CS534
Assignment #1: search
Due: February 10, 2023 @ 11:59pm

This assignment will require you to explore a variety of search techniques variants of the (modified) N-puzzle. Parts 1 and 2 are fairly straightforward application of class concepts. Part 3 requires more thought. Note: in addition to the programming component of developing the algorithms, there is a moderate amount of analysis/writeup required as well. If you finish your code just before the deadline, **you will not finish the assignment**. Start right away.

Note: this assignment is completely new, so there could be parts that are poorly written. Please contact me right away for clarifications.

The modified N-puzzle

The puzzle your program will solve is extremely similar to the N-puzzle we discussed in class. There is a square grid of size N by N. Each grid space is either blank or has a distinct number [1, N^2-1]. A valid move is to slide a tile into an adjacent blank square. To change things up, there are some important differences from the original N-puzzle:

1. There *can be* multiple blank squares. You are guaranteed to have at least 1 blank square and at least 1 square with a number on it.
2. Moves have variable cost. The cost to move a tile is equal to the number on the tile.
3. There are multiple goal states. You can either have the tiles towards the top-left of the puzzle (so the blanks are at the end), or towards the bottom-right of the puzzle (so the blanks are at the beginning). All of the blanks must be together; blanks have no particular order amongst themselves.

Here is a sample puzzle, where B represents a blank square.

3 B 7
B 2 1
6 8 5

Legal moves are sliding the 3 tile to the right, the 7 tile to the left, the 2 tile either left or up, and the 6 tile up. To slide the 6 tile up costs 6; to slide the 3 tile down costs 3.

Here is one possible solution to the above puzzle:

1 2 3
5 6 7
8 B B

Here is the second possible solution:

B B 1
2 3 5
6 7 8

Part 1: A* search

You will solve the modified N-puzzle using A* search. You will experiment with the teleporting tiles and the sliding tiles heuristics discussed in class. You will create versions of those heuristics that consider the tile's weight and those that ignore it.

What your program should input and output

Your program should accept as **command-line arguments**:

1. The file name of the board to read in
2. Which heuristic to use (sliding, greedy (Part 3))
3. Whether the heuristic should take into account tile weight (true vs false)?

For example:
npuzzle board1.txt sliding true
Would read in board1.txt, use the sliding tiles heuristic, and take into account tile weight. Not following this protocol will cost you points. I have to grade the assignments myself. Make my life easy.

Your program should output:

1. The sequence of moves to solve the puzzle in human readable form. Since there can be multiple blanks, you cannot simply give a number (as there could be multiple blank squares a single number could move into). Instead, for the sample board above, write something like "2 up" or "2 left" to make it clear which direction the 2 tile is moving.
2. The number of nodes expanded in your search.
3. How many moves the solution required
4. The cost of the solution
5. The estimated branching factor

Analysis for A*

Consider the sliding tiles heuristic where it makes use of tile weight and when it ignores it.
Experimental approach:

1. Pick a board size your program can solve in about 30 seconds.
2. Have about 20% of the squares be blank. It is fine to simply approximate. For example, for a 6x6 = 36 squares * 20% = 7 blanks.
3. Perform 10 runs and take the average performance.

Questions:

1. What is the branching factor when tile weight is used? When it is not used?
2. How does solution quality differ between the two versions of the heuristic?
3. How does the branching factor differ between the two heuristics?
4. Why does accounting for weight result in a lower branching factor? Explain it in terms of the A* search.
5. Does the heuristic that accounts for weight dominate the unweighted heuristic?

Part 2: Hill climbing

You will now write a program that solves the modified n-puzzle using greedy (hill climbing) search. Your program has a limited amount of time, so must budget its time wisely. Taking the move that takes it closest to the solution is usually the best option. But a mix of annealing, sideways moves, and restarts will result in better performance. Remember, the goal is to solve the board efficiently, not spend a large percentage of the search time very close to a solution but never quite getting there!

Your program should accept as **command-line arguments**:

1. The file name of the board to read in
2. How long to run for.

For example:
greedy board2.txt 12
Would read in board2.txt and spend up to 12 seconds trying to solve it. Greedy search can balance speed and solution quality. Tweaking your search to use its allotted time to find a better solution will be viewed favorably.

As with the A* search, your program should output:

1. The sequence of moves to solve the puzzle in human readable form. Since there can be multiple blanks, you cannot simply give a number (as there could be multiple blank squares a single number could move into). Instead, for the sample board above, write something like “2 up” or “2 left” to make it clear which direction the 2 tile is moving.
2. The number of nodes expanded in your search.
3. How many moves the solution required
4. The cost of the solution
5. The estimated branching factor

Writeup for greedy search

1. Explain how you decided to have your program invest its time. Provide some data for how you balanced annealing, restarts, and just greedily moving towards the goal.

2. How large of a board could your greedy search solve? How does its runtime compare to A* on identical-sized boards?
3. How good of a solution did your greedy search find? How much worse was it than A*?

Part 3: Putting it all together

As you saw in Part 1, A* works more efficiently with a more accurate heuristic. For Part 3 you will experiment with using greedy search to create a more accurate heuristic for your A* search. The notion of a heuristic function is a broad one; a heuristic function is anything that is: 1) simpler to compute than the search itself, and 2) helpful in guiding the search. A handcrafted function is one approach, but there are others.

Consider a greedy search. It quickly finds a solution (or a near solution). Could that solution be used as a cost estimate for A* search, and in effect, a heuristic function? You will take your search from Part 2 and simplify it to use as a heuristic for Part 3. Remember, the goal of using this greedy search is not to directly find a good *solution*. It is simply to quickly find a good *next move*. Therefore, you (probably) do not want to make use of restarts or annealing.

One simple approach is to do a greedy search until it reaches a local maxima. Once a local maxima is found, you have the cost of the greedy search to get to that point, and the estimated cost to get to the goal. There are more complex approaches you can try as well, but those are left for you to think about.

Writeup for Part 3

Explain how you developed your greedy-based heuristic. What approach did you use? Why?

Compare your greedy heuristic to the sliding tiles (with weight) from Part 1. Compare them in terms of:

1. Solution speed: how long did the search take to complete?
2. Branching factor
3. Solution quality: did one find better solutions than the other?
4. Heuristic speed: how long does each heuristic take to compute?

Is your greedy heuristic admissible? Would you recommend using the greedy heuristic instead of sliding tiles?

Turning in your assignment

Assignments should be submitted via the course Canvas page. Please submit your:

1. Source code
2. Instructions for running it
3. Writeup