that sufficiently encapsulates the game of peg solitaire was devised, and explains how each section of code contributes to the model. This is continued into a period of testing on a smaller test environment to confirm the model functions as expected, and some preliminary testing as to the major design choices encountered up until that point.

Due to the presence of a high degree of symmetry in peg solitaire, the next part of the project looks at which types of symmetry can be detected and broken. Both reflections and rotations of the board, and the choice that is involved when presented with independent moves, are addressed. The addition of these features is then empirically evaluated, and these results are used to guide the design of the best model. An additional feature of identifying dead ends ahead of time is also implemented and tested.

The culmination of the previous chapters of design and testing are then applied to the European version of peg solitaire, and a solution is sought.

Finally, the conclusion reviews the work that has been performed and discusses extensions that may be implemented to further the model.

2. Context survey

The game of peg solitaire is well studied, and has a wide range of literature. Most analysis of the game has been done in the field of mathematics, as well as being explored in computer science, and more specifically constraint programming.

English peg solitaire has been studied using constraint programming in [1], which looks at integer programming and constraint programming approaches. Both approaches are analysed, and extensions to the game are also considered, such as 'Fool's Solitaire' and 'Long-hop' solitaire. The theme of solving solitaire with a constraint programming model is shared with this paper, but the approach is markedly different: where the game is tackled in [1] by encoding all possible transitions on the English board, and searching for a solution in terms of these transitions, this project takes a representation of the state of the board as transitions are made, and a solution is sought in the form of a sequence of these states. The way in which that board state is encoded in this paper also has the benefit that it can be adapted to fit alternate boards, or initial/goal conditions, although this comes at the expense of some of the power and accuracy that is achieved in [1].

Other branches of computer science have also been applied to the game: A.I. search methods, such as depth-first and breadth-first search, and iterative deepening, are used on both the English game and a variation using a triangular board in [2] and [3]; string rewriting systems are discussed in [4], and then applied to English peg solitaire. The attention given to the game is nothing new, and in fact drew the attentions of Alan Turing in the 1950s, about which he wrote a letter on "how to do the solitaire puzzle" which was recently featured in [5].

A lot of work on mathematical properties of the game of peg solitaire has been done: an entire chapter of [6] is devoted to it, which looks at several concepts, methods of solving the game, and some alternate goals to reducing the board to a single peg; graph theory is used to analyse the game in [7], and test whether arbitrary boards are solvable; [8] looks at the possibility of moving from one given position to another; and [9] looks at mathematical results of the game, and related puzzles.

The board styles that are studied also varies, as can be seen from those that were mentioned above. The English board, being the standard and most popular incarnation of the game, receives the most attention, with the European board of this project being touched on lightly in [6]. The triangular style of board is also mentioned on a number of occasions, and [10] looks at a one dimensional version of the game, both as a puzzle and a two player game.

3. Methodology

3.1 Constraint satisfaction problems

The game is modelled as a constraint satisfaction problem (CSP), i.e. a set of decision variables are defined, each decision variable is given a domain of possible values, and they are associated with a set of constraints. An assignment of values to the decision variables is then sought, such that all of the constraints are satisfied. This assignment is equivalent to a solution of the original problem, given that it was modelled in the correct manner.

A class of CSPs is a collection of problems that have the same definition, in terms of some parameters. An instance of the problem class is defined by instantiating these parameters to some values.

For example, the game of sudoku can be modelled as a class of CSPs. A decision variable is defined for each of the 81 squares, each variable is given the domain of the integers from 1 to 9, and constraints are added to enforce the rules that each row, column, and usual 3x3 square must contain each of the nine distinct integers. A single game of sudoku can be defined within this class of CSPs by assigning values to certain positions, and a solution to the game is found by finding a solution to the CSP with a constraint solver.

3.2 Software and language

This project uses St Andrews' own constraint programming software Savile Row and Minion. The model is written in Essence' (read as "essence prime"), which is a high level, solver independent constraint programming language. Savile Row is a modelling assistant, which translates the model from Essence' into input language that is specific to a constraint solver. Minion is a solver for constraint satisfaction problems, and takes the solver specific code from Savile Row and returns a solution that satisfies the CSP.

Constraint models written in Essence' have the following structure, as outlined in [11]:

1. Header with version number: `language ESSENCE' 1.0`
2. Parameter declarations with `given` (optional)
3. Constant definitions with `letting` (optional)
4. Decision variable declarations with `find` (optional)
5. Constraints on parameter values with `where` (optional)
6. Objective (optional)
7. Solver Control (optional)
8. Constraints

The model for this project will use parts 1, 3, 4, 7, and 8 of this generic structure, and hence has the form:

```
language ESSENCE' 1.0

letting <constants are defined here>

find <decision variables and their domains are defined here>

<solver control is added here>

such that <constraints are specified here>
```

Section 4 will expand on this template, and explain in detail how each part of this structure contribute to the model, except for solver control which will be discussed in section 7.

3.3 Peg solitaire

A model is formulated that describes the problem in its simplest form, with just the rules of the game encoded. Once this had been completed, some preliminary testing is performed to confirm that the model behaves as expected, and satisfactorily describes the problem as intended.

The next step is to address the symmetry present in the game, and break as much of it as possible, in order to form a more intelligent model that reached solution more quickly. Both symmetry of the board and the choice of moves are addressed. Again, this is followed by a stage of testing to confirm that the design choices that were implemented functioned as expected, and to gauge the effect they had on the solution.

Finally some additional code is added to try and improve the solution, and identify dead end as they occur in search, and its effectiveness is confirmed through testing.

3.4 Modelling as a problem class

Due to the nature of using a CSP approach to this problem, testing and debugging the model is somewhat challenging. The only feedback that is received from the model, and can be used to help guide the design process, is in the form of valid solutions to the given CSP.

This is combated by writing a model for the problem class of the game of peg solitaire on a board that can be inscribed within a square, and which allows horizontal and vertical piece movement only (i.e. diagonal moves are not permitted). This proved extremely useful, as the model, and any additional features, could then be tested on alternate board configurations.

A small board, i.e. an arrangement of 14 pegs that only took a matter of seconds to find a solution to, was helpful in confirming the model was working, and investigating the differences of major design choices. A slightly larger board, i.e. 24 pegs that took longer to solve, was useful to test the effectiveness of additional features that were subsequently added to the model.

Other methods that were used to gain feedback on how a model was performing were: using the constraint solver to return all solutions to a given problem, as opposed to just the first solution that is found, which was especially useful in evaluating the symmetry breaking constraints; varying the number of moves that the solution looked for, so that a subsequence of moves was found, as opposed to finding all steps from the initial state to the goal state.

4. Modelling Peg Solitaire

4.1 State representation

In order to model this problem, a representation of the state of the board is chosen. An `n*n` matrix of binary values is used to describe the state of the board at any given step, where a `1` represents a peg being present in the given position, and a `0` represents the slot at that location being empty. A `0` is also used in the corners of the board to represent the superfluous matrix entries that don't represent any location on the board. This is illustrated in Figure 4 below, using the initial state of the European game as an example, with the excess corner squares highlighted in red.

0	0	1	1	1	0	0
0	1	1	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
0	1	1	1	1	1	0
0	0	1	1	1	0	0

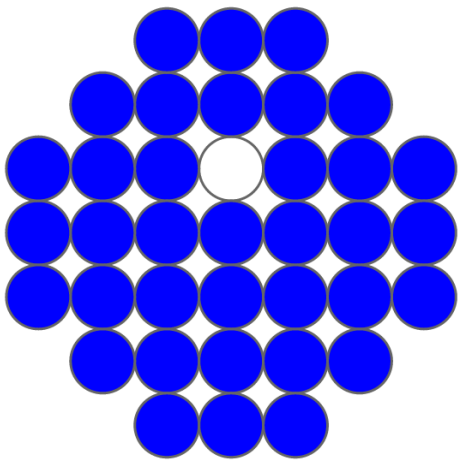


Figure 4

As mentioned in section 3, defining a model for the class of CSPs is useful, as the variation of solitaire that is being solved can be easily altered by simply changing the dimension of the matrix to the smallest square that will cover the size of the board, and fixing any unnecessary matrix entries to `0`.

This representation is extended to find a solution of the peg solitaire game, by finding a sequence of matrices that fit the above description. The first and last matrices in this sequence must satisfy the initial and goal conditions, and constraints are applied to pairs of subsequent matrices to ensure that the board can go from one state to another in a single, legal move. As such, a solution is obtained by applying the moves that transform the board at each given step, beginning at the initial state, and arriving at the goal state.

4.2 Extending the model to multiple boards

The solution is parameterised by `n` and `noBoards`, where `n` is the size of the square within which the board is inscribed, and `noBoards` is the length of the sequence of boards that is being sought; for the game of European peg solitaire the assignments `n=7` and `noBoards=36` are made. These are defined in the constant definitions section of the model, along with some frequently used values that are assigned shorthands, with the lines

```
letting n be 7
letting noBoards be 36
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)
```

so that `binaryDom` represents the domain of `0` and `1`, `nDom` is the domain of integers from `1` to `n`, and `noBoardsDom` is the domain of integers from `1` to `noBoards`.

The model searches for a solution defined by the single decision variable `Boards`, which is a matrix described by the line

```
find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom
```

so that the individual, generic entry `Boards[i,j,k]` is the state of the position at the j^{th} row and k^{th} column, on the i^{th} board. Although only a single decision variable is defined, it is composed of `noBoards*n*n` entries, which is `1764` for the European game.

The definition of the board is complemented by fixing the excess matrix entries to zero by adding constraint, such as

```
forAll i : noBoardsDom . Boards[i,1,1] = 0
```

which ensures the upper- and left-most corner of each board is always zero. Similar constraints are applied to the other 11 entries highlighted in Figure 4 for the European board.

Since the game of peg solitaire is reversible, i.e. a player can move to an earlier stage in the game by making moves in reverse, the game can readily be modelled in either direction: a 'forward' model that finds a solution in the intuitive way of going from the initial state to the goal state, or a 'backward' model that begins at the goal state and adds pegs to reach the initial state. Both methods were tested, and this will be explored in more detail in section 5. For the duration of this chapter, the 'backward' model will be considered.

