Empirical Analysis

Using recursive backtracking on Peg Solitaire

Lindiwe Mncwabe - 1118055

Clifford Ralikhwatha - 0412610

Thomas Johannsen - 721988

2017

Contents

List of Figures

1 Introduction

1.1 Overview

This project is a joint effort by the above authors to perform empirical analysis on our algorithm that solves various peg solitaire boards. The algorithm reads in the initial game board and the goal state and then proceeds to solve the board by means of a recursive backtracking algorithm. Due to complexities discussed later, we implemented a hash table to stored previously checked board states to speed up the process.

1.2 Aims

The aim was to write an algorithm using Java, that would solve the classic peg solitaire board in a reasonable time. Once the basic functionally was working the next goal was to solve additional boards, of various shapes and sizes, to compare how the algorithm performs.

2 Peg Solitaire

Peg solitaire is a very old single player game, that is played on a board of 33 holes in a cross formation. The aim is to jump pegs/marbles/stones over each other, removing one in the process, until just a single peg is left in the centre of the board. Different variations exist, some that use more than the standard 33 pegs, others that use less.
Figure 1 shows an example of a valid move.

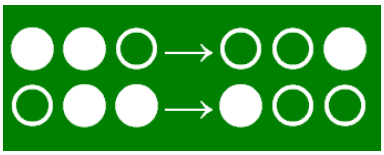

Figure 1: Example of a move

2.1 Theory

The Game

It soon becomes clear that the game is mathematically very complex. The first move has 4 possibilities, each of those 4 moves has its own 4 possibilities. From this we can extrapolate that the number of possibilities grows at the rate(O^4), in actual fact its less than that, due to not all 4 possibilities always being valid moves and many of them resulting in the same board state. None the less the problem grows at an exponential rate.
The exact number of different sequences for the game is $5 * 10^{20}$ [1], $4 * 10^{16}$ of these are solutions. It turns out that all of those solutions are rotations and symmetries of one unique sequence. All solutions use exactly 31 moves, by the design of the game a move removes a peg, so no solution is better or worse than any other.

Recursion

The first thing you always learn about recursion is that you shouldn't use it unless you know what you are doing. The idea is that the recursive functions at some point calls itself, this can very easily result in an infinite loop. The reason we can use it for this problem is that we can control the recursion, at each step we remove a peg, this means that after 31 recursions we only have 1 peg left, at which point the game is over. We can never get to a depth of greater than the original number of pegs we started with.

Backtracking

As the name suggest, backtracking is a method that makes an initial guess and then tests if that guess was correct, if not we backtrack along the sequence traversed so far and try a different move. Applied to peg solitaire; once we reach a board state that isn't the final solution, but also has no possible moves left we know that we have made a mistake at some point, so we undo our last move. This is done by adding the peg that was removed back onto the board and then trying a different jump instead. Once all options have been exhausted we backtrack a step further.

2.2 Design

To begin we need to represent the board somehow. There are many way to do this, but we decided to store it as a 7x7 matrix, where a 0 represents out of bounds, 2 is a peg and 1 is a hole. It turns out that 1 and 2 are interchangeable because the reverse of the solution would be building the board from just 1 peg to a full board with a single hole in the middle. Well that is the exact same problem with the definition of 1 and 2 swapped. We set 1 and 2 as defined so that the final board state has a hash value of 0. A sequence is considered a dead end, if we cannot perform any further moves and the board is not solved. Any solution we find is accepted, there is no need to find further solutions, as every solution has the same length.

	0	1	2	3	4	5	6
0	0	0	1	1	1	0	0
1	0	0	1	1	1	0	0
2	1	1	1	1	1	1	1
3	1	1	1	2	1	1	1
4	1	1	1	1	1	1	1
5	0	0	1	1	1	0	0
6	0	0	1	1	1	0	0

Figure 2: Matrix representation of the solved board

The algorithm runs through the matrix trying on every element to perform a jump, in the order left, up, down, right and then storing that move in a separate array. The recursion is then called and the algorithm scans for the next possible jump. Once it has been determined that the sequence is a dead end, the board state is saved to the hash table and we backtrack one step to a shallower recursion level. Here we try to jump from the same position, but into a different direction, if possible, otherwise we scan for the next element on which we can perform a jump/.

