Assignment 3

B351 / Q351

Comprehension Questions Due:
Tuesday, September 24th, 2019 @ 11:59PM

Initial Due:
Tuesday, October 1st, 2019 @ 11:59PM

1 Summary

- Using various search algorithms, solve the traditional 8-puzzle generalized to any $n^2 - 1$ puzzle.
- Demonstrate your understanding of heuristic admissibility.
- Incorporate uninformed, Uniform Cost Search expansion, and A* expansion to solve these various puzzles.

We will be using Python 3 so be sure you aren't using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the above deadline without incurring a late penalty.

You may collaborate on this assignment (optional) **but you can only share ideas**. Any shared code will not be tolerated and those involved will be subjected to the university's cheating and plagiarism policy. **If you discuss ideas, each person must write a comment at the top of the assignment file naming the students with whom ideas were shared.**

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

In this assignment's programming component, you will utilize the starter framework that we have provided to build various search algorithms and expansion algorithms that have been discussed in lecture.

3.1 Data Structures

3.1.1 Board

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

1. `matrix`: a double subscripted list containing the current puzzle state
2. `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. `__repr__(self)`: A string of code to produce a board.
4. `__eq__(self, other)`: Checks two boards for equality.
5. `duplicate(self)`: Creates a copy of given board.
6. `find_element(self, elem)`: Returns a tuple (row, col) that gives the position of the element in the board.
7. `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.
8. `__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:

1. `board`: The board that belongs to this state.
2. `parent`: The parent state that this state came from.
3. `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).
4. `f_value`: The heuristic value of the board plus the cost to get from the initial board to this board. (May be set to 0.)

and has the following methods:

1. `__init__(self, board, parent, depth, f_value = 0)`: 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. `__lt__(self, other)`: Checks if a state's fValue is less than that of another state.
3. `__eq__(self, other)`: Checks if two States contain the same Board. This does not compare parents or depth.
4. `__str__(self)`: The string representation of a given state.
5. `__repr__(self)`: A string of code to produce a given state.
6. `printPath(self)`: Prints the path to the given state.

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:

1. `uninformed_solver(start_board, max_depth, goal_board)`
   Looping function that calls `bfs()` until a goal state has been found or until STOP has been returned. It does not consider States below `max_depth`. If the goal is reached, this function should return the Goal State, which includes a path to the goal. Otherwise returns None.

   - `start_board`: The start board.
   - `max_depth`: The maximum depth of searching for the Goal State.
   - `goal_board`: The solved board.

2. `findManhattanDist(current_board, goal_board)`
   Function which uses the Manhattan Distance heuristic to give values to the States. This is an example heuristic function that we have provided for testing..

   - `current_board`: The current board for which we are calculating a heuristic value.
   - `goal_board`: The solved board.

3. `informed_solver(current_board, goal_board, f_function)` Looping function that calls `informed_search()` until it finds a solution (a State object) or until STOP has been returned. If the goal is reached, this function should return the Goal State, which includes a path to the goal. Otherwise, returns None.

   - `start_board`: The starting board.
   - `goal_board`: The solved board.
   - `f_function`: Either the `ucs_f_function()` or `a_star_f_function_factory()` which allows this looping function to consider either Uniform Cost Search or A* Search when testing.

4. `ucs_solver(current_board, goal_board)` Function which performs the `informed_solver()` function recursively using Uniform Cost Search.

   - `start_board`: The starting board.
   - `goal_board`: The solved board.

5. `a_star_solver(current_board, goal_board, heuristic` Function which performs the `informed_solver()` function recursively using A* Search.

   - `start_board`: The starting board.

- `goal_board`: The solved board.
- `heuristic`: A heuristic function, i.e. either the given Manhattan Distance function or your own implemented heuristic found in `new_heuristic()`.

3.2 Objective

Your goal is to provide the implementations to the following functions:

1. `fringe_expansion`
2. `bfs`
3. `ucs_f__value_function`
4. `a_star_f__value_function_factory`
5. `my_new_heuristic`
6. `informed_expansion`
7. `informed_search`

to the following specifications:

3.2.1 fringe_expansion(current_state, fringe)

Write a function that adds the possible states that we can get to from the current state to the end of the fringe. This function should not return or yield anything but just update the contents of the fringe.

- `current_state`: The current state that is to be used to generate children.
- `fringe`: A list that holds the states that have been added to the fringe.

3.2.2 bfs(fringe, max_depth, goal_board)

Write a function that implements a single iteration of the BFS algorithm by considering the first state from the fringe; **NOT** the whole thing.

1. Return `STOP` if the fringe is empty.
2. If the depth of the current state is greater than the `max_depth`, skip the current state and continue searching.
3. Return the current state its board is the `goal_board`.
4. Otherwise expand the `fringe` and continue.

3.2.3 ucs_f_value_function(board, current_depth)

Write a function that takes a board and depth and returns the f-value that board should have in a uniform-cost search scenario.

