UNIVERSITY OF THE WITWATERSRAND

COMS3005: ADVANCED ANALYSIS OF ALGORITHMS

Peg Solitaire Backtracking Assignment

October 26, 2017

*B*y Bancroft, E. (879192)
*A*nd Chalom, J. (711985)

1 Introduction

The purpose of the assignment is to implement and analyse a version of the Peg Solitaire game and the backtracking algorithm (to play the game).

2 Background

Peg Solitaire is a board game which has a number of holes that can be filled with pegs. We have chosen to use the European style of board which has extra positions on the board 2. In this style most of a grid has peg holes excluding three per corner. The aim of the game is to remove pegs until only one peg remains. This is the position one row directly above the central peg, a row above that and the left most peg in the top row. Moves are made when pegs jump over a peg and are placed in an open position. Then peg which is jumped over is then removed. This move can happen in both horizontal and vertical directions. In the European variant, the game has three possible optimal terminal states. Its is possible to hit sub-optimal states where there are more pegs left on the board but no possible moves left [1].

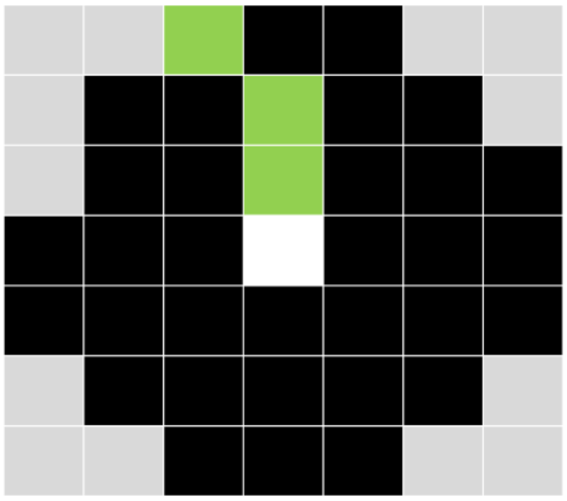

Figure 1: Diagram of an European Peg Solitaire Board Where the Black Squares Represent Peg Positions, Green Are Terminal Positions, Grey Are Not Positions and White is the Central Pixel.

The backtracking algorithm is similar to a brute force approach to finding solutions to problems but is more systematic. It attempts to follow a logical series of decisions in solving these problems and when a block state occurs the algorithm will backtrack to previous decisions and choose different paths until a terminal (complete) state is reached. The full set of solutions to a problem can be found by continuing to run the algorithm until all paths have been searched but that is not always necessary.

2.1 Recursive Algorithm

```
FindSolution($start, final, path$)
if $start.numPegs \leq final.numPegs$
    return ($start = final$)
else
    for each jump $J \in [0,n) \times [0,m) \times \{$NORTH, EAST, SOUTH, WEST$\}$
        if $J$ is a legal jump for $start$
            $start.makeMove(J)$
            $path.push(J)$
            $found$ = FindSolution($start, final, path$)
            if $found$
                return TRUE
            else
                $start.makeReverseMove(J)$
                $path.pop()$
    return FALSE
```

Figure 2: Recursive Algorithm [2]

2.2 Stack Based Algorithm

The stack based algorithm we used is adapted from the recursive version.

3 Implementation

3.1 Technology Used

We made use of c++ 11, its standard libraries and OpenMP to time our results. Our results are saved as comma seperated value files which are then processed and graphed by libre office.

3.2 How To Compile and Run

In order to compile and run the code:
Go to root folder of the project and run make.
Then run ./bin/game.out to run the game.

Commandline Parameters:
usage: ./bin/game.out -rr -m
Random state -rr
Full state -rf
Run Stacked Based Backtracking -rb
Run Recursive Backtracking -recurse
Manual -r
Help -h

3.3 Our Termination Conditions

3.3.1 Recursive Implementation

3.3.2 Stack Implementation

3.4 Problems We Encountered

3.5 Experimental Setup

4 Theoretical Analysis

We assume that our basic opertion used for analysis in the backtracking algorithm for peg-solitaire is generating a new state which is playing a valid move or jumping a peg into a valid empty space (and removing a peg between them). Determining if there are no more moves, and also getting every valid move for a peg are both assumed to take constant time, which is the speed of each conditional by the number of board elements i.e. 49 elements.

The best case complexity of backtracking for peg-solitaire would be the case where only one path needs to be generated for any number of pegs i.e. no backtracks occur because a game win is found at the end of the first path. In this case the complexity is the number of pegs left on the board or the length of the found path. If we assume this number is represented by the variable n, then the best case complexity is $O(n)$.

In the worst case complexity event no game win is possible so every possible path has to be traversed by the algorithm. Each peg has the potential to move in four directions but that is unlikely. From our observations of the game being played out moves on a board from a start state tends to allow on average two possible moves per peg when moves are available. This means that on average (based on our observations) the game has a branching factor of 2. Since each path can be considered to be a branch on a tree data structure and the number of branches is the number of pegs which is assumed to be n. This means that the total search space (game-space) size is 2^n and so the worst case complexity is directly propotional to this number i.e. the worst case complexity is $O(2^n)$.

5 Results

6 Empirical Analysis

7 Conclusion

8 Group Member Contribution

Member	**Evan Bancroft 879192**	**Jason Chalom 711985**
Game	75%	25%
Back Tracking Algorithm	50%	50%
Report	25%	75%

Table 1: Contributions of Group Members By Task

Acknowledgements

All drawn diagrams were drawn using http://draw.io/ and charts were made with Libre Office. All the programming was done in c++ using OpenMP for its timing functions.

References

[1] Douglas Wilhelm Harder. Peg solitaire. https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/.

[2] Charles E. Leiserson. Lab 5: Backtracking search. http://courses.csail.mit.edu/6.884/spring10/labs/lab5.pdf, 2010.