Backtracking Algorithm Applied to English Peg Solitaire

Author: Ryan Cook
Student Number: 691153
Date: 20/10/2014
Group members:

Brendan Bell	598468
Candice Swarts	607087

Introduction

Abstract

Through the course of this report, myself and 2 fellow computer science students at the University of Witwatersrand attempted to gain a deeper insight into certain aspects of the backtracking algorithm. This was done by firstly coding and theoretically analyzing the algorithm's complexity and finally running a series of experiments to test the theory.

The algorithm, along with certain program specifics is appended in this document.

Theory

To get an expression for the time complexity of the solve() algorithm, we will consider its main goal - to reduce a board consisting of p pegs, ($p > 1$, $p \in \mathbb{R}$) down to one final peg.

To begin, consider the move:

$$1\ 1\ 0 \rightarrow 0\ 0\ 1$$

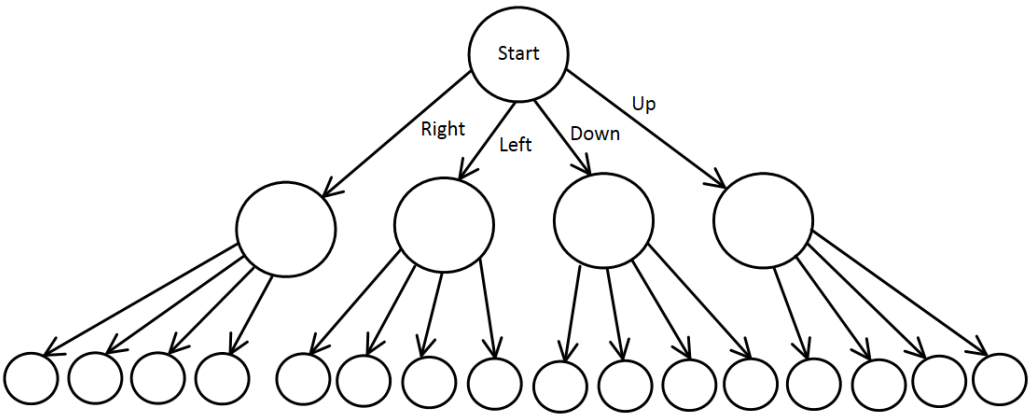
This move converts 2 pegs into one, i.e.: every valid move eliminates one peg from the board.

Now we assume that a board of size nxn has $n^2 - 1$ pegs - this is a worst case for our algorithm.

This means, that after $(n^2 - 1) - 1 = n^2 - 2$ pegs have been eliminated, our board will be left with one peg and we are done. This will be the height of our algorithm's decision tree.

We now need to consider how many possible ways there are to make these $n^2 - 2$ moves to complete our board. To do this, we draw a decision tree as follows:

Decision tree implemented by the solve() algorithm



From the root node, we begin our algorithm on some board, b. We now find the open peg in the board and perform a move into this open peg through one of four possible moves (Up, Down, Left or Right).
Once we have performed this move, we run a new instance of solve() on a new board, a where a ≠ b and a has one less peg than b.

There are 2 key points of insight that can be gained from this tree:

1. For every step down the tree (from parent to root node), one peg is removed from the board.
2. For every board, i, there are 4 possible boards that can be produced from i.

From point 1, we are able to determine the height of the decision tree. We know that for a board of p pegs, the decision tree will have a height p-1.

From point 2, we are able to determine the growth rate of our algorithm (assuming that our algorithm traverses the tree from root to node for each possible outcome).

We will now determine the growth rate of the decision tree and finally try attempt to relate this to the height of the tree to determine the number of iterations that our algorithm will run in the worst case.

Table showing the growth rate of decision tree

Move Index(i)	Possibilities
0	4
1	16
2	64
…	…
i	$4^i = 2^{2i}$

From this table we can see that for every step from parent to child node, the number of possible boards is increased by 4^2. We were able to calculate the general formula for the growth rate, g, as a function of the move index, i, and got a value of:

$$g(i) = 4^i = 2^{2i}$$

Now as discussed in point one, the maximum height of our tree is related to the number of pegs decremented by one, i.e. p-1. We now substitute this value into our growth rate function to get:

$$g(p) = 2^{2p-2}$$

Hence, the growth rate of our algorithm can be no worse than 2^{2p-2} and our algorithm can be said to be of $O(2^p)$ which is an exponential growth rate. This implies that our problem is of NP complexity