4.3 Enforcing legal moves

Using the 'backward' model, the first board must satisfy the goal condition, and so has a single peg. In transitioning to the second board, this peg then makes a jump two steps in any direction, and an additional peg is added in the space between the jumping pegs original location and new location.

This is broken down into three conditions, and constraints are applied to the model to ensure that each of these constraints are held. Firstly, the number of pegs in each board increases by one at each step; secondly, the number of matrix entries that are different between each pair of neighbouring boards is exactly three; and thirdly, if a peg is removed from a location in a transition, then two pegs are added in the fashion described above.

The first condition, the increasing number of pegs, was first encoded into the model with a couple of constraints, as follows:

```
sum(flatten(Boards[1,..,..])) = 1,

forAll i : int(1..noBoards-1) .
 sum(flatten(Boards[i,..,..])) = sum(flatten(Boards[i+1,..,..])) - 1,
```

which defines the sum over all entries of the first board to be `1` with the first constraint, and is complemented by the second constraint so that the sum over all entries of each subsequent board increases by just one. On revision, these were consolidated to the single constraint

```
forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i
```

which utilises the fact that the number of pegs in each board is equal to the index of that board.

The second condition, that exactly three entries of each pair of neighbouring matrices are different, is then added to the model with the constraint

```
forAll i : int(1..noBoards-1) .
 ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3
```

which takes the sum of the absolute value of the differences between each corresponding location of each pair of boards, and ensures that it is exactly three.

Finally, the third condition regarding the addition of pegs is achieved with a constraint of the form

```
forAll i : int(1..noBoards-1) .
 forAll j : nDom .
  forAll k : nDom .
   LHS -> RHS
```

which ranges over every board, and enforces the constraint that if the left hand side (LHS) is true, then the right hand side (RHS) must also be true. This is used to ensure only legal moves are made using the left hand side

```
LHS = ( Boards[i,j,k] - Boards[i+1,j,k] = 1 )
```

which checks the value of subtracting the value at position `(j,k)` of each board with the same position of the previous board. If the value of this is 1, as is being tested by the LHS of the constraint, then a peg has been removed. If this does hold then the right hand side

```
RHS = ( ( Boards[i+1,j+1,k] - Boards[i,j+1,k] = 1 ) /\
        ( Boards[i+1,j+2,k] - Boards[i,j+2,k] = 1 ) ) \/ ...
```

is enforced, which takes the difference between the two positions below the position `(j,k)` and forces them both to be `1`. This is encapsulated in parentheses, three further clauses are added for each of the other three directions, and they are combined to form a disjunction, so that at least one of them must hold.

The game is completed by manually setting the location of the vacant slot in the ‘initial’ board using

```
Boards[noBoards,3,4] = 0
```

and setting the location of the last remaining peg in the ‘goal’ state by adding

```
Boards[1,2,4] = 1
```

so that the model has everything it needs to find a solution. The model can be seen in full in Figure 5.

4.4 Complete model

```
language ESSENCE' 1.0

letting n be 7
letting noBoards be 36
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom, nDom, nDom] of binaryDom

such that

  Boards[1,2,4] = 0,

  Boards[noBoards,3,4] = 0,

  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  forAll i : int(1..noBoards-1) .
    forAll j : nDom .
      forAll k : nDom .
        ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
          ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
            ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

          ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
            ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

          ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
            ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

          ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
            ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),

forAll i : noBoardsDom . Boards[ i , 1 , 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , 1 , 2 ] = 0,
forAll i : noBoardsDom . Boards[ i , 2 , 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , 1 ,n-1] = 0,
forAll i : noBoardsDom . Boards[ i , 1 , n ] = 0,
forAll i : noBoardsDom . Boards[ i , 2 , n ] = 0,
forAll i : noBoardsDom . Boards[ i ,n-1, 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 2 ] = 0,
forAll i : noBoardsDom . Boards[ i ,n-1, n ] = 0,
forAll i : noBoardsDom . Boards[ i , n ,n-1] = 0,
forAll i : noBoardsDom . Boards[ i , n , n ] = 0
```

Figure 5

5. Testing the model

5.1 Arrow configuration

Running the model on the board for European peg solitaire, as described in Figure 5, returned no solution after a number of hours, so a test environment was necessary to confirm that the model was working as a whole, and also to perform some preliminary evaluation of the different models mentioned so far.

For this purpose a small configuration of 14 pegs into an arrow shape was chosen, and is shown in Figure 6 below.

0	0	1	0	0
0	1	1	1	0
1	1	0	1	1
0	1	1	1	0
0	1	1	1	0

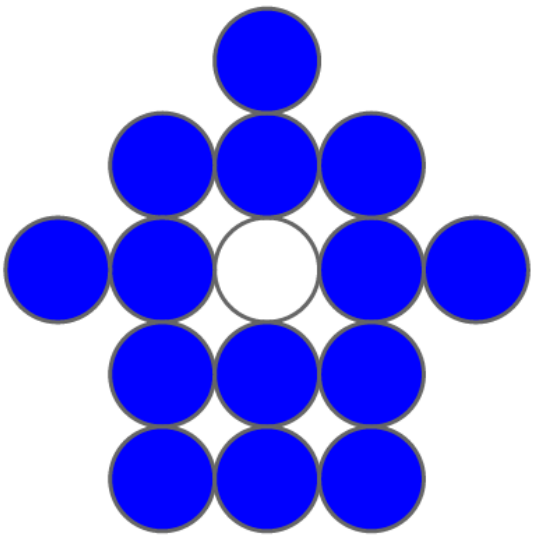


Figure 6

The model was adapted for this game by assigning `n=5` and `noBoards=14`, and fixing the ten locations highlighted above to zero for all boards. This model is shown in full in Figure 7 below.

After some minor debugging, the model ran as expected and returned a solution which satisfied all constraints, and hence a solution to the game that the model described. Using this board configuration as a test environment, experiments were run to evaluate the effectiveness of the models covered so far.

```
language ESSENCE' 1.0

letting n be 5
letting noBoards be 14
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

such that

  Boards[1,4,3] = 1,

  Boards[noBoards,3,3] = 0,

  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,


  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  forAll i : int(1..noBoards-1) .
      forAll j : nDom .
        forAll k : nDom .
          ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
            ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
              ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

            ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
              ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

            ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
              ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

            ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
              ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),

  forAll i : noBoardsDom . Boards[ i , 1 , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , 2 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , 1 ] = 0,

  forAll i : noBoardsDom . Boards[ i , 1 , n ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , n ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 ,n-1] = 0,

  forAll i : noBoardsDom . Boards[ i , n , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i ,n-1, 1 ] = 0,

  forAll i : noBoardsDom . Boards[ i , n , n ] = 0,
  forAll i : noBoardsDom . Boards[ i ,n-1, n ] = 0
```

Figure 7

5.2 Direction of solving

As touched on in section 4.2, the moves that one takes to solve a game of peg solitaire are reversible, and so when devising a model for the game, either direction can be chosen. Both a forward and backward model were considered in this case, and their performance evaluated. These were also compared to a model that used bidirectional search.

Both the forward and backward models that have been discussed are unidirectional search methods, i.e. they begin with a state that satisfies one of the end conditions, search for a set of steps that progress from this state, and end when they have found a series of moves that leads to state that satisfies the other end condition. Bidirectional search runs two concurrent searches, one from each of the end conditions, and ends when they find a common state that both searches have in common. This is often beneficial, as two smaller searches can take less time that a single, larger search.

In the context of peg solitaire, this is done by searching from the initial state, taking forward moves, and simultaneously searching from the goal state, taking moves in reverse. Both searches are performed for half the number of moves required for a solution, and a constraint is added so that they must both reach the same state.

This is implemented in Essence' for the arrow configuration by declaring two decision variables, with the lines

```
find BoardsForward  : matrix indexed by [int(1..noBoards+1),nDom,nDom] of binaryDom
find BoardsBackward : matrix indexed by [int(1..noBoards)  ,nDom,nDom] of binaryDom
```

with the assignment `noBoards=7`. The constraints that are used to enforce legal moves between pairs of boards are applied to these decision variables as above, and using the respective constraint for each direction. A final constraint

```
forAll i: int(1..n) .
 forAll j : int(1..n) .
  FM[noBoards+1,i,j] = BM[noBoards,i,j]
```

is added so that the final board is the same for both searches, and that their individual solutions can be combined to form a solution for the game.

The performance of each model is analysed by looking at three factors: the time taken for Savile Row to convert the model into input language for Minion, called 'Savile Row Time'; the time taken for Minion to solve the problem after it has been converted by Savile Row, referred to as 'Minion Time'; and the number of nodes in the constraint model that is passed from Savile Row to Minion, called 'Minion Nodes'. The final metric is a useful measure, as it gives an indication of how complex the problem is after it has been processed by Savile Row, and a lower complexity often correlates with a faster solution.

5.3 Results

The results for the three models on the arrow configuration are shown below in Figure 8, where time is measured in seconds.

Model	Minion Nodes	Minion Time	Savile Row Time
Forward:	2486	0.144	0.939
Backward:	78	0.101	0.988
Bidirectional:	3837	0.174	1.071

Figure 8

The game itself is solved in roughly one second by all models, with the Forward and Backward models outperforming the Bidirectional model in all aspects. This is most likely due to the fact that, even though the model is performing smaller searches, it can take some time to find a common board arrangement that can be reached by both directions of the search.

Between the two remaining models, the times taken to return a solution are very similar (1.083s for Forward, 1.089s for Backward), whereas the number of nodes each model have a much clearer disparity, with the Backward model being less complex. As the size of the board increases, as well as the number of pegs involved, the more simple Backward model should have a less significant increase in time taken to produce a solution. For this reason, the Backward model is chosen as the primary candidate to model the game, and will form the basis of further refinement.

