COMPSCI 383 – Spring 2020

Homework 1

Due Thursday 2/13/2020 at 11:59pm

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.

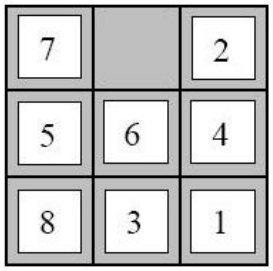
Code must be written in a reasonably current version of Python (>3.0), and be executable from a Unix command line. You are free to use Python's standard modules for data structures and utilities, as well as the pandas, scipy, and numpy modules.

Solving the 8-puzzle Problem with Search

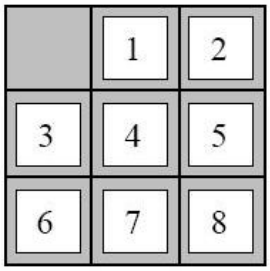
The goal of this assignment is to deepen your understanding of different search strategies. You will be writing a solver for a modified version of the 8-puzzle problem mentioned in class. The 8-puzzle consists of eight tiles on a 3x3 grid, with one open ("blank") square. In the classic problem, the goal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

In this assignment, you will consider a modified version of the problem where the goal state can be any arbitrary configuration. Your solver must take a start state and goal state as input, perform a search over the state space, and return a solution when possible.



Start State



Goal State

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 will be required to implement **two** different heuristic functions: misplaced tile count, misplaced tile Manhattan distance. For the example above, the misplaced tile count for the Start State is 7 (all tiles except #2 are out of place), and the Manhattan distance is 15 (3+0+2+1+2+2+3+2).

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.

Command Line Format

The solver should all be callable from a single file, called `solver.py`. Your program will take the start and goal states as command line arguments[1], each specified as a ~~ten~~ *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 `702564831`.

Your program should expect the following command line arguments (in order):

- A keyword specifying which search strategy/heuristic to use, assumed to be one of "bfs", "ucost", "greedy-count", "greedy-manhat", "astar-count", or "astar-manhat"
- The start state in the format described above
- The goal state in the format described above

For example, to execute a Greedy Best-First search using the Manhattan distance heuristic:

```
> python solver.py greedy-manhat 702564831 86753012
```

Output Format

Solutions will be specified in a similar manner, but printed to stdout. For each state on the solution path, print out the move made (one of "up", "down", "left", or "right") followed by the nine-digit board representaton, with a tab character ("\t") in the middle. For example:

```
start 312458607
left  312458670
down  312450678
right 312405678
right 312045678
down  012345678
…
```

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

- Total path cost of solution (each step has a cost of 1)
- Total number of search nodes added to the frontier queue

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

- Total number of search nodes selected from the frontier queue for expansion

For example:

```
path cost: 47
frontier: 19
expanded: 13
```

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.

Supplied Code

To get your 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, 'left': None,
'right': 123407568}
```

Testing

You should test your solver on a variety of test cases. You might want to generate these programmatically, by starting with a puzzle in a goal state and randomly perturbing it using some number of legal moves. Good test cases will show differences between the different search strategies in terms of performance or optimality. You are also welcome to share your test cases on Piazza, along with the output stats for your solver.

What to Submit

You should submit any code required to run your solver (`puzz.py`, `solver.py`), and a `readme.txt`.

Your readme.txt file should indicate:

- Your name(s)
- The best meal you've ever eaten
- A short (3-4 sentence) summary of findings from examining the performance metrics on different test cases. Which methods are most efficient? How does the choice of heuristic affect results?
- Notes or warnings about what you got working, what is partially working, and what is broken

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.

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.