CS 401 :: Homework 2

Ryan Magdaleno

February 12, 2024

Problem 1: Vertex-weighted shortest paths. Let $G = (V, E)$ be an undirected graph and let $w : V \to \mathbb{R}_{\geq 0}$ be a function that assigns non-negative weights to the vertices of G. We define the *weight* of a path $P = v_1, ..., v_k$, denoted by weight(P), to be the sum of the weights of the vertices in P; that is

$$\text{weight}(P) = \sum_{i=1}^{k} w(v_i).$$

Let $s, t \in V$. Design a polynomial-time algorithm that computes a path from s to t of minimum weight, if one exists. Show that your algorithm is correct and that is runs in polynomial time.

Solution ::

Because this problem deals with an undirected graph with non-negative weights, we can use a graph algorithm that already exists such as Dijkstra's Algorithm. An issue however is that Dijkstra's Algorithm only works with non-negative edge weights, not vertex weights like in this problem. Therefore I'd like to modify the problem a little to fit Dijkstra's Algorithm.

In a typical implementation of DA (will be calling Dijkstra's Algorithm this now) we use a priority queue where each element is the tentative distances from the source like vertex s in this problem. I'd like each element in the priority queue to now represent the vertex along with its cumulative weight from the source s.

In a typical DA implementation we have some distances container that keeps track of all cumulative edge weights, we will now use the vertex weights. Using this we can now compute the tentative distances based on said vertex weights and update them as the algorithm progresses through the graph.

Distance container updates are now simply the cumulative weight of the path from the source vertex s to the current vertex u. Updating a neighboring vertex v's distance if the following condition is met:

$$dist(u) + w(v) < dist(v)$$

where $w(v)$ is the weight of vertex v.

The algorithm terminates when vertex t is dequeued from the priority queue, which means we have found a shortest path. If the graph is explored without ever finding t this indicates there is no connecting path from s to t.

From this we can reconstruct the minimum-weight P from s to t using a predecessor array, where $prev(u)$ returns the previous vertex on P.

Algorithm $MinP(G, V, s, t)$:

```
Initialize priority queue PQ;
pred ← array of size |V|;
dist ← array of size |V|;
foreach v ∈ V do
    dist(v) ← ∞;
    pred(v) ← empty;
    PQ.enqueue(v, ∞);
end
dist(s) ← 0;
PQ.enqueue(s, 0);
while PQ is not empty do
    u, weight_u ← PQ.dequeue();
    if u = t then
        return BuildP(pred, s, t);
    end
    foreach neighbor vertex v of u do
        if weight_u + w(v) < dist(v) then
            dist(v) ← weight_u + w(v);
            pred(v) ← u;
            PQ.enqueue(v, weight_u + w(v));
        end
    end
end
return "s and t are not connected in G.";
```

Polynomial-Time Complexity Proof ::
Given that $n = |V| =$ the number of vertices in G and $m = |E| =$ is the number of edges in G, if we use a binary heap for our priority queue, we can be sure that our modified DA implementation is:

$$O((n + m) \cdot \log n)$$

Initializing the predecessor array, distance array, and priority queue all take linear time, which in this case is:

$$O(n + m)$$

The main while loop in our modified DA implementation iterates at most once for each vertex in the graph, resulting in:

$$O(n + m)$$

The enqueue, dequeue, and distance updates if implemented with efficient data structures such as a binary heap will result in:

$$O(\log n)$$

Path reconstruction is also linear in n time, the worst case being a single path through every vertex resulting in a time complexity of:

$$O(n + m)$$

If we are to combine all these terms together, keeping in mind the inner loop, we come to a time complexity of:

$$O((n + m) \cdot \log n)$$

This time complexity is indeed polynomial time.