The better performance of the Backward model is likely due to the constraints being written around how the pegs move on the board. Dead ends in the search would correspond to unsolvable board positions, e.g. in the Forwards model, remaining pegs being too far away from each other, which happens quite regularly when arbitrary moves are chosen. In contrast, the Backward model begins with a single peg, and other pegs can only be added around it with legal moves outward, and so the pegs tend to form a cluster in the board.

Another factor which may have an effect is the value ordering in Minion, which is a standard increasing value ordering; i.e. when a value is to be assigned to a variable, and there is more than one option, the lowest is tried first. The start of the sequence of boards in the Backward model is predominantly 0s, in contrast to the first few boards in the Forward model, so the Backward model is a better fit to the variable ordering of the solver.

6. Refining the Model

Once the model had been confirmed to work, methods of refining and improving the model were explored. These included breaking symmetry in the board, breaking symmetry in the choice of moves, and spotting dead ends earlier in the search.

6.1 Symmetry in constraint programming

The concept of symmetry is especially prevalent in constraint programming, as it can be taken advantage of to make drastic improvements in the performance of a model. In general, symmetry is a transformation of an object that maintains it's shape or structure, as illustrated with the three lines of symmetry of a triangle below in Figure 9. To complete the set of symmetries, rotations of 120^0 and 240^0 are added, which also map the triangle onto itself. The European board layout is also displayed in Figure 10, with the four lines of reflectional symmetry. Again, three rotations are added to complete the set of symmetries.

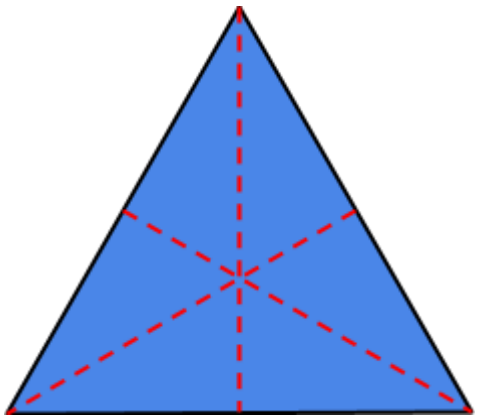


Figure 9

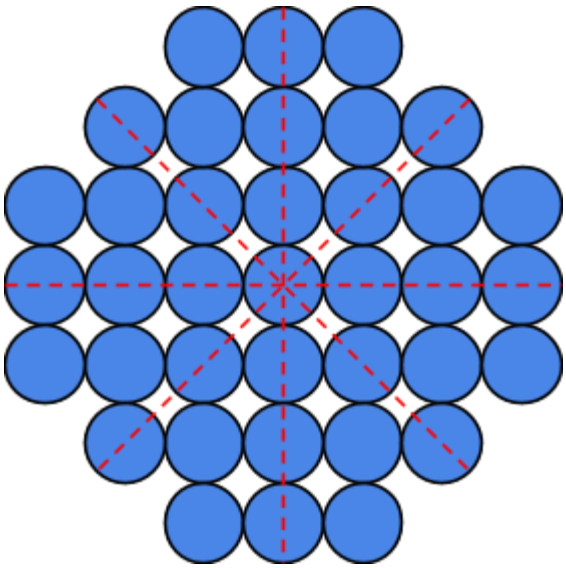


Figure 10

Symmetry between assignments in constraint programming also preserves attributes of those assignments, in particular solutionhood; i.e. all assignments that are symmetric to a solution are also solutions, and all assignment that are symmetric to a non-solution are not solutions either. This is extremely useful, as it helps prevent the model wasting time repeatedly searching for a solution in the same dead end. By adding symmetry breaking constraints, some valid solutions are no longer accepted by the model, which is permissible as long as one solution in the equivalence class is still found. The added constraints help to guide the search by discounting assignments that don't satisfy them, and although solutions are ruled out of the model, they can be regained after search is completed by applying the symmetries that were broken using those constraints to the accepted solution.

6.2 Reflectional and rotational symmetry

As shown earlier in Figure 10, the board is highly symmetric, containing four reflectional symmetries and three rotational symmetries. The reflection through the vertical line of symmetry will be considered first, and the constraints will be similarly applied to the other reflections and rotations afterwards.

Since the board contains this reflectional symmetry, some of the states that the board goes through as pegs are added may also contain this reflectional symmetry, e.g. the first state with a single peg. Given that this symmetry is present in a state, next moves can be categorized into pairs that are reflections of each other. This is shown below in Figure 11.

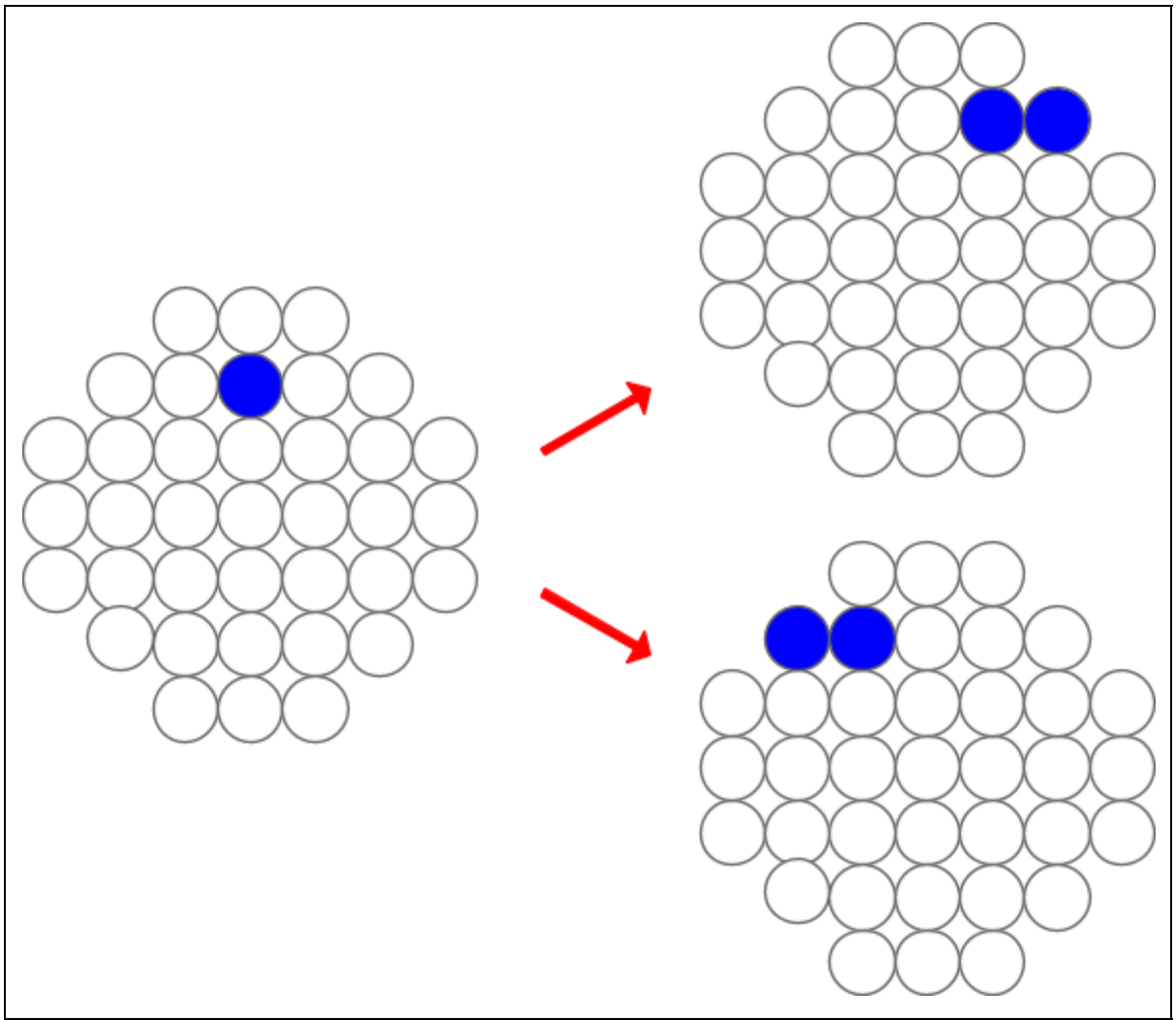


Figure 11

Since these states are symmetric to one another, only one of them needs to be considered in the search for a solution. If a solution is eventually found, then the reflection can be applied to each board state that is traversed, and the corresponding lost solution is regained.

In order to detect this reflection, and enforce the restriction as mentioned above, a representation of the board states with this reflection applied has to be available for comparison. Another decision variable is added as

```
find RB : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom
```

which is identical to the Boards variable in the model, and where RB stands for 'reflected board'. The values of each entry in the matrix is defined in terms of the Boards matrix, as follows

```
forAll i : noBoardsDom .
 forAll j : nDom .
  forAll k : nDom .
   RB[i,j,k] = Boards[i,j,n+1-k]
```

so that RB is a sequence of boards that corresponds exactly to Boards, where each state in the sequence goes through the reflection that is being considered.

Any one state that is passed will have this symmetry if it is the same both in the Boards and RB matrices, which is how it's presence will be detected. When present, a constraint will be applied to the following index so that only one of the moves will be allowed by the model.

This is done with the lines

```
forAll i : int(1..noBoards-1) .
 (forAll j : nDom . forAll k : nDom . Boards[i,j,k] = RB[i,j,k] )
 -> flatten(RB[i+1,..,..]) <=lex flatten(Boards[i+1,..,..])
```

where lexicographical ordering is enforced between the next states of Boards and RB. Since the orderings greater than or less that could be used here, both were considered and tested, which will be covered in the next chapter. It is worth noting that strict inequalities cannot be used here, as a transition from a symmetric state to another symmetric state must be allowed.

This concept can also be applied to the other three reflections, and three rotational symmetries that are also present within the board, by simply adding further auxiliary variables and assigning their values in terms of the Boards matrix as required. These are illustrated below in Figure 12, with each assignment rule labelled with it's corresponding reflection or rotation.

After these assignments are made, the second constraint is applied in the same fashion: if the symmetry is detected at the i^{th} position then an ordering is enforced on the $i+1^{th}$ states so that only one of the pair of moves are accepted.

