COMPSCI 383 – Fall 2020

Homework 1

Due Friday, September 18th at 5:59pm ET

You are encouraged to discuss the assignment in general with your classmates, and may optionally collaborate with one other student. If you choose to do so, you must indicate with whom you worked. Multiple teams (or non-partnered students) submitting the same code will be considered plagiarism.

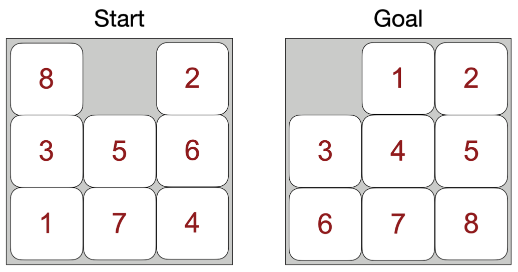
Code must be written in a reasonably current version of Python (>3.6), and be executable from a Unix command line. You are free to use Python's standard modules for data structures and utilities, as well as the pandas, scipy, and numpy modules if you really want, but you must write the code for the search algorithms yourself.

Solving a Weighted 8-puzzle Problem with Search

The goal of this assignment is to deepen your understanding of different search strategies. You will be writing a solver for a modified version of the 8-puzzle problem mentioned in class. The 8-puzzle consists of eight tiles on a 3x3 grid, with one open ("blank") square. The goal state consists of the open square in the upper left, with the other tiles arranged in numeric order from left to right.

Valid moves are Up, Down, Left, and Right, which shift a tile into the open square. Depending on the position of the open square, not all of those moves may be available — in the example below, the valid moves from the start state are Up, Left, and Right.

We will modify the problem to include different path costs between states. In our problem, the cost of transitioning from one state to the next is equal to the square of the number on the tile that gets shifted. For the start state above, moving the Up move has a cost of 25, while the Left move has a cost of 4.



Your solver must take a start state as input, perform a search over the state space, and return a solution path to the goal states when possible. In addition, your program will track the efficiency of the search process and report back performance metrics (more details below).

Search Strategies

For this assignment, you must implement **four** different search strategies: Breadth-First, Uniform-Cost, Greedy Best-First, and A*. As discussed in class, these methods (especially the latter three) are quite similar, and you are strongly encouraged to structure your solution in a way that reuses code.

For Greedy Best-First and A*, you should use a modified version of the misplaced tile Manhattan distance (see AIMA p. 103) that takes the different transition costs into account. For the start state depicted above, there are five misplaced tiles. The weighted Manhattan distance would be calculated as follows:

> Tile 1: 1^2 x 3 (one right, two up) = 3
> Tile 2: 0 (already in place)
> Tile 3: 0 (already in place)
> Tile 4: 4^2 x 2 (one left, one up) = 32
> Tile 5: 5^2 x 1 (one right) = 25
> Tile 6: 6^2 x 3 (two left, one down) = 108
> Tile 7: 0 (already in place)
> Tile 8: 8^2 x 4 (two right, two down) = 256
> Total: 424

In addition, to avoid looping searches, you'll want to make sure you follow the Graph-Search paradigm as shown on p.77 in AIMA.

An Open Question

In designing this assignment, it was unclear to the course staff whether the introduction of unequal transition costs would affect the solutions produced (something to ponder: what does this say about the topology of the state space?). We are going to investigate this issue empirically as a class. To do so, your solver must be able to run in "unweighted" mode, where the transition cost for all moves is 1 and the heuristic used is traditional Manhattan distance. You'll report your findings about the hypothesis that the weights do not affect the solution.

Command Line Format

The solver should all be callable from a single file, called `solver.py`. The start state and search strategy will be specified by command line arguments[1] in the following order:

- A keyword specifying which search strategy/heuristic to use (one of `bfs`, `ucost`, `greedy`, or `astar`)
- The start state specified as a nine digit string, with `0` denoting the blank square. The first three digits represent the top row, the next three the middle row, and the last three the bottom row. For the Start State depicted above, the command line representation is `802356174`.
- An optional `--noweight` flag that will instruct the program to use equal transition costs and an unweighted Manhattan distance heuristic (if applicable)

For example, to execute a Greedy Best-First search using the weighted transition costs and distance heuristic:

```
> python solver.py greedy 802356174
```

[1] See https://stackabuse.com/command-line-arguments-in-python/ for a good primer

Output Format

Solutions will be specified in a similar manner, but printed to stdout. For each state on the solution path, print out the iteration number, the move made (one of `up`, `down`, `left`, or `right`), and the nine-digit board representation, all separated with tab characters ("\t").

