Alexis Lara-Umana
CSCI 164
April 13, 2022

Search w/ Eight Puzzle (and 15-Puzzle)

I. Purpose & Scope

- The goal of this project is to build an artificial intelligent that can solve *8- and 15-puzzle problems* using different search methods.
- This project is a software that can be used to create number-puzzle games and a solver for the problem using *Artificial Intelligence* by implementing multiple search method. In addition, the software is designed to help the user to identify the different benefits of each search method and their downfalls.

II. The Problem w/ Extra

- Issues I've come to have whilst building my program, along the lines of small issues here and there. While building the program for this project, I tried my hand for the first time at Python, and in doing so I'm realizing the similarities and differences in doing so, unlike C++. First off, when doing this project, I got too comfortable with plugging in the variables directly into the code itself for the inputs, so converting strings to integers on a grid for the functions themselves to work properly from the user's side is something I struggle with. And then for the inputs as well, my program works well with an 8 or 15 puzzle, but the only thing I'm having difficulty with is how to get the program to work for both letters and numbers together. Lastly, a side note would be that I couldn't figure out how to work with iterative A*, so in my repository there are 3 files, the 15Puzzle is mainly for reference, and 8(P_15)Puzzle is what my initial code was, where the build was more precise and better structured but I struggled to have it run properly even though, hypothetically, it meets all the criteria but I just can't run it on an online compiler with ease, and lastly was my NPuzzle code that runs smoothly but has the issues mentioned beforehand. So, my report will mainly be on the last file since it's the one that runs more reliably, but it's not as complete as the second file unfortunately. As for members that I worked with in class, I'm not too sure mainly because I worked on my submission alone, but I did ask another classmate for their opinion on my work and for tips on how to adjust certain parts, my peer being Michael Nugent.

III. Definitions

- Heuristic function: a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow (estimation cost from the current state to the goal state), and we use Manhattan distance as a Heuristic function for this problem.
  Manhattan distance: the distance between two points measured along axes at right angles.
  GValue: the actual cost from the initial state to reach the current state.
  FValue: GValue + Heuristic function .

IV. Code Specifications

Class Node:
Node class Is used as nodes for the Goal tree class, in addition to having the expansion function and any other cost functions.

Attributes:
state: the state of the n-puzzle in the current node.
GValue(g): the depth of the current node (root depth is 0).
parent: pointer to the parent of the node (root node doesn't have a parent). Action: the action that led into creating the state within the node (Left, Right, Up, Down) (root node doesn't have an action).
children: list of the node's children (expansions).

Functions:
__init__ (state, g, parent, action, children): it's the constructor of the Node class that initiate the attributes' values of the node by the passing parameters. equal(state): return true if the parameter's state is equals to current, false otherwise.
mannhaten_distance (): calculate the distance between every square and its goal position measured along axes at right angles.
get_h(): return the heuristic value from the current state to the goal state, using the Manhattan distance.
get_f(): return the F value of the node, which is f = GValue + HValue.
expand(): return a list of all possible states generated directly from the state within the current node.
move(state, x1, y1, x2, y2): Move the blank space in the given direction (x2,y2) and return a new state after moving the blank. And it returns None if the positions value is out of limits. (Used by expand function)
copy(state): return a similar created matrix of the given node. (Used by move function)
find(state,x): return the position of (Specifically) the blank space. (Used by expand function).

• **Class GoalTree:**
Goal tree class is the main class of the software, having the search methods and the closed list, in addition to other things.

Attributes:
root: the root node of the tree.

Functions:
__init__(initial_state): it's the constructor of the GoalTree class that initiate the attributes' values of the GoalTree by the passing parameters.

Static - isGoal(state): goal test function, return true if the parameter is a matrix of increasing order (is the goal), false otherwise.
Solve(strategy): the interactive method of the class takes a strategy (search method), then return a tuple of results that will solve the puzzle according to the strategy.

results: a tuple that has (sol, g, processed_nodes, max_stored_nodes, flag, root.state):
- sol.state: the solutions' state of the problem.
- sol: a list of moves (e.g. right, left, ...) that if fallowed will solve the puzzle. - g: the cost of the solution (i.e., the number of moves to reach the solution).

- processed_nodes: the number of nodes that has been tested using the goal function.
- max_stored_nodes: the maximum number of nodes stored concurrently in the frontier.
- flag: True if it has found a solution, False otherwise.

bredthFirst(): return a tuple of results that will solve the puzzle according to the breadth first search strategy.
UniformedCost(): return a tuple of results that will solve the puzzle according to the uniformed cost search strategy.

depthFirst(): return a tuple of results that will solve the puzzle according to the depth first search strategy.
depthLimited(l): return a tuple of results that will solve the puzzle according to the depth limited search strategy, with respect to l.

iterativeDeepening(): return a tuple of results that will solve the puzzle according to the iterative deepening search strategy.
greedy(): return a tuple of results that will solve the puzzle according to the best- first search strategy.

aStar(): return a tuple of results that will solve the puzzle according to the A* search strategy.
Static - solution(node): return a list of actions that lead to the solution.