Any combination of these can be implemented at the same time, and the effectiveness of these added constraints is explored in the next chapter.

`B90[i,j,k] = Boards[i,n+1-k,j]`	`RB2[i,j,k] = Boards[i,n+1-j,k]`
90^0 rotation	Horizontal reflection
`B180[i,j,k] = Boards[i,n+1-j,n+1-k]`	`DB1[i,j,k] = Boards[i,k,j]`
180^0 rotation	Diagonal reflection 1
`B270[i,j,k] = Boards[i,k,n+1-j]`	`DB2[i,j,k] = Boards[i,n+1-k,n+1-j]`
270^0 rotation	Diagonal reflection 2

Figure 12

6.3 Independent move symmetry

Another type of symmetry that is present in the game of peg solitaire is the choice between two or more independent moves. A set of moves are described as independent if the spaces on the board that they effect do not overlap. An example of a pair of independent moves, taken from a solution of the arrow configuration, is shown in Figure 13.

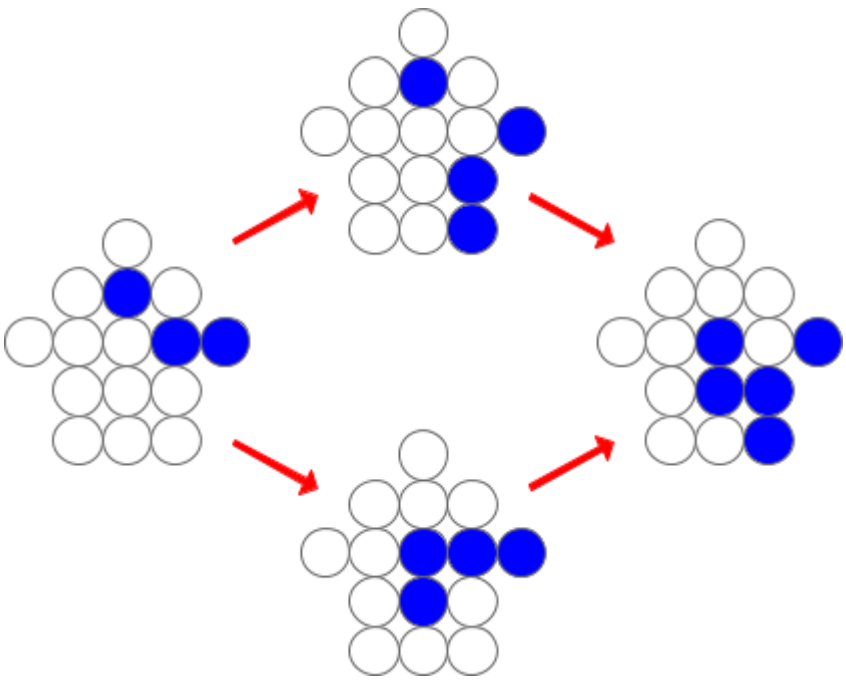


Figure 13

The two subsequences that begin with the lefthand board and end with the righthand board, by going through different intermediate positions, are symmetric in the sense that they arrive at the same destination by taking different paths. Although the above situation only contains two alternative, this increases as the number of moves in the set increases. Given n independent moves, any permutation of these moves is acceptable, so that means there are n! combinations. Couple this with the fact that, if there are multiple occasions of independent moves in a solution, then the total number of paths is the product of the number of choices in each case, then it is apparent that breaking this symmetry should be very effective.

In order to break this symmetry, the model must be able to detect when subsequent moves are independent, and only accept a single order of the possible moves. Using the properties of the state representation, the presence of a pair of independent moves is relatively straightforward with

```
forAll i : int(2..noBoards-2) .
 ( (sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+2,j,k] |) = 6 ) -> RHS
```

which sums the absolute difference between every location on board states that are two steps apart and, if it is equal to six, the righthand side is enforced.

The difficulty with imposing the choice of move comes due to the fact that, given the current setup of encoding only the state, there is no way to identify how a state was reached. A method to identify moves in terms of the change of state is required, and was found using the method illustrated in Figure 14:

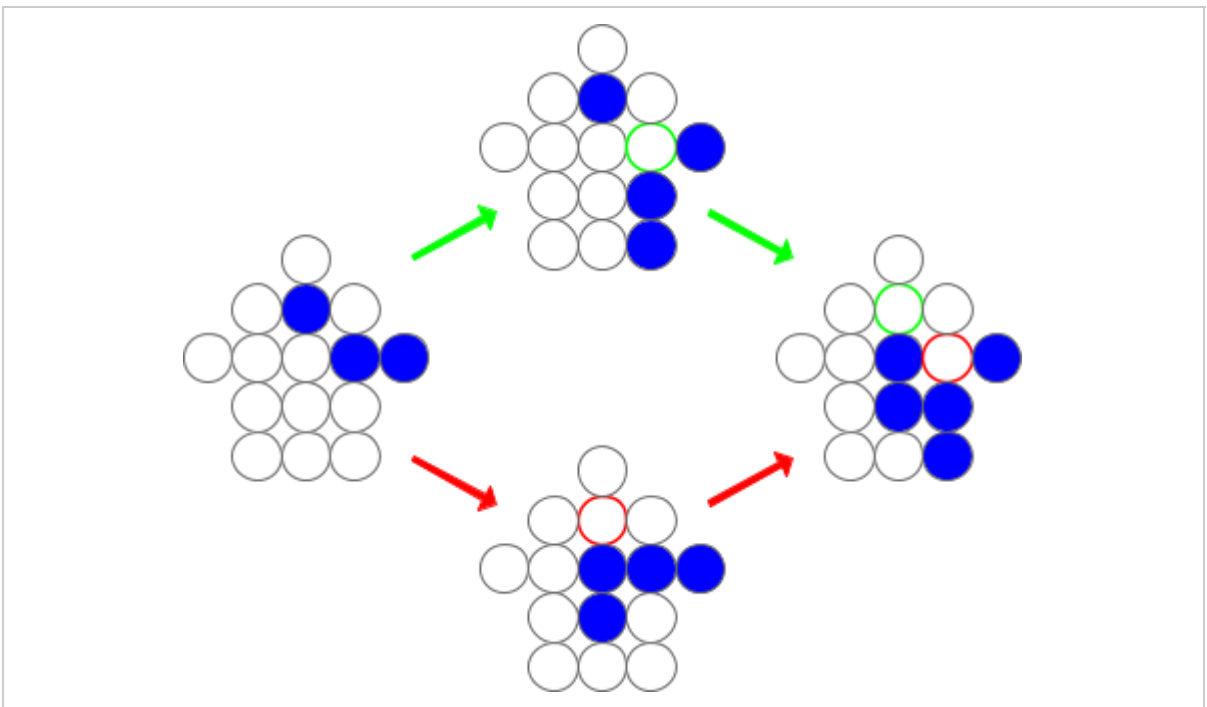


Figure 14

The two options are now colour coded, with green and red circles around the location where the jumping peg has moved from during their respective transitions. Now at the righthand state, there is a way of discerning which path was taken. Another auxiliary variable is required for this, which is added with

```
find transitionPeg : matrix indexed by [int(2..noBoards),int(1,2)] of nDom
```

which is a matrix of `n-1` pairs of values, defined as the location that the moving peg has just jumped from with the constraint

```
forAll i : int(2..noBoards) .
 forAll j : nDom .
  forAll k : nDom .
   ( Boards[i-1,j,k] - Boards[i,j,k] = 1 )
   -> (transitionPeg[i,1] = j /\ transitionPeg[i,2] = k)
```

so that these are identified for all transitions. The independent move symmetry is finally broken by completing the earlier constraint with

```
RHS = transitionPeg[i+1,..] <lex transitionPeg[i+2,..]
```

Again, since an ordering is used to define which option is taken, both orders were tried and tested, and will be detailed in the next chapter.

6.4 Identifying dead ends

The final refinement that is added to the model is some additional code to spot dead ends in the search without having to complete the assignment. Each time a dead end is reached, the constraint solver must backtrack and search for an alternate solution, so any assistance to identify these earlier in the search will improve performance of the model.

In order to be able to add a peg to the backwards model, there must be two spaces beside each other in a board state: one for the moving peg to slot into, and one beside it for the new peg to be added. Therefore, every board except for the last one must have two `0` entries that are side by side. This is added with

```
forAll i : int(1..noBoards-1) .
 exists j : int(1..n-1) .
   exists k : int(1..n-1) .
     ((Boards[i,j,k] + Boards[i,j,k+1] = 0) \/ (Boards[i,j,k] + Boards[i,j+1,k] = 0))
```

which checks every pair, both horizontally and vertically. This generic encoding must be tailored for specific boards, so that the dummy `0`s in the corners of the board are excluded from the check, and don't satisfy the constraint when they shouldn't.

7. Evaluating the model

7.1 Square configuration

A slightly larger test configuration is used to run these refinements, so that the time taken to reach a solution is more than as trivially short as a second. The configuration is shown below in Figure 15.

1	1	1	1	1
1	1	0	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

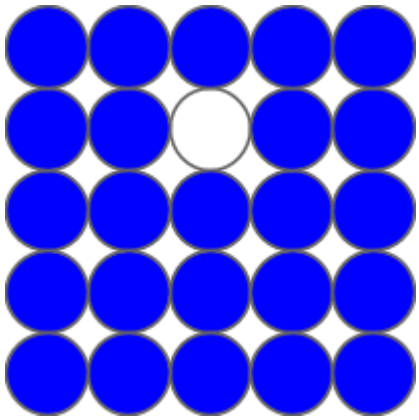


Figure 15

This configuration is run with the model from section 4 with assignments `n=5` and `noBoards=24` and the code in full is displayed below in Figure 16. It reaches solution in ~25 minutes, so the effectiveness of adding the refinements discussed in the previous chapters can be judged comparatively. It is also helpful that there are no excess squares in corner that some of the code needs to be tailored to.

