CAB203 Project: Pegs

Ryan Indrananda: n10852565

1 Introduction

The pegs puzzle board consists of regularly spaced holes of which amount can vary. It starts with a given amount of the holes filled with pegs and some holes left empty, for example:

$$[XXooXX]$$

where 'X' = hole with peg in, and 'o' = an empty hole.

The goal of the puzzle is to determine the shortest set of moves from a given arrangement of the board which will ultimately leave it with only one hole filled with a peg and the rest of the holes left empty, e.g.

$$[oXoooo]$$

A 'valid' move exists if two peg-filled holes and an empty hole are directly adjacent, such as:

$$[oXX]\ or\ [XXo]$$

A move states that the peg furthest away from the empty hole can 'jump over' the peg in the middle, then the middle peg is removed from the board.

2 Problem

Let us describe the problem in its entirety. We can describe the starting configuration in terms of the index of the holes in the board. For example, a 6-hole board may be visualized as: $(x_0, x_1, x_2, \dots, x_5)$. Note that the indices start from zero to keep consistent with later implementation with Python. Thus, for our example $[XXooXX]$, $x_0 = 'X', x_1 = 'X', x_2 = 'o'$, etc. A 'solved' configuration is found when the amount of $x_j = 'X'$ is exactly one, signifying one peg on the board and the rest of the holes are empty. Storing a given configuration to check if it is solved, possibly by using a tuple or a list appended by the use of the indices, is in this

report unnecessary, as finding the amount of $x_j = 'X'$ for an arrangement can be implemented with default, built-in Python functions.

Returning to the discussion of valid moves, $[oXX]$ indicates a possible move left, achieved taking the rightmost peg and inserting it into the empty leftmost hole then removing the peg it jumps over, resulting in a $[Xoo]$ configuration. $[XXo]$ thus works vice versa. If the starting arrangement is an already solved board (one peg and the rest of the holes empty), then the sequence of moves to find the solution is just zero, in other words: zero moves are needed to achieve a solved board. However, if we find that a starting arrangement leads, after at least one move, to a board where there are multiple pegs left and no possible moves, then there is no possible complete sequence of moves thereafter that will lead to a 'solved' board.

A valid left move $[oXX]$ can be done by swapping x_j = rightmost peg with x_{j-2} = empty hole, then removing the peg 'X' in x_{j-1} and replacing it with an empty hole, 'o'. This results in $[Xoo]$.

A valid right move $[XXo]$ can be done by swapping x_j = leftmost peg with x_{j+2} = empty hole, then removing the peg 'X' in x_{j+1} and replacing it with an empty hole, 'o'. This results in $[ooX]$.

With the defined mechanics, we can continue discussing the problem in terms of graphs. We can let graph $G = (V, E)$, where V = vertices = the set of configurations of the puzzle (pegboard arrangement), and E = edges = moves to go from one configuration to another. The configurations *u* and *v* are adjacent in *G* if there is a valid move in the arrangement that changes *v* to *u*. Depending on the arrangement of the pegboard, there might be the possibility of both a left and right move. Furthermore, there might also be a possibility where taking a left move will result in an unsolvable arrangement (multiple pegs left with not possible moves), but the puzzle is solvable if a right move is taken instead. However, no legal moves can be done to take *v* back to *u* to directly backtrack and test that possibility, thus *G* is a directed graph.

In summary: find the shortest sequence of moves from a starting arrangement to a 'solved' arrangement (one peg left and all other holes empty).

3 Solution

In this report, a depth first search approach was taken to solve this problem. Essentially, this approach starts at a root node, which for this puzzle is defined as the starting arrangement of the puzzle board, then explores each possibility of moves to as far an extent as possible before backtracking.

The pegs puzzle solution method to solve and find the shortest sequence of moves from starting arrangement to solved arrangement will return a path as a list of moves from start to solved, remembering the defined valid moves in the previous 'Problem' section. Therefore, the shortest path is:

$$x_{starting} = v_1, v_2, \dots, v_n = x_{solved}$$

where $x_{starting}$ = starting arrangement, and x_{solved} = solved arrangement, and v_n = vertices, or arrangements found after conducting moves from $x_{starting}$ until x_{solved} is found. v_n to v_{n+1} is given by applying either a left or right move as appropriate to v_n.

The sequence of moves is therefore:

$$M(v_1, v_2), M(v_1, v_2), \dots, M(v_{n-1}, v_n)$$

or, in other words, moves from v_1 that will result in v_2, and so on and so forth. If no path is found, our method should show such a result.

4 Implementation

The implementation makes use of the built-in Python RegEx library. Various functions were defined, such as the **solvedBoard** function which determines whether a given arrangement (stipulated within its parameters) has a valid move to solve, is already solved, or unsolvable. This function incorporates the **findPatternL** and **findPatternR** functions, which finds whether, based on the function, the defined $[oXX]$ or $[XXo]$ patterns exist within a given arrangement (vertex). These functions then return the index x_j where a peg is valid to move either left or right in conjunction with the defined mechanics of the puzzle.

In terms of doing the moves themselves, the **moveLeft** and **moveRight** functions take a given arrangement, then properly moves the pegs in conjunction with the defined mechanics of the puzzle and returns the resulting arrangement. It is important to note that these functions

only iterate the first instance found of a possible left or right move, as iterating all instances simultaneously will result in a different, and ultimately incorrect game board.

To store and remember the moves, the functions **pegIndexesL** and **pegIndexesR** create a tuple out of the index x_j of a movable peg and the possible direction it can take then returns that tuple.

The main function **pegsSolution** makes use of all previously stated functions. It first initializes the list of moves **pegMovelist** and assigns the starting arrangement to an arbitrary variable, mainly to not alter the actual puzzle board and provides the possibility of backtracking. If the starting arrangement is already solved, it returns an empty list, signifying no moves needed to solve the arrangement. While the starting arrangement is not yet solved, based on the possible direction of moves, it first conducts the move, assigns it to the arbitrary variable, then checks whether the resulting arrangement (neighbour N of the previous arrangement) is unsolvable. If it is unsolvable but a right move beforehand is possible, then it undoes the move by assigning the arbitrary variable to a right move instead. This is the backtracking portion of the implementation. If an arrangement is completely unsolvable, the function returns *None*, meaning there are no possible solutions. For every valid move without resulting in an unsolvable arrangement, the tuple generated from either **pegIndexesL** and **pegIndexesR** is appended into the **pegMovelist** list, and the new arrangement generated is assigned to the actual gameboard. Until the while loop breaks (when **solvedBoard** results in 'solved'), it will continue to conduct these moves. Once a solved arrangement is found (one peg left with all other holes empty), the **pegsSolution** function will return the full **pegMovelist**, corresponding to $M(u, v)$ defined beforehand.