- **Global Functions:**
  Solvable(state): return true if the given state is solvable, false otherwise. random_state(n): generate random states with n dimension through expansion to ensure solvability.

number_of_inversion(state): return the number of inversions of a given state. (Used by Solvable function)
row_of_blank_from_bottom(state): return the row of the blank from the bottom of a given state. (Used by Solvable function)

- **Min-Heap Functions (Used as the frontier of the uniform, greedy and A* strategy searches):**
  HeapPush(heap, item)**:** Push item onto the heap, maintaining the heap invariant.
  HeapPop(heap): Pop the smallest item off the heap and return it, maintaining the heap invariant.

  _siftdown(heap, startpos, pos): “heap” is a heap at all indices >= startpos, except possibly for pos. pos is the index of a leaf with a possibly out-of-order value. Restore the heap invariant. (Used by HeapPush function)
  _siftup(heap, pos): sifting up the heap to restore the heap invariant.

- **Usage of built-in libraries:**
  - From collections class we import deque function to help us in the breadth-first search. (Used in bredthFirst function)
  - From random class we import * that help us to make a random initial state. (Used in random_state function)
- **The project has been implemented by *Python* language. The implementation code has been attached with the document.**

V. Results

```
Choose an algorithm [Breadth first, Depth first, Uniform cost, Depth limited, Iterative deepening, Greedy, A*] ->
BF

---------- Output Information ----------
Time taken: 0:00:00.008144
G-value (level solution found in goal tree): 18
Processed nodes: 68
Max stored nodes: 31
Do we find solution: True
Solution:
['Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Right', 'Up', 'Left',
'Down', 'Down', 'Right']
Solution state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

```
Choose an algorithm [Breadth first, Depth first, Uniform cost, Depth limited, Iterative deepening, Greedy, A*] ->
a*

---------- Output Information ----------
Time taken: 0:00:00.306973
G-value (level solution found in goal tree): 20
Processed nodes: 582
Max stored nodes: 230
Do we find solution: True
Solution:
['D', 'L', 'L', 'U', 'R', 'D', 'D', 'L', 'U', 'R', 'R', 'U', 'L', 'L', 'D', 'D', 'R', 'U', 'R', 'D']
Solution state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

