Donan928, Denan895

1-D Solitaire Report

Introduction

One-dimensional solitaire presents a unique and strategic problem revolving around a strip of holes (O), where a series of pegs (I) are initially placed in some holes. The game's main objective is to reduce the configuration using specific moves until there's a singular peg left standing. A move consists of a peg jumping over an adjacent peg into an empty hole. Subsequently removing the jumped-over peg from the configuration and turning it back into an empty hole. By reviewing this logic, only some combinations can be deemed as "winnable". E.g. if we are using three pegs, the only winnable starting position is: "O I I O I O"

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5
O	I	I	O	I	O
O	O	O	I	I	O
O	O	I	O	O	O

Whereas using a starting position of "O X X X O" will only result in a loss

Index 0	Index 1	Index 2	Index 3	Index 4
O	I	I	I	O
I	O	O	I	O

OR

Index 0	Index 1	Index 2	Index 3	Index 4
O	I	I	I	O
O	I	O	O	I

Which instantly results in a loss as there's no adjacent peg to jump over and there's more than 1 peg still standing in the combination.

Donan928, Denan895

Implementation

When creating an implementation for this problem, our plan was to start at the winning position and work our way back to generate all the possible moves while concurrently checking if the position is winning. This approach uses the breadth-first search (BFS) algorithm to check possible peg states.

For a starting position consisting of 7 pegs, we were able to work out 8 winnable states as listed:

1. O O I I O I O I O I O I O O O O O O O O O O O O O
2. O O O O I I O I O I O I O O I I O O O O O O O O O O O
3. O O O O O O I I O I O O I I I I O O O O O O O O O O O
4. O O O O O O I I O I O I I I O I O O O O O O O O O O O
5. O O O O O O I I O I O I O O I O I I O O O O O O O O O
6. O O O O O O O O I O I I I I I I O O O O O O O O O O O
7. O O O O O O O O I I I I O I I I O O O O O O O O O O O
8. O O O O O O O O I I O O I I I O I I O O O O O O O O O

Breadth First Search

This implementation uses a form of the breadth first search algorithm as we are checking every possible combination. To achieve this, we start at the winning position of “O I O” and reverse engineer the state using moves to turn it into its “previous state”. For example:

- From the state “I O O” we can generate “O I I”

or

- From the state “O O I” we can generate “I I O”

Pruning

Each newly reverse engineered state is stored into a queue, ready to be analyzed. Once we poll() the state from the queue, the *“checkDuplicateOrReverse”* method is used to check if we’ve already been to this state or if it’s a reversed position from a state we’ve visited. This allows us to efficiently prune states that we’ve already visited, saving us a large amount of unnecessary computation as we already know these positions are losing.

Identifying Win States

A state is considered a winning state if the number of pegs in the state equals the integer input. Here is an example using three pegs to demonstrate this process:

- Initial state: “O O O I O O O”

- Possible moves from winning state “O I O”:
    - Move “O I O” to “I I O” -> generates state “O O O I I O O O”
    - Move “O I O” to “O I I” -> generates state “O O O I I O O ”
- The generated states are then added into the queue for further processing.
- As the program continues, these states are checked for validity, ensuring they adhere to the rules and are not duplicates or reversals of previously seen states.

If a state is unique and considered a winning state, it is then added to a list of winning states ready to be output.

Move Generation

We start at the winning state and we work our way back. The *“generateMovesForState”* method generates new states by making valid moves from the current state. The method checks each possible triplet in the current state and generates new states by applying valid moves. All states are evaluated like this until we reach the end of the initial state.

As we are generating our moves in this way we save a lot of time computing as we are only aiming to generate winning paths and not calculating and checking if each and every combination is a winning state. If we were to go down this path, the time taken will scale exponentially with the number of layers we may have to search. As such, breadth first search is effective in iteratively finding the states and categorizing them as states we have seen, both in reverse and the original order. Therefore, we will not have to search these states and we are able to go through the states efficiently and we will come across a winning state much sooner.

Summary

Breadth first search is used efficiently to solve the 1-d solitaire problem due to the way we have approached it. By starting the search at the winning states and working our way back through possible moves, we are able to find valid previous states. States deemed invalid are efficiently pruned as we are storing valid states and their reversals in a set, reducing the computation required to find winning states. The breadth first search approach ensures we are only generating winning paths, avoiding the need to calculate if each and every combination is winning. This method scales very efficiently, especially when dealing with a large number of pegs.