COMPSCI 383 – Spring 2022

Homework 1 Coding

Due Friday, February 18th at 9pm 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.

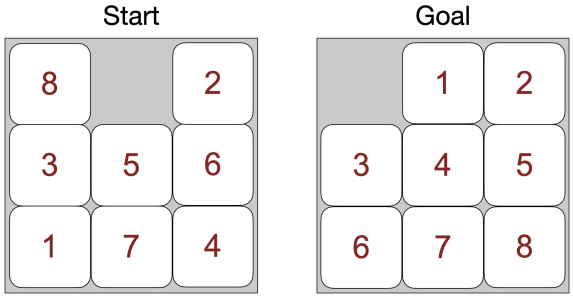
Code must be written in a reasonably current version of Python (>3.7), and be executable from a Unix command line. You are free to use Python's standard modules for data structures and utilities, as well as the pandas, scipy, and numpy modules if you want, but you must write the code for the search algorithms yourself.

Solving a Weighted 8-puzzle

The goal of this assignment is to deepen your understanding of different search strategies, and understand the ramifications of using different heuristics. You will be writing a solver for a modified version of the classic 8-puzzle problem mentioned in lecture. The 8-puzzle consists of eight tiles on a 3x3 grid, with one open ("blank") square. The possible configurations of tiles comprise a state space, where the goal state has 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 classic problem to include different transition costs between states. In our version, the cost of moving from one state to the next is equal to the square of the number on the tile that gets shifted. For example, in the start state on the right, the Up move has a cost of 25 (5×5), while the Left move has a cost of 4 (2×2).



Your solver must take a start state as input, perform a search over the state space, and return a solution path to the goal state 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 **three different search strategies**: Uniform-cost, Greedy best-first, and A*. The supplied code includes an implementation of Breadth-first search as an example.

For Greedy best-first and A*, you must implement **three different heuristic functions** to estimate the cost of the shortest path to the goal. The first two are the "number of misplaced tiles" and "Manhattan distance" heuristics described on p. 103 of AIMA (h_1 and h_2, respectively). The third heuristic (h_3) is a modified version of Manhattan distance that takes the different transition costs into account. For the start state depicted above, the value of h_3 is computed as follows (note that the blank is not counted in the cost):

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

To avoid looping searches, you'll want to make sure you follow the Graph-Search paradigm as described in Lecture 1.

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 `successors()` method. Sample usage:

```
import puzz
state = puzz.EightPuzzleBoard("123470568")
print(state)
print(state.pretty())
succs = state.successors()
print(succs)
d = succs['down']
print(d.pretty())
```

The above produces the output:

```
123470568
1 2 3
4 7 .
5 6 8
{'up': 123478560, 'down': 120473568, 'right': 123407568}
1 2 .
4 7 3
5 6 8
```

In addition, the file `pdqpq.py` contains an implementation of a generic priority queue to use to hold the frontier for the best-first search algorithms. You are required to use this module for

your implementation Uniform-cost, Greedy best-first, and A*. The class provides methods to add and remove items, check for inclusion, etc.:

```
import pdqpq
k = "Kipo"  # this can be any hashable object
w = "Wolf"
q = pdqpq.PriorityQueue()
q.add(k, 42)
q.add(w, 19)
if "Kipo" in q:
    print("Yes")
else:
    print("No")
e = q.pop()
print(e)
print(len(q))
```

The above produces the output:

```
Yes
Wolf
1
```

Note that `pdqpq.PriorityQueue` is based on a min-heap, so it pops the element with the lowest valued priority.

Lastly, the file `solver.py` contains a skeleton for your search code. This file contains the `solve_puzzle()` function, which you must complete. This function gets invoked to perform the actual search given a start state and strategy (see the docstrings for details on the parameters). How you fill in this function is completely up to you, but we highly suggest that you model your solutions after the implementation for `BreadthFirstSolver` (using the `PuzzleSolver` class will save you some code). Whatever you do, do not modify the API (the parameters or return type) of `solve_puzzle()`, as the autograder will be calling that function and examining the output to test your solution.

The `solve_puzzle()` function must return a dictionary whose entries document how the search played out (again, see the docstrings for details). Note that if there is no solution for a particular start state, your program should output a dictionary that does not include the `path` or `path_cost` entries.

As discussed in lecture, the different search methods are quite similar, and you are strongly encouraged to structure your solution in a way that reuses code. That said, there are some key differences between BFS and the ones you must write (particularly in regards to when the algorithm checks for goal states), so be mindful.

You *should not modify* `puzz.py` or `pdqpq.py`. The *only* file you must modify is `solver.py`.

Testing Your Code

While we will test your code by importing and directly invoking the `solve_puzzle()` function, you can test your solver by running `solver.py` from the command line or launching it from within an IDE. When run this way, you must specify at least two command line arguments[1]:

- 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`.
- One or more keywords specifying which search strategies and heuristics to use, each one of the following: `bfs`, `ucost`, `greedy-h1`, `greedy-h2`, `greedy-h3`, `astar-h1`, `astar-h2`, or `astar-h3`.

For example, to execute a BFS and A* (using a standard Manhattan distance heuristic):

```
> python solver.py 802356174 bfs astar-h2
```

The above should produce the output:

```
solving puzzle 802356174 with bfs
solving puzzle 802356174 with astar-h2

flavor      astar-h2         bfs
--------  ----------  ----------
length            30          26
cost             692         728
frontier     179,310     159,580
expanded     175,479     140,495
```

Note that your output may differ slightly from the above, depending on how ties are broken. Additionally, supplying the keyword tag `all` for the search flavor will run all variants.

You should test your solver on a variety of solvable and non-trivial (> 10 moves) test cases, as well as any “edge cases” you can come up with (e.g., where the start state is the same as the goal state, or the start state is unsolvable). Run your solver using each search strategy as you implement them, and look at the results output to see if they’re behaving as expected with regards to efficiency, completeness, and optimality.

Additionally, you must supply three test cases of different difficulty, and add them to the `get_test_puzzles()` function in `solver.py`. (You might consider generating these programmatically by starting with a puzzle in a goal state and randomly perturbing it using some

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

number of legal moves.)

Your solution should run in a “reasonable” amount of time: less than 5 minutes for finding a solution path for any valid start state. The public tests on Gradescope will check for this to let you know if you’ve engineered things correctly.

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 the due date, but we will make tests available that will check the format of your return values. Note that your solutions will only be tested on well-formed (but not necessarily solvable) boards. Code that does not run will not receive credit.

What to Submit

Submit any code required to run your solver (`solver.py`, and any other supporting files), along with a `readme.txt` which includes:

- Your name(s)
- The best meal(s) you’ve ever eaten
- Any significant references you consulted when writing your code
- Notes or warnings about what you got working, what is partially working, what is broken, and any other feedback you have on the assignment