Proof of Correctness

The algorithm is a depth first recursive algorithm, with backtracking. To speed up the process we prune dead branches. By the nature of it being depth first it will scan every possible move. This guarantees that we find a solution, if the board is solvable. We can be sure that the pruning doesn't skip any untested moves, as it only stored boards that have been exhausted. If we look at figure 3, we search to the left of the root node, till we can't any more. We get to (1), this isn't the solution, so we save the current board state to the hash table and backtrack and then continue to (2). This is also not the solution, so we save the hash and backtrack, repeating the procedure till (3) has been backtracked and hashed. When we get to (4) we see that it has the same hash value as (3), this means that it is a rotation of a dead end board, so we can completely skip this branch and backtrack a step.

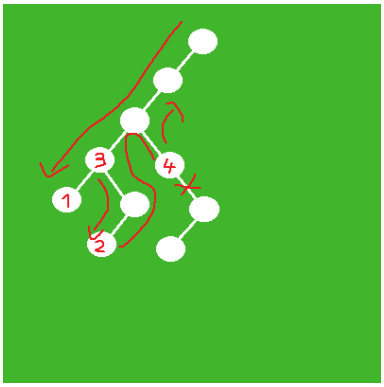

Figure 3: Depth First Search

Hash Table

The idea behind the hash table was to store boards that have already been checked, as in many cases doing jump A and then jump B has the same result as B then A. If we know that AB doesn't yield a result, there is no need to run through the branch of BA. Due to the high level of symmetry in the board, we decided that not only should a branch be pruned if it has a dead end board state, but also if the board state is a rotation of a dead end state.

Design 1

The first idea for the hash table was that every element of the matrix has a unique value assigned to it. We include all 49 element of the matrix, in case we want to test a board that uses a different shape from the standard cross. Out of bounds elements and elements with no pegs are considered 0 and a peg is considered a 1. This results in a 49 bit binary hash value. We considered two possible ways to compare the hash values of rotated board states. First rotating the matrix by multiplying by a rotation matrix and then hashing it, or identifying the pattern in the hash value, as all elements are rotated to a different element we rearrange the hash value. Both of these methods seemed a bit cumbersome so we came up with design 2 instead.

Design 2

Instead of using 49 unique values we mapped and colour coded the symmetries of the board. Each element has 4 or 8 equivalent positions, we sum these positions and multiply each by a different power of 10, to obtain a value that is unique to that board state and its congruent siblings.
Figure 3 shows the design of the hash map we ended up using. The central piece is the only one with no symmetries, so we store it as a boolean, by making the hash positive or negative depending on whether it has a peg or not.

	0	1	2	3	4	5	6
0	0	0	1	1	1	0	0
1	0	0	1	1	1	0	0
2	1	1	1	1	1	1	1
3	1	1	1	2	1	1	1
4	1	1	1	1	1	1	1
5	0	0	1	1	1	0	0
6	0	0	1	1	1	0	0

	100000000
	10000000
	1000000
	100000
	10000
	1000
	100
	10
	1
	-1

Figure 4: Colour coded hash map

2.3 Experimentation

The first draft of the algorithm didn't make use of the hash table. We read in the start board and starting performing jumps and backtracking and let that run for a while. It found the 29th jump after trying 10 000 jumps, and the 30th after 90 000 but 30 hours and over 100 billion jumps later, it hadn't found the last move to solve the problem.

It was clear that this was far too slow, even worse we had no way of predicting how long it would take to find the final move to solve the problem. It could at any stage be the next jump, or it could still be hours away. Considering how many possibilities there are in total we thought it was unlikely that we would find the solution any time soon. We should instead try optimise the search, find a way to make it smarter or faster.

