CS 6150: HW 4 – Graph algorithms: shortest path

Submission date: Friday, November 10, 2023, 11:59 PM

This assignment has 6 questions, for a total of 55 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers. As default, n represents the number of nodes while m represents the number of edges in a graph.

Question	Points	Score
Safest path	7	
Shortest and simplest path	7	
Going electric	7	
Rendezvous location	7	
Lucky paths	12	
All-Pairs Shortest Path	15	
Total:	55	

Question 1: Safest path . . . . **[7]**

Let $G = (V, E)$ be a directed graph with n vertices and m edges. Imagine that the edges represent the streets of a dangerous city, and our goal is to get from vertex s to vertex t in the safest possible way. For an edge e, we are given the safety probability $p_e \in (0, 1)$. For a path consisting of a sequence of edges, the safety probability is defined as the product of the safety probabilities of the edges on the path.

Given the probabilities p_e for all edges, show how to find the safest (path with maximum safety) going from s to t. You may use Dijkstra's algorithm as a subroutine. [*Hint:* Apply Dijkstra on a graph with appropriately chosen edge weights.]

Present the general idea	3 points
Provide correct algorithm	2 point
Provide convincing explanation of correctness	2 point

Answer:

Derive a graph G' from G by setting the weights $w_e = -\log(p_e)$ for each edge e. Since $p_e \in (0, 1)$, we can see that $\log(p_e) < 0$ and thus $w_e = -\log(p_e) > 0$. Now the new graph G' consists of positive weights for edges. Thus, we can apply Dijkstra's algorithm to find the shortest path from s to t in the modified graph G', corresponding to the safest path in the original graph G.

Correctness: Our target is the find the path R from s to t such that $\prod_{e \in R} p_e$ is maximized. We can see that since the log function is monotonic increasing, the path that maximizes $\prod_{e \in R} p_e$ is the same as the path that maximizes $\log \prod_{e \in R} p_e = \sum_{e \in R} \log(p_e)$. This equals to the path that minimizes $-\sum_{e \in R} \log(p_e) = \sum_{e \in R} -\log(p_e) = \sum_{e \in R} w_e$. This is exactly the result of Dijkstra's algorithm on the modified graph G'.

Runtime: Calculating weights w_e takes $O(m)$ time. The Dijkstra algorithm takes $O((m + n) \log n)$. m is the number of edges while n is the number of nodes.

Question 2: Shortest and simplest path . . . . **[7]**

Let $G = (V, E)$ be a directed graph with **integer** edge lengths $\{w_e\}$. In class, we saw that Dijkstra's algorithm finds the shortest path (in terms of total length) between given vertices u and v in time $O((m + n) \log n)$ time. However, this does not pay attention to the number of "hops" (i.e., the number of intermediate vertices) on the path. In graphs where there are multiple shortest paths (in terms of total length), suppose we wish to find the one with the smallest number of hops.

Show how to solve this problem also in time $O((m + n) \log n)$ time. [*Hint:* What if you slightly perturb the original weights?]

Present the general idea	3 points
Provide correct algorithm	2 point
Provide convincing explanation of correctness	2 point

Answer:

The idea is to perturb the edges a bit but in a way that we preserve $w(e_i) < w(e_j)$.
For example, like the directed graph in Fig 1.
In the original graph, there exist two shortest paths from $A \to E$:
$A \to B \to C \to D \to E$ and $A \to G \to H \to E$. Each has length 4.
Thus, if we add some small value ϵ on each edge.
The length of two paths is changed to:
$A \to B \to C \to D \to E$: $4 + 4\epsilon$
$A \to G \to H \to E$: $4 + 3\epsilon$.
The path $A \to G \to H \to E$ becomes the shortest path. The number of hops is put into consideration in the shortest path finding.
However, we must be aware that the ϵ cannot be selected too large such that it will dominate the path

selection.
For example, if $\epsilon = 2$. The length of path $A \to F \to E$ becomes $5 + 2\epsilon = 9$, which is less than path $A \to G \to H \to E$ $(4 + 3\epsilon = 10)$. This will change the original shortest-path candidates.
Thus, we pick $\epsilon < \frac{1}{m}$. For any path, the perturbation can only at most add $< m \cdot \frac{1}{m} = 1$ volume to the path length. The edge lengths w_e are integer, which means the difference between the length of the shortest path and other path must be ≥ 1. So the perturbation cannot influence the original shortest-path candidates.
Running time: Since we have an extra preprocessing/perturbing step of $O(m)$ and everything else is the same, the total running time will be $O((m + n) \log n)$

