CS2040 Tutorial 10 Suggested Solution

Week 12, starting 31 Oct 2022

Q1 Mario 3D

Suppose you are playing Super Mario. Mario wants to reach his princess to save her from the enemies. Mario and the princess are in an **R*C** grid, together with **N** enemies who each take up 1 cell in the grid. Mario can move {up, down, left, right} but never diagonally. The princess and her enemies do NOT move.

2 examples:

In both examples, the game is a 5*4 grid

Gray cells represent those occupied by enemies, $\mathbf{N_A}$ == 4 and $\mathbf{N_B}$ == 3

In grid A, the princess is doomed as Mario cannot reach her

In grid B, there are several ways for Mario to reach the Princess, the shortest having distance of 4

Grid A

(0,0) Mario	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)
(4,0)	(4,1)	(4,2) Princess	(4,3)

Grid B

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1) Mario	(1,2)	(1,3)
(2,0)	(2,0)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)
(4,0)	(4,1)	(4,2) Princess	(4,3)

Given a grid with the positions of the **N**+2 game characters, how can you:

(a) detect whether Mario can save the princess

(b) find the min number of cells Mario needs to move to be in the same square as the princess

Assuming Mario can reach the princess, how do you find each of the following, independent of one another, given that each cell also has:

(c) a positive **calorie** cost that Mario will incur on moving into the cell - to find the **min total number of calories** Mario will burn upon reaching the princess

(d) a **difficulty** level (could be -ve, 0, +ve) to move into the cell - to find the **min total difficulty** Mario needs to go through to be in the same square in the princess

(e) a **difficulty** level (could be -ve, 0, +ve) to move into the cell - to find the **most difficult cell** that is along the "easiest" path from Mario to the princess, "easiest" being determined solely by the most difficult cell in that path

Find an efficient algorithm for each of **(a)** - **(e)** and state its time complexity

Answer

Firstly, this is an undirected graph with **RC** vertices and less than 4**RC** edges. Each cell without enemy is a vertex and the border between adjacent cells are edges. There are no edges to cells occupied by enemies. Since the edges from/to each vertex are already known, the graph is implicit – it is not necessary to build an adjacency list to store the graph.

Instead of maintaining a 1D array to mark vertices while performing some algorithm, a 2D **R*****C** array can be used instead to correspond to each vertex. This prevents DFS/BFS from visiting the same node repeatedly, or worse, getting stuck in a cycle.

(a) Use DFS/BFS to check whether Mario and princess are in the same connected component. Start at Mario's cell and return true if princess is visited. Otherwise, return false.

(b) Use BFS to find the shortest distance between Mario and princess. Start at Mario's cell. At each vertex u, for each neighbour (u, v) that does not already have a computed distance, the dist(Mario, v) == 1 + dist(Mario, u). Store all distances in a **R*****C** grid (in fact, the distances grid can be initialized to -1 and double up as the visited grid)

DFS **does NOT work** because there can be more than one path from one vertex to another. DFS may take a longer path from Mario through a vertex u to another vertex v, but the dist(Mario, u) may not be optimal. Even if dist(Mario, u) gets updated later, outgoing edges from u will not be processed again so dist(Mario, v) may not be optimal.

Both **(a)** and **(b)** run in O(**RC**) time as each cell and each edge is visited at most once.

Parts **(c)** - **(e)** work with weighted graphs, unlike parts **(a)** - **(b)**. Therefore the graph should be directed, as the weight of u->v and v->u may be different.

(c) The min total calorie count will be the same as the weighted shortest path from Mario's to princess' cell with the destination cell calorie count as each edge weight.

Since the calorie cost cannot be negative, we can use Dijkstra's algorithm with Mario's cell as source. Terminate when princess is visited (i.e. when dequeued, **NOT** when an edge to princess is relaxed). This runs in O(**RC**log(**RC**)) time if indirect heap / lazy deletion is used.

(d) Since the difficulty can be negative, Dijkstra's algorithm may not work as once a vertex has been visited, it will never be visited again.

Use Bellman Ford's algorithm with Mario's cell as the source. After the algorithm runs to completion, return the distance at princess' cell. The algorithm runs in O($\mathbf{R}^2\mathbf{C}^2$) time.

Bellman Ford's algorithm does NOT rely on edge weights being non-negative, but ensures that more and more vertices have correct distance estimates each outer loop pass.

The pre-condition here is that there are no negative cycles. If there is, and cells can be visited repeatedly, then another **RC**-1 outer loop passes of Bellman Ford's algorithm can be used to mark vertices which have distances affected by negative cycle(s). If princess is affected, then the distance should be -INF instead.

