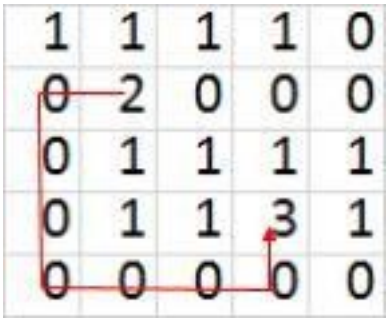
2018 Winter Lab1: Search

In this lab, you will write search functions to find a path reach a destination from a start location. Be sure to look at the test code in `main.py` to better understand the requirements. This file contains test cases you may use to verify your code. Grading will use a similar set of test cases. You are welcome to write additional tests in `main.py` to test your solution, but you should not remove or alter any of the tests that are there.

Part 1:

The task in this part is to write two search functions to help a robot find its way to a destination. The robot exists on a 2D square grid (represented in code as a list of lists) that we'll call a map. It can move in all 4 cardinal directions (not allowed to move along the diagonals) one cell at a time.
The starting location of the robot is marked on the map with the number 2 and the goal with the number 3. Walls are denoted by a 1, and empty space (that the robot can move through) is a 0.

Sample 5x5 map



Robot starting location at coordinates[1][1]
Goal at coordinates[3][3]

You must write the breadth first function `bfs(testmap)` and the depth first search function `bfs(testmap)`. These functions search for and label a path from start to goal on a given map. Each function will take in a map represented as a list of lists of numbers, a 2D array. Each input will have one 2 (the starting location) and one 3 (the goal or ending location). All other squares will be 0s or 1s.

Your functions should return the map with a path marked by 5s and all other explored cells marked by 4s. That is, you will return a list of lists of numbers where 5s mean that the robot should move through that cell on the path and 4s mean that your dfs or bfs explored that cell but that it is not part of the final path. Looking at the above example every cell along the red

arrow line would have a 5 with bfs or dfs while the location of 4s will depend on the algorithm.

For our automated grading scripts everyone needs to consider adjacent cells in the same order. That is, for both dfs and bfs you should consider adjacent cells in the following order: East, North, West, South (in coordinate form: [y][x+1], [y+1][x], [y][x-1], [y-1][x]). For example maps (before and after solving), see `main.py`.

Considerations:

- The starting and ending locations (2 and 3) are part of the path and should be marked with a 5 in the returned (output) map.
- The running time of your algorithm cannot be longer than 5 seconds for a 15x15 or smaller map, otherwise it will fail the grading tests. (The tests in main.py are 10x10, we encourage you to write your own)
- All maps will have only one possible path from the starting location to the goal (to make it easier).
- There will be no loops in the maps (to make it easier).

Part 2:

The task in this part is to write an A* search algorithm to find the path with lowest cost. You will be provided two Python dictionaries. One represents the distances between all pairs of locations, and the other represents: 1) the existence of a road between locations; and 2) time to traverse the road. In the time dictionaries, a numeric value represents a time, while `None` means there is no road between the locations. See `main.py` for examples. Notice that there are multiple time dictionaries in `main.py` - this simulates different traffic patterns at different times of day.

The evaluation function used in the A* search is f(x) = g(x) + h(x), and represents the estimated of reaching the goal via node `x`. For this task, g(x) represents the time required to reach `x` from the start location. Our heuristic function h(x) is the straight-line distance from x to the end. Taken together, g(x) + h(x) gives a "reasonable estimate" of the time it will take to reach the goal location via `x`. When prioritizing nodes in the 'fringe', if two nodes have the same f(x) value, the secondary ordering should be based on the name of the location represented in the node. For example, (17,'Cinema') will be before (17,'YWCA').

Your goal is to complete the function `a_star_search(dist_map, time_map, start, end)`, and to output a score dictionary. The keys of this score dictionary OUTPUT should be every location your search considers (i.e., every node that is expanded), and the values are another dictionary INNER. The keys in dictionary INNER are the neighboring locations, and the values are the results of calculating f(x) for that neighbor. Thus, OUTPUT[loc1][loc2] will give the result of f(loc2) when expanding the node for loc1. See `main.py` for examples.

The output should include all nodes that were expanded. This means that you need to include the start location but NOT the end location. (It is important that your data format match the expected, so everyone has the same results and we can autograde.)