3.2.4 a_star_f_value_function_factory(heuristic, goal_board)

Given a heuristic function and a goal board, returns an f-value FUNCTION that takes a board and a depth and returns an f-value, as in the A* algorithm. (Using the `lambda` keyword here may be helpful.)

3.2.5 new_heuristic(current_board, goal_board)

Write a function that takes `current_board` and `goal_board` as arguments and returns an estimate of how many moves it will take to reach the goal board. Your heuristic must be admissible (never overestimate the cost to the goal), but it does not have to be consistent (never overestimate step costs). It should always return a value ≥ 0, be a more accurate estimate than (be greater than on average) the Manhattan Distance heuristic, and function competently for A* searches. It may not correlate linearly with the Manhattan Distance.

3.2.6 informed_expansion(current_state, fringe, f_function

Write a function that expands the fringe using the f-value function provided. Note that States automatically sort by their f-values. This function should not return any value but just update the contents of the fringe (HINT: Use heap).

3.2.7 informed_search(fringe, goal_board, f_function, explored)

Write a function that implements a single iteration of the A*/UCS algorithm by considering the top-priority state from the fringe. (NOTE: explored is a dictionary mapping boards to their lowest encountered f-values.)

1. If the fringe is empty then stop.
2. Otherwise, get the top state from the fringe.
3. If the current state's board has already been seen and the current state's f-value is not smaller than the previous f-value, then skip this state and continue.
4. Add the current state's board and its f-value to the explored dictionary.
5. If the current state is the `goal_board`, return the state.
6. Otherwise expand the fringe and continue.

4 Tests

Very simple tests have been given for you in A3.py. Running this file will run the tests. Be sure to use these to make sure your implementations are working generally as requested. These test cases are NOT all encompassing. Passing these does not mean your implementation is completely correct. Each test uses a puzzle that is only 2 moves away from the goal. In order to save submissions,

it is advised to write additional unit tests for more complex puzzles as well as to cover cases that these tests do not. One great example is the Competency section of the rubric. This is not covered through the given tests.

5 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

Finally, remember that this is an individual assignment.

5.1 fringe_expansion (10%)

Criteria	**Points**
Tries to make a child board for each possible move	2
Ignores moves that did not successfully produce a child state	1
Initializes child states with the appropriate board and depth	5
Adds each new child state to the end of the fringe	2
Total	10

5.2 bfs (10%)

Criteria	**Points**
Stops when the fringe is empty	1
Pops the first state in the fringe: the current state	3
Ignores states deeper than max depth	2
Returns the goal state when found	2
Appropriately expands the fringe	2
Total	10

5.3 ucs_f_value_function (5%)

Criteria	**Points**
Returns the appropriate f-value for a Uniform Cost Search	5
Total	5

5.4 a_star_f_value_function (10%)

Criteria	Points
Returns a function that accepts a board and a depth and returns an f-value	3
The function returned applies the given heuristic to the board provided at run-time and the given goal board	5
The function returned produces the appropriate f-value for an A* search	2
Total	10

5.5 new_heuristic (15%)

Criteria	Points
Returns numerical values; does not call Manhattan distance; does not reimplement Manhattan distance; is not always zero	required
Never returns a value below zero	2
Is admissible (never overestimates distance to goal) with respect to the standard goal board	5
Is admissible given any possible goal board	3
Returns a value that is closer to the actual distance than (greater than) the Manhattan distance, on average	5
Total	15

5.6 informed_expansion (15%)

Criteria	Points
Tries to make a child board for each possible move	2
Ignores moves that did not successfully produce a child state	1
Initializes child states with the appropriate board and depth	5
Initializes child states with the appropriate f-value	4
Adds each child state to the fringe using heappush	3
Total	15

5.7 informed_search (10%)

Criteria	Points
Only performs a single search iteration	required
Stops when the fringe is empty (returns STOP)	1
Uses heappop to get the highest priority state from the fringe	1
Ignores states whose boards have already been explored, if they do not have a lower f-value than before	2
Properly records current board and f-value in explored	2
Returns the goal state when found	2
Appropriately expands the fringe	2
Total	10

5.8 Competency (25%)

Criteria	Points
Recognizes goal board	5
Finds optimal path for boards that are less than five moves away from the goal	5
Finds optimal path for boards that are less than ten moves away from the goal	5
Finds optimal path for boards that are less than fifteen moves away from the goal	5
Finds optimal path for boards that are less than twenty moves away from the goal	5
Total	25

6 Bonus (10%)

Implement IDS in any way you choose. You will want to write multiple helper functions; be sure to document these appropriately.

6.1 ids(start_board, goal_board, final_depth)

Takes a start board and goal board and then performs multiple depth-first searches, with the maximum depth increasing from 0 all the way to final depth. If there is a solution within `final_depth` moves, then `ids` should return the found board. Otherwise, return None.