Goals

Our primary aim is to run a series of experiments through which we can either verify or disprove our theory regarding the complexity of the backtracking algorithm depicted in solve() (see appendix). Furthermore, we aim to develop ideas of how to reduce the complexity of this algorithm

Method

Apparatus

The apparatus used in our experiment are as follows:

Algorithms:
- Backtracking algorithm

Theoretical analysis:
- Analysis of backtracking algorithm

Program:
- Implementation of backtracking algorithm

Input Data:
- Set of peg solitaire inputs

Computer:
- Hardware:
  - Core i5-4460 3.2 Ghz
  - 8 Gb DDR3 1666 RAM
- Software:
  - Windows 7 64 bit
  - Netbeans IDE 8.0.1
  - JRE 1.8.0_25
  - JDK 1.0.0_20

Experiments

In the case of our solve() algorithm, there were two ways to vary the amount of pegs on a board:

1. Use the same size board with less pegs on it
2. Use a varying board size with maximum amount of pegs

We ran an experiment for each of these cases.

Experiment 1 - Testing the time taken to solve an increasingly solved peg solitaire board.

For this experiment we ran the solve() algorithm on a given board. Once the board was solved, we stored the steps taken to solve the board in an arraylist which we will refer to as moves. Finally, using the initial board we took the following steps:

Run the solving algorithm
Record the time taken to solve()
Perform the first move in moves on the original board
Run the solving algorithm
Repeat till the algorithm ran with just 2 pegs (best case)

Since many steps were taken to solve the initial board, we decided to run the test a predetermined amount of times (a factor of the amount of pegs on the board).

Experiment 2 - Testing the time taken to solve a board of increasing size.

For this experiment, we tested the peg solitaire solving algorithm on peg solitaire boards of increasing size.

NB: Due to our algorithm's large growth rate, we limited size of input boards up to a size of at most 8x8.

Experiment 2 was carried out as follows:

solve() was run on a 3x3 peg solitaire board (containing 8 pegs)
Time taken to solve() was recorded
Input board size was increased to (n+1)x(n+1)
solve() was run on new board
Time was recorded
Above steps were repeated for board sizes 3x3 to 8x8

Data

Experiment 1 - Time taken for varying peg count and constant board size

Input data used for this experiment:

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

This board was the first set of input data used. As this board was incrementally solved (correctly), a new test was run on the child board. Here is a list of the moves that were performed on this board in order to get a solvable child board:

Table showing moves performed to get solvable child board

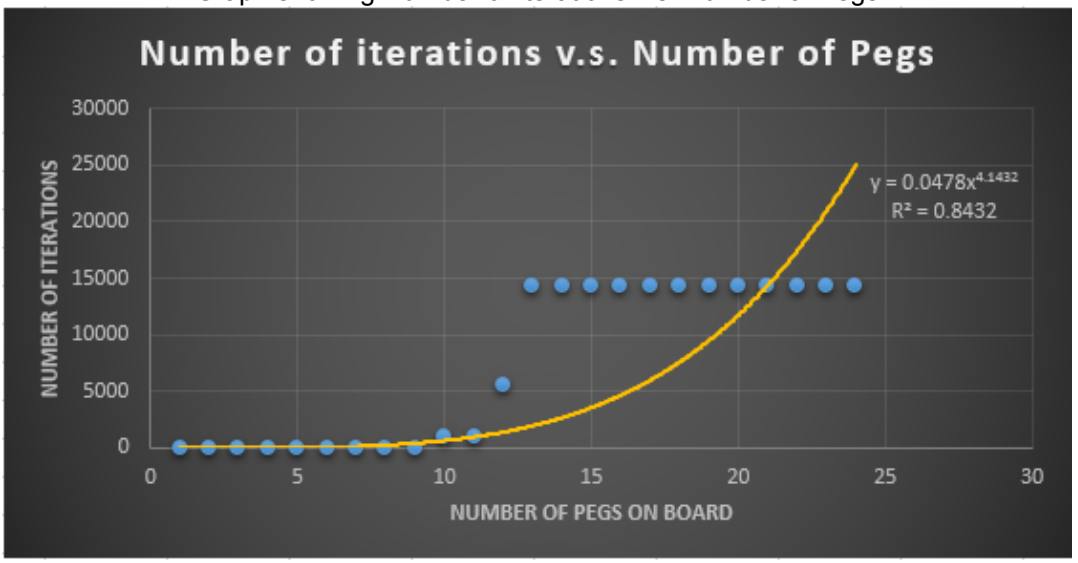
Move Index	Direction	Starting Position
0	EAST	(1,2)
1	NORTH	(1,0)
2	EAST	(0,2)
3	NORTH	(0,0)
4	SOUTH	(0,3)
5	SOUTH	(1,4)
6	WEST	(2,2)
7	NORTH	(0,1)
8	SOUTH	(0,4)
9	NORTH	(2,0)
10	SOUTH	(2,3)
11	WEST	(3,4)
12	WEST	(4,0)
13	NORTH	(2,0)
14	WEST	(4,1)
15	NORTH	(2,1)
16	WEST	(3,3)
17	SOUTH	(1,4)
18	EAST	(0,2)
19	SOUTH	(4,3)
20	EAST	(2,2)
21	NORTH	(4,1)
22	SOUTH	(4,4)