After getting a point of reference for how long the model takes to solve this board with no added heuristics, each of three heuristics was applied in turn and the results were recorded, followed by all three combinations of pairs of heuristics, and finally all were implemented at once. The results can be seen below in Figure 17.

As expected, each additional heuristic helped to simplify the model, in the sense that less nodes were passed from Savile Row to Minion, and both breaking independent move symmetry and spotting dead ends helped to reach a quicker solution. A surprising result was that by adding constraints to break all reflectional and rotational symmetry actually resulted in Minion taking a longer time to find a valid solution.

Each of this heuristics was then tested in turn, with more detailed experiments, to find how best they could be implemented.

```
language ESSENCE' 1.0

letting n be 5
letting noBoards be 24
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

such that

  Boards[1,1,3] = 1,

  Boards[noBoards,2,3] = 0,

  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  forAll i : int(1..noBoards-1) .
    forAll j : nDom .
      forAll k : nDom .
        ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
          ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
            ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

          ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
            ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

          ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
            ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

          ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
            ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) )
```

Figure 16

Symmetry breaking	Minion Nodes	Minion Time	Savile Row Time
None:	82,931,603	1464.89	1.625
Rotation/reflection:	81,444,151	6543.12	3.519
Independent move:	7,412,473	241.01	2.040
Dead end spotting:	72,871,377	1398.27	1.655
Reflections + Ind moves:	7,265,990	775.16	4.204
Reflections + Dead ends:	71,498,773	6098.56	4.111
Ind moves + Dead end:	6,707,916	240.08	2.450
Full symmetry breaking:	6,581,927	714.73	4.121

Figure 17

7.2 Reflectional and rotational symmetry

To confirm the extent to which these symmetry breaking constraints were working, the arrow configuration from chapter 5 was taken into consideration again. However, instead of searching for just one solution to the problem, Minion is run with the `-all-solutions` flag which returns a solution file for every possible assignment that is valid. When run on the original model, 290 solutions are returned, and when all rotational and reflectional symmetry is broken this is reduced to 145. Since the board used in this configuration only has the single reflectional symmetry, it's less likely that the other symmetries will appear, and indeed when the symmetry from just this reflection is broken the total number of results is also 145. This result confirms that the reflectional symmetry is broken in full, as the reflection can be applied to each of the valid solutions to regain one of the discounted solutions, bringing the total back up the 290.

Revisiting the results gotten in Figure 17, when applying all of these symmetry breaking constraints to the square configuration, the time taken goes up from ~25 minutes to almost two hours. After some analysis of the constraints, this is most likely due to them being too restrictive, and making it hard for a valid solution to satisfy all of them.

To test these further, each individual symmetry is broken in turn. The independent move symmetry breaking and dead end spotting are included in all tests, and both orderings were considered. The results are shown in Figure 18.

Symmetry breaking	Ordering	Minion Nodes	Minion Time	SR Time
None		6,707,916	240.080	2.450
All		6,581,927	714.730	4.121
Vertical reflection	Boards <= RB1	6,581,927	346.762	2.603
	Boards >= RB1	1,569,646	71.824	2.517
Horizontal reflection	Boards <= RB2	6,707,916	447.900	3.944
	Boards >= RB2	6,707,916	411.824	3.847
Forward diagonal	Boards <= DB1	6,707,916	408.501	3.562
	Boards >= DB1	6,707,916	424.299	3.999
Backward diagonal	Boards <= DB2	6,707,916	395.225	3.517
	Boards >= DB2	6,707,916	407.712	3.542
90^0 rotation	Boards <= B90	6,707,916	400.623	3.373
	Boards >= B90	6,707,916	358.913	2.904
180^0 rotation	Boards <= B180	6,707,916	352.518	2.737
	Boards >= B180	6,707,916	368.704	3.108
270^0 rotation	Boards <= B270	6,707,916	366.785	3.239
	Boards >= B270	6,707,916	396.605	3.898

Figure 18

As can be seen, with all other factors fixed and just varying the type of reflectional or rotational symmetry that is broken, the performance varies. For all symmetry breaking, except for along the vertical line of symmetry, there is no reduction in the number of nodes that Minion is given to solve, so the problem is not being made any simpler. Indeed the time compared to not having them at all is increased, so the added constraints means the solver takes longer to find a valid solution.

On the other hand, breaking symmetry along the vertical line of symmetry does have a noticeable effect, with both choice of orderings reducing the number of nodes. The reason that this alone stands out as making a difference is that it is immediately relevant, because the very first board position is symmetric to itself along this line. This is true of the square configuration that is used for these tests, and also applicable to the European board, which makes the square board a suitable choice for testing symmetry breaking which will eventually be applied to the European board.

The first ordering, although having a slight decrease in nodes, again increases the search time, and so the added simplicity the model gets due to breaking the symmetry in this way is outweighed by the cost in time it takes the solver to find a solution. On the other hand, the second ordering suits the problem much better, with a drastic decrease in the number of nodes, resulting in a much shorter search, to about ⅓ of the original time. The major difference in performance of the choice of ordering here again is mainly attributable to the value ordering heuristic of Minion. Enforcing `Boards <=lex RB1` tends to put the `1`s on the left of each entry of `Boards`, `0`s on the right, and vice versa for states of `RB1`. As touched on earlier, Minion assigns its values in increasing order, and when assigning values it fills matrices from the bottom right hand corner upwards. As it tries to assign values to `Boards` first, tending to have `0`s on the right hand side as opposed to `1`s fits this value assignment order best.

As the symmetry breaking heuristic along the vertical axis of symmetry, with the ordering as described, made a significant improvement to the search, it is added to the model. The next experiment that is run is to test if any further symmetry breaking constraints can be added to improve the model again. Each of the six additional symmetries constraints are added to the model, and ran again to find a solution. The results are below in Figure 19.

As displayed in the results, the addition of any of the other symmetries does not help to simplify the model, as the number of nodes stays the same throughout. Furthermore, the time taken to reach a solution is slightly increased each time. Again, this is due the fact that more restrictions are being applied to the assignments, so the model has to work a little harder to meet all the requirements.

Since no additional symmetry breaking constraints is helping to improve the solution, they will not be included in the final model, and just the single reflection will be addressed.

Symmetry broken, order	Minion Nodes	Minion Time	Savile Row Time
RB1 <= Boards only	1,569,646	71.824	2.517
RB1 + RB2 <= Boards	1,569,646	82.823	3.037
RB1 + RB2 >= Boards	1,569,646	84.393	2.928
RB1 + R90 <= Boards	1,569,646	84.168	3.125
RB1 + R90 >= Boards	1,569,646	86.705	2.956
RB1 + R180 <= Boards	1,569,646	88.764	2.874
RB1 + R180 >= Boards	1,569,646	78.778	2.755
RB1 + R270 <= Boards	1,569,646	77.714	2.983
RB1 + R270 >= Boards	1,569,646	88.357	2.774
RB1 + DB1 <= Boards	1,569,646	76.603	3.136
RB1 + DB1 >= Boards	1,569,646	82.656	2.812
RB1 + DB2 <= Boards	1,569,646	88.642	2.825
RB1 + DB2 >= Boards	1,569,646	83.477	2.824

Figure 19

7.3 Independent move symmetry

Testing of the symmetry breaking constraints for independent moves again began with adding them to the model for the arrow configuration, and running to find all possible solutions. The results are somewhat surprising, with the number of total solutions being drastically reduced, and also a different number of solutions found for the two possible orderings.

When applied with `transitionPeg[i+1,..] <lex transitionPeg[i+2,..]` as described in section 6, the number of solutions was reduced from 290 to 9, and when this was reversed so that the ordering `transitionPeg[i+1,..] >lex transitionPeg[i+2,..]` was imposed, the number of solutions was 11.

Although the constraint itself is defined to identify just a pair of independent moves, this is sufficient to cover a sequence of independent moves of any length. Given a sequence of n independent moves, any subsequence of length 2 within this sequence would be identified by the symmetry breaking constraint. Therefore, the constraint gets applied to each of the n-1 subsequences of length two, and in fact contains the scope to break all independent move symmetry. It is this feature of the constraint to work in tandem with itself that makes it a very effective heuristic, and why it reduces the number of solutions to such a degree.

In order to investigate why different orderings resulted in a different total number of solutions, some miniature puzzles involving the addition of pegs were tested to try and isolate the behaviour. The puzzle depicted in Figure 20 proved to highlight this.

