Algorithms - Assignment 6

Due Tuesday, April 23rd by 11:59 pm (I'll have extra office hours Monday the 22nd)

This assignment is a combination of written problems and programming problems. The first two are writte, the last three are programming. Please submit your assignment as a single jupyter notebook file, as you did for assignment 4. For problem 1b, you are allowed to draw the graph by hand and scan and attach an image if that is easier for you.

(1) Perform Dijkstra's shortest paths algorithm on the following graph (graph α) with A as the starting node. Along the way you should

  (a) Create a table showing the intermediate distance values of all nodes after each iteration of the algorithm (i.e. there should be a column for each time a new node is released from the priority queue, and a row for every node). Make sure to include an extra column and row labelling everything (i.e. have an 'A' left of the A row, a 1 above the first iteration column, etcetera), so the graders can understand what they are looking at. Alongside these distances you should also include the current best predecessor (i.e. the *prev* values associated with each node). You don't need to include the state of the priority queue after each iteration, but you will almost certainly need to write it down on paper in order to carry out the algorithm.

  (b) Afterwards, draw the shortest-paths tree.

(2) Consider a pixel map of a tropical archipeligo (figure β), like figure α which I absolutely did not steal from somewhere else. The gray tiles represent land, while the blue tiles represent ocean. We want an algorithm which, given this map in the form of an $n \times n$ matrix of 0s and 1s (0 representing water, 1 representing land), returns the number of distinct islands. We will consider two tiles to be a connected landmass if they are adjacent to one another (this will include corners touching, so all 8 tiles surrounding a tile should be considered adjacent). Describe using words and pseudocode (but don't actually program unless you want to) an algorithm which does this.

Your algorithm should work in linear time with respect to the number of cells (so if we have a square $n \times n$ matrix as input, linear runtime would actually be $O(n^2)$, not $O(n)$). Be clear about how you are representing this as a graph problem, and how your graph would be represented in code.

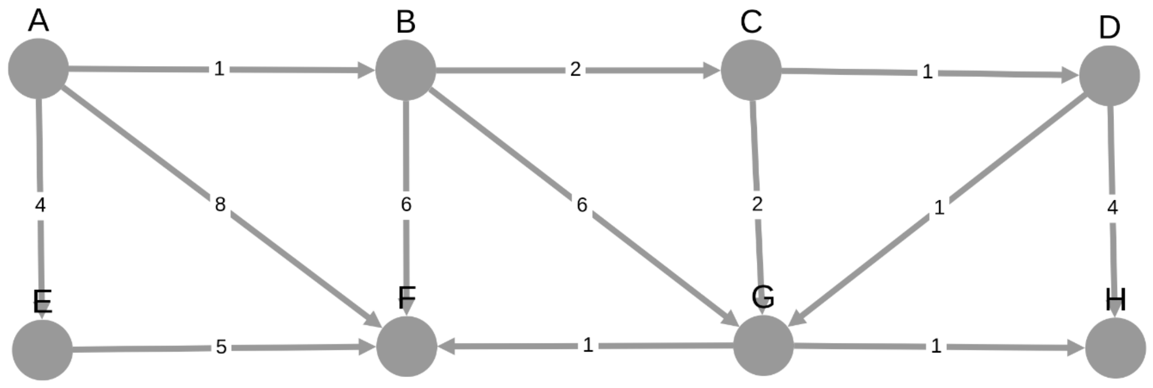


Figure 1: Graph α

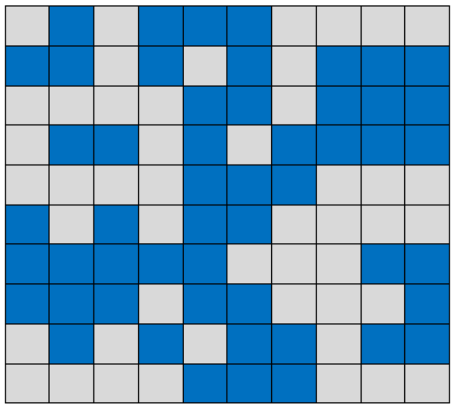


Figure 2: Figure β

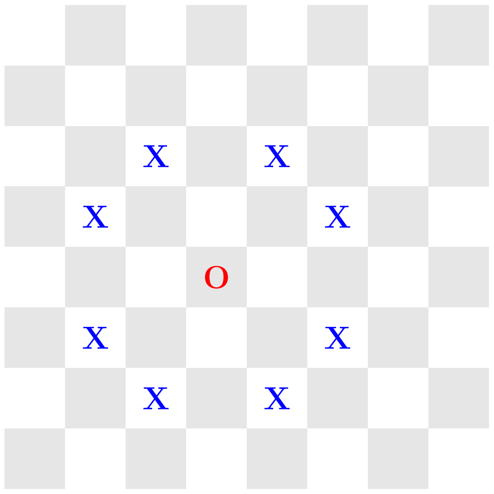
(*Hint:* This problem really should have been on the previous homework, although both breadth-first and depth-first searches can work here. Try and express this as a graph connectivity problem.)