Note that EAST ⇒ UP, WEST ⇒ DOWN, NORTH ⇒ RIGHT, SOUTH ⇒ LEFT and that the positions go from 0 → 4 for both x and y (x goes from top left position downwards).

Table showing recorded values of the time taken and number of iterations when running solve() on the above input

Number of Pegs	Time taken (ms)	Number of iterations
24	83	14287
23	77	14286
22	98	14285
21	85	14284
20	105	14283
19	97	14282
18	82	14281
17	85	14280
16	90	14279
15	96	14278
14	66	14277
13	99	14276
12	51	5519
11	55	964
10	54	963
9	42	42
8	43	41
7	8	9
6	7	8
5	8	7
4	3	4
3	2	3
2	2	2
1	0	1

Graph showing Number of Iterations v.s. Number of Pegs

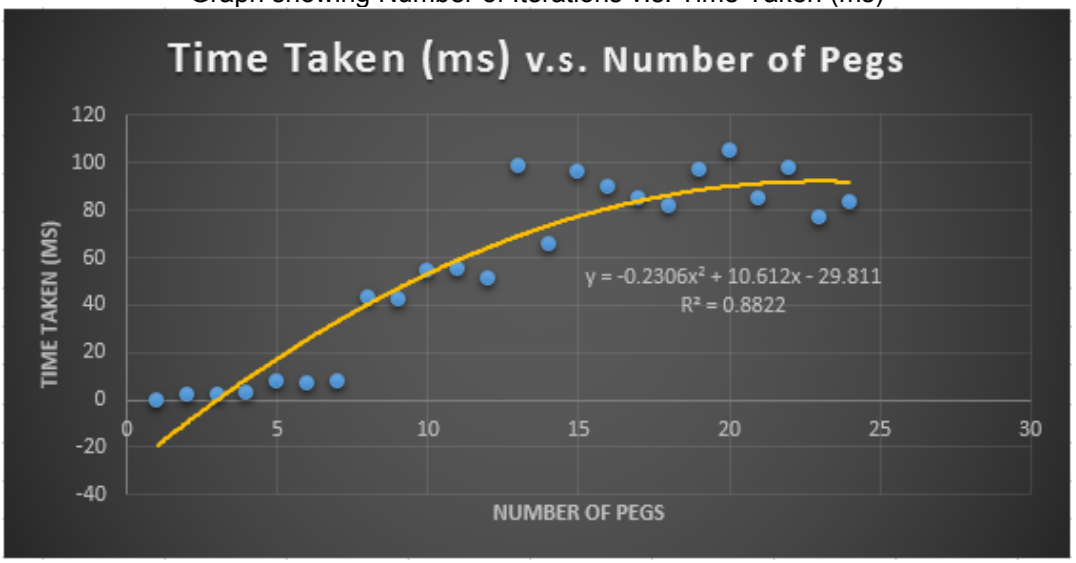


As seen from the graph, the number of iterations seems to limit at a value of approximately 14000. The reasoning for this will be discussed in the next section.

The value, R^2 = 0.8432, represents the linear regression for this graph and is a measure for how accurately the generated line of best fit “fits” the data points.

This value for linear regression tells us that the trend line does not accurately fit the data points on the graph.

Graph showing Number of Iterations v.s. Time Taken (ms)



For the above graph, I was unable to generate an exponential trend line that remotely fits the data set. The one you see above is polynomial of the second degree.
The linear regression is not of an acceptable value.

Clearly the results for this experiment seem to disagree with the theory of this experiment, but this will be discussed under the “Discussion” section.

Experiment 2 - Time taken for varying board size with maximum peg count

Through conducting the described experiment, our group found that a 3x3 board was unsolvable and that a 7x7 board ran the algorithm for over 20 hours. We recorded the following results for the 4x4, 5x5 and 6x6 boards and then added some extra cases to our experiment in the effort to obtain more data. I have added these extra cases as an extra experiment (experiment 2.1).

