COMS3: Advanced Analysis of Algorithms Assignment

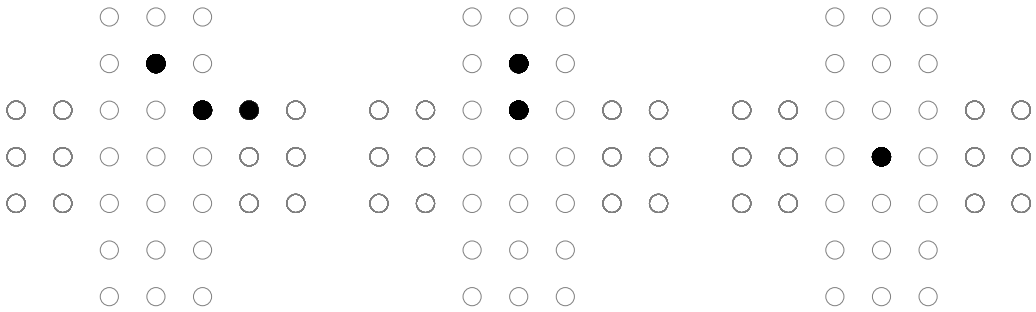
Tau Merand 908096 Vincent Varkevisser 705668

October 31, 2017

Introduction

The game of peg solitaire is a one player game played on a 33 holed cross shaped board that involves jumping pegs over other pegs, in a manner similar to checkers. The rules are as follows:

1. A move consists of jumping a peg over an orthogonal neighbor into an empty space. The peg that was jumped over is then removed from the board.
2. Pegs can only jump onto an empty space.
3. The game is won if the final peg is in the centre space.
4. If no pegs can legally move or the final peg is not in the centre the game is lost.



(a) A start state

(b) The right most peg jumps over its left neighbour

(c) The winning state

Figure 1: A winning set of valid moves

Solution Technique

Recursive backtracking using depth first search was chosen as the method for state space exploration. Unlike the brute force approach to finding a solution to a game, the game tree is not fully precomputed and then explored. Instead in Algorithm 1 the tree is generated by choosing a random legal move performing it and then evaluating the resultant state. If the resulting state is not the winning state a random legal move is chosen and performed on that state. This is repeated until a state with no legal moves is reached. If that terminal state is the winning state then the sequence of moves taken to reach that state is returned. If it is not the winning state then the algorithm backtracks to the previous state and choses a different move. Thus the game tree is generated in a depth first manner as it is explored. Due to the nature of peg solitaire it is sufficient to return the first solution found as the optimal solution (see Algorithmic Analysis)

```
1  Function backtrack
       input  : An initial, possibly un-winnable, board state.
       output: A sequence of moves to get from the initial state to the
                winning state, if a winning state cannot be reached the
                sequence should be empty.
2      initialState ← The initial state
3      legalMoves ← A list of all the legal moves for initialState
4      result ← Empty list to store moves from the initial state to the
        winning state
5      foreach move in legalMoves do
6          state ← The state after playing move on initialState
7          if state is a winning state then
8              result ← move
9              break
10         end
11         childResult ← backtrack(state)
12         if childResult is not empty then
13             Prepend move to childResult
14             result ← childResult
15             break
16         end
17     end
18     return result
```

