Assignment 3 Programming
B351 / Q351
Due: September 24th, 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.

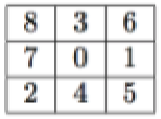
1. You may collaborate **with one partner** on this assignment (optional).
2. Your files will be run against Moss to check for plagiarism within the class outside of your partnership.
3. **If you are working with a partner, each of you have to submit a file to the autograder. Please write a comment indicating you work with a partner and name said partner.**
4. Keep your git updated by pushing your files when you're done (and within the submission period). Extra credit is given for this step.

1.1 What to Submit

Push your code to github when you are done, submit a file named 'a3.py' to the autograder.

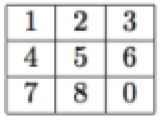
2 Background

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.



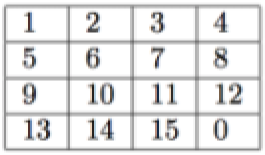
8	3	6
7	0	1
2	4	5

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.

3 Programming

3.1 Data Structures

3.1.1 Board

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

- **matrix** - a double subscripted list containing the current puzzle state
- **blankPos** - a tuple containing the (row, column) position of the blank.

The following are Board's object methods:

1. **__init__(self, matrix)**: Constructor for the board class. Takes a double-subscripted list as an argument.
2. **__str__(self)**: The string representation of a board.
3. **__eq__(self, other)**: Checks two boards for equality.

4. **duplicate(self)**: Creates a copy of given board.
5. **find_element(self, elem)**: Returns a tuple (row, col) that gives the position of the element in the board.
6. **slide_blank(self, move)**: Function for sliding the blank space around. Takes a tuple (Delta x, Delta y) that represents how far you want to move in each direction.
7. **__hash__(self)**: Hash function that maps a board to a number.

3.1.2 State

This class is implemented in the State.py file. This class encapsulates the following data members:

- **board** - The board that belongs to this state.
- **parent** - The parent state that this state came from.
- **fValue** - The heuristic value of the board and the cost to get from the initial board to this board.
- **depth** - The depth in the move tree from the original board that this board can be found in (the number of moves the puzzle has undergone thus far).

1. **__init__(self, board, parent, fValue, depth)**: Constructor for a State object. This function takes as arguments the board that corresponds to this state, the parent state, the fValue of the state, and the depth of the state in the state tree.
2. **__eq__(self, other)**: Checks if two states are equivalent.
3. **__lt__(self, other)**: Checks if a state's fValue is less than that of another state.
4. **__str__(self)**: The string representation of a given state.
5. **__bool__(self)**: Establishes the fact that the boolean value of a state should be considered True.
6. **printPath(self)**: Prints the path to the given state.
7. **__hash__(self)**: A hash function mapping the given state to a real number.

3.1.3 a3.py

This is the main file in which you will be writing code. The following is a list of functions that are provided for you:

<u>**Already Completed Functions**</u>

- **uninformed_solver(board, limit, goal_board, mode)**
  Looping function that calls either depth-first search or breadth-first search until a goal state has been found or until the limit has been reached.
  - **board** The starting board.
  - **limit** The maximum times the function is allowed to check if a state contains the goal board, **NOT** the maximum depth of the state tree.
  - **goal_board** The solved board.
  - **mode** If True, the function uses breadth-first search (BFS) to expand the fringe, otherwise it uses depth-first search (DFS).
- **findManhattanDist(current_board, goal_board)**
  - **current_board** The board for which we are calculating a heuristic value
  - **goal_board** The solved board.

  Example heuristic function that we have provided you with so that you can test the other functions you have to implement.
- **informed_searches(fringe, limit, goal_board, explored, mode)**
  - **fringe** List of states that are in the fringe. States that have not yet been explored.
  - **limit** The limit on how many times the function check if a state is the goal state.
  - **goal_board** The solved board.
  - **explored** The set of states that have already been explored.
  - **mode** If True A* is used to expand the fringe and uniform cost search is used otherwise.
- **informed_solver(current_board, limit, goal_board, mode)**
  - **current_board** The starting board.

- **limit** The limit on how many times the function check if a state is the goal state.
- **goal_board** The solved board.
- **mode** If True A* is used to expand the fringe and uniform cost search is used otherwise.

