GA HW4

Piyush Hinduja

November 2023

Question 1

To solve the problem, we will use the same Dijkstra's Algorithm, just by changing the weights.
Instead of using safety values as edge weights, we can use $-log_e(safetyvalues)$ as edge weights.
This will reverse the order of incrementing safety values, that is, in case of safety values we have , $0.6 > 0.5 > 0.1$, but, $-log_e(0.6) < -log_e(0.5) < -log_e(0.1)$.
This means shortest distance will have max safety and vice versa.
Hence now we can easily apply Dijkstra's to find the min distance or min edge weights on this updated graph as follows.
We will maintain 2 data structures, a 1D array of size 'n' which will store the minimum distance from 'u' till each node and the parent node using which it is getting that distance.
Initially, for all nodes the distance will be set to infinity except the start node 'u' where it will be 0 and parent nodes of each node will also be NULL except 'u' where it will be 'u'.
Another data structure which we will maintain will be a priority queue which will store all the nodes which can be accessed next, i.e. once we traverse a node, we will add all of it's reachable nodes along with their distance values required to reach till them.
Initially, this priority queue will contain just the start node 'u' along with distance '0' in the form (distance, node) (0, u).
Now, at each iteration, we will pick the closest node from our priority queue which will be having minimum distance value and whenever we pick a new node from queue, we will update our both data structures, i.e., we will add new nodes which can be reached from new node ONLY IF the distance value of that reachable node is lesser than the one in our 1D array (because if a node is reachable via some other path with lesser safety, there's no point in updating it).
We will keep iterating or keep taking elements till we haven't reached our target node 'v' (we are told that a path exists from u to v, hence we will reach 'v' at some point).
Finally after reaching 'v', we will backtrack through all the parent nodes until we reach 'u' and return that path.

Pseudo-Code:
Initialize a 1D array 'distance' of size 'n' with all distances as infinity.
Initialize a priority_queue with (0, u).
Until 'v' is reached:
    new_node = next closest element from priority_queue
    reachables = All nodes reachable from new_node
    for all nodes in reachables:
        if(distance required to reach that 'node' < distance[that node]):
            distance[that node] = new distance value
            parent[that node] = current node
            If that node is already in priority queue with greater associated distance, replace it with this instance.
    priority_queue.add(reachable nodes)

Running Time:
i) Here, initialization of priority queue takes $O(logn)$ time because adding one element to PQ takes $O(logn)$ time.
Then initialization of 1D array takes $O(n)$ time for n nodes.
ii) Now coming to the main loop, taking the node with minimum distance from priority queue takes $O(logn)$ time and we take out each node once, hence, $O(nlogn)$.
iii)Updating the distances of edges in 1D array each time takes constant time but updating them in priority queue takes $O(logn)$ time and for all edges it will take $O(m * logn)$ time.
iv) Backtracking will take at most $O(n)$ time.

Hence adding everything up, we get:

$$O(logn + n + nlogn + m * logn + n) = O((n + m)logn)$$

Correctness:
Claim 1: Path returned is the safest one
At each step (for each node), we are updating all the nodes reachable from current node with min distance (i.e. max safety), until we reach the target node.
Therefore, starting from source till target will give us the safest path.
Claim 2: Reaching target node
Since we are running algorithm till we reach target node and we know that target node is somewhere in the graph, we can say we will eventually reach 'v'.

Question 2

In this problem, we are asked to find the path with shortest path and shortest hops.
The main idea to find the shortest path is applying Dijkstra'a algorithm along with a priority queue (same as question 1) which takes $O((m+n)\log n)$ time.
Now to find the shortest path with minimum hops, we can add a small value $'e'$ (e.g. 0.000001) and then proceed further with finding the shortest path.
This extra step will take maximum time $O(m)$ for all edges in graph which would be ignored in front of $O((m+n)\log n)$.
Now when we will reach our target node through multiple shortest paths, we will compare the values after decimal point of those paths and the one having least value after decimal point will be our shortest paths with shortest hops.
More the number of edges a path will encounter, the value after decimal will also keep increasing, that is, if we have 2 shortest paths from u to v, path one coming through edge weights 2, 2, 2 and path two coming directly with edge weight 6.
No if we perturb our weights by adding 0.001 to each edge weight, path length for path one will be 6.003 where as path two will have 6.001.
Hence, we will pick path 2 since it has less value after decimal.
Finally, we can back track using the saved parent nodes with each distances in distance array until 'u' and return the shortest path with min hops.

Algorithm:
Add $e = 0.0001$ to all the edge weights.
Initialize a 1D array 'distances' of size 'n' with all distances as infinity and their parents as Null.
Initialize a priority_queue with one element as (0, u).
Until 'v' is reached:
    new_node = next closest element from priority_queue
    reachables = All nodes reachable from new_node
    for all nodes in reachables:
        if(dist of reachable node from 'u' via current_node < distances[node]):
            distances[node] = dist of reachable node from 'u' via current_node
            priority_queue.add(reachable nodes)
            If that node is already in priority queue with greater associated distance, replace it with this instance.

Now if 'v' can be reached with more than 1 shortest paths:
    Return the path having least value after decimal.