The input data used for this experiment is as follows:

Board 1

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

Board 2

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

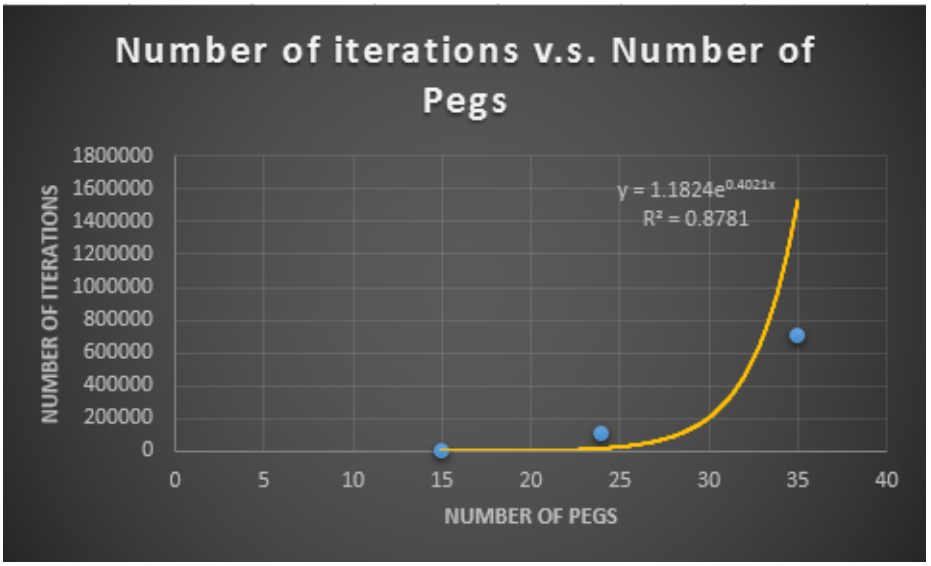
Board 3

1	1	1	1	1	1
1	1	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	1	1	1

Table showing recorded data of number of iterations as well as time taken for specific input data

Number of Pegs	Number of Iterations	Time taken (s)
15	190	0.072
24	103504	2.168
35	701832	701.832

Graph showing Number of Iterations v.s. Number of Pegs on the board



From the graph, the number of iterations as a function of number of pegs is given by:

$$I(p) = 1.1824 \cdot e^{0.4021\,p}$$

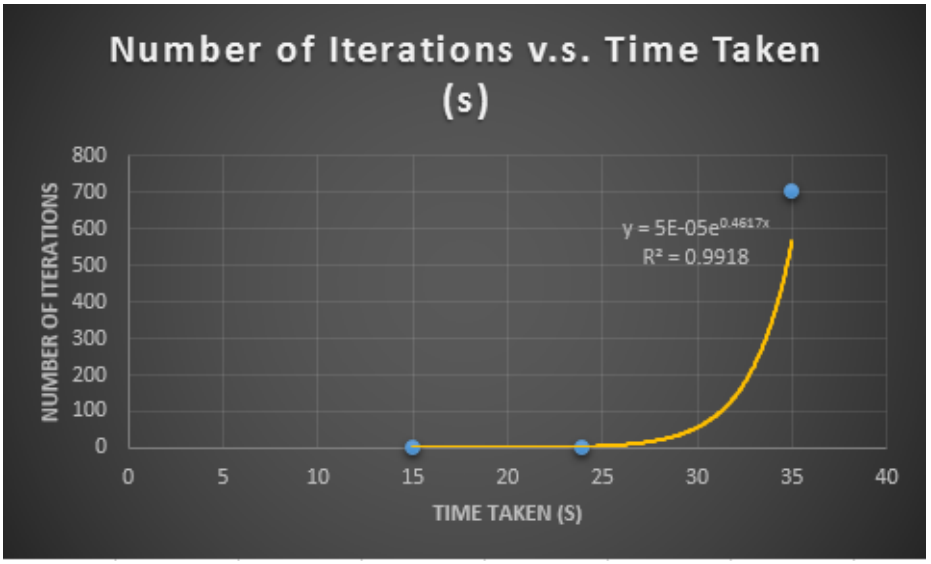
In the above equation, the coefficient of p, 0.4021, affects the amount of iterations taken to solve for constant p.