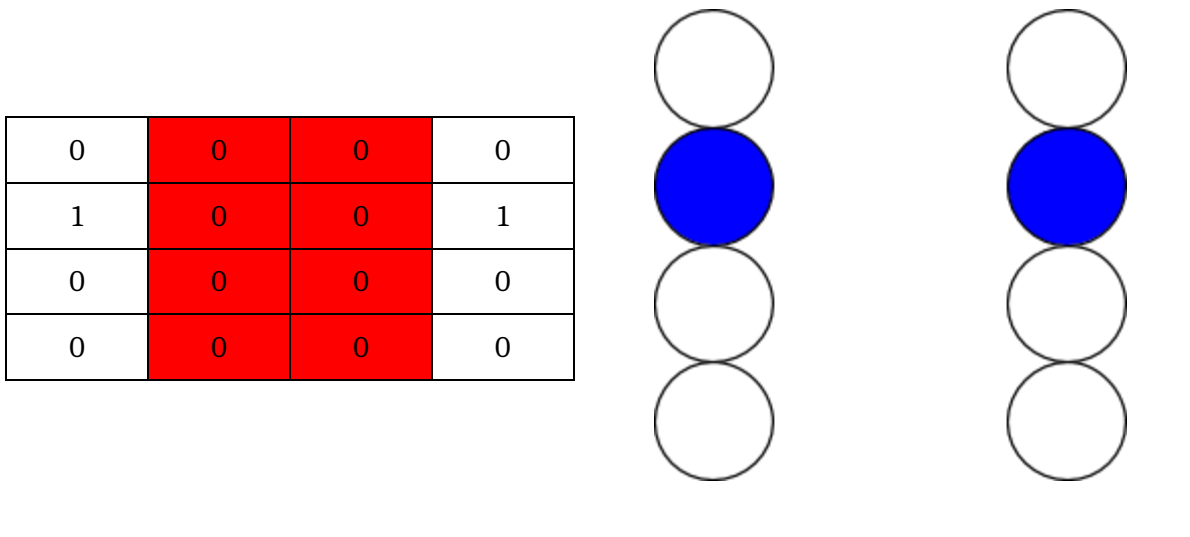
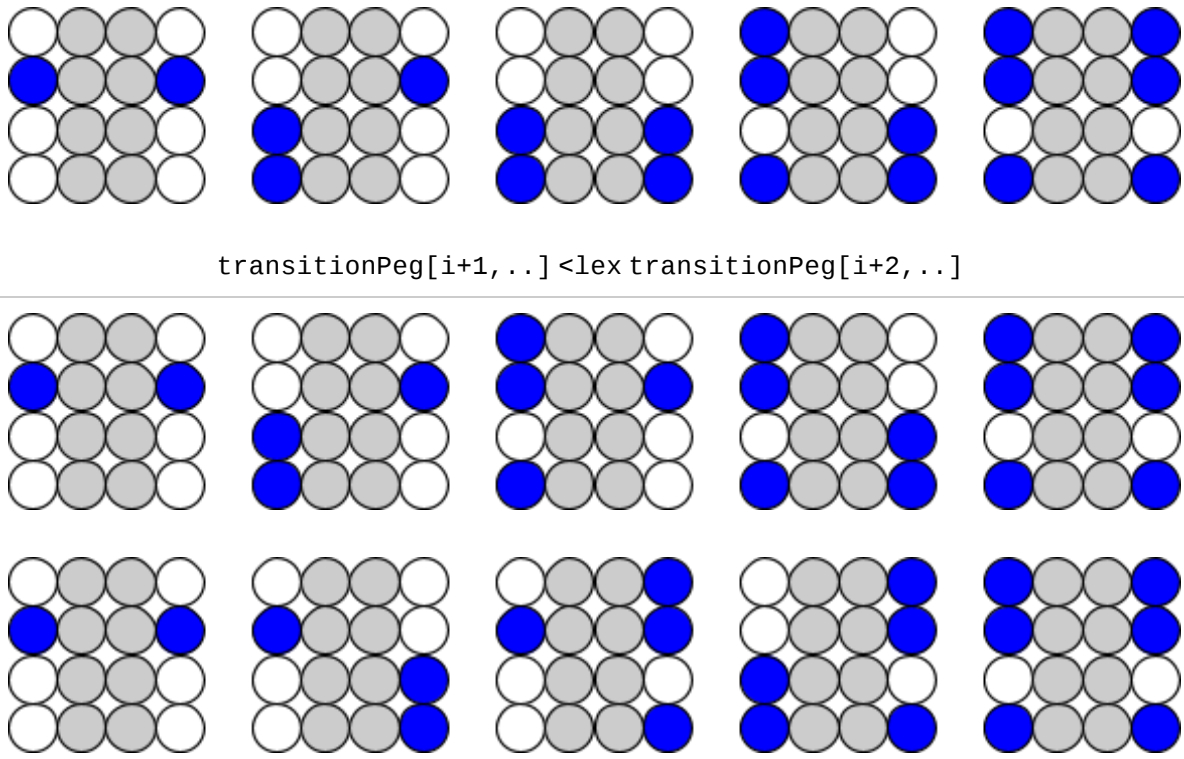


Figure 20

Two separate lines of four slots are defined, which are not overlapping or adjacent to each other. A sequence of four moves is sought, following the usual rules of peg solitaire that have been followed so far. There is only one solution to this, which is to make two additions to each column, but by running this as a constraint model, or simply manually playing with the moves, it can be seen that there are six different combinations that the moves can be played in. Since all six sequences result in the same configuration, it would be ideal if the symmetry breaking constraint reduces these to a single possibility.

Running this model while breaking independent move symmetry gave the results as follows, in Figure 21, with the solutions labelled with the orderings that permitted them. Note that the central columns to complete the square are included to help visualising the board, but cannot be moved into, and are coloured in grey to represent this.



`transitionPeg[i+1,..] <lex transitionPeg[i+2,..]`

`transitionPeg[i+1,..] >lex transitionPeg[i+2,..]`

Figure 21

As displayed, the first ordering finds one valid solution, whereas the second ordering finds two. The key here is that the puzzle is designed so that it can be organised into a sequence that contains two pairs of independent moves, or just one pair of independent moves. The first ordering permits the former, and the second ordering permits the latter. Clearly the first ordering is preferable, as it reduces the number of solutions, and this corresponds to the smaller number of solutions that were found when applied to the arrow configuration.

Both orderings were tested on the square configuration to further compare the two, and the results are in Figure 22 below. All experiments include symmetry breaking on the vertical axis that was decided on in the previous section.

Independent move breaking	Minion Nodes	Minion Time	Savile Row Time
None	71,498,773	6098.56	4.111
tPeg[i+1,..] <lex tPeg[i+2,..]	1,569,646	71.824	2.517
tPeg[i+1,..] >lex tPeg[i+2,..]	161,886,259	7583.56	2.808

Figure 22

These results serve to reinforce the previous feedback that using the `<lex` ordering serves to improve the model the most, and as such will be also included in the final model.

7.4 Identifying dead ends

The final addition to the model that is to be tested is the additional code that spots dead ends in advance in the search. The results of this experiment are shown in Figure 23, where the reflectional and independent move symmetry decided upon in the previous sections are applied in both cases.

Dead end spotting	Minion Nodes	Minion Time	Savile Row Time
Yes	1,569,646	71.824	2.517
No	1,756,892	82.978	2.577

Figure 23

As expected, the dead end makes a small improvement to the search, as it identifies dead ends one step earlier. It is maintained in the model as it can be seen to improve it.

7.5 Solver control

One feature of the constraint model that hasn't been addressed yet is the solver control that was mentioned in chapter 3. As described in [11] "Essence' also supports some rudimentary options for controlling which variables the solver will branch on, and which variable ordering heuristic it will use".

The variables which the model assigns first are declared with the line `branching on`, and the options for the heuristic are `static`, `conflict`, `sdf` and `srf`. As the constraints are designed to identify symmetries in board states, and then apply constraints where necessary to later board states, the best way to assign values is by filling the boards completely, in an ascending order; e.g. complete all entries on the first board, then the second, etc. For this reason the solver control

```
branching on [ Boards[i,..,..] | i : noBoardsDom ]

heuristic <choice of heuristic>
```

is added. The previous results have all branched on the variables in this order, and used the `conflict` heuristic. Figure 24 shows the complete model run with each heuristic, and displays why this choice was made.

Heuristic	Minion Nodes	Minion Time	Savile Row Time
conflict	1,569,646	71.824	2.517
static	1,684,140	72.084	2.590
sdf	1,684,140	77.980	2.820
srf	1,684,140	77.551	2.412

Figure 24

While the difference in times is not much throughout, the conflict heuristic performs the best. This is due in part to the fact that it helps to simplify the model slightly, which is indicated by the lower number of nodes compared to the other three.

8. Solving European peg solitaire

Based on the results drawn from the previous chapter, the best performing model was compiled and the European board was encoded with the model. This meant assigning `n=7` and `noBoards=36`, breaking symmetry along the vertical axis of symmetry and of independent moves, identifying dead ends earlier, and branching on the matrices of Boards in ascending order using the conflict heuristic. This is shown in full below in Figure 26.

```
language ESSENCE' 1.0

letting n be 7
letting noBoards be 36
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find RB1    : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find transitionPeg : matrix indexed by [int(2..noBoards),int(1,2)] of nDom

branching on [ Boards[i,..,..] | i : noBoardsDom ]

heuristic conflict

such that

  Boards[1,2,4] = 1,

  Boards[noBoards,3,4] = 0,

  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  forAll i : int(1..noBoards-1) .
    forAll j : nDom .
      forAll k : nDom .
        ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
          ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
            ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

          ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
            ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

          ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
            ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

          ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
            ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),


  forAll i : noBoardsDom . Boards[ i , 1 , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , 2 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , 1 ] = 0,

  forAll i : noBoardsDom . Boards[ i , 1 ,n-1] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , n ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , n ] = 0,
```

```
forAll i : noBoardsDom . Boards[ i ,n-1, 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 2 ] = 0,

forAll i : noBoardsDom . Boards[ i ,n-1, n ] = 0,
forAll i : noBoardsDom . Boards[ i , n ,n-1] = 0,
forAll i : noBoardsDom . Boards[ i , n , n ] = 0,

forAll i : int(1..noBoards-1) .
  ( exists j : int(3..5) .
    ( (Boards[i,j,1] + Boards[i,j,2] = 0) \/
      (Boards[i,j,6] + Boards[i,j,7] = 0) ) ) \/

  ( exists j : int(2..6) .
    ( (Boards[i,j,2] + Boards[i,j,3] = 0) \/
      (Boards[i,j,5] + Boards[i,j,6] = 0) ) ) \/

  ( exists j : nDom .
    exists k : int(3..4) .
     (Boards[i,j,k] + Boards[i,j,k+1] = 0) ) \/

  ( exists k : int(3..5) .
    ( (Boards[i,1,k] + Boards[i,2,k] = 0) \/
      (Boards[i,6,k] + Boards[i,7,k] = 0) ) ) \/

  ( exists k : int(2..6) .
    ( (Boards[i,2,k] + Boards[i,3,k] = 0) \/
      (Boards[i,5,k] + Boards[i,6,k] = 0) ) ) \/

  ( exists k : nDom .
    exists j : int(3..4) .
      (Boards[i,j,k] + Boards[i,j+1,k] = 0) ),

forAll i : noBoardsDom .
  forAll j : nDom .
    forAll k : nDom .
      RB1[i,j,k] = Boards[i,j,n+1-k],

forAll i : int(1..noBoards-1) .
  (forAll j : nDom . forAll k : nDom . Boards[i,j,k] = RB1[i,j,k] ) ->
    flatten( RB1[i+1,..,..]) <=lex flatten(Boards[i+1,..,..]),

forAll i : int(2..noBoards) .
  forAll j : nDom .
    forAll k : nDom .
      ( Boards[i-1,j,k] - Boards[i,j,k] = 1 ) ->
        (transitionPeg[i,1] = j /\ transitionPeg[i,2] = k),

$ Break independent move symmetry
 forAll i : int(2..noBoards-2) .
   ( (sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+2,j,k] |) = 6 ) ->
     transitionPeg[i+1,..] <lex transitionPeg[i+2,..]
```

Figure 25

Unfortunately the model did not return a solution after being run for a number of hours. In order to work towards a solution the constraint that is applied to the final board state is removed, and the `noBoards` is reduced, so that the model searches for a sequence of legal moves from the initial state, of increasing lengths. The results are displayed in Figure 26.