Running time:
Dijkstra's part:
i) Initialization: $O(nlogn)$ (Initializing the priority queue with single element).
ii) Main Loop: Taking the node with min dist (each time for a node, 'n' nodes total) takes $O(nlogn)$ time and updating priority queue for each edge takes

$O(mlogn)$ time.
iii) Backtracking the path using associated parent nodes will take $O(n)$ time in worst case.
Hence, total time $O((n+m)logn+n)$.
Extra time:
i) Updating each weight takes $O(m)$ time.
ii) Checking for more than one shortest paths takes *constant* time.

Therefore, total running time becomes $O((n+m)logn+n+m) = O((n+m)logn)$.

Correctness:
Claim 1: Getting the path with shortest distance.
We can prove this by induction as follows:
Base case: Shortest distance from source node to itself is zero and that is true, (0, u), in our algorithm.
Inductive step: If we assume that after 'k' steps we are getting a shortest path till a node 'x', then by selecting the closest node to n (lets say y), we know we are getting the shortest path from u to y at k+1 step.
Claim 2: Selecting the path with smallest hops
Since we are adding a small constant term e to edge weights (which will have no impact on shortest path), the path with longer hops will have larger accumulation of e compared to the one with smaller hops.
Hence we can say, both of our claims are proved.

Question 3

In this problem, we are just asked that can we reach from A to B or not, we aren't asked any kind of optimal solution.
So the simplest approach we can think of is similar to BFS traversal.
We will maintain a queue in which each element will have 2 values: node number and the range of electric after reaching that node.
Initially 'q' will have (A, 50).
Whenever we remove an element 'x' from the queue, we add all the nodes which could be reached from 'x' in <= remaining miles of our electric.
While adding new nodes will will update the range of our electric based on if current node is a supercharger or not.
Now, while adding a new node to 'q' there is a chance that node is already in the queue, hence, we will keep that value of node which will have maximum range value associated to it.
We will follow these iterations until we reach target node 'B' or we have an empty queue.

Pseudo-code:
Initialize a queue 'q' with start node initially.
while q is not empty:
    x = topmost element of q
        If x is B:
            return true
        Updated range at nodes reachable from 'x' will be:
            If next_node is a supercharger:
                range = max_range
            Else:
                range = range at current node - distance required to reach next_node
        Add new nodes which could be reached from 'x' with current range.
        If node i already in 'q':
            Keep that instance of node which has the maximum range.

Running Time:
Running time complexity of BFS is $O(m+n)$ because in worst case we will have to visit all the nodes and edges.

Correctness:
Since we are traversing each node reachable from source node and the nodes reachable from those connected nodes and so on, if the target node would be reachable from source node given the range, edge weights and supercharger positions, we will encounter it.

Question 4

In this problem, we are asked to find a meeting point (w) for two persons (A and B) such that the maxdist(A,w), dist(B,w) is minimum.
We know that, Dijkstra's algorithm (of a growing ball of radius 'r') ensures us the shortest path from the start node to all the nodes in the ball of radius 'r'.
So we can start by applying Dijkstra's on 2 nodes (A and B) simultaneously and keep moving forward by taking the next closest node (this can be done using priority queues).
After taking each step, we will check the nodes which are newly added to queue (or ball) for both the algorithms.
If we find a common node, whether it has been added to the queue in current iteration or previous iterations, we will stop the procedure and return that node, else we will continue.

Algorithm:
Initialize two priority queues, qa and qb, such qa has (0, A) has first entry and qb has (0, B) as first entry.
Until qa or qb is empty:

Take the next node from qa (which is the one with closest distance) and add all it's neighbouring nodes to qa.

Perform the same step for qb.

Compare all the reachable nodes from both A and B.

If a same node is found, return that node.

Else, continue.

Running time:
Running time of this algorithm is same as that of Dijkstra's, $O((m+n)logn)$ (Explained above in question 1 and 2) since we are applying it twice.
Hence, $O(2(m+n)logn) \approx O((m+n)logn)$

Correctness:
Claim: We are getting maxdist(A,w), dist(B,w) as minimum.
Since we are applying Dijkstra's on both the nodes A and B simultaneously, we can say that at each step the visited nodes (in radius 'r') are at the minimum distances from their respective source nodes A and B.
Now, if we find a node (w) common between the visited nodes of A and B, we can say that, that node is at the minimum possible distance from A and B both.
Hence, we can conclude that maxdist(A,w), dist(B,w) as minimum.

Question 5

a)

First to build a graph G' containing paths from u to v which are multiples of 3, we can use the BFS approach just with a modification as follows.
While adding a node to queue, we will also add the distance required to reach that node from 'u' and whenever we will encounter 'v' we will check if the distance required to reach 'v' through that path is a multiple of 3 or not, if it is, we will save the path.
A normal BFS would stop at this point after reaching 'v' but we will continue until our queue is empty, that is, we will traverse all the paths of graph to find all the possible paths from u to v.
We can then construct a graph G' containing all the saved paths.

Moving further, we need to find the shortest path from graph G', from u to v.
This can be done by applying Dijkstra's algorithm (similar to question 1) from u to v.

