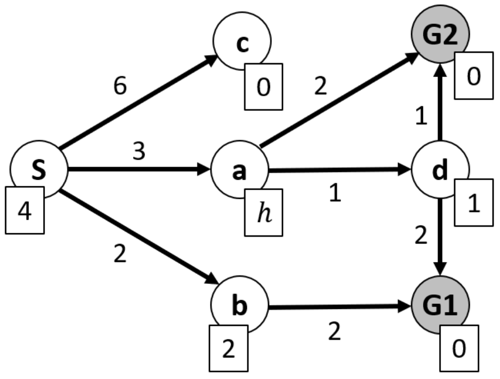
COMS 4701: Artificial Intelligence

Homework 2

Due: October 1, 2019

1 Search, Search, Search! (15 points)

In the search graph below, S is the start state and G1 and G2 are the goal states. Costs are shown along the edges and heuristic values are shown in the rectangles next to each node. Assume that search algorithms expand states in alphabetical order when ties are present. All search algorithms implement graph-search and finish after expanding the first goal state (not both).



(a) List the order of the states expanded by iterative deepening search starting with $l = 1$ (corresponding to state S), and clearly label the iteration (max depth level) to which the sets of expanded states correspond. **Note that states may be expanded multiple times in different iterations.** Give the path solution returned. Is it optimal?

(b) Give the range of non-negative values for the missing heuristic h that would make the overall heuristic function admissible. Do the same to make the heuristic function consistent. Be sure to justify why your values in each case are correct.

(c) Suppose $h = 0$. List the order of the states expanded and the path solution returned by A* search. Is it optimal?

2 Cryptarithmetic (20 points)

Cryptarithmetic problems can be composed of all sorts of words, like the ones below:

$$\begin{array}{cccc} & & G & O \\ + & & T & O \\ \hline & O & U & T \end{array}$$

Recall that a solution is an assignment of a unique number between 0 and 9 (inclusive) to each letter. The two two-digit numbers represented by GO and TO must also add up to the three-digit number represented by OUT, and the leftmost digits of all numbers cannot be 0. Do not consider carryover variables in this problem.

(a) List all the variables and their domains for this problem after applying unary constraints (domain reduction) **only**. Note that we are not yet making any formal assignments.

(b) Make a LCV assignment to the MRV variable and perform forward checking to prune the domains of the variables sharing a constraint with the assigned one (don't forget about $Alldiff$ constraints!). Indicate the variable you assigned and the remaining domains of the rest.

(c) Repeat the above step for the next MRV variable.

(d) Finally, write down a binary constraint between the two remaining unassigned variables and perform arc consistency between them to remove all invalid domain values. Make the remaining variable assignments and give the final solution.

3 Tic-Tac-Twist (15 points)

Two players are playing a modified tic-tac-toe game in which grid spaces have point values. X seeks to maximize the total score, and O seeks to minimize it. When the game is over, the values of the X spaces add to the score, and the values of the O spaces subtract from the score. In addition, a win for X is worth an additional +3 points, a win for O is worth an additional −3 points, and a draw is worth an additional 0 points.

2	5	2
5	1	5
2	5	2

X		
X		O
O	O	X

The point values for the grid spaces are shown on the left above. The right shows the current board. If the game were to end at this point without any win or loss, the total score would be −3.

(a) It is the X player's turn to make a move. Draw out the entire game tree with the current board state as the root. Be sure to clearly show MAX nodes, MIN nodes, and terminal nodes. Terminals correspond to game states in which either a player has won or a board that is filled up. In addition, sketch out the game state for each terminal node (you can just draw them below the tree leaves), and write down the overall score of each.

(b) Propagate the terminal values up through the nodes of the tree and indicate the values of each node as would be returned by the minimax algorithm. What is the best action for X to take, and what is the expected score when the game ends?

4 n-Puzzle (50 points)

You will implement and compare search strategies for solving the n-puzzle. The idea is that you have n tile pieces numbered 1 to n on a square grid with $n+1$ spaces. There is also a single empty grid space, represented in this program with 0. Starting with an arbitrary configuration of the tiles, the game is to slide the tiles around via the empty space (equivalently, swap the empty space with an adjacent tile) until the tiles are ordered 0 to n, going left to right and then top to bottom.

