PROJECT 1

PEG SOLITAIRE

ABHINAV MISHRA (SBU ID: 109838041)

Peg Solitaire Solution Using Iterative Deepening Search (Uninformed)

- **Algorithm [1]**

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH( problem, depth)
      if result ≠ cutoff then return result
    end
```

- **Implementation**

This algorithm has been implemented to solve the peg solitaire problem. Implementation can be found in source file *peg_solitaire_IDS.py*.

- **Invocation**

The source file contains the test case to be executed. Replace the *input_state* with a different test case if the source is required to be tested with difference test case.
For instance, presently source file contains following test case:

```
input_state = [
    ["_", "_", "0", "0", "0", "_", "_"],
    ["_", "_", "0", "X", "0", "_", "_"],
    ["0", "0", "X", "X", "X", "0", "0"],
    ["0", "0", "0", "0", "X", "X", "0"],
    ["0", "0", "0", "X", "0", "X", "0"],
    ["_", "_", "X", "0", "0", "_", "_"],
    ["_", "_", "X", "0", "0", "_", "_"],
]
```

This can be replace with another test case as,

```
input_state = [
    ["_", "_", "X", "X", "X", "_", "_"],
    ["_", "_", "X", "X", "X", "_", "_"],
    ["X", "X", "X", "X", "X", "X", "X"],
    ["X", "X", "X", "0", "X", "X", "X"],
    ["X", "X", "X", "X", "X", "X", "X"],
    ["_", "_", "X", "X", "X", "_", "_"],
    ["_", "_", "X", "X", "X", "_", "_"],
]
```

On executing the program, it asks if pruning is required. If yes, enter *"Yes"*, else *"No"*.
On complete execution, execution result, followed by move performed, memory consumed and time consumed will be printed.

- **Results and Observation**

Results are based on the following test case:

```
input_state = [
    ["_", "_", "0", "0", "0", "_", "_"],
    ["_", "_", "0", "X", "0", "_", "_"],
    ["0", "0", "X", "X", "X", "0", "0"],
    ["0", "0", "0", "0", "X", "X", "0"],
    ["0", "0", "0", "X", "0", "X", "0"],
```

```
    ["_", "_", "X", "0", "0", "_", "_"],
    ["_", "_", "X", "0", "0", "_", "_"],
]
```

*

Implementation	**Memory Consumed**	**Time Consumed**	**States Expanded**
Without pruning	24.250000 MB	4628.774166 ms	644
With pruning (Pagoda Function)	24..246094 MB	4460.264921 ms	568

We can observe that on applying pruning no. of expanded states reduced which is also reflected in memory consumption and time consumption. Pruning technique applied in this implementation is Pagoda Function, which will be discussed in detail later.

Peg Solitaire Solution Using A*

- **Algorithm [2]**

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated,
initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start,
goal)
     while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)
         remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] +
dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score <
g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] +
heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from,current)
    total_path := [current]
    while current in came_from:
        current := came_from[current]
        total_path.append(current)
    return total_path
```

- **Implementation**

This algorithm has been implemented to solve the peg solitaire problem. Implementation can be found in source file *peg_solitaire_A*_h1.py*.

In the implementation of A* three heuristic approached have been applied,

1. Manhattan Distance
2. Move Possibility
3. State Penalty

- **Invocation**

Invocation of this implementation is similar to IDS invocation discussed before.
On executing the program, it will ask for heuristic approach to be selected. After selection of heuristics, it will ask if pruning is required.
On completion, program will list result followed by moves performed, memory consumed and time consumed.

- **Results and Observation**

Test configuration followed is same as followed for IDS.

1. Manhattan Distance

In this heuristic, Manhattan distance of each peg position is evaluated with respect to center position. States with lowest Manhattan Distance are considered for expansion during the search execution.

Implementation	**Memory Consumed**	**Time Consumed**	**States Expanded**
Without pruning	8.042969 MB	60.311079 ms	100
With pruning (Pagoda Function)	8.035156 MB	41.359901 ms	98

2. Move Possibility

In this heuristic, for every state no of possible moves is calculated. State with highest no of possible moves is selected for expansion during search.

Implementation	**Memory Consumed**	**Time Consumed**	**States Expanded**
Without pruning	9.097656 MB	274.351120 ms	713

With pruning (Pagoda Function)	9.031250 MB	256.072998 ms	673

3. State Penalty

In this heuristic, penalty is assigned to each peg position based on possibility to reach goal state. Following penalty matrix has designed to determine the best node to expand during search.

```
---404---
---000---
4030304
0001000
4030304
---000---
---404---
```

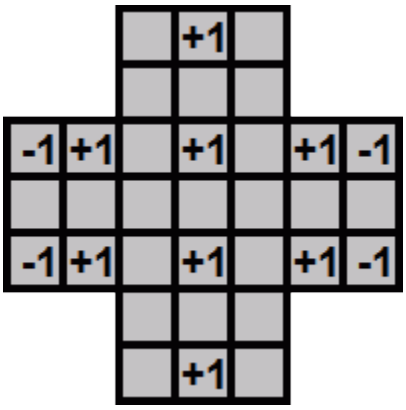
Reason for assigning 4 on corners is the inability of replacing pegs at these position by any other peg. Positions with weight 3 are assigned this weight due to their equal probability to move to corners (a bad state) and to another equivalent position on the board. Center has been given a weight as 1 to make sure there is no peg on it till the goal state is reached.

Implementation	**Memory Consumed**	**Time Consumed**	**States Expanded**
Without pruning	8.285156 MB	92.732191 ms	213
With pruning (Pagoda Function)	8.289062 MB	95.798969 ms	212

It is observed that Move Possibility heuristic performs worst due to the face that it does not take closeness to goal state into consideration and just concentration on forward movement in the search space. Manhattan Distance over performs State Penalty. This is owing to the fact that this heuristic considers all peg positions and their contribution in achieving the goal state where as State Penalty heuristic doesn't consider all peg positons.

Pruning Technique- Pagoda Function [3]

A Pagoda Function, is some function which gives a value to any position of the pegs on the board, and which has the property that any jump performed on the board will not increase the value of function. If a jump lowers the value of a pagoda function below the value of the final board position, then the game will be impossible to finish. This pagoda function value is calculated by simply adding together all the values of all the filled holes on the board.
Pagoda function employed in this implementation is,



References:

[1] Problem Solving and Search, Prof. I.V. Ramkrishnan, Spring 2015

[2] http://en.wikipedia.org/wiki/A*_search_algorithm

[3] http://www.jaapsch.net/puzzles/pegsolit.htm