Algorithm 1: A standard recursive backtracking using DFS

But because pegs are indistinguishable, game states where pegs are in the same position are identical, regardless of the moves taken to arrive at that state. Thus exploring the state space looking for a sequence of states leading to the winning state will probably involve evaluating the same states many times. Thus a significant speed up can be achieved by saving states that are known to not lead to the winning state as shown in Algorithm 2. A hash set of infeasible states is kept as the game tree is explored with states inserted into this set if it is not a winning state and there are no legal moves from that set or if all of its children are already in the set.

```
1  Function dynamicBacktrack
       input  : An initial, possibly un-winnable, board state.
       input  : A set of all states that have been searched and are
                known to be un-winnable.
       output: A sequence of moves to get from the initial state to the
                winning state, if a winning state cannot be reached the
                sequence should be empty.
2      initialState ← The initial state
3      infeasibleSet ← The set of un-winnable states
4      legalMoves ← A list of all the legal moves for initialState
5      result ← Empty list to store moves from the initial state to the
         winning state
6      foreach move in legalMoves do
7          state ← The state after playing move on initialState
8          if state is in infeasibleSet then
9              continue
10         end
11         if state is a winning state then
12             result ← move
13             break
14         end
15         childResult ← dynamicBacktrack(state, infeasibleSet)
16         if childResult is not empty then
17             Prepend move to childResult
18             result ← childResult
19             break
20         else
21             Add state to infeasibleSet
22         end
23     end
24     return result
```

Algorithm 2: Recursive backtracking using DFS and dynamic programming methods

Analysis

Algorithmic Analysis

For any given state S let P be the number of pegs and E be the number of empty spaces. There are 33 spaces on a board so $E = 33 - P$
For any legal move a peg must jump over an orthogonal neighbour and that neighbour is then removed. Therefore for any initial state a solution must consist of exactly $P - 1$ moves. If more moves were performed then there would be no pegs left and if fewer were performed the resultant board wouldn't be the final state. The best case for a depth first algorithm is going to be if the correct move was selected and performed at each level of the tree, thus the best case for our backtracking DFS, with or without the dynamic programming methods, would be $O(P - 1) = O(P)$
On the other hand the worst case would be when the entire game tree has to be explored to find a solution. One can estimate an upper bound on the number of legal moves that can be made in any given game state in two ways. Firstly pegs have to jump orthogonally so each peg could jump, at most, in 4 directions. Alternatively since pegs have to jump into empty spaces, each empty space could be jumped into by at most 4 pegs. For a state that has 32 pegs and 1 empty space, e.g. the standard start state, there only 4 legal moves. Whereas in a state with 2 neighbouring pegs and 31 empty spaces there are only 2 possible moves. Therefore an upper bound on the number of moves could be the minimum$\{4P, 4E\}$ but $4P < 4(33 - P)\ \forall P \leq 16$ giving an upper bound of $4 \times 16 = 64 = 2^6$ on the number of moves that can be made in any state.
As discussed above the number of moves to find a solution is $P - 1$ but for some invalid paths the moves may lead to an invalid state that while $P > 1$, no legal moves can be performed due to the relative positions of the remaining pegs. Thus $P - 1$ is an upper bound on the depth with which any branch must explored, i.e. the maximum height of the game tree is $P - 1$.
If we consider the root node to be at depth 0 of the tree, at depth 1 there are at most 2^6 states, at depth 2 there are $(2^6)^2$ states. Following this pattern, at the maximum depth of $P-1$ there are $(2^6)^{P-1}$ terminal states. Therefore the total number of states in the game tree is $1+(2^6)+(2^6)^2+(2^6)^3+...+(2^6)^{P-1} = \sum_{i=0}^{i=P-1}(2^6)^i = \frac{(2^6)^P-1}{(2^6)-1}$ Therefore the worst case would be $O(2^P)$ which is equivalent to $O(2^N)$ where N is the number of moves in the solution.