The coefficient of the exponential term, 1.1824, affects the amount of iterations taken to solve for constant p (although at a much slower rate than the coefficient of p).

The value on the graph, $R^2 = 0.8781$, represents the linear regression of the best-fit curve i.e. how well the generated exponential curve fits the given data set.

The value $R^2 = 0.8781$ tells us that the function I(p) does not very accurately represent the number of iterations for a board of p pegs.

Graph showing Number of Iterations v.s. Time Taken to solve



From the graph, the time taken as a function of number of pegs is given by:

$$t(p) = 5^{-5} \cdot e^{0.4617\,p}$$

The coefficient of p, 0.4617, affects the time efficiency for solving a board with p pegs.

The coefficient, 5^{-5}, affects the time efficiency for a board with peg count, p, i.e. the time taken for

constant amount of pegs.
This value for R^2 is very close to $R^2 = 1$ and hence the function for t(p) accurately represents the performance.

Experiment 2.1 - Extension of experiment 2

Board 1 - 4x4

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

Board 2 - 4x5

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

Board 3 - 5x5

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

Board 4 - 5x6

1	1	1	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	1	1

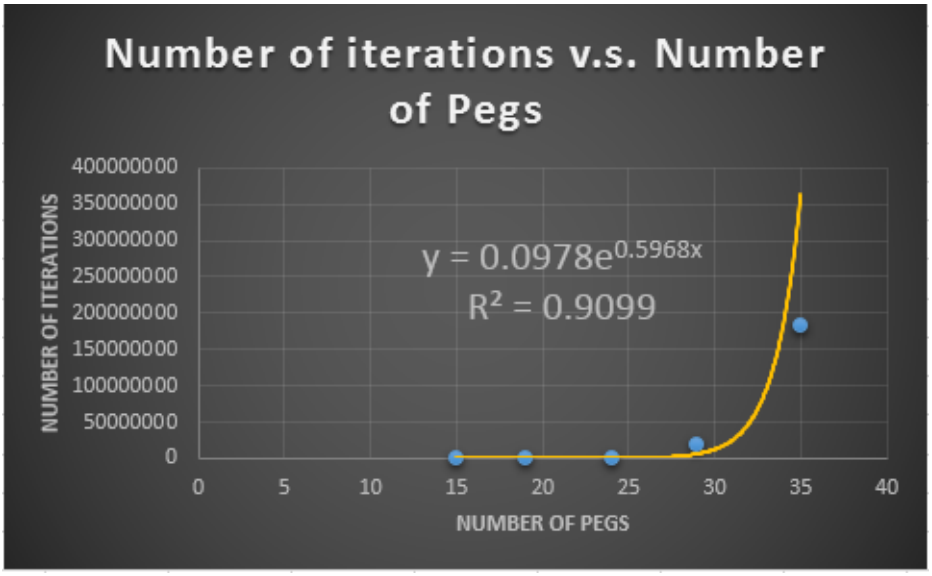
Board 5 - 6x6

1	1	1	1	1	1
1	1	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	1	1	1

Table showing recorded data of number of iterations as well as time taken for specific input data

Number of Pegs	Number of Iterations	Time taken (s)
15	190	0.072
19	32311	0.102
24	103504	2.168
29	18948524	31.278
35	182220026	309.362

Graph showing Number of Iterations v.s. Number of Pegs on the board



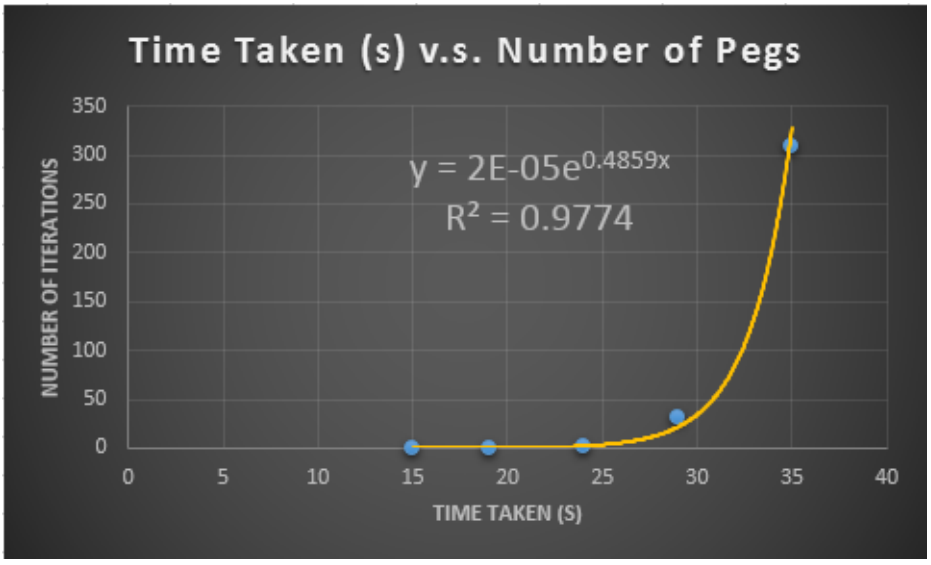
From the graph, the number of iterations as a function of number of pegs is given by:

$$I(p) = 0.0978 \cdot e^{0.5968\,p}$$

In the above equation, the coefficient of p, 0.5968, affects the amount of iterations taken to solve for constant p.
The coefficient of the exponential term, 0.0978, affects the amount of iterations taken to solve for constant p (although at a much slower rate than the coefficient of p).

The value $R^2 = 0.9099$ tells us that the function I(p) semi-accurately represents the number of iterations for a board of p pegs.

Graph showing Number of Iterations v.s. Time Taken to solve



From the graph, the time taken as a function of number of pegs is given by:

$$t(p) = 2^{-5} \cdot e^{0.4859\,p}$$

The coefficient of p, 0.4859, affects the time efficiency for a board with p pegs.
The coefficient, 5^{-5}, affects the time efficiency for a board with peg count, p, i.e. the time taken for

constant amount of pegs.
This value for R^2 is very close to $R^2 = 1$ and hence the function for t(p) accurately represents the performance.

Discussion

Experiment 1

By looking at the table of recorded data for experiment 1, we can already see some horrific looking results. Moves 1 through 5 seem to take 1 more iteration from the previous move and this pattern is repeated randomly throughout the results. I believe this arises as a case of pure chance since the probability of choosing the correct traversal of our decision tree grows as follows:

NB: This may seem tedious and unnecessary, but the argument does lead to some important observations regarding complexity.

For a move index, i, there are 2^{2i} different possible solutions to the board. This means that from move i, there are p -1 - i (p = number of pegs, open or closed) possible traversals from a root at position i to the leaves of the tree (which hold the problem solution). Hence the probability of selecting the correct traversal is given by:

$$P(i) = \frac{1}{p-1-i}$$

This comes about as a result of our algorithm failing to check if a node is empty (contains no peg) before testing if it can make a valid move. This issue is addressed toward the end of the Discussion section.

Now from P(i), we can multiply the probabilities of getting the first 1 moves correct in a row together where p = as follows:

$$P(0) \cdot P(1) \cdot \ldots \cdot P(11) = \text{Some Extremely Small Number}$$

The point of this argument is firstly to show that for every board, the time taken to solve the puzzle is dependent on how ‘lucky’ we are when our algorithm begins it’s brute force search i.e. we could find the solution on the first try or in approximately 2^{2i} many tries. This is because this algorithm is analogous to a depth-first search and hence attempts to perform a full traversal from root to leaf as soon as possible.

The second point I wish to make here is that due to the random selection of traversal routes, in order to collect viable data, we would need to test p different boards (p the amount of pegs) since our algorithm works through the board from top left to bottom right and hence finds the different possibilities at different times in each of these cases (see lines 03 and 04 of solve()).

Following the above insight, I would reject the data from experiment 1 as a valid representation of the algorithm’s performance. The data does however provide a good means to understand the probabilistic nature of backtracking.

Experiment 2 - Combined

I originally kept experiments 2 and 2.1 apart since I wanted to keep the board size increase as constant as possible, yet after conducting both experiments distinctly, the evidence proved that the algorithm solved problems regardless of board dimensions and its solving time/number of iterations was only related to the amount of pegs on the board.

For this part of the discussion I will consider the graphs plotted from the experiment data and analyse their trend lines and the equations accompanying them. I will also talk about the linear regression values produced by both plots. I will use the experiment 2.1 as the basis for my discussion - I thought it was useful to leave experiment part 2 in the report since it provided me with insight as to how the algorithm solved regardless of board dimensions.

Number of Iterations v.s. Number of Pegs

From a general point of view, this graph produced a result that relates very closely to our theory. The equation of the trend line in this graph provides us with two calibration coefficients as pointed out in the Data section. The equation coefficient (0.0978) adjusts the trend line based on the efficiency of our implementation of the solve() method.
If we were to code the method in such a way that solve() was more efficient, this would be the value which would vary.
If we were to implement methods to 'trim' our decision tree i.e. remove unnecessary moves, the exponential term's coefficient would decrease and so would the rate of exponential growth of our graph.