(e) For good efficiency, we can view this as a **minimax path** problem, solved by building a minimum spanning tree and search its edges – this is for tutorial 11. The important thing is that you should be able to see that this is **NOT a SSSP problem**. This algorithm will take O(**RC**log(**RC**)) time.

Another method is to view this as a **searching problem**. There are up to **RC** candidate answers, that can be taken from the difficulty level in each cell.

Assume a most-difficult difficulty level **D**. DFS from Mario's cell till princess is reached, but only take edges that have weight of at most **D**. This is because no edge should have a difficulty (edge weight) greater than **D**. When princess is reached, return the coordinates of the cell with difficulty level **D** along that path taken. One such search for a given difficulty level **D** will take O(**RC**) time

Instead of trying out all **RC** possible candidates, we only need to try around log(**RC**) of them. Sort the candidate difficulty levels and then "binary search" to find the lowest difficulty level that would successfully let you reach princess (somewhat similar to floor / lower bound). Sorting will take O(**RC**log(**RC**)) time and searching around log(**RC**) times will take a total of O(**RC**log(**RC**)) time. Therefore, this algorithm also runs in O(**RC**log(**RC**)) time.

Q2 SSSP In a DAG

In each of **V** passes, Dijkstra's algorithm confirms the distance estimate of the reachable unconfirmed vertex with the minimum distance from source. This works because the confirmed vertices will never be updated again when there are **NO negative-weighted edges**.

Now if we have a weighted **directed acyclic** graph, can we modify Dijkstra's algorithm to run in O(**V** + **E**) time, and at the same time allow negative-weighted edges (but no negative-cycles) to be present? The output of your algorithm should be **another** weighted **directed** graph containing the shortest-path spanning-tree.

Hints: How do you

- remove the O(log **V**) cost per vertex and per edge
- pick vertices to confirm, so that all confirmed vertices will **NEVER** be revisited

?

Answer

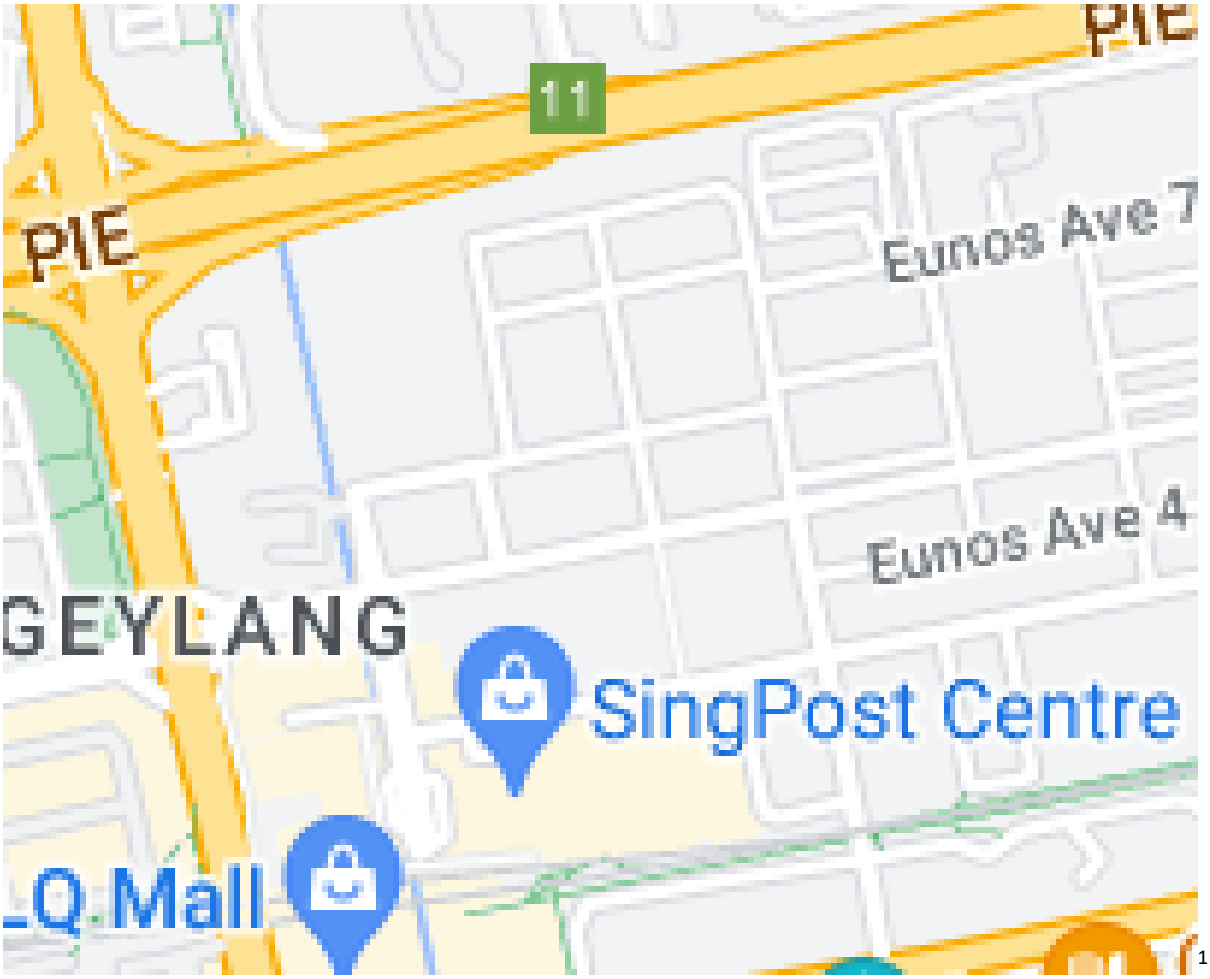
Each time, instead of confirming the vertex with the lowest distance estimate, confirm a vertex that has no more incoming edges left to relax.