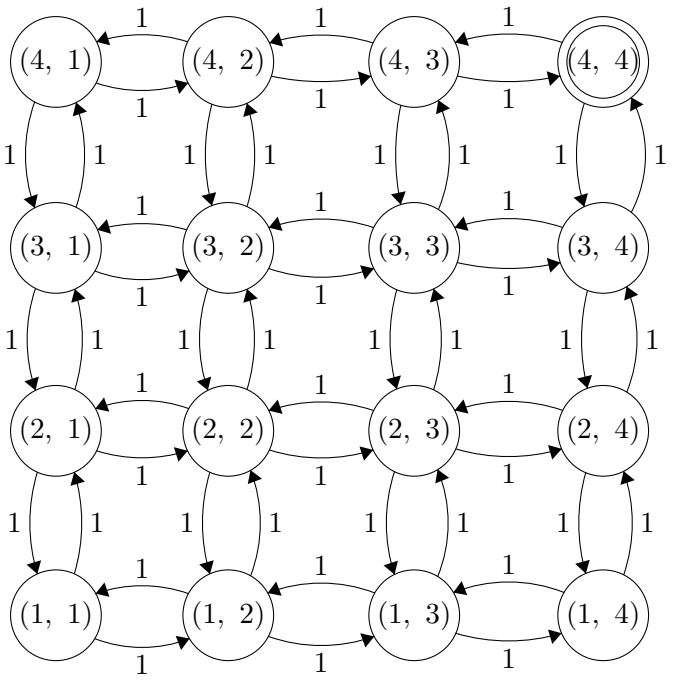
Problem 2: Robot navigation. Consider a robot that moves inside a room, represented by a 2-dimensional $n \times n$ square grid. Every location of the grid is indexed by a pair of integers (i, j), where $i, j \in \{1, ..., n\}$. Location (i, j) may either be empty, or it might contain an obstacle. The robot is allowed to move only in empty locations. Apart from its position, the state of the robot also consists of its orientation, which can be either North, South, West, or East. At every step, the robot can either move by one position forward in its current orientation, or it can change its orientation by 90°.
Hint: Express the below problems as shortest-path computations in some graph.

(a) Suppose that initially the robot is positioned at $(1, 1)$ and has orientation North. The goal is to reach position (n, n) in any orientation. Design an algorithm that computes a routing for the robot, with a minimum number of steps. Your algorithm should run in time polynomial in n.
Solution ::

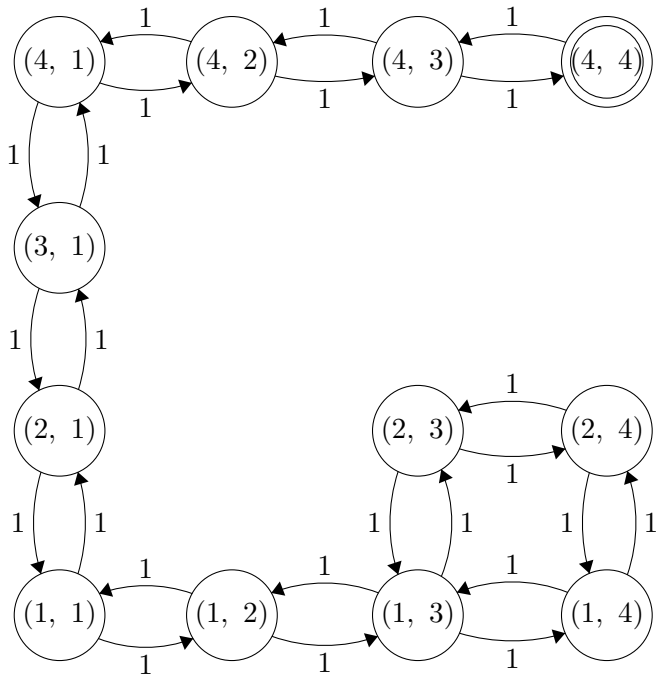
I would like to use an existing algorithm for shortest path traversal, Dijkstra's Algorithm. This problem however is not able to keep robot orientation in mind. I would like to construct a graph G which takes orientation in mind with its edges. For each cell that does not have a obstacle we will create four outgoing edges. These four edges will all have a weight of 1 each. For example the north orientation edge will be directed towards the cell $(i, j + 1)$, so for cell $(1, 1)$ the northern edge would be directed towards $(1, 2)$. The robot cannot go into the containing walls of the grid or into an obstacle so when initially setting up the orientation edges, we must not create invalid edges into vertices that are not within the grid or that contain obstacles.

Each cell will be a vertex, but each of those should become four sub-vertices, making it so the robot can only move forward one position or change orientation, the four sub-vertices will be connected to each other, and they will move towards another set of four sub-vertices representing another cell.

For example, here's an example of a grid converted with no obstacles, $n = 4$. Due to limitations of my drawing, vertex (1, 1) for example is really four vertices each connected together, the (1, 1) north sub-vertex will go to vertex (2, 1)'s north sub-vertex, all edges including the ones going to another sub-vertex will have an edge weight of 1, to allow for DA to find a shortest path keeping orientation in mind:



Here's an example with $n = 4$ and obstacles $= \{(2,2),(3,2),(3,3),(3,4)\}$



Using Dijkstra's Algorithm the robot should be able to find the shortest path from $(1,1)$ to (n,n).