This is the biggest clue of how to reduce the number of iterations of the solve() algorithm since as the graph grows with larger inputs, the rate at which the exponential trend line goes from horizontal to vertical would be much less. This can be demonstrated with a simple derivative.
Consider the following equation representing the actual trend line with variable coefficients:

$$y = ae^{bp}$$

Now consider the rate of change of y with respect to a and b respectively:

$$\frac{dy}{da} = e^{bp}$$

And

$$\frac{dy}{db} = pae^{pb}$$

Clearly, as we increase a, the effect on the function reflects an exponential increase.
Now as we increase b, the same effect occurs, only the rate of increase reflects a direct proportionality to the number of pegs as well.

Consider the above in this sense:

The ratio of the affect of increasing a to the affect of increasing a $\left(\frac{dy}{da} : \frac{dy}{db}\right)$ is given by:

$$\frac{dy}{da} \div \frac{dy}{db} = pa$$

This means that if we wanted to reduce this algorithm's run time, we would get the biggest effect by focusing our efforts on attempting to reduce b.

The regression for this function was 0.9099 which, in a general physics experiment, is usually an unacceptable value. It tells us that the error in our trend line was approximately 9% where results are only usually accepted when they fall within a 5% error margin.
I feel that in order to get a more accurate trend line, we should have performed n·m tests (assuming the board's dimensions are nxm) on each input board by putting the open peg in a different position for each run and finally taking an average of these results to use as our single point (or plotting them all since this would have the same effect) and plotting this in our scatter plot.
I feel that the 9% error is acceptable under the circumstances that computers are an extremely variable test platform with events such as context switching, process race conditions for memory and or memory allocation as well as many other arbitrary events playing a part in influencing our results. Ideally, for the most accurate results, we would want to perform our experiments on an operating system that is able to allocate a single, dedicated core (since no multithreading is used here) and perhaps a dedicated DIMM including it's memory bus solely to the program invoking solve(). This would produce the least noise in our data, but would require us to go to new lengths.

Time Taken v.s. Number of Pegs

Regarding the graph trend line function, I would pose a similar argument as above. We can clearly see an exponential increase in time taken to solve as a function on the number of pegs.
Our value for regression was within the acceptable 5%.

One interesting point to note here is how the number of iterations computed by solve() is irrespective of the computer running it whereas the time taken is not. This means that if we were to run the same tests on many distinct computers and plot time taken v.s. number of iterations, we would have a nice constant with which we could compare the different platforms' efficiency in running this algorithm.

Discussion of the solve() algorithm

The first interesting point to discuss here is how this algorithm is analogous to the depth-first search algorithm. This is because the algorithm attempts to traverse the decision tree from its root to its first possible leaf node and upon failing, it backtracks up the tree and attempts the next possible traversal to a leaf node.
Now a depth-first search has [1] worst case complexity of $O(b^{d+1})$ where b is the branching factor (4 = 2^2 in our case) and d is the depth of the tree (p-1 in our case). The best case for the algorithm is hence $O(d)$.
It is also worth noting that the worst space is O(bd) since the only storage necessary is the current traversal stack of maximum depth d where for each of these iterations it may store at most b paths (say it is traversing the fourth path, paths 1 through 3 are still in memory). This means that our algorithm has a small space usage increase (at leas when compared to its complexity increase).

The second point to discuss is complexity functions obtained from the trend lines generated by our input data. Where these functions do indeed verify that the growth rate of our algorithm, relative to the number of pegs on the board, is exponential ($O(c^n)$), they do not match the exact function predicted by our theory $\left(O\left(c^{n^2}\right)\right)$.
There is more than one reason for this, the first being the ambitious assumption that every peg on every board has 4 possible moves. This is not true in the case where a peg is on the edge of the

board or one row in from the edge of a game board. Although in defense of our assumption, our algorithm does check every possible move regardless of position in the board and handles the case where a move would put the peg out of bounds.

Possible ways to ‘trim’ (a better way to say this would be “optimize our algorithm”) our algorithm’s decision tree includes not attempting to perform a move in the case that the move will land the peg outside the board’s bounds i.e. handle this prematurely.
Another method to decrease work done would be to not perform any checks on a peg in the case that the current peg being checked is an out of bounds or closed (‘0’) peg (this error is not reflected in our theory since it is handled in our code, but could be handled better as is the case for the point above).

