CSE 368: Assignment 1

Introduction

In this assignment you will use both uninformed and informed search algorithms to solve a slide puzzle. You will be implementing graph search, iterative deepening Depth-First Search, A* tree search, and A* graph search. The only file you will need to implement is searches.py; slideproblem.py contains class definitions and some helper functions you'll be using.

Slideproblem.py

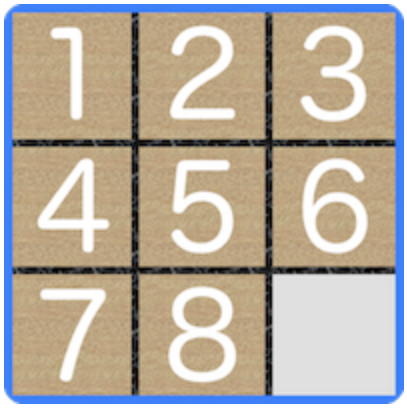
First, we will talk about the classes and some useful helper functions in slideproblem.py. Recall, that in search problems we have the following:

- Initial State: $s \in S$
- Actions: $A \subset S \times S$
- Cost: Cost(s, a, s') -> $\mathbb{R}$
- Function to find applicable actions -> Actions(s): {a1, a2,…, an}
- Function that returns result of action -> Result(s, a) -> s
- Goal Test: Test for completion

The class **problem** is what defines a search problem and is essentially what the above portion is (with the exception of cost, which will be covered when we talk about Node). Here are some useful methods from Problem:

- apply(self, a, s): This function applies an action a, in some state s and returns our new state. **For this problem, the only valid actions are 'U', 'L', 'R', and 'D', which move the empty space up, down, left, and right respectively**. Essentially, it's a function that returns the result of an action given an action and a state.
- applicable(self,s): This function returns a list of valid actions that can be done in some state s.
- goalTest(self,s): This function just checks if the provided state is equal to the goal state. Returns true if it is, false otherwise.

The class **state** is what defines the state of our board at any given time.



The above image is an example of what our state can be. If you attempt to print a state object, you would get something as follows:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Note that the 0 represents an empty space in this grid format. Lastly, state contains an important method called *toTuple()*. What this method does is it prints out the grid format above, but in tuple form. So we would get ((1, 2, 3), (4, 5, 6), (7, 8, 0)) returned. This will be useful when you implement BFS and need to see if a node has been explored already. Just check if this state exists in your explored set.

The class **node** is what defines a node in your search tree. The variable nodeCount just maintains how many nodes have been created in your tree. Invoking the print function on a Node object will print the node's ID, it's cost, and the **state**. The constructor of Node takes 4 arguments, a parent (of type Node), an action which denotes the action the parent has to take to get to the child, a cost which is the cost of reaching that node, and a state which is the state of our node after the action has been applied.

Lastly, there are two methods that are not apart of any classes, but reside in the file itself. One method is *child_node(n , action, problem)*. What this does is it creates a child node for a node n, based on the action variable. So suppose we wanted to create a child from node A. If we perform an action, it would be something like child = child_node(A, 'U', problem). Note that all the methods you will be implementing have a 'problem' field, so that will be passed in. The next method is *solution(node)* which returns two things, the list of actions that bring you from node to the solution, and the cost to reach the solution.

Uninformed Search

(20 points) Breadth-First Search: For implementing graph search you will edit the method graph_bfs(self, problem).

Iterative Deepening Depth-First Search: For implementing IDDFS, you will be editing a couple of methods.

- *recursiveDL_DFS(self, lim,problem):* This method you don't have to edit. Note that it calls on depthLimitedDFS with an initial state, a limit and problem object provided by the user.

- ***(20 points)*** *depthLimitedDFS(self, n, lim, problem)*: This is where you will implement a depth limited version of DFS. This function should be recursive and when you explore the nodes of the children, make sure you lower the limit by 1.

- ***(20 points)*** *id_dfs(self,problem)*: This is where you implement iterative deepening depth-first search. Here you will call *recursiveDL_DFS(self, lim,problem),* while increasing your depth, d. Once you have reached the max depth (which you will define) you will stop searching.

Informed Search

- **(20 points each: Total 40 points) A* Search:**
  - A* Tree:
    a_star_tree(self, problem): This is where you will implement A* tree search. Here you will call heuristic function. You will need to use an appropriate data structure, such as heapq. You will also need sorting functions, such as sort.
  - A* Graph:
    a_star_graph(self, problem): This is where you will implement A* graph search. It's similar to a_star_tree, except that you will need to keep track of visited nodes if you don't want to process them multiple times.

(Extra Credit) 4*4 slide

- **(20 points)Adapt the code to solve 4*4 slide problem:**
    - solve4x4(p : problem): This is the function you will add to your code which should solve the larger board. You can use any search method you want, but you must add a new function solve4x4(p: Problem), which should solve the larger board. Solving the 4x4 can be very time consuming, if you're still using the same heuristics as before. Think of new heuristics to implement for this problem, also, your results will not be graded if they don't find the answer in under a minute. Lastly, separately submit this file to autograder. It will not be autograded; rather, manually graded by TAs.