Algorithm $Dijkstra\_Plus\_New\_G(obstacles, n)$:

```
Initialize n × n G;
foreach o ∈ obstacles do
  | removeConnectedEdges(G, o);
end

// Dijkstra's Algorithm from here onwards:
Initialize priority queue PQ;
pred ← array of size n^2;
dist ← array of size n^2;
foreach v ∈ G do
  | dist(v) ← ∞;
  | pred(v) ← empty;
  | PQ.enqueue(v, ∞);
end
dist((1,1)) ← 0;
PQ.enqueue((1,1), 0);
while PQ is not empty do
  | u, weight_u ← PQ.dequeue();
  | if u = (n,n) then
  |   | return BuildP(pred, (1,1), (n,n));
  | end
  | foreach neighbor edge e of u do
  |   | if weight_u + w(e) < dist(v) then
  |   |   | dist(e) ← weight_u + w(e);
  |   |   | pred(e) ← u;
  |   |   | PQ.enqueue(e, weight_u + w(e));
  |   | end
  | end
end
return "(1,1) and (n,n) are not connected in G.";
```

Polynomial-Time Complexity Proof ::
Firstly, we build the $n \times n$ grid graph G. For each empty cell, we check all four directions for if there is a valid cell to create an edge towards. This results in a $O(n^2)$ time complexity for building the graph G.

Assuming the worst case number of vertices and edges we get $|V| = n^2$, $|E| = 4n^2$. Dijkstra's Algorithm runs in $O((|V| + |E|) \cdot \log |V|)$ time. The reason for why in the worst case $|V| = n^2$ is that if there are no obstacles we can have at most n^2 vertices in our graph G. For edges in the worst case, if there are no obstacles then no vertices get removed, which means no edges get removed. There are four directions and in the worst case $|E| = 4n^2$.

Therefore, the time complexity of Dijkstra's Algorithm in this case is:

$$O((n^2 + 4n^2) \cdot \log n^2) = O(n^2 \cdot \log n^2)$$

(b) Suppose that there are k robots in the room. At each step every robot moves simultaneously. Two robots cannot occupy the same location and they cannot pass through each other. You are given the starting position and orientation of each robot and its destination. Design an algorithm that computes the shortest sequence of steps that move all the robots to their respective destinations. The running time of your algorithm should be $n^{O(k)}$.

Solution ::

Let us represent the graph G like in part a, where obstacle vertices are removed and each cell is a set of four vertices. This problem however has us control multiple robots. The goal is to avoid collisions during movements. We can run Dijkstra's Algorithm on all robots towards the given destination coordinates. We should then check how many robots are on each shortest path, we should prioritize moving robots with the lowest amount of robots on its path. So we should sort the number of encountered robots on each k_i robot's shortest path. If it occurs that there is a collision immenent, we will stop one robot for that movement, this way it allows for one robot to proceed on its shortest path. After that we would run Dijkstra's Algorithm again to check the new shortest path for all robots and repeat these steps until the kth robot reaches its respective destination.

This is assuming that all robots and their respective destination coordinates are connected by atleast one path, if the first Dijkstra's Algorithm search doesn't find a path, then the robot(s) that couldn't find their path should be removed.

dests is the coordinates expressed in tuple form (x, y), $x \geq 1$, $y \geq 1$
obstacles is the vertices to remove from G.
$kRobotPos$ is the current robot positions.
Algorithm $kRobotsDijkstra(dests, obstacles, kRobotPos, n, k)$:

```
Initialize n × n G;
robotPaths ← array of size k;
foreach o ∈ obstacles do
 | removeConnectedEdges(G, o);
end
while k > 0 do
 | foreach robotPos ∈ kRobotPos do
 | | dest ← dests(robotPos);
 | | robotPath ← Dijkstra(G, robotPos, dest);
 | | robotPaths.append(robotPath);
 | end
 | robotPaths.sort();
 | foreach path in robotPaths do
 | | if collision(G, path) then
 | | | continue;
 | | end
 | | moveRobotOnPath(G, path);
 | | k ← k − 1;
 | end
end
```

Time Complexity Proof ::

There are $4n^2 = n^2$ vertices in G, there are $8n^2 = n^2$ edges in G. In the worst case, for each cell, there will be 8 edges, 4 for the orientation vertices and 4 for out going edges to other cells. Therefore construction of graph G takes:

$$O(n^2)$$

Because we run Dijkstra's Algorithm on every move we need to first check DA's time complexity and check the largest amount of moves before the kth robot reaches its destination. Dijkstra's Algorithm for this graph takes:

$$O((n^2 + n^2) \cdot \log n^2) = O(n^2 \cdot \log n^2)$$

In the worst case the amount of moves is linked to the size of the grid so therefore the amount of moves is some constant times n. There is also a sorting step of $O(k \cdot \log k)$