First we found that a minor change to the order of the jumps can make a difference. Initially we had: right, left, up, down, but it turns out that if the first 2 jumps are opposite directions it takes a lot longer than if they are 90° rotations. Which direction is chosen as first or second... makes very little difference due to the symmetry of the game.

We wanted to half the search depth, by trying to solve the problem from both sides, 1 algorithm that starts from the beginning and another that starts from the solution and they work towards each other and meet at the middle. We weren't quite sure how to implement this, so we started doing some research and came across the work of Masashi Kiyomi and Tomomi Matsui [2]. They use three different optimisation methods, the first is to set up the problem as a integer problem and solve the feasibility of that at each step, as we don't know much about integer problem solving we skipped this.

The next two points they make are to use a hash table to avoid repetition and solving forwards as well as backwards. The last two ideas were similar to our ideas, so we decided to spend some time thinking how we would implement them.

We decided not the split the problem in half, and just solve it as is, as we couldn't come up with a good idea on how to store each board state, considering that if a later recursion of the one meets an earlier recursion of the other, we would need to store the moves performed getting to that board. This means storing not only the hash value of half depth board states, but also the moves to get to them.

2.4 Results

The hash table showed incredible improvements, so much so that we decided we didn't need pursue the integer problem method.
At first we were storing hash values randomly and repeatedly in the hash table, but still cut the execution of the problem to 4.6 seconds. Once we started trying bigger boards (40+ pegs), the run time sky rocketed, because the hash table grew with every jump tried. Very soon the lookup in the massive table resulted in the coded being slower than before. We then opted to store the hash values in a red-black tree, so that they remain sorted and unique, for faster looking up. This cut the time to 200ms and reduced the number of hash values by about 30%, while improving look up from linear to log(n) time.
As we will show, there are three important details to be considered for a run time execution of a board.

- The number of starting pegs
- The shape in which the pegs are set up
- The number of out of bounds fields

Unfortunately we can't directly compare all of these, as not all designs are solvable. This means that an unsolvable board with the same amount of pegs as a solvable board will result in many many more jumps, because the algorithm only ends once every branch has been exhausted.
We designed some different shapes, with different number of starting pegs, ran them, and plotted the outcome.

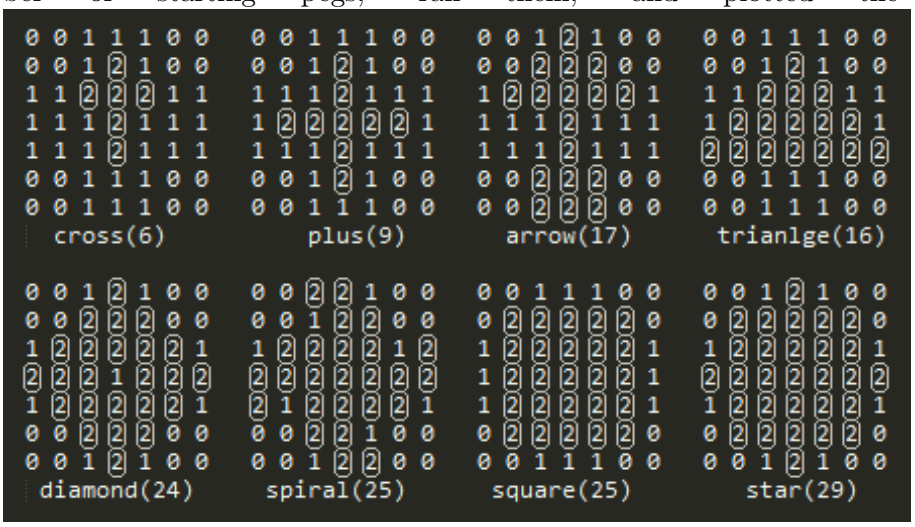

Figure 5: Different Starting Boards

The hash table made no difference at all for small boards, cross and plus cross, on the number of jumps or the run time.
For all bigger boards it's not even worth running without the hash table, they takes minutes of hours to solve.