Additionally, you must print out the following search efficiency metrics:

- Total path cost of solution
- Total number of states added to the frontier queue
- Total number of states popped from the frontier queue and expanded

For example:

```
0     start   312458607
1     left    312458670
2     down    312450678
3     right   312405678
4     right   312045678
5     down    012345678
path cost: 163
frontier: 121
expanded: 71
```

If there is no solution for a particular start state, your program should output `no solution`, followed by the performance metrics. To avoid overly long runtimes, you may halt your search after expanding 100,000 nodes and print `search halted` followed by the performance metrics.

Your solution should run in a "reasonable" amount of time (less than ten minutes for finding a path of ten moves or less). The public tests on Gradescope will check for this to let you know if you've engineered things correctly.

Supplied Code

To get you started, we've included a Python module that will handle representing the 8-puzzle board called `puzz.py`. Once you create an 8-puzzle object, you can generate successor states using the different methods. Sample usage from an interactive Python session:

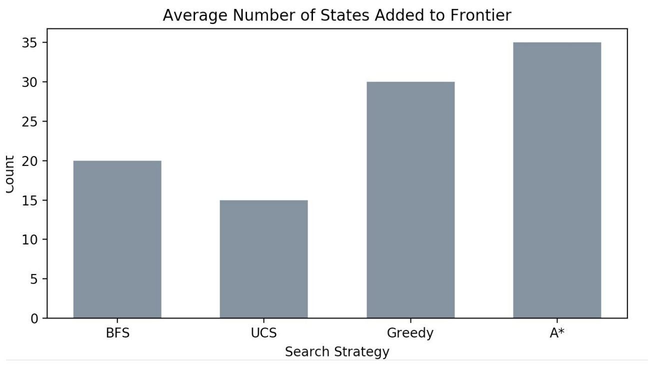
```
>>> import puzz
>>> b = puzz.EightPuzzleBoard("123470568")
>>> b.success_up()
123478560
>>> b.success_right()
123407568
```

```
>>> b.success_up().success_right()
123478506
>>> b.success_up().success_up()   # evaluates to None
>>> b.successors()
{'up': 123478560, 'down': 120473568, 'right': 123407568}
```

In addition, the file `pdqpq.py` contains an implementation of a priority queue to use for your best-first search algorithms.

Testing and Presenting Results

Test your solver on at least ten non-trivial (> 10 moves) and solvable test cases. You might consider generating these programmatically, by starting with a puzzle in a goal state and randomly perturbing it using some number of legal moves. Run your solver using each search strategy, and keep track of the efficiency metrics. (for producing these results, do not use the `--noweight` flag)



Using those results, produce three bar graphs (by hand or programmatically[2]), one for each search efficiency metric, showing the *average* values for each search strategy (see the example above, made with fake results data).

Next, try to answer the research question about whether the unequal transition costs affect the solutions produced. To do so, run your solver using A* search on your test cases, this time including the `--noweight` flag and checking to see whether the solutions differ.

Grading

We will run your program on a variety of test cases. Your grade will be proportional to the number of test cases you pass. The test cases for grading will not be available to you before grading, but we will make tests available that will check the format of your output.

[2] https://www.geeksforgeeks.org/bar-plot-in-matplotlib/

Note that your solutions will only be tested on well-formed boards. We are not going to feed your program incorrectly formatted input, so you need only concern yourself with handling input in the format described in the assignment.

Code that does not run will not receive credit.

What to Submit

Submit any code required to run your solver (`puzz.py`, `pdqpq.py`, `solver.py`), a `readme.txt`, and a pdf containing your comparison plots called `plots.pdf`.

The `readme.txt` file should include:

- Your name(s)
- The best meal(s) you've ever eaten
- Your conclusion about whether including the path costs affects the solutions found. If you don't think they do, hypothesize why in 1-2 sentences. Otherwise, provide at least one test case where A* finds a different solution path when the `--noweight` flag is used.
- Notes or warnings about what you got working, what is partially working, and what is broken, and any other feedback you have on the assignment