PEG SOLITAIRE

Robin Cockett

November 19, 2019

The problem

Write a program to solve (English) peg solitaire problems.

Peg solitaire is a game in which one is given a board with pegs filling certain positions and the aim is, using a succession of jumps, to reach another goal board position. English peg solitaire is played on a cross shaped board (see below) with 31 pegs.

Program description

The program should have three phases: a setting up stage in which the problem is specified, a search phase, and a solution demonstration stage.

Setting up: You must be able to specify to the program:

- The start state either as a list of filled positions or as a list of empty positions. Check with the user that the start configuration is correct by showing him the board.
- The goal position should be obtained similarly and verified.

Search: For this phase the most important factor is to have an effective representation of the search problem which will minimize the necessary search. I recommend the use of several pagoda functions (to detect states from which the solution cannot be reached) together with independence checking.

With *all* the above tricks (pagodas and independence) you should be able to solve the full problem in less than ten seconds.

Recall that the efficiency with which you represent the problem will have a linear effect on the time it takes to solve the problem. The failure to detect non-solvable states (using pagoda functions) and independent moves will have an exponential effect! If you use more than one pagoda function you should check that they do detect different failure modes.

Demonstration: You should allow the user to step through the solution just found by showing the board positions in sequence, and prompting the user at each stage.

Representing the board

A very simple representation of the board is to number each square:

	a	b	c	d	e	f	g
1			2	3	4		
2			12	13	14		
3	20	21	22	23	24	25	26
4	30	31	32	33	34	35	36
5	40	41	42	43	44	45	46
6			52	53	54		
7			62	63	64		

Then a board can be represented as a list of numbers (which for efficiency one may want to keep in order). Performing a jump then is achieved by removing the start point of the jump, the jumped position, checking that the end position of the jump is not occupied, and then adding the end position into the list. Furthermore, jumps can easily be determined by adding 1 to move across the board and 10 to move vertically.

Sample problems

Here are some sample problems. Your aim is to solve as many of these as possible (you should definitely be able to solve (1), (2), and the halfdead state of (3). The start positions are indicated by "x" and the goal position by an "o."

1. Crossbow:

	a	b	c	d	e	f	g
1				o			
2							
3							
4		x	x		x	x	
5		x	x	x	x	x	
6				x			
7							

2. Longbow:

	a	b	c	d	e	f	g
1				o			
2							
3	x						x
4	x	x		x		x	x
5		x		x		x	
6			x	x	x		
7				x			

3. Not quite dead:

	a	b	c	d	e	f	g
1			x	x	x		
2			x		x		
3	x	x	x	x	x	x	x
4	x		x	o		x	x
5	x	x	x	x	x	x	x
6			x		x		
7			x		x		

Not quite dead has the following solution

$[e2-e4, g3-e3, g5-g3, d3-f3, g3-e3, b3-d3, c1-c3, e1-c1, c4-c2, c1-c3]$

at which stage one has reached the following half dead position:

	a	b	c	d	e	f	g
1							
2							
3	x		x	x	x		
4	x			o	x	x	
5	x	x	x	x	x	x	
6			x		x		
7			x		x		

The following moves reduce it down to the following position

$[c6-c4, a5-c5, a3-a5, d5-b5, a5-c5, c4-c6, c7-c5, f5-d5, e7-e5]$

	a	b	c	d	e	f	g
1							
2							
3			x	x	x		
4				o	x	x	
5			x	x	x		
6							
7							

From which a finish can be found easily.

Try solving these steps in the not quite dead problem ..

4. Now try removing any one peg with the goal of ending with that one hole filled (the top middle peg is one of the easier options and is helped by having a heavily weighted pagoda function).

   Can you show how to get from the middle peg removed to the not quite dead position?

Pagoda functions

The idea of a pagoda function is to give a weighting of a board position which is such that any jump does not increase the weight of the board (and, indeed, may decrease the weight). This means that if at any stage the weight of the goal position exceeds the weight of a current search position it can be abandoned as the goal is not reachable.

Using pagoda functions dramatically reduces what has to be searched. To solve the full game in any reasonable amount of time it is necessary to use a number of pagoda functions (in various orientations).

Here are some examples of pagoda functions. you should choose them so as to be as sparse as possible yet giving as much weight as possible to the goal position(s).

The basic centre weighting is:

	a	b	c	d	e	f	g
1			0	0	0		
2			0	1	0		
3	0	0	0	0	0	0	0
4	0	1	0	1	0	1	0
5	0	0	0	0	0	0	0
6			0	1	0		
7			0	0	0		

A stronger asymmetric center weighting (for a center game the two symmetries of this would be most effective):

	a	b	c	d	e	f	g
1			0	0	0		
2			0	1	0		
3	-1	1	0	1	0	1	-1
4	0	2	0	2	0	2	0
5	-1	1	0	1	0	1	-1
6			0	1	0		
7			0	0	0		

A last couple which are worth remembering:

	a	b	c	d	e	f	g
1			-1	0	-1		
2			1	0	1		
3	0	0	0	0	0	0	0
4	0	1	1	0	1	0	1
5	0	0	0	0	0	0	0
6			1	0	1		
7			-1	0	-1		

A strong weighting to the top:

	a	b	c	d	e	f	g
1			0	21	0		
2			0	13	0		
3	-8	8	0	8	0	8	-8
4	5	5	0	5	0	5	5
5	-3	3	0	3	0	3	-3
6			0	2	3		
7			0	1	0		

Independence

Two jumps are independent, that is can be done in any order, if their footprints are disjoint. The footprint of a move is the start position, the jumped position, and the landing position. If moves are independent one want to make sure they are performed only in a preferred order in the search. If you allow both orders in the search it can double the search time for each independent pair (making the running time unacceptable).

To check independence you need to keep a history of the jumps performed. When you do the next jump you check to see whether the previous jumps are independent. If so the current jump must be in the "right" order. If a non-independent jump is detected then, of course it is impossible to change the order ...