Therefore in one move there is:

$$O((k \cdot \log k) \cdot k \cdot (n^2 \cdot \log n^2)) = O((k^2 \cdot \log k) \cdot (n^2 \cdot \log n^2))$$

All together if we have an $n \times n$ grid the time complexity would be:

$$n^{O((k^2 \cdot \log k) \cdot (n^2 \cdot \log n^2))}$$

Where in the worst case, we have some n sized grid and therefore have n amount of moves, where on each move we must do Dijkstra's algorithm k number of times.

Problem 3: Finding a cheap flight. Let $G = (V, E)$ be a directed graph, where V is a set of cities, and E represents all possible flights between the cities in V. For every edge $u, v \in E$, you are given the duration of a direct flight from u to v, denoted by $d(u, v)$, which is an integer. For example, if you are at city u at time t, and you take a direct flight to v, departing at time $t' \geq t$, then you arrive at v at time $t' + d(u, v)$. For every $\{u, v\} \in E$, you are given a timetable of all available direct flights from u to v, for some interval $\{0, ..., T\}$, where $T > 0$ is an integer. That is, for any $\{u, v\} \in E$, you are given a list of pairs of integers $((t_u, v, 1, c_u, v, 1), ..., (t_u, v, k, c_u, v, k))$, where the pair (t_u, v, i, c_u, v, i) denotes the fact that there is a direct flight from u to v that departs at time t_u, v, i, and costs c_u, v, i dollars. Design an algorithm that given a pair of cities $u, v \in V$, computes the cheapest possible route that starts at u at time 0, and ends at v at time at most T. The running time of your algorithm should be polynomial in $|V|$, and T.
Solution ::

This problem is giving us a directed graph G that represents cities as vertices and flights between cities as edges. We are trying to find the cheapest flight path from city u to city v, and must end at most at time T.

We will need to keep track of each cities minimum costs at each step of the way. First we need to initialize a container that keeps track of each city's cost at each time t_i up to T, for example cell (i, j) will represent city i at time j, $0 \leq j \leq T$. We will make this $|V| \times T$ table and call it R

To start off we must set city u at time 0 like so:

$$\mathrm{R}[u][0] \leftarrow 0$$

For all other cities in V we will set their initial cost to infinity, we are essentially going to minimize the cost as we traverse the graph.

Now we are to go over all times from 0 to T for each city u. We are to also check for each city u, each outgoing flight from u to v at our current time t. We will update the table on the condition that $t' + d(u, v) \leq T$, the new min cost would become:

$$R[v][t' + d(u, v)] \leftarrow min(R[v][t' + d(u, v)], R[u][t] + c_u)$$

The algorithm would then return the minimum cost from $R[v][T]$ if it's not infinity, infinity indicates there was not a path to the final destination. This solution constructs the cheapest paths using the graph, in a sort of greedy manner.

Algorithm $cheapest(G, T, V, u, v)$:

```
Initialize R ← container with dimensions |V| × T;
foreach city u' ∈ V do
    t ← 0;
    while t ≤ T do
        R[u'][t] = ∞;
    end
end
Mark start city: R[u][0] ← 0;
t ← 0;
while t ≤ T do
    foreach city u do
        foreach (t_u, v, i, c_u, v, i) do
            if t + d(u, v) ≤ T then
                R[v][t + d(u, v)] ← min(R[v][t + d(u, v)], R[u][t] + c_u);
            end
        end
    end
    t ← t + 1;
end
if R[v][T] = ∞ then
    return "No route";
end
else
    return R[v][T];
end
```

Time-Complexity Analysis ::
Initializing the R container takes two nested loops, the outer in $O(|V|)$ time and the inner loop in $O(T)$ time, all together this is:

$$O(|V| \cdot T)$$

There are three nested loops, the outer loop iterates t from 0 to T, therefore this is:

$$O(T)$$

The next loop layer iterates over all cities u, the number of cities is $|V|$, therefore it is:

$$O(|V|)$$

The next loop layer iterates over each outgoing flight from u, which could have be $|E|$ if you consider the worst case, therefore it is:

$$O(|E|)$$

Combining our loops we get the following time complexity if you consider the worst case scenario:

$$O(T \cdot |V| \cdot |E|)$$

This time complexity satisfies the constraint that it must be a polynomial in $|V|$, and T.