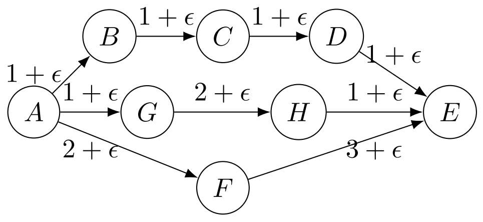

Figure 1: DAG Example in Q2

Question 3: Going electric . . . . **[7]**
Imagine you just bought your dream EV, and you found out its range is R miles. You're planning a road trip from place A to place B (vertices on a directed graph (V, E) with edge lengths $\{w_e\}$). Given the locations of all the superchargers on the graph, find out if it's feasible to go from A to B using the superchargers along the way. (Assume that the car is fully charged at the start vertex A.) Specifically, give an algorithm and show that its running time is polynomial in n, m. [For clarity, you do not need to worry about getting back from B to A.]

Present the general idea	3 points
Provide correct algorithm	2 point
Provide convincing explanation of correctness	2 point

Answer:

We use a DFS approach to solve the problem. The traversal tracks the EV's remaining range C by checking for distance traveled (w), and recharging at supercharger stations during the traversal. By checking if the EV can reach each subsequent node with its current range, and recharging at supercharger nodes, the algorithms explores all viable paths within the range contraints. The algorithm's runtime is polynomial in the number of nodes (n) and edges (m), as it traverses each edge at most once and performs a constant amount of work at each node.

Algorithm 1: Algorithm for Question 3: Going electric

```
Result: Whether the EV can afford the road trip from A to B(True or False)
Given a direct graph (V, E) and R miles as EV capability;
$C$ means the remaining miles the EV can afford when arriving at the current $node$;
$w(node \to child)$ means the consumption of EV when passing the edge $node \to child$;
record is an array of size $n$, recording the highest remaining miles when the EV visits this node! Initially, all element is -1!;
Initially call function checkPath($A$, $R$);
;
Function checkPath(node, C):
if node is B then
    return True
end
if C is less than or equal the record[node] then
    return False
end
if node has a supercharger then
    set C = R (EV is charged as full.)
end
record[node] = C;
for child of node do
    if $C - w(node \to child) >= 0$ then
        result = checkPath(child, $C - w$);
        if result == True then
            return True
        end
    end
end
return False
```

The algorithm is like DFS. Let's be clear that $w_e \geq 0$. n is the number of nodes in the graph equal $|V|$. m is the number of edges in the graph equal $|E|$.
For some node $v_i \in V$, $0 \leq i \leq n-1$ and $i \in Z$, the worst case is going through all its children.
Let's define ne_i as the neighbor of node v_i. Thus, we say for each $v_j \in ne_i$, there exists $e_{ij} \in E$. e_{ij} is the directed edge from node v_i to node v_j. Define $|ne_i|$ as the number of neighbour of node v_i
So the run time of the algorithm can be shown as(C is some constant number)

$$\sum_{v_i \in V} (C + |ne_i|) \leq Cn + m \in O(n+m)$$

.
Thus run time of the designed algorithm is $O(n+m)$, a polynomial of n and m.

Question 4: Rendezvous location . . . . **[7]**

Alice and Bob are located at vertices A and B of a directed graph (V, E) with edge lengths $\{w_e\}$. They wish to find a place to meet that is as close as possible to both of them. Specifically, they wish to find a $w \in V$ such that $\max\{d(w, A), d(w, B)\}$ is as small as possible. [As usual, $d(\cdot, \cdot)$ denotes the shortest path distance in the graph.]

Give an algorithm with running time $O((m+n)\log n)$ that finds such a w. [*Hint:* The view of Dijkstra's algorithm growing a ball around a vertex may be useful.]

Answer:
The algorithm is as below:

Present the general idea	3 points
Provide correct algorithm	2 point
Provide convincing explanation of correctness	2 point