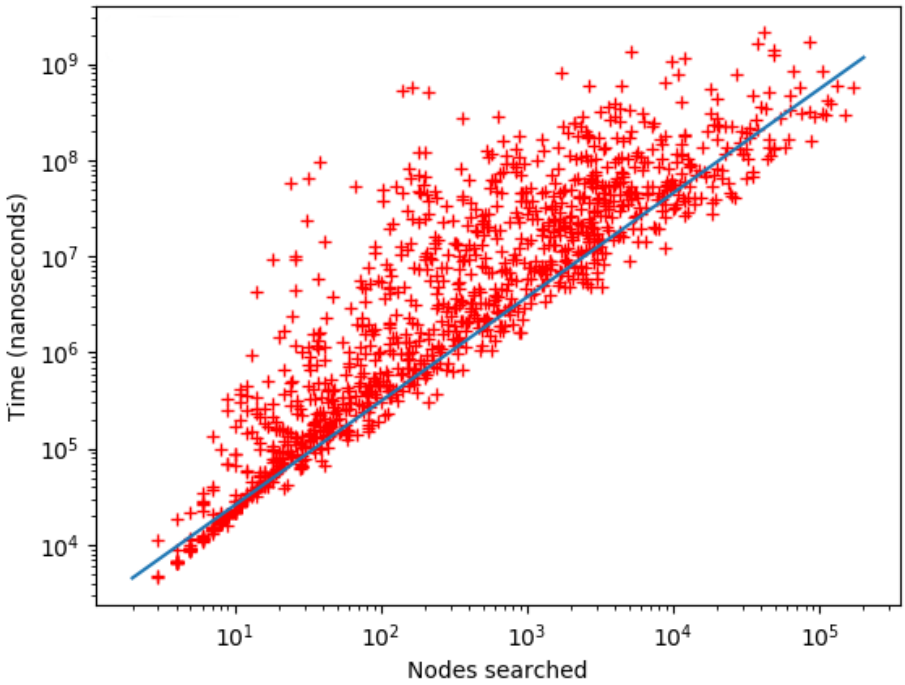


Figure 2: Graph comparing running time with the number of elements searched

Empirical Analysis

Figure 2 shows that the running time can be fit to the line $2133.73x^{1.083}$ with an R^2 value of 0.81 meaning the algorithm is of $O(n^{0.81}) \approx O(n)$ complexity in the number of elements of the tree searched.
Figure 3 shows that the running time can be fit to the line $2000e^{0.6x}$ meaning the algorithm is of $O(e^n)$ complexity in the length of the solution.

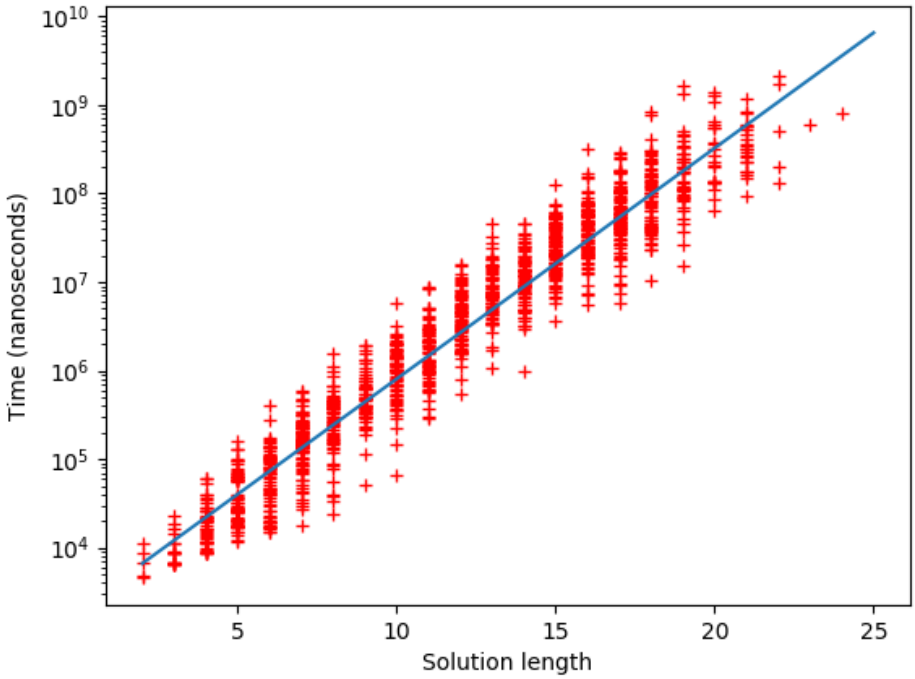


Figure 3: Graph comparing running time with the length of the solution found