Conclusion

In view of our goals, we have verified that the complexity of the backtracking algorithm is indeed exponential.
We were, however unable to verify the exact predicted complexity of $O(2^{2p})$ due to the inability to produce a valid worst case input data sample - since this would most likely pose more work than than this report itself.
I would hence classify the experiment a success, with room for improvement.
As for part 2 of the experiment goal, I have learnt a lot regarding recursive functions and made suggestions in the Discussion section to optimize the solve() algorithm and will put suggestions to better the approach to exponential growth graphing problems under the Suggestions section after this.

Suggestions

I have two suggestions that may pose a method to reduce the complexity when solving graph problems of an exponential complexity.

Implementing multithreaded backtracking

When a closed (‘1’) peg is found, 4 possible moves are attempted, each of which can result in a new board (which is a subset of the original board) and a solve() is run on this new board.
If we were to multithread our Solver method to invoke solve() on a new thread for each level of recursion, perhaps we could speed up the solving process of the graphing problem, since the average modern CPU handles up to 8 concurrent threads.
A few issues arise from this proposition however which would need to be handled. Here is a list of possible issues:
1. Too many threads will be spawned and context switching overhead time will outweigh the actual processing time. Perhaps we could invoke thread.sleep() before invoking a new solve thread to try reduce this, but it would still spawn 4 child threads at each level of recursion in the worst case.
2. Race conditions would create huge deadlock potential. Perhaps we could clone the board object and pass each thread a new instance of the board. This would have huge space requirements however, but is this such a bad thing? Today’s physical computing technology is currently focused on increasing the number of threads handled as well as storage space for RAM as well as cache memory. In light of this, my proposed method seems to be following the hardware trends of today’s computers.
Perhaps CUDA programming holds the solution to the exponential growth rate problem?
See Nvidia’s website for more info on this topic (a google search should suffice).

Inverting the algorithm logic at key moments

The logic of our program is to look for a closed (‘1’) peg and check it for possible moves.
When the board starts, there are more closed than open pegs on the board. Logic hence states that we should rather start by searching for open pegs and check them for possible occupation (moves into the open space), although once the amount of open pegs overtakes the amount of closed pegs we have the problem that we are checking for open pegs on a board of mostly open pegs (this is the symmetry of the problem).
Now if we were to have two versions of the solve() algorithm, say one that looks for closed pegs (as ours does) called solveClosed() and one for open pegs called solveOpen(), perhaps we could implement the following:

Let p be the number of closed ('1') pegs on the board with dimensions nxm.
Assume that the board starts with q = (n·m) - 1 closed pegs.

$$\text{solve}() = \begin{cases} \text{solveClosed}() & \text{if } p < \frac{q}{2} \\ \text{solveOpen}() & \text{if } p \geq \frac{q}{2} \end{cases}$$

This method would require its own theoretical analysis, yet from observation, the logic seems sound.

References

[1] Depth-First Search complexity analysis

Source: Online
Group: Massachusetts Institute of Technology
Year: 2003
Title: MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2003
Notes on the Complexity of Search September 23
URL: http://www.ai.mit.edu/courses/6.034b/searchcomplex.pdf
Viewed: 19 October 2014

[2] Backtracking algorithm

Source: Online
Group: Massachusetts Institute of Technology
Year: March 30, 2010
Title: Concepts in multicore programming
URL: http://courses.csail.mit.edu/6.884/spring10/labs/lab5.pdf
Viewed: 1 October 2014

[3] Personal Knowledge - Physics major lab experiments

I feel it is necessary to state that I am currently doing a physics major where we were required to do lab reports on a weekly basis and rigorous graph and trend line function analysis was required of us. I have applied those skills in this report and hence the reason for not citing any sources concerning that knowledge (it is personal property).

Appendix

Algorithm - solve( ) [2]

Input:
a valid nxn peg solitaire board containing ‘closed’ positions (contains a peg), open positions (contains no peg) and out of bounds positions (cannot hold any form of peg).
Output:
The final board (with one peg on it) along with the number of times solve() was entered OR the text “Unsolvable”.

```
00    print counter++
01    if board has one peg left
02        return true
03    for i from 1 to n
04        for j from 1 to n
05            for each direction (north, south, east, west)
06                if move from i,j in direction is valid
08                    make the move on the board
09                    add the move direction to path
10                    if(solve() on this new board)
11                        return true
12                    else
13                        undo last move
14                        pop last move from path
15    return false
```