Algorithm 2: Algorithm for Question 4: Rendezvous location

```
Result: The node $w$ in the graph or no such node!
Given a direct graph (V, E) ;
Settings;
Keep two minimum heaps $H_A$ and $H_B$ with distance as key, storing pair (distance, node):
  representing until now, the shortest path from A and B to the node;
Keep two sets $S_A$ and $S_B$ recording the nodes which have already been picked by the algorithm;
Keep two variables $D_A$ and $D_B$ to record the shortest path from A and B to some node until now.
  Initially $D_A = D_B = 0$;
Push pair (0, A) to $H_A$ while (0, B) to $H_B$;
w(node→child) $\geq 0$ means distance from node→child;
;
if A and B are both leaf nodes then
    return No such node exists!
end
while $H_A$ or $H_B$ not empty do
    if ($D_A \leq D_B$ and $H_A$ is not empty) or ($H_B$ is empty) then
        (dis, node) = head of $H_A$;
        if node in $S_B$ then
            return $w$=node
        end
        for child of node do
            if child not in $S_A$ then
                push (dis+w(node→child), child) into $H_A$;
            end
        end
        push node into $S_A$;
        $D_A += dis$;
    end
    else
        (dis, node) = head of $H_B$;
        if node in $S_A$ then
            return $w$=node
        end
        for child of node do
            if child not in $S_B$ then
                push (dis+w(node→child), child) into $H_B$;
            end
        end
        push node into $S_B$;
        $D_B += dis$;
    end
end
return No such node exists!
```

The algorithm is like two Dijkstra algorithms starting from A and B concurrently. Let's be clear that $w_e \geq 0$. n is the number of nodes in the graph equal $|V|$. m is the number of edges in the graph equal

$|E|$.
Explanation of pseudocode:
The key point of the algorithm is to make the shortest path from A and B to some point w as equal as possible. Thus, we use Dijkstra's algorithm starting from A and B concurrently.
The path from A to the node already in S_A is shorter than the one to be treated in H_A. When the node popped from H_B for processing, it is considered as the pivot of the shortest path from B. The remaining nodes in H_B must result in a longer path than the current one popped from H_B. The ones already in S_B are not valid due to not being checked by the A part. So if the node is in S_A, it is the one we are looking for.
The same argument is for H_A and S_B.
Thus, with this, we can guarantee that the resulted vertex w is a minimum of $max(d(w, A), d(w, B))$.

Runtime check:
Let's define n_A as the maximum number of nodes in the heap H_A while n_B as the maximum number of nodes in the heap H_B. It's easy to show $n_A + n_B \leq n$
Thus, when we pop head from heap! It will take *log* time to reorder the nodes! So It will take no larger than

$$n_A \log n_A + n_B \log n_B \leq n \log n$$

for pop operations!
Then, for the operation of pushing nodes into a heap, it also takes log time to reorder the structure! let's define $v_i^A \in V$, $0 \leq i \leq n-1$ and $i \in Z$ as the node of graph which is processed by heap operation of A side while v_j^B of B side.
Let's define ne_i^A as the neighbor of node v_i^A while ne_j^B of v_j^B. Define $|ne_i^A|$ as the number of neighbour of node v_i^A while $|ne_j^B|$ as the number of neighbour of node v_j^B

So the run time of the heap push operation is no larger than

$$\sum_{v_i^A \in V} |ne_i^A| \log n_A + \sum_{v_j^B \in V} |ne_j^B| \log n_B \leq (\sum_{v_i^A \in V} |ne_i^A| + \sum_{v_j^B \in V} |ne_j^B|) \log n = m \log n$$

.
Thus run time of the designed algorithm is $O((n + m) \log n))$.

Question 5: Lucky paths..**[12]**

Calvin would like to travel from vertex u to vertex v in a directed *unweighted* graph $G = (V, E)$. Being quirky, Calvin believes that paths whose lengths are not multiples of 3 are unlucky, and avoids them at all costs. Thus, Calvin's goal is to find the shortest path from u to v whose length is a multiple of 3. [He is OK with visiting the same node or traversing the same edge multiple times in order to achieve this goal.]

(a) **[6]** Consider the following algorithm: first construct a graph $G' = (V, E')$ whose vertex set is V, where there is an edge ij if and only if there is a directed path of length 3 from i to j in G; next, find the shortest path from u to v in G'. Show how to complete this algorithm so as to return the desired shortest path (i.e., return all the vertices on the path). Also discuss its correctness and running time.

Complete algorithm	2 points
Discuss correctness	2 point
Discuss performance	2 point

Answer:

The algorithm is mostly complete, there are two major missing steps:

1. The vertices in between each i and j during the construction step need to be saved somewhere so we can extract them back out to reconstruct the full path at the end. We can accomplish this by storing them in a dictionary while constructing G', so we can recall them with $dict([i, j])$.

2. The original path must be produced from the one found in G'. Here we just use our dictionary from step 1 while following the previous pointers left behind by the shortest path algorithm. In between any two vertices u and v, add the vertices from $dict([u, v])$ to the list of path vertices.

