Assignment 3 Programming

B351 / Q351

Due: February 13th, 2018 @ 11:59PM

1 Summary

In this assignment you will work with heuristic admissibility and uninformed and A* search by applying and testing a general search procedure to the traditional 8-puzzle generalized to any $n^2 - 1$ puzzle.

You may collaborate **with one partner** on this assignment (optional). Your files will be run against Moss to check for plagiarism within the class outside of your partnership.

2 Programming

2.1 Background

The 8-puzzle is one of a family of classic sliding tile puzzles dating to the late 1800s and still played today. A puzzle is solved when the numbered tiles are in order from the top left to the bottom right with the blank in the bottom right corner.

8	3	6
7	0	1
2	4	5

Unsolved 8-Puzzle

1	2	3
4	5	6
7	8	0

Solved 8-Puzzle

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

Solved 15-Puzzle

A legal move is made by simply moving a tile into the blank space. We will be denoting the blank space with a 0 to make computation simpler. If you want to familiarize yourself with the game, you can here: `https://murhafsousli.github.io/8puzzle/#/`.

2.2 Data Structures

2.2.1 Board

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 0]]
```

The class 'Board', implemented in the `Board.py` file has two attributes:

- `.matrix` - a double subscripted list containing the description of the current puzzle state
- `.blankPos` - a tuple containing the (row, column) position of the blank (denoted as 0)

The following are Board's object methods:

1. `print(Board)` - Printing the board itself will make a call to the __str()__method and will print a string representation of the Board
2. `Board1 == Board2` - Asking if two boards are equal will make a call to the __eq()__method which returns true if all elements in Board1 are in the same position as Board2.
3. `duplicate()` - Returns a copy of the board object
4. `findElement(elem)` - Returns a tuple containing the (row,col) position of the given element
5. `slideBlank(dir)` - Since this problem is a bit easier to think about as moving the blank around rather than moving tiles into the blank position, slideBlank returns the board after swapping the blank and the value in the direction specified by dir. dir should be (0,1) to move right, (0,-1) to move left, (-1,0) to move down, and (1,0) to move up.

2.2.2 Node

This class is implemented in the `Node.py` file. The class encapsulates the following data members:

- `.board` - The board described above that belongs to the node.
- `.parent` - The `Node` that `.board` came from after applying a legal move. This should be initialized to 'None' if `.board` is the initial board.
- `.fValue` - The heuristic value of the board + the cost from the initial board to this board (if this doesn't make sense look up the A* algorithm).
- `.depth` - The depth in the move tree from the original board that this board can be found in. This should equal the number of moves the puzzle has undergone.

2.3 test.py

1. `shuffle(board, n)` - Returns the board after performing n random moves on it. This will be useful for testing your various search algorithms.

2.4 a3.py

This file contains the necessary functions to perform uninformed and A* search.

2.4.1 Uninformed Search

1. `uninformedExpansion(currentNode, frontier, goalBoard)` - In this function you will add all possible successive children of currentNode to the end of the frontier list. Successors are Nodes with boards that are the result of performing one valid move on the original board. The successors are appended to the end of the frontier to maintain its FIFO structure. This function should not return any value but rather just update the contents of frontier in memory

2. `uninformedClient(board, limit, goalBoard, mode)` - This function should loop, calling uninformed search each time, until it finds the goal board or the limit is reached. The limit means the maximum number of times checking if a node contains a goal board, NOT the maximum depth of a solution can be. If the goal is reached this function should return the goal node and `None` otherwise.

3. `uninformedSearch(frontier, limit, goalBoard, mode)` - In this function implement BFS when `mode` is false.

2.4.2 Informed Search

1. `fastSearchClient(board, limit, goalBoard, astar)` - Similar to uninformedClient this function should loop,calling fastSearch each loop, until the goal is found or the limit is reached. The limit means the maximum number of times checking if a node contains a goal board, NOT the maximum depth of a solution can be. It should perform A* search when astar is true and uniform cost search when astar is false. If the goal is reached this function should return the goal node and `None` otherwise.

2. `fastSearch(frontier, limit, goalBoard, explored, astar)`- This function, similar to uninformedSearch(), should return True when the limit is reached or the goal node if the goal is found. In this case frontier is a heapq `https://docs.python.org/3/library/heapq.html`. This should perform A* when astar is true and uniform cost search otherwise.

3. `uniformCostExpansion(currentNode, frontier, goalBoard, explored)` - Here you are to expand the frontier using the uniform cost search algorithm. Note that the fValue attribute of a node should be exactly equal

to its depth since the cost of a single move is just 1. Again, you should not return the frontier, but rather just update its contents in memory.

4. `aStarExpansion(testNode, frontier, goalBoard, explored)` - Here you are to expand the frontier using the A* search algorithm. Note that the fValue attribute of a node should be calculated as the actual cost from the initial node to the current node + the heuristic value of the current board. We have given you a heuristic, number of missed tiles, which you can use to make sure your search is working. Again, you should not return the frontier but rather just update its contents in memory.

5. Create a Heuristic - You must design and implement your own heuristic that is better than the number of missed tiles. This must be an admissible and consistent heuristic.

3 Grading

- `uninformedExpansion()` – 10 %
- `uninformedClient()` – 5 %
- `uninformedSearch()` – 5 %
- `fastSearchClient()` – 10 %
- `fastSearch()` – 10 %
- `uniformCostExpansion()` – 20 %
- `aStarExpansion()` – 30 %
- Heuristic – 10 %