(3) Consider the 8x8 chess board pictured below. The red circle represents a knight chess piece, and the 8 X marks represent the different legal moves that a knight can make. To represent position on the board: top-left cell will be position $(0, 0)$, bottom left will be $(0, 7)$, top right will be $(7, 0)$, and bottom right will be $(7, 7)$. Our problem will deal with a *generalized* chess board, i.e. an $n \times n$ grid, but with the same convention for representing positions.

(a) Write a python function which, given the dimension of the game board n, an initial position, and a destination position, returns the minimum number of moves that a knight needs to make in order to move from that initial position to the final position, *as well as* a specific sequence of moves which will get them to that position. (Represent the moves with two-letter strings, i.e. 'ul' should represents moving two cells up and one cell left.)

You'll want to use breadth first search for this, but I like this exercise because the actual graph will be *implicit* to your solution: don't try to force an adjacency list into your code. I recommend making a helper function which calculates the resulting position of a move applied to a knight in a given position (or an error code if the move is invalid), and then using that function in order to check for 'edges'. As a quick way of checking your program, the minimum number of moves required to travel from the position (0,7) to (7,0) is six, and there are many possible ways of getting there in this many moves (some of which might surprise you). Don't jump to thinking your program is wrong if the path it returns looks weird at first (like I did).

(b) What is the runtime with respect to n?



(4) In this problem, you will use the priority queue data structure that you created in homework 4 in order to implement Dijkstra's algorithm.

(a) Before that though, we need to make a few changes/additions to it (I tried to leave you with something which would be perfect for Dijkstra in that homework, but my planning was not perfect.) Don't worry, this part shouldn't take long.

- Replace the increase_priority(self, task, k) function with a function called change_priority(self, task, k). Instead of adding k to the current priority, it should simply replace the old priority with k. This is better suited for Dijkstra.
- The biggest addition we need to make is that our priority queue needs a min-mode, in which the lowest numbers have the highest priority. We can do this in same way that we talked about turning a max-heap into a min-heap.
  In the constructor, add an additional default argument called mode, which defaults to the string 'max'. The constructor should initialize an attribute of the same name to that argument. (I.e. add the line self.mode = mode).
  All we need to change in order to make everything work in min-mode is add a few checks to the enqueue(self, task, priority), dequeue(self, task, priority),

and change_priority(self, task, k) methods. For enqueue, simply add a check which multiplies the input priority by -1 if the mode attribute is set to 'min' before using it. This will make it so that behind the scenes, all of the priorities will be negative, although the user will never actually see this. When the user dequeues something, they should have their original priority returned, not the negative priority, so add a mode check to that method which multiplies by -1 again before returning the when in min-mode. Finally, in the change_priority method, add a mode check which multiplies k by -1 when in min-mode before using it. That should be all you need to change in order to switch your priority queue between a max mode and a min mode.

- We will add two more dunder methods which aren't strictly necessary, but will your queue easier to work with and thus make Dijkstra a little more straightforward to implement. These are __len__(self) and __contains__(self, task). The first is what python looks for when len(thing) is called. Have your queue return the size of the underlying heap when len(Q) is called (one line of code). __contains__(self, task) is what python looks for when you write an if statement that checks for membership. I.e.

```python
l = [1,2,3,4]
if 1 in l: print("Found it")
```

  When python sees the 'in' here, it looks for a method called __contains__(1), which is supposed to return a True or False value. Make such a method check to see if a task is currently in the queue (you can do this in constant time using the data_locations dictionary attribute).

(b) With that out of the way, your queue should be ready for use in Dijkstra. Program it the way we discussed in class. Your function dijkstra(G,s) should take two arguments, a weighted graph G and a starting node s. Further instructions:

- The graph should be a weighted adjacency list in the form of a dictionary of dictionaries. Rather than try to explain what exactly that means here, I've included an example graph on canvas. This graph I've included is precisely the graph α from question 1 of this assignment. You already know what the output should be for this graph, so you can use it to see if your code is working.
- Have the starting node s default to the first node in the adjacency list. After that and before doing anything else, your function should test to see if the starting node is an actual node in the graph. If it isn't, then return an error message and stop.
- The function should return two outputs: dists, a dictionary of shortest distances to s for each node, and prevs, a dictionary of predecessors in the shortest paths tree, so that shortest paths can be recovered.

(5) (a) Now implement Bellman-Ford in python. It should take identical inputs as your function for Dijkstra did (a weighted adjacency list and a starting node) and return the same predecessor and distance dictionaries. Remember: Bellman-Ford is a brute(ish) force algorithm, and shouldn't be difficult to program. Mine only

amounted to about 20 lines of code (same for Dijkstra, excluding the priority queue). You can assume that all edge weights are positive, and so there is no need for programming negative cycle detection.

(b) I've included on canvas a python function which generates random graphs (with positive edge weights). Use this to run some tests comparing Bellman-Ford with Dijkstra, creating plots in MatPlotLib like usual. You should see Dijkstra massively outperforming Bellman-Ford. Remember though that Bellman-Ford still has its uses: it works when there are negative edge weights, unlike Dijkstra.