*Most of the outputs are like this for 3x3 (8-Puzzle)

```
Choose an algorithm [Breadth first, Depth first, Uniform cost, Depth limited, Iterative deepening, Greedy, A*] ->
DF

---------- Output Information ----------
Time taken: 0:00:02.103704
G-value (level solution found in goal tree): 276
Processed nodes: 3302
Max stored nodes: 1732
Do we find solution: True
Solution:
['Right', 'Right', 'Up', 'Left', 'Left', 'Up', 'Left', 'Up', 'Right', 'Down', 'Right', 'Right', 'Down', 'Left', 'Left',
'Down', 'Left', 'Up', 'Right', 'Down', 'Right', 'Right', 'Up', 'Left', 'Left', 'Down', 'Left', 'Up', 'Right', 'Up',
'Right', 'Up', 'Left', 'Down', 'Down', 'Down', 'Right', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left',
'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up',
'Left', 'Up', 'Right', 'Right', 'Down', 'Left', 'Down', 'Right', 'Up', 'Left', 'Down', 'Left', 'Up', 'Right', 'Right',
'Down', 'Right', 'Down', 'Left', 'Left', 'Left', 'Up', 'Right', 'Down', 'Right', 'Right', 'Up', 'Left', 'Left', 'Left',
'Up', 'Up', 'Right', 'Down', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Up', 'Right', 'Down', 'Right', 'Down',
'Down', 'Left', 'Up', 'Right', 'Up', 'Left', 'Up', 'Left', 'Down', 'Right', 'Down', 'Down', 'Right', 'Up', 'Left',
'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Left', 'Down', 'Right', 'Up', 'Up', 'Up', 'Left',
'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Down', 'Down', 'Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down',
'Right', 'Up', 'Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Down',
'Right', 'Right', 'Up', 'Up', 'Left', 'Down', 'Left', 'Down', 'Right', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left',
'Left', 'Up', 'Right', 'Right', 'Up', 'Left', 'Down', 'Left', 'Down', 'Right', 'Right', 'Up', 'Up', 'Left', 'Down',
'Right', 'Down', 'Left', 'Left', 'Up', 'Right', 'Up', 'Left', 'Down', 'Down', 'Right', 'Right', 'Up', 'Left', 'Down',
'Left', 'Up', 'Right', 'Right', 'Down', 'Left', 'Left', 'Up', 'Up', 'Right', 'Down', 'Left', 'Down', 'Right', 'Right',
'Up', 'Left', 'Up', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Right', 'Down', 'Right', 'Up', 'Up', 'Left',
'Down', 'Left', 'Up', 'Right', 'Right', 'Down', 'Left', 'Up', 'Right', 'Down', 'Left', 'Left', 'Up', 'Right', 'Right',
'Down', 'Down', 'Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Left', 'Up',
'Right', 'Right', 'Down', 'Down']
Solution state:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
```

```
Choose an algorithm [Breadth first, Depth first, Uniform cost, Depth limited, Iterative deepening, Greedy, A*] ->
ID

---------- Output Information ----------
Time taken: 0:00:02.387156
G-value (level solution found in goal tree): 414
Processed nodes: 7400
Max stored nodes: 3888
Do we find solution: True
Solution:
['R', 'U', 'R', 'D', 'D', 'D', 'L', 'U', 'L', 'U', 'U', 'L', 'D', 'R', 'R', 'R', 'D', 'D', 'L', 'L', 'U', 'L', 'U',
'U', 'R', 'R', 'R', 'D', 'L', 'U', 'R', 'D', 'L', 'L', 'D', 'R', 'R', 'U', 'U', 'L', 'D', 'D', 'D', 'R', 'U', 'L', 'L',
'U', 'R', 'U', 'R', 'D', 'D', 'D', 'L', 'L', 'L', 'U', 'U', 'R', 'D', 'D', 'R', 'U', 'U', 'L', 'D', 'R', 'D', 'L', 'L',
'U', 'R', 'D', 'R', 'U', 'L', 'L', 'U', 'U', 'R', 'R', 'R', 'D', 'L', 'U', 'L', 'L', 'D', 'D', 'R', 'R', 'U', 'R', 'D',
'L', 'D', 'R', 'U', 'U', 'L', 'L', 'U', 'L', 'D', 'R', 'R', 'U', 'L', 'L', 'D', 'D', 'R', 'U', 'L', 'U', 'R', 'R', 'D',
'L', 'D', 'R', 'U', 'U', 'L', 'D', 'R', 'R', 'D', 'D', 'L', 'U', 'L', 'D', 'R', 'R', 'U', 'U', 'L', 'L', 'U', 'L', 'D',
'R', 'R', 'R', 'D', 'D', 'L', 'L', 'U', 'R', 'D', 'R', 'U', 'U', 'L', 'D', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R',
'D', 'R', 'U', 'L', 'L', 'L', 'U', 'R', 'R', 'D', 'R', 'U', 'L', 'L', 'D', 'R', 'U', 'R', 'D', 'L', 'U', 'L', 'D', 'L',
'U', 'R', 'R', 'D', 'R', 'U', 'L', 'D', 'L', 'U', 'R', 'R', 'D', 'L', 'U', 'L', 'D', 'R', 'R', 'U', 'L', 'D', 'L', 'U',
'R', 'R', 'D', 'D', 'D', 'L', 'U', 'R', 'U', 'L', 'D', 'D', 'R', 'U', 'U', 'L', 'D', 'R', 'D', 'L', 'U', 'U', 'R', 'D',
'L', 'D', 'R', 'U', 'U', 'L', 'D', 'L', 'L', 'U', 'U', 'R', 'D', 'L', 'D', 'R', 'R', 'U', 'R', 'D', 'D', 'L', 'U', 'R',
'U', 'L', 'D', 'D', 'R', 'U', 'L', 'U', 'R', 'D', 'D', 'L', 'U', 'U', 'R', 'D', 'L', 'D', 'R', 'U', 'U', 'U', 'L', 'L',
'D', 'R', 'U', 'R', 'D', 'D', 'D', 'L', 'U', 'U', 'L', 'D', 'R', 'D', 'R', 'U', 'U', 'L', 'L', 'D', 'R', 'U', 'R', 'D',
'D', 'L', 'U', 'L', 'D', 'R', 'R', 'U', 'L', 'L', 'D', 'R', 'U', 'R', 'D', 'L', 'L', 'U', 'R', 'R', 'U', 'L', 'D', 'L',
'D', 'R', 'R', 'U', 'U', 'L', 'D', 'R', 'D', 'L', 'L', 'U', 'R', 'U', 'L', 'D', 'D', 'R', 'R', 'U', 'L', 'D', 'L', 'U',
'R', 'R', 'D', 'L', 'L', 'U', 'U', 'R', 'D', 'L', 'D', 'R', 'R', 'U', 'L', 'U', 'R', 'D', 'D', 'L', 'L', 'U', 'R', 'D',
'R', 'U', 'U', 'L', 'D', 'L', 'U', 'R', 'D', 'R', 'D', 'L', 'L', 'U', 'U', 'R', 'D', 'L', 'D', 'R', 'R', 'U', 'U', 'L',
'L', 'D', 'R', 'U', 'R', 'D', 'D']
Solution state:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]


** Process exited - Return Code: 0 **
```

***The rest of the outputs are in this format for 4x4 (15-Puzzle)**

****Also included extra algorithms for the sake of my own personal use and comparison**