CAB203 Project: Pegs

Daniel Mendels: n11070871

1 Introduction

The classic board game Peg Solitaire consists of a wooden board with holes or indentations for the pegs. Traditionally, the board is arranged with the centre peg removed.

Figure 1
[Peg Solitaire with Wooden Marbles]

Note. From Peg Solitaire with Wooden Marbles by Fishpond, n.d. (https://www.fishpond.com.au/Toys/Peg-Solitaire-Game-with-Wooden-Marbles-Reizen/0612750292132)

A player makes a valid move by 'jumping' a peg orthogonally over an adjacent peg and into a hole, removing the jumped peg. Note that a peg may not jump over an empty space. The game proceeds until only one peg remains in the centre, like so (where 'o' represents an empty space, and 'X' represents a peg):

$$\begin{bmatrix} & o\ o\ o & \\ & o\ o\ o & \\ o\ o\ o\ o\ o\ o\ o \\ o\ o\ o\ X\ o\ o\ o \\ o\ o\ o\ o\ o\ o\ o \\ & o\ o\ o & \\ & o\ o\ o & \end{bmatrix}$$

In this report, we consider a variation of the traditional Peg Solitaire board game, titled *Pegs*. Pegs differs from Peg Solitaire in that there is only one row of pegs and holes with a length of ≥ 3. Furthermore, the starting and end points may be found anywhere within the board. The goal of the Pegs puzzle is, given a starting arrangement of pegs and holes, to find the shortest sequence of moves that results in a board with a single peg and all other spaces empty, like so:

$$[\ o\ o\ X\ o\ o\ o\ o\ ]$$

or determine that there is no such sequence of valid moves.

2 Problem

We begin by describing the Pegs puzzle rigorously. An arrangement of pegs on the game board is called a *configuration* of the puzzle. We will map a configuration to a one-dimensional matrix, $x = [x_0,\ x_1, x_2 \dots]$, where each x_j represents a character literal of a peg or hole of the configuration ('o' or 'X'). The size of the matrix is given by the length of the configuration at the instantiation of the puzzle. Note that the indices start from 0 for consistency with the later implementation in Python. Each x_j represents the value of the space in position j of the configuration. A possible starting configuration is given by:

$$[X, X, o, X, o]$$

According to this notation, a possible solved configuration is given by:

$$[o, X, o, o, o]$$

The solved configuration will be called x_{solved} and the starting configuration will be x_{start}.

We now describe the legal peg moves, as outlined in the puzzle's specification. A left move is made legally when a hole is followed by two pegs, allowing the rightmost peg to jump left, leaving empty spaces in the wake of the path it has travelled. The opposite is true for a right move. Furthermore, a peg may never move outside the bounds of the configuration, over an empty space (in the instance of $[o, o, X]$, a left move would be considered illegal), nor perform a move that would land the peg on top of another.

Examining the arrangement of x_j's in a configuration to describe all legal moves, the following becomes apparent:

- A left move is possible if:
  - $\exists x_j \in x\ (\left[x_{j-2}, x_{j-1}, x_j\right] = [o, X, X])$
  - Where $\left[x_{j-2}, x_{j-1}, x_j\right] = [X, o, o]$ after the move is performed
- A right move is possible if:
  - $\exists x_j \in x\ (\left[x_j,\ x_{j+1}, x_{j+2}\right]) = [X, X, o])$
  - Where $\left[x_j,\ x_{j+1}, x_{j+2}\right] = [o, o, X]$ after the move is performed

Note that if a configuration contains the substring $[o, o, X, o, o]$, and is not already solved, no set of valid moves will result in a neighbouring secondary peg, rendering this configuration *unsolvable*.

Now that we have discussed all possible legal moves, we can define the problem in terms of a graph. Let $G = (V, E)$ be a graph with V containing the set of legal configurations for the puzzle. Two configurations u and v are considered adjacent in G if there is a legal move that changes u to v. Note that once a move has been made, the resulting configuration is unable to be reversed, as a peg has been removed. Hence, G is a directed graph.

The pegs puzzle's solution can be summarised as finding a valid solution to the starting configuration, from x_{start} to x_{solved}.

3 Solution

The standard approach to finding a valid path to the desired solution, from x_{start} to x_{solved}, involves a depth first search (DFS). We will be using a modified version of the

method provided in CAB203. To solve the pegs puzzle, we will calculate G dynamically, based on the configuration provided. There is no limit to the length of the original configuration, hence the number of unique configurations is infinite, and G must be calculated independently for each solution.