Solve first n steps:	minion Nodes	minion Time	Savile Row Time
n = 30	2500	1.60	5.341
n = 31	8521	2.15	6.005
n = 32	28555	3.30	5.554
n = 33	75338095	5905.15	6.373
n = 34	n/a	n/a	n/a

Figure 26

The model easily finds solutions of up to 32 steps in a matter of seconds. A sequence of 33 board states is more difficult, with the model returning a solution in ~98 minutes. When `noBoards` was set to 34 the model was run for 12 hours without returning a solution.

Although much improvements to the model have been made by breaking both symmetries mentioned, and identifying dead ends, the enormity of the search space, and the difficulty in reaching a position where the final moves fit the board correctly, proved too much for the model in this case.

9. Conclusion and further work

The goal of this project at the outset was to solve the European variation of peg solitaire using a constraint programming model. A model that represents the state of the board was devised, and through testing on smaller board arrangements, refined by breaking symmetries and identifying dead ends. Although this original goal was not accomplished, the progress that has been made in the model that is evident from testing on the square board is very satisfactory.

One of the major positive results from this project has been the scope of the model. As opposed to a model that is only usable for a single variation of peg solitaire, the devised model can be adapted to search for a solution to any peg solitaire game that is inscribed in a square board, and allows horizontal and vertical peg movement.

The further development of this project could be taken in two directions: the original goal of finding a solution to European peg solitaire can be pursued, which would involve working on the specificities of this board. More in depth analysis of the game could be performed, in order to find better heuristics to guide the search; e.g. identifying dead ends much earlier than covered in this dissertation.

An alternate direction to continue work on this project would be to focus more on the generality of the model design, and how it can be parameterised more properly for use as an all purpose, peg solitaire solver. Currently the framework is in place, but each time a new board is considered the code within the model must be changed to fit, e.g. constraints are used to identify the unused corner squares, as opposed to parameters. Work could be continued by consolidating the Essence' model to just the rules and heuristics involved in solving the game, and that takes the size of the board, number of states, and a list of corner squares to be excluded via a parameter file. Once the model is completely parameterised, general rules of gameplay could be analysed to help improve the model further.

10. References

[1] Jefferson C, Miguel A, Miguel I, Tarim A. Modelling and Solving English Peg Solitaire. *Computers and Operations Research* 33(10), pages 2935-2959, 2006.
[2] Bell G. Notes on solving and playing peg solitaire on a computer. *arXiv preprint arXiv:09033696*. 2009.
[3] Bell G. Solving triangular peg solitaire. *Journal of Integer Sequences*. 2008;11(2):3.
[4] Ravikvmar B. Peg-solitaire, string rewriting systems and finite automata. *Springer*. 1997;233-242.
[5] Dowd V. Playing solitaire with Turing. *BBC News*. 6/6/2014. http://www.bbc.co.uk/news/magazine-27701207 (accessed 19/8/2014).
[6] Berlekamp E, Conway J, Guy R. *Winning ways for your mathematical plays, Volume 4*. 2nd ed. Wellesly, MA: A.K. Peters Ltd; 2001.
[7] Beeler R, Paul Hoilman D. Peg solitaire on graphs. *Discrete Mathematics*. 2011; 311(20): 2198--2202.
[8] Ramaré O. A stronger model for peg solitaire, II. *arXiv preprint arXiv:08010679*. 2008.
[9] Beasley JD. The ins and outs of peg solitaire. Oxford: Oxford University Press; 1992.
[10] Moore C, Eppstein D. One-dimensional peg solitaire, and duotaire. *More games of no chance*. 2001; (42): 341--350.
[11] Nightingale P, Rendl A. Essence' tutorial. Savile Row 1.6

11. Appendices

11.1 Ethics pre-assessment form

UNIVERSITY OF ST ANDREWS

SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self -Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your preliminary ethics considerations.

PROJECT TYPE (please ✓)

☐ Staff
☐ Undergraduate
☑ Postgraduate
  ☑ MSc
  ☐ PhD

PROJECT TITLE

Solving European Peg Solitaire with Constraint Programming	
Name of researcher(s)	Niall Colfer
Name of supervisor (for student research)	Dr. Ian Miguel

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self-audit has been conducted (Please ✓)

☑ YES ☐ NO

Ethical Issues (Please ✓)

☑ There are **NO** ethical issues raised by this project
☐ There are ethical issues raised by this project

Signed Niall Colfer **Print Name** NIALL COLFER **Date** 7/5/14
(Student Researcher(s), if applicable)

Signed Miguel **Print Name** IAN MIGUEL **Date** 7/05/14
(Lead Researcher or Supervisor)

This form must be date stamped and held in the files of the School Ethics Committee. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

1. Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing? **YES NO**

For example:

Will you be surveying, observing or interviewing human subjects?

Will you be testing any systems or programs on human subjects?

Will you be collecting data from computers or networks used by human subjects?

If YES, *full ethics review may be required*

If NO, go to question 16, then sign this form and return to the School Ethics Committee Secretary.

Potential physical or psychological harm, discomfort or stress

2. Will your participants be exposed to any risks greater than those encountered in their normal working life? **YES NO**

If YES, *full ethics review required*

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g., walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g., ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback.

3. Do your experimental materials comprise software running on non-standard hardware? **YES NO**

If YES, *full ethics review required*

Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and PDAs is considered non-standard.

4. Will you offer incentives to your participants? **YES NO**

If YES, *full ethics review required*

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

Data Protection

5. Will any data collected from the participants not be stored in a secure and anonymous form? **YES NO**

If YES, *full ethics review required*

All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

6. Will you collect more data than are required for your immediate experimental hypotheses? **YES NO**

If YES, *full ethics review required*

Any collection of personal data should be adequate, relevant and not excessive in relation to the purposes for the collection.

Informed consent

7. Will participants participate without their explicit agreement to participate, or their explicit agreement that their data can be used in your project? **YES NO**

If YES, *full ethics review required*

If the results of the experiments are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant. Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

8. Will you withhold any information about your experiment or materials from your participants? **YES NO**

If YES, *full ethics review required*

Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

9. Are any of your participants under the age of 18? **YES NO**

If YES, *full ethics review required*

Working with human subjects under the age of 18 requires you to obtain an Enhanced Disclosure from Disclosure Scotland.

10. Do any of your participants have an impairment that may limit their understanding or communication? **YES NO**

If YES, *full ethics review required*

Additional consent is required for participants with impairments.

11. Are you or your supervisor in a position of authority or influence over any of the participants? **YES NO**

If YES, *full ethics review required*

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

12. Will participants participate without being informed that they can withdraw at any time? **YES NO**

If YES, *full ethics review required*

All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

13. Will participants participate without being informed of your contact details and those of your supervisor? **YES NO**

If YES, *full ethics review required*

All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

14. Will any participants not have the opportunity to discuss the experiment and answer questions at the end of your experimental session? **YES NO**

If YES, *full ethics review required*

You must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.

Conflicts of interest

15. Do any conflicts of interest arise? **YES NO**

If YES, *full ethics review required*

For example:

Might research objectivity be compromised by sponsorship?

Might any issues of intellectual property or roles in research be raised?

Funding

16. Is your research funded externally? **YES (NO)**

If YES, is the funder missing from the 'currently automatically approved' list on the UTREC website? **YES NO**

If YES, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

You (and, where appropriate, your supervisor) should now sign this form and return it to the School Ethics Committee Secretary.

Check whether you need to submit a full Ethics Application Form as well. If you have a supervisor, also check with them. If still in doubt, please contact the School Ethics Committee for advice.

Where appropriate, your experiments must use information sheets based on the examples provided on the Ethics website (http://www.cs.st-andrews.ac.uk/ethics), and these should be submitted with your final report.

11.2 Ethics supervisor checklist

PRELIMINARY ETHICS SELF-ASSESSMENT CHECKLIST FOR PROJECT SUPERVISORS

Please read this page carefully and then sign it together with the attached ethical self-assessment form before returning them to the ethics secretary.

If you have are unsure whether or not a full ethics application is required, please email ethics-cs@st-andrews.ac.uk.

A full ethics application **is not required** if:

1) The answer to question 1 on the self-assessment form is no;
2) The answer to question 1 on the self-assessment form is yes AND the answer to questions 2-15 is no.

A full ethics application **is required** if:

1) The answer to any of questions 2-15 on the self-assessment form is yes;
2) The project involves data from any social networking sites (Facebook etc.);
3) The project involves health data (e.g. working with NHS projects);
4) The project involves working with children (a Disclosure Scotland form is also required so this should be flagged up well in advance of starting the project).
5) The answer to question 1 is yes AND supervisor/ethics convenor thinks a full ethics form is necessary for some reason not covered by the above.

Please complete applicants' name	Niall Colfer

"I verify that as supervisor, in addition to reading through all of the student's completed self-assessment form, I have also checked that a full ethics form **should not** be completed."