Because large chunks of the algorithm are provided by the question as black box routines the pseudocode looks a little sparse, but here it is:

```
Algorithm 3: Mystery Lucky Paths
1 Construct G' and dict
2 Find shortest path from S' to E' in G'
3 X = E'
4 while X is not S' do
5     add X to path
6     add dict([X.prev, X]) to path
7 return path
```

This algorithm returns the shortest path because each edge in G' represents a hop that maintains the "multiple of 3" property. We start at a multiple of 3 (0) and with each edge we remain a multiple of 3, so the shortest path that reaches E' will be the shortest path with a length multiple of 3. When reconstructing the finished path, we use the original vertices instead of the summarized ones, so the answer meets the original requirements.

The runtime of this algorithm depends heavily on the runtime of lines 1 and 2, steps which were provided to us beforehand. However, line 1 can be accomplished with a naive "for each vertex, run BFS and stop at length 3" approach, which gives us an upper limit of O(n(n+m)). Line 2 can simply be done with BFS, since G' is just an unweighted graph, which gives us O(n+m). The final path reconstruction takes O(n) in the worst case, since the path may use all vertices in the graph but each one can only be pulled out once. We can bound the runtime overall with O(n(n+m)), though it could potentially run faster if line 1 was solved faster.

(b) **[6]** By modifying the standard breadth first search procedure, show how to compute the answer in time $O(m + n)$, where as usual, n, m are the number of vertices and edges of G respectively. [*Hint:* instead of just remembering if a vertex is visited, remember the length of the path used to visit the vertex, modulo 3.]

Present the general idea	2 points
Provide correct algorithm	2 point
Provide convincing explanation of correctness	2 point

Answer:

Since we know we need to modify BFS to solve this problem, let's start with the base version (simplified):

Algorithm 4: BFS simplified

```
Insert S into queue
while queue not empty do
    pop V from queue
    if V is not marked then
        if V is E then
            stop and do wrap-up
        mark V
        add all unmarked neighbors of V to queue
```

First, we need some means of tracking our path length mod 3, otherwise we wouldn't know if we had met the "multiple of 3" condition. When adding a vertex to the queue, let's also associate it with some value mod 3 representing the length of the path taken to reach it.

Next, we must also account for cycles and paths of different lengths. Regular BFS completely blocks out all extraneous paths with the marks, because revisiting vertices is redundant work and gets us stuck in an infinite loop. Here however, if we have two paths to the same vertex that have different lengths mod 3, then these are not redundant and can reach other vertices with different mod values. Instead of marking vertices as "visited", we should mark them with the mod value used to reach them, and only stop if we reach them with the same value. If we have a different mod value, we should keep going. This will allow us to reach the same vertex up to 3 times, but only if we have a different mod value each time.

Finally, we need to change the end condition. Instead of simply popping E out of the queue, we need to pop E with a value of 0 in order to stop, otherwise we keep going. During the wrap-up stage where we extract the path itself we will also need to include up to 3 different previous pointers, one for each possible mark. The new pseudocode for all of these changes is as follows:

Algorithm 5: 3BFS

```
1  Insert (S, 0) into queue
2  while queue not empty do
3      pop (V, d) from queue
4      if V is not marked with d then
5          if V is E and d is 0 then
6              break
7          mark V with d
8          add all neighbors of V that don't have the (d+1 mod 3) mark
9  X = E, d = 0
10 while X is not S and d is not 0 do
11     add X to path
12     X = X.prev(d)
13     d = d-1 mod 3
14 return path
```

(line 12 is getting the previous vertex that corresponds to the current mod value as discussed above.)

This algorithm works for the same reasons that regular BFS works: the FIFO nature of the queue means that the first path to reach E (represented by its endpoint vertex inside the queue) was the shortest path available. Our modifications have expanded the number of times we can visit vertices but only as necessary, and the ending requirement has been changed to meet the new restriction, both modifications discussed above.

In the absolute worst case we visit every vertex in the graph a maximum of 3 times, the equivalent of 3 BFSes. Therefore the runtime of this algorithm is O(3(m+n)) or O(m+n).

Question 6: All-Pairs Shortest Path . . . . **[15]**

The goal of this problem is to introduce you to the All-Pairs Shortest Path (APSP) problem that we didn't get a chance to study in class. Given a directed graph $G = (V, E)$ with non-negative edge lengths $\{w_e\}$, we define the *distance matrix* M as the $n \times n$ matrix ($n = |V|$ as usual) whose i, j'th entry is the shortest path distance between vertices i and j. Given the graph (vertices, edges and lengths), the goal of the APSP problem is to find the matrix M.

In what follows, let G be an **unweighted, undirected** graph (all edge lengths are 1). Thus, in this case, shortest path from one vertex u to the rest of the vertices can be found via a simple BFS. (Thus the APSP problem can be solved in time $O(n(m + n)) = O(n^3)$.)