Figure 6 shows the number of pegs plotted against the number of jumps and the run time till the solution is found or all moves have been tried.

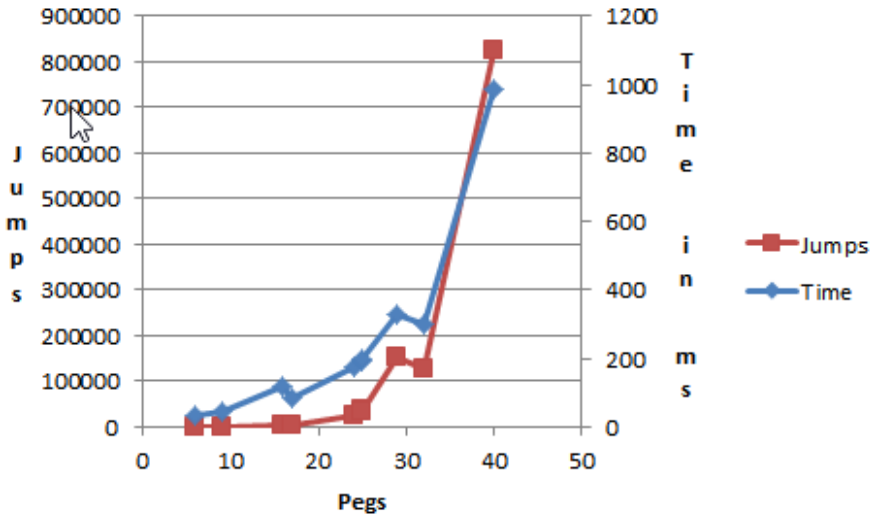

Figure 6: Graph of results

2.4.1 Interpretation

The first thing that should be noted is that the 40 peg board is the only one that wasn't solved, in the graph. Therefore we would expect it to take longer, as all moves were tried. However when we forced a similar situation on the smaller boards, by setting an unreachable goal for each. We found that they had very similar results to their original ones, barely noticeable on the graph. Which is why we decided included 40 in the graph, chances are if a board of 40 were solvable it wouldn't be much faster then the unsolvable one. The graphs takes the form of en exponential curve, which we would expect, as added a peg not only increases the length of the sequence, it also adds more possible moves to consider at each level. Interestingly we notice two points that don't lie on the curve, those are at 29/32 and 16/17. We can't be sure if the first value is high or the second value is low. If we look at figure 5, we see that 16 pegs is the triangle, which has 12 possible first moves, 17 pegs is the arrow, which only has 8 starting moves. The arrow also has 2 groups of pegs joined by a single line. From the start and due to the shape, the triangle has more possible moves than the arrow at every step.
If we look at the next disturbance, we have 29 pegs, the star, and 32, the classic board. The star, even though it uses less pegs, has a bigger playable board. The standard has 16 elements out of bounds and 33 playable fields, while the star has 12 out of bounds and 37 playable elements. This results in the star being the more complex puzzle, as at any given stage there are more possible jumps to consider, due to the extra playable fields. Also from the start there are 4 possible jump for the classic, while the star has 8 jumps.

For 25 pegs we found two separate solvable boards, the spiral and the square. Their run time is almost identical, as the jumps to completion differs by about 200 (35540 and 35332). This surprised us, as the square has 12 out of bounds compared to the 16 of the spiral, so we expected the square to take longer. Although if we count the possible starting moves the square has 12 to the 16 of the spiral. The two differences counteract each other and almost perfectly cancel out in this case
. The trend is clear though that as the number of pegs increase the solution takes exponentially longer to find, with the shape causing some variance from the trend.

3 Conclusion

References

[1] Durango Bill. Durango Bill's 33 Hole Peg Solitaire. http://www.durangobill.com/Peg33.html. [Online; Accessed October 2017].

[2] Masashi Kiyomi and Tomomi Matsui. Integer programming based algorithms for peg solitaire problems. In *International Conference on Computers and Games*, pages 229–240. Springer, 2000.