Signature: Miguel Date: 02/05/14

Name: Dr. Ian Miguel

11.3 Arrow configuration model in full

```
language ESSENCE' 1.0

letting n be 5
letting noBoards be 14
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find RB     : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find transitionPeg : matrix indexed by [int(2..noBoards),int(1,2)] of nDom

branching on [ Boards[i,..,..] | i : noBoardsDom ]

heuristic conflict

such that

  $ Fix the intial peg
  Boards[1,4,3] = 1,

  $ Specify empty slot in final board
  Boards[noBoards,3,3] = 0,

  $ Calculate the number of pegs at each state
  $ Modelled from 'goal' state to 'initial' state
  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  $ Exactly three squares are different in each pair of subsequent boards
  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  $ If a peg is removed in a transition, then two adjacent pegs are added in the
same transition (four clauses -> one for each direction)
  forAll i : int(1..noBoards-1) .
      forAll j : nDom .
        forAll k : nDom .
          ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
            ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
              ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

            ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
              ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

            ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
              ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

            ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
              ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),

  $ Set board configuration, by fixing entries to zero
  forAll i : noBoardsDom . Boards[ i , 1 , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , 2 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , 1 ] = 0,

  forAll i : noBoardsDom . Boards[ i , 1 , n ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , n ] = 0,
```

```
  forAll i : noBoardsDom . Boards[ i , 1 ,n-1] = 0,

  forAll i : noBoardsDom . Boards[ i , n , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i ,n-1, 1 ] = 0,

  forAll i : noBoardsDom . Boards[ i , n , n ] = 0,
  forAll i : noBoardsDom . Boards[ i ,n-1, n ] = 0,

  $
  $ Symmetry breaking:
  $

  $ RB is a complete reversal of Boards
  forAll i : noBoardsDom .
    forAll j : nDom .
      forAll k : nDom .
        RB[i,j,k] = Boards[i,j,n+1-k],

  $ Break reflectional symmetry
  forAll i : int(1..noBoards-1) .
    (forAll j : nDom . forAll k : nDom . Boards[i,j,k] =    RB[i,j,k] ) ->
flatten(Boards[i+1,..,..]) <=lex flatten(   RB[i+1,..,..]),

  $ Define the location of the transition peg
  forAll i : int(2..noBoards) .
    forAll j : nDom .
      forAll k : nDom .
        ( Boards[i-1,j,k] - Boards[i,j,k] = 1 ) -> (transitionPeg[i,1] = j /\
transitionPeg[i,2] = k),

  $ Break independent move symmetry
   forAll i : int(2..noBoards-2) .
     ( (sum j : nDom . sum k : nDom . | Boards[i+2,j,k] - Boards[i,j,k] |) = 6 ) ->
transitionPeg[i+1,..] <lex transitionPeg[i+2,..]
```

11.4 Square configuration model in full

```
language ESSENCE' 1.0

letting n be 5
letting noBoards be 24
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find RB1    : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find transitionPeg : matrix indexed by [int(2..noBoards),int(1,2)] of nDom

branching on [ Boards[i,..,..] | i : noBoardsDom ]

heuristic conflict

such that

  $ Fix the intial peg
  Boards[1,1,3] = 1,

  $ Specify empty slot in final board
  Boards[noBoards,2,3] = 0,
```

```
  $ Calculate the number of pegs at each state
  $ Modelled from 'goal' state to 'initial' state
  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  $ Exactly three squares are different in each pair of subsequent boards
  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  $ If a peg is removed in a transition, then two adjacent pegs are added in the
same transition (four clauses -> one for each direction)
  forAll i : int(1..noBoards-1) .
      forAll j : nDom .
        forAll k : nDom .
          ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
            ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
              ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

            ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
              ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

            ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
              ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

            ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
              ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),

  $ Dead end spotter
  $ Every board must have two 0s side-by-side
  forAll i : int(1..noBoards-1) .
    exists j : int(1..n-1) .
      exists k : int(1..n-1) .
        ( (Boards[i,j,k] + Boards[i,j,k+1] = 0) \/
          (Boards[i,j,k] + Boards[i,j+1,k] = 0) ),

  $
  $ Symmetry breaking
  $

  $ RB1 is a reflection of Boards
  forAll i : noBoardsDom .
    forAll j : nDom .
      forAll k : nDom .
        RB1[i,j,k] = Boards[i,j,n+1-k],

  $ Break reflectional symmetry
  forAll i : int(1..noBoards-1) .
    (forAll j : nDom . forAll k : nDom . Boards[i,j,k] =  RB1[i,j,k] ) ->
      flatten( RB1[i+1,..,..]) <=lex flatten(Boards[i+1,..,..]),

  $ Define the location of the transition peg
  forAll i : int(2..noBoards) .
    forAll j : nDom .
      forAll k : nDom .
        ( Boards[i-1,j,k] - Boards[i,j,k] = 1 ) ->
          (transitionPeg[i,1] = j /\ transitionPeg[i,2] = k),

  $ Break independent move symmetry
   forAll i : int(2..noBoards-2) .
     ( (sum j : nDom . sum k : nDom . | Boards[i+2,j,k] - Boards[i,j,k] |) = 6 ) ->
       transitionPeg[i+1,..] <lex transitionPeg[i+2,..]
```

11.5 European board model in full

```
language ESSENCE' 1.0

letting n be 7
letting noBoards be 34
letting binaryDom be domain int(0,1)
letting nDom be domain int(1..n)
letting noBoardsDom be domain int(1..noBoards)

find Boards : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find RB1    : matrix indexed by [noBoardsDom,nDom,nDom] of binaryDom

find transitionPeg : matrix indexed by [int(2..noBoards),int(1,2)] of nDom

branching on [ Boards[i,..,..] | i : noBoardsDom ]

heuristic conflict

such that

  $ Fix the intial peg
  Boards[1,2,4] = 1,

  $ Specify empty slot in final board
  $ Boards[noBoards,3,4] = 0,

  $ Calculate the number of pegs at each state
  $ Modelled from 'goal' state to 'initial' state
  forAll i : noBoardsDom . sum(flatten(Boards[i,..,..])) = i,

  $ Exactly three squares are different in each pair of subsequent boards
  forAll i : int(1..noBoards-1) .
    ( sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+1,j,k] | ) = 3,

  $ If a peg is removed in a transition, then two adjacent pegs are added in the
same transition (four clauses -> one for each direction)
  forAll i : int(1..noBoards-1) .
      forAll j : nDom .
        forAll k : nDom .
          ( Boards[i,j,k] - Boards[i+1,j,k] = 1 ) ->
            ( ( Boards[i+1,j+1, k ] - Boards[ i ,j+1, k ] = 1 ) /\
              ( Boards[i+1,j+2, k ] - Boards[ i ,j+2, k ] = 1 ) ) \/

            ( ( Boards[i+1,j-1, k ] - Boards[ i ,j-1, k ] = 1 ) /\
              ( Boards[i+1,j-2, k ] - Boards[ i ,j-2, k ] = 1 ) ) \/

            ( ( Boards[i+1, j ,k+1] - Boards[ i , j ,k+1] = 1 ) /\
              ( Boards[i+1, j ,k+2] - Boards[ i , j ,k+2] = 1 ) ) \/

            ( ( Boards[i+1, j ,k-1] - Boards[ i , j ,k-1] = 1 ) /\
              ( Boards[i+1, j ,k-2] - Boards[ i , j ,k-2] = 1 ) ),

  $ Fix corner triplets as empty throughout
  forAll i : noBoardsDom . Boards[ i , 1 , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , 2 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , 1 ] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 ,n-1] = 0,
  forAll i : noBoardsDom . Boards[ i , 1 , n ] = 0,
  forAll i : noBoardsDom . Boards[ i , 2 , n ] = 0,
```

```
forAll i : noBoardsDom . Boards[ i ,n-1, 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 1 ] = 0,
forAll i : noBoardsDom . Boards[ i , n , 2 ] = 0,
forAll i : noBoardsDom . Boards[ i ,n-1, n ] = 0,
forAll i : noBoardsDom . Boards[ i , n ,n-1] = 0,
forAll i : noBoardsDom . Boards[ i , n , n ] = 0,

$ Dead end spotter
$ Every board must have two 0s side-by-side
$ Problem specific coding so that empty corners don't pass the constraint
forAll i : int(1..noBoards-1) .
  ( exists j : int(3..5) .
    ( (Boards[i,j,1] + Boards[i,j,2] = 0) \/
      (Boards[i,j,6] + Boards[i,j,7] = 0) ) ) \/

  ( exists j : int(2..6) .
    ( (Boards[i,j,2] + Boards[i,j,3] = 0) \/
      (Boards[i,j,5] + Boards[i,j,6] = 0) ) ) \/

  ( exists j : nDom .
    exists k : int(3..4) .
      (Boards[i,j,k] + Boards[i,j,k+1] = 0) ) \/

  ( exists k : int(3..5) .
    ( (Boards[i,1,k] + Boards[i,2,k] = 0) \/
      (Boards[i,6,k] + Boards[i,7,k] = 0) ) ) \/

  ( exists k : int(2..6) .
    ( (Boards[i,2,k] + Boards[i,3,k] = 0) \/
      (Boards[i,5,k] + Boards[i,6,k] = 0) ) ) \/

  ( exists k : nDom .
    exists j : int(3..4) .
      (Boards[i,j,k] + Boards[i,j+1,k] = 0) ),

$
$ Symmetry breaking
$

$ RB1 is a reflection of Boards
forAll i : noBoardsDom .
  forAll j : nDom .
    forAll k : nDom .
      RB1[i,j,k] = Boards[i,j,n+1-k],

$ Break reflectional symmetry

forAll i : int(1..noBoards-1) .
  (forAll j : nDom . forAll k : nDom . Boards[i,j,k] = RB1[i,j,k] ) ->
    flatten( RB1[i+1,..,..]) <=lex flatten(Boards[i+1,..,..]),

$ Define the location of the transition peg
forAll i : int(2..noBoards) .
  forAll j : nDom .
    forAll k : nDom .
      ( Boards[i-1,j,k] - Boards[i,j,k] = 1 ) ->
        (transitionPeg[i,1] = j /\ transitionPeg[i,2] = k),

$ Break independent move symmetry
 forAll i : int(2..noBoards-2) .
   ( (sum j : nDom . sum k : nDom . | Boards[i,j,k] - Boards[i+2,j,k] |) = 6 ) ->
     transitionPeg[i+1,..] <lex transitionPeg[i+2,..]
```