To create the graph G, we will first compute all possible moves for the starting configuration, adding each *child configuration* to an adjacency list. This process is repeated over all child configurations and their descendants, until a solved configuration is found, or all possible moves have been made, creating a directed graph of all vertices and their children.

The DFS method produces a path as a sequence of vertices (configurations). For the completion of a puzzle solution algorithm, a sequence of moves is required. Modifying the DFS to remember the moves taken between vertices is not necessary as computing the move taken between an adjacent pair of configurations is straightforward. Suppose we have an edge (u, v) where $u_0, u_1, u_2 = [X, X, o]$ and $v_0, v_1, v_2 = [o, o, X]$. Using the formulas discussed in the problem section above, the move that takes u to v is:

$$M(u,v) := \begin{Bmatrix} Left & : \exists u_j \in u\ ([u_{j-2}, u_{j-1}, u_j] = [o, X, X]) \\ & and\ \exists v_j \in v\ ([v_{j-2}, v_{j-1}, v_j] = [X, o, o]) \\ Right & : \exists u_j \in u\ ([u_j, u_{j+1}, u_{j+2}] = [X, X, o]) \\ & and\ \exists v_j \in v\ ([v_j, v_{j+1}, v_{j+2}] = [o, o, X]) \end{Bmatrix}$$

With these methods clarified, we can state the solution to the pegs puzzle. Given the starting configuration x_{start} find a path

$$x_{start} = v_1, v_2, \ldots, v_n = x_{solution}$$

The sequence of moves is then

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

If there is no such path, then the DFS will indicate so.

4 Implementation

The implementation of an algorithmic solution to the peg puzzle closely follows techniques discussed in the CAB203 course. The adjacency list to graph function required modification to allow for the creation of a directed graph. The function **`adjacencyListToGraph`** takes an adjacency list, declares the vertices, V, before iterating through the neighbours of v to create a set of tuples containing the graph's edges. Both the vertices and edges are then returned. For the implementation described in the CAB203 course, the function requires all child vertices to include their parent vertex in their list of neighbours. However, as the peg puzzle forms an undirected graph, modifications were made to the code to add all neighbours as vertices before iterating over each to create a graph. Although this creates several redundant tuples, they were automatically ignored due to the edges being a set. This modified function is called **`adjacencyListToDirectedGraph`**.

The directed graph function **`findPath`** was also provided as part of the CAB203 course and imported from **`digraphs.py`**. This function takes the following arguments: Vertices (V), Edges (E), v_{start} and v_{end}. By parsing the directed graph G, discussed above, as well as x_{start} and $x_{solution}$ returns the path of x taken to reach the solution. This function works by recursively parsing G and v_{end} through the **`findPath`** function. It follows the edges of G from x_{start} through the graph until it finds $x_{solution}$, dynamically changing the v_{start} value and saving it to a list. The list containing all the vertices traversed to reach the solution is then returned; returning **`None`** instead if no path was found.

The remaining functions are all specific to Pegs. Puzzle configurations are represented as strings since Python functionality allows for strings to be indexed similarly to a list of character literals. **`validMoves`** returns the possible moves that can be made to a configuration, following the conditions described in Section 2. **`makeMove`** implements $M(u, v)$ from Section 3 directly. **`moveMade`** takes two arguments: a starting configuration and a subsequent configuration and applies the **`makeMove`** function to the starting configuration until it has found the move that was performed to get to the subsequent configuration's state. **`createAdjacencyList`** is a helper function that takes the initial configuration and a local variable that stores the adjacency list as parameters. The function obtains the neighbourhood of the original configuration by applying **`validMoves`**, adding the neighbourhood to the adjacency list. Recursively doing the same for all subsequent child configurations.

The main function is **`pegsSolution`**, which takes a starting configuration, first analyses the state of the configuration to check if it is already solved or if it contains the unsolvable sub-string from Section 2. Declaring these conditions before creating or searching through a graph greatly reduces the computational time of the program. **`pegsSolution then`** builds an adjacency list using **`createAdjacencyList`** function described above before parsing this into **`adjacencyListToDirectedGraph`** function to create a directed graph of all possible moves. The graph is then fed into the **`findPath`** function, which returns the list of configurations from starting configuration to the solution. The function iterates over the list of configurations using **`moveMade`** to convert the configurations to moves, adding them to a list. **`pegsSolution`** returns this list to the user, like so:

$$[(1, 'R'), (3, 'L')]$$

5 References

Fishpond. (n.d.). *Peg Solitaire Game with Wooden Marbles* [Photograph]. Fishpond. https://www.fishpond.com.au/Toys/Peg-Solitaire-Game-with-Wooden-Marbles-Reizen/0612750292132