This means we can first perform topological sort, then confirm distance estimates of vertices and relax all outgoing edges in topological ordering. As not all vertices may be reachable from source, first initialize all distance estimates to +INF except for the source, which has a distance of 0.

Since no priority queue is needed and every vertex and edge is visited exactly once, it takes O(**V** + **E**) time if we have an adjacency list of the graph.

```
ArrayList<LinkedList<Edge>> runDijkstraOnDAG(
     ArrayList<LinkedList<Edge>> adjList, int sourceIdx
     ) {

   // initialization
   int[] dists = new int[adjList.size()];
   Edge[] ssspTreeEdges = new Edge[adjList.size()]; // incoming edges
   for (int idx = 0; idx < adjList.size(); idx++)
      dists[idx] = INF; // some constant
   dists[sourceIdx] = 0;

   Queue<Integer> topoOrder = findTopoOrder(adjList); // week 11 lecture

   // relax edges in topo order
   while (!topoOrder.isEmpty()) {
      int fromIdx = topoOrder.poll();
      // no need to mark visited as each vertex visited exactly once
      for (Edge edge : adjList.get(fromIdx)) {
         int newDist = dists[fromIdx] + edge.weight;
         if (dists[fromIdx] == INF) newDist = INF; // INF + +ve -> INF
         if (newDist < dists[edge.to]) {
            dists[edge.to] = newDist;
            ssspTreeEdges[edge.to] = edge;
         }
      }
   }

   // build SSSP tree graph
   ArrayList<LinkedList<Edge>> ssspTree = new ArrayList<>();
   for (int idx = 0; idx < adjList.size(); idx++)
      ssspTree.add(new LinkedList<Edge>());

   for (int to = 0; to < adjList.size(); to++)
      if (ssspTreeEdges[to] != null)
         ssspTree.get(ssspTreeEdges[to].from).add(ssspTreeEdges[to]);

   return ssspTree;
}
```

Question 3 (Online Discussion) – Noosepaper



[1]

You are a software developer at Lion Noosepaper & Co, and you want to find the shortest time it takes for a delivery truck to exit your *base* and reach a *road segment*. For simplicity, you may assume that every *road segment* is situated in between 2 *road junctions*.

A *road segment* can be on a 1-way road, or a 2-way road. As delivery takes place early in the morning, the roads are clear, therefore the estimated travel time on a *road segment*, regardless of valid direction, is (given and) fixed. For simplicity, there are only 2 kinds of *road junctions*:

Cross Junction

T Junction

Junction orientations may be rotated.

Some *road junctions* do not allow right turns, some allow u-turns (in Singapore, u-turns at junctions are not legal unless otherwise indicated =( ). Each *road junction* also has a (given) different estimated time taken to

[1] Image cut from Google Maps, 2022

turn left, go straight ahead, turn right, and/or u-turn if legal (turn left time may be different from go straight ahead time).

Given:

- the position of the **N** *road junctions* and **M** *road segments*
- the nature of each of the N road junctions – cross / T? Can turn right? Can u-turn?
- the nature of each of the **M** *road segments* – 1-way / 2-way
- the relationships between the *road junctions* and *road segments*
- the turn left / straight ahead / turn right / u-turn times, if exists, to clear each of the **N** *road junctions*
- the travel time taken to clear each of the **M** *road segments*
- the position of a base (one of the **M** *road segments*) where your truck starts from

you are to **efficiently find** the shortest time it takes to hit a desired road segment

How do you model the graph, what algorithm should you use, and what time complexity is required?