We have provided skeleton code in the `npuzzle.py` file. You will modify this file by filling in the pre-defined functions step by step. States are represented as tuples of tuples. Inner tuples represent a given row of the grid. For example, the state below would be represented as `((1,4,2),(0,5,8),(3,6,7))`. When you first run the program, you should see the following:

```
python npuzzle.py
1 4 2
0 5 8
3 6 7

===BFS===
Path to goal: None
Cost of path: 0
States expanded: 0
Max frontier size: 0
Total time: 0.000s
```

The first part of the output is a representation of an initial 8-puzzle state. The rest of the output displays information about the performance of the search algorithm. Path to goal is a list of actions taking the initial state to the goal; cost of path is the length of that list (assuming uniform costs). We also track the total number of nodes expanded in the search process, the maximum size of the frontier over the entire search, and the runtime of the algorithm.

There are also a few utility functions. Two that you may find useful for your search implementations are `get_successors()` and `goal_test()`. The former returns a list of valid (action, state) tuples from the current state (actions are one of four strings in fixed order: "Left", "Right", "Up", or "Down"). The latter returns True if the queried state passes the goal test and False otherwise.

4.1 Breadth-First Search (15 points)

Complete the function `bfs()`. It should implement breadth-first search starting at the initial state passed in as an argument. Your algorithm should maintain a frontier and an explored set. For BFS, the frontier should act as a queue; you can use a Python list implementation and the corresponding `append()` and `pop()` methods. Each queue item corresponds to a state, which in turns keeps track of the state's parent and the corresponding action. You are free to represent this information in any way you wish—one efficient way is to use a separate "parents" dictionary with a child state as key and a (parent, action) tuple as value, so that queue items are states only.

You should implement the algorithm according to the `Graph-Search` pseudocode provided in class. That is, a node should be placed in the queue only if it is not already in the frontier or explored set. (You will likely find that the hard test case will not even complete in Python if using a tree-search implementation.) When the goal state is found, follow each successive node's parent all the way back up to the initial state to return the full path.

Important: Testing list membership in Python takes $O(n)$ time, which can significantly slow down your search when checking against the frontier during expansion. Testing set membership (*e.g.*, the explored set) is much faster, taking about $O(1)$ time due to Python hashing optimization. We therefore recommend that you maintain a "frontier set" in parallel to the actual frontier queue, whose sole purpose is to allow for fast frontier membership testing.

4.2 Depth-First Search (10 points)

Complete the function `dfs()`. It should implement depth-first search starting at the initial state passed in as an argument. The change from your BFS implementation should be minimal—simply use a stack instead of a queue. With a Python list, you can change the location in which you pop items as compared to the queue implementation. The same note regarding a "frontier set" in addition to the actual frontier stack applies from above.

4.3 A* Search (25 points)

For A* search, we will use two heuristic functions: `misplaced_heuristic` and `manhattan_heuristic`. Complete both of these functions so that they take in a given board state and return the corresponding heuristic value. The former simply sums the number of tiles that are out of place as compared to the goal state; the latter sums all Manhattan distances between the misplaced tile locations and their desired locations.

These heuristic functions will be passed in as arguments to `astar`, which you will complete. Since both heuristics are consistent, you can continue implementing a `Graph-Search` algorithm with an explored set. To implement the frontier as a priority queue, you can use the `heapq` module, which provides the functions `heappush()` and `heappop()`. Since the priority of an item in the heap will require both the cumulative cost and the heuristic value of the state, you will also need to keep track of costs in the search nodes (in addition to parent and action).

Important: In class we discussed the possibility of updating the priority of a node in the frontier if it is added again later. The bad news is that `heapq` does not allow for an easy way to do this. The good news is that you would never encounter this situation for n-puzzle. Since all costs are uniform, the first time that a state is added to the frontier also corresponds to the cheapest path to get there. So you can safely go without a priority comparison check in your implementation.

4.4 Finishing Up

As you finish implementing each individual algorithm, you can uncomment the corresponding blocks of code in `__main__` for testing. For submission, please uncomment all code blocks there and run your program twice, once for the "easy" test case and once for the "hard" one. The optimal costs that you should be seeing are 7 and 26, respectively. Copy/paste the entire output from both tests into your writeup. Submit both the writeup with the code output and your solutions for the written problems as well as your completed `npuzzle.py` file on Gradescope. **Please be sure to author your codefile in the comments at the top of `npuzzle.py` before submission.**