3.2 Objective

Your goal is to provide implementations to the following functions:

1. fringe_expansion
2. breadth_first_search
3. depth_first_search
4. ucs_expansion
5. a_star_expansion
6. my_own_heuristic

to the following specifications:

The following functions have to do with uninformed search.

3.2.1 fringe_expansion(current_state, fringe, goal_board)

Add all possible successive children states of the current_state to the end of the fringe.

- **current_state** The current state that is to be used to generate children to add to the end of the fringe.
- **fringe** A list that holds the states that have been added to the fringe.
- **goal_board** The solved board.

TO DO:
Write code that adds new game States to the end of the fringe.

3.2.2 breadth_first_search(fringe, limit, goal_board)

Explore the fringe using BFS.

- **fringe** - The list of states that are in the fringe.
- **limit** - The limit on how many times the function check if a state is the goal state.
- **goal_board** - The solved board.

TO DO:
Implement breadth-first search to expand the fringe. This function should return the current state if the goal board is found, True if the limit is reached, and it should expand the fringe otherwise.

3.2.3 depth_first_search(fringe, limit, goal_board, visited)

Implement the visited call to ignore the visited States. **NOTE:** There is the possibility that DFS may never find the goal.

- **fringe** - List of states that are in the fringe. States that have not yet been explored.
- **limit** - The limit on how many times the function check if a state is the goal state.
- **goal_board** - The solved board.
- **visited** - A list that keeps track of what states have already been visited/expanded.

TO DO:
This function should grab a state from the fringe and see if it has been visited before, skipping it if so. It should then return the current state if it is equal to the goal board, True if the limit has been reached and it should expand the fringe otherwise.

The following functions have to do with informed search.

3.2.4 ucs_expansion(current_state, fringe, goal_board, explored)

Uniform-cost expansion function.

- **current_state** The current state that we are looking at.
- **fringe** The states that haven't been explored yet.
- **goal_board** The solved board.
- **explored** The set of states that have already been explored.

TO DO:
Implement the uniform-cost search algorithm. You should first generate the possible children states of the current state. Add the current state to the set of states that have already been explored and then check if any of the children have already been explored. If a state has already been explored it should be ignored. Then you need to check if each of the child states is in the fringe, adjusting its position according to UCS. Then, all children that are not already in the fringe should be added to it.

3.2.5 a_star_expansion(current_state, fringe, goal_board, explored)

A* expansion function.

- **current_state** The current state that we are looking at.
- **fringe** The states that haven't been explored yet.
- **goal_board** The solved board.
- **explored** The set of states that have already been explored.

TO DO:
Implement the A* search algorithm. You should first generate the children states of the current state, being sure to calculate the heuristic value corresponding to each state. Then, you need to check if each of the child states already exists in the fringe, and if so, handle the conflict based on the overall heuristic value of that state. Finally, you need to check if each of the child states exists in the set of explored states. If it does, you either remove it from the set of explored states and add it back to the fringe or leave it alone. Of course at the end, all valid child states should be added to the fringe.

3.2.6 my_own_heuristic(current_board, goal_board)

Create your own heuristic function that is to be used with the A* search algorithm.

- **current_board** The board for which a heuristic value needs to be calculated.
- **goal_board** The solved board.

TO DO:
Implement this function. For testing purposes, in the heuristic() method, replace findManhattanDist() with your heuristic. It should pass the same tests for A*.

4 Grading

- **fringe_expansion()** - 10%
- **breadth_first_search()** - 15%
- **depth_first_search()** - 15%
- **ucs_expansion()** - 20%
- **a_star_expansion()** - 30%
- **my_own_heuristic()** - 10%

5 Bonus

Implement IDS with the following parameters:

5.0.1 ids_solver(board, limit, goal_board)

Similar to uninformed_solver()

- **board** The starting board.
- **limit** The maximum times the function is allowed to check if a state contains the goal board, **NOT** the maximum depth of the state tree.
- **goal_board** The solved board.

5.0.2 ids(fringe, limit, goal_board, horizon)

- **fringe** - List of states that are in the fringe. States that have not yet been explored.
- **limit** - The limit on how many times the function check if a state is the goal state.
- **goal_board** - The solved board.
- **horizon** - The depth of the search.