Running time:
The BFS part involves traversing all the nodes and edges and in worst case, we might have to travel each edge for all the nodes which will lead to a running time $O(n * m)$.
The Dijkstra's algorithm will take $O((m + n) * logn)$ time.
Hence overall running time would be, $O(n * m)$.

b)

To solve this problem in $O(m + n)$ time, we will use a modified BFS approach as follows:
We will maintain two data structures, a normal queue to store the neighbours of visited nodes and a 1D array to store min distances of each node.
Initially the distance array will have distance corresponding to each node as infinity except the 'u' node where it will be 0. The queue will also have (u, 0) initially where 0 indicates distance of u from u.
We will start with taking the start node 'u' from queue. Then add all the nodes reachable from 'u' along with their distances from u which will be in the format $distance\%3$ and also the parent node which lead to them with that distance, example if I am reaching node D with $distance\%3 = 1$ from C then in distance[D] I will store (1, C).
If the current distances of reachable elements are less than the ones in main distance array, update the distance array.
Then we perform the same iterative step of taking the next element, adding it's neighbours and updating the distances until we get 'v' where $distance\%3 = 0$, that is, distance from u to v is a multiple of 3.

After reaching 'v' which has $distance\%3 = 0$ we will backtrack using the saved parent node of each node until we reach 'u' backwards and return the path.

Algorithm:
Initialize a queue 'q' with one element (u, 0).
Initialize a distance array of size n with distance of u=(0, u) and other distances (infinity, Null).
while true:
    if next element in 'q' is 'v':
        if distance required to reach 'v'%3==0:
            return the path using backtracking
        else add the neighbours of 'v' and update the distance array.
    else:
        After taking out the next element, add all it's neighbouring nodes to 'q' and update the distance array if neighbouring nodes can be reached in a more efficient way.

Running time:
In worst case, we will have to visit all the nodes (n) and all the edges (m) in the graph which will lead to a running time of $O(m + n)$.

Correctness:
Claim 1: We will reach to 'v' from 'u'.
Since we are starting from 'u' and traversing all the neighbouring nodes for all the nodes in the queue, then if there exists a path from 'u' to 'v', we are guaranteed to reach 'v'.
Claim 2: The path which we get is a multiple of 3.
Before returning the path, we check it's length, if it is not a multiple of 3, we will check for other paths.

Question 6

a)

Here, we need to prove that given a graph G with adjacency matrix A, the shortest distance between two nodes i and j is the smallest k where $A^k_{ij} > 0$.
We could prove this using induction as follows:

Base Case:
Base case for this problem is $A^0 = I$ where I is the identity matrix with all non diagonal elements as zero.
This case is true since no two nodes have shortest distance of zero between them.

Inductive case:
Let's assume that shortest distance between two nodes i and j is smallest k such that $A^k_{ij} > 0$.
Now for the next iteration, k+1, we will have, $A^{k+1}_{il} = \sum A^k_{ij} * A_{j,l}$, where j stands for all nodes from 1 to n.
We know from k^{th} iteration, that there exists a path from i to j ($A^k_{ij} > 0$) and if there is a direct edge from j to any node l ($A_{j,l} > 0$), then we will get $A^k_{ij} * A_{j,l} > 0$.
And we know that in unweighted graph, if two nodes are directly connected, their distance will be 1 which is shortest for all neighbouring graphs.
Hence, if we are getting a shortest path from i to j from first k iterations and we know that j is directly connected to l, we are guaranteed to get a shortest path from i to l in the $(k+1)^{th}$ iteration.

b)

Here, we are given that multiplying two NxN matrices take $O(n^{2.5})$ time.
We also proved above that for any two nodes i and j, if 'k' is the length of shortest path, then $A^k_{ij} > 0$.
This means we will have to multiple A with itself for k number of times.
And if one matrix multiplication takes $O(n^{2.5})$ time, 'k' matrix multiplications will take $O(k * n^{2.5})$.
Also, in time $O(k*n^{2.5})$, we will get all the pairs of edges whose $shortestpaths \leq k$.

c)

We are asked to prove that the probability of sampling $2 \log_e n * \frac{n}{k}$ vertices from the graph such that they don't lie on path of length k (i to j) is $\leq \frac{1}{n^2}$.
The probability of choosing a vertex such that it lies on a path of length k is $\frac{k}{n}$ and NOT choosing such vertex is $1 - \frac{k}{n}$.

And for $2\log_e n * \frac{n}{k}$ vertices, probability would be

$$(1 - \frac{k}{n})^{2\log_e n * \frac{n}{k}}$$

.
We are given that $1 - x \leq e^{-x}$, Hence, Probability of sampling $2\log_e n * \frac{n}{k}$ such that they are not in path of length k+1 is,

$$\leq e^{-\frac{k}{n} * 2\log_e n * \frac{n}{k}}$$

$$\leq e^{-2\log_e n}$$

$$\leq e^{\log_e n^{-2}}$$

$$\leq n^{-2}$$

$$\leq \frac{1}{n^2}$$

Hence Proved.

References

1) Striver - Youtube