Let A denote the *adjacency matrix* of the graph, i.e., an $n \times n$ matrix whose ij'th entry is 1 if ij is an edge, and is 0 otherwise. Now, consider powers of this matrix A^k (defined by traditional matrix multiplication). Also, for convenience, define $A^0 = I$ (identity matrix).

(a) [**5**] Prove that for any two vertices i, j, the distance in the graph $d(i, j)$ is the smallest $k \geq 0$ such that $A^k(i, j) > 0$.

Present the general idea	3 points
Provide correct proof	2 point

Answer:
Let's prove by induction.
Base case: $k = 1$, A is the adjacency matrix of the graph. If the shortest path from i to j is equal to 1, $A(i, j) > 0$. Otherwise, the distance between the two vertices i and j is greater than $k = 1$.

Suppose the statement is held for $k = q > 1 \in Z$, which means if the shortest path between vertex i and p is q, then $A^q(i, p) > 0$ while $A^{q-1}(i, p) = 0$.
Let's select one vertex j such that $A^q(i, j) = 0$ while $A(p, j) = 1$, meaning the shortest path from i to j must be greater than q and there exists an edge between p and j. Thus, it's trivial to construct the shortest path from $i \to j$ by combining the shortest path from $i \to p$ and the edge between p and j. The resulting path has length $q + 1$. Let's calculate the related entry of A^{q+1}:

$$A^{q+1}(i, j) = \sum_{l=0}^{n-1} A^q(i, l) \cdot A(l, j) \geq A^q(i, p) \cdot A(p, j) > 0$$

and

$$A^q(i, j) = 0$$

Thus, if the shortest path between i and j is equal k, thus, $A^k(i, j) > 0$ while $A^{k-1}(i, j) = 0$

(b) [**4**] The idea is to now use fast algorithms for computing matrix multiplications. Suppose there is an algorithm that can multiply two $n \times n$ matrices in time $O(n^{2.5})$. Use this to prove that for any parameter k, in $O(kn^{2.5})$ time, we can find $d(i, j)$ for all pairs of vertices (i, j) such that $d(i, j) \leq k$. In other words, we can find all the *small* entries of the distance matrix. Let us see a different procedure that can handle the "big" entries.

Present the general idea	3 points
Provide correct proof	2 point

Answer:
We have proved in part (a) that the distance is equal to the smallest $k \geq 0$ such that $A^k(i,j) > 0$. We can simply keep another matrix B of the same size (with all entries marked to some large value N [in this setting even k+1 works]) and do an iterative calculation of the powers of A. We know that non-zero entries of A give the pairs of vertices with the path of length 1. Let (i,j) be such a pair. We set $B(i,j) = 1$. Assume we are at step m and we have calculated A^m, we can check A^m for non-zero entries. Let $A^m(i,j)$ be non-zero, then we check $B(i,j)$ and if it is N we change it to m. Observe that the update will only happen if a path hasn't been found before and from our argument in part (a), this will be the shortest path. We can see that at each step $t-1$, we take $O(n^{2.5})$ to get A^t from A^{t-1} and checking for non-zero entries and updating takes just $O(n^2)$. So for each step the complexity is $O(n^{2.5})$ and since all the ones with $d(i,j) \leq k$. Thus, overall it takes $k-1$ steps so the time complexity is $O(kn^{2.5})$.

(c) [**6**] Let i,j be two vertices such that $d(i,j) \geq k$. Prove that if we sample $(2\ln n) \cdot \frac{n}{k}$ vertices of the graph uniformly at random, the probability of not sampling any vertex on the shortest path from i to j is $\leq \frac{1}{n^2}$. [*Hint:* You may find the inequality $1 - x \leq e^{-x}$ helpful.]

Present the general idea	3 points
Provide correct proof	3 point

Answer:
We can see that the number of vertices on the shortest path from i to $j(i,j$ inclusive) is $d(i,j)+1$. Due to sampling uniformly at random, the probability of the sampled vertex being one of the vertices in the shortest path is $\frac{d(i,j)+1}{n} \geq \frac{k+1}{n}$. Thus, the probability of the sampled vertex not on the shortest path is $1 - \frac{d(i,j)+1}{n} \leq 1 - \frac{k+1}{n} \leq e^{-\frac{k+1}{n}}$. Due to sampling independently, the probability of none of the $2\ln n\frac{n}{k}$ vertices on the shortest path is $(1 - \frac{d(i,j)+1}{n})^{2\ln n\frac{n}{k}} \leq (1 - \frac{k+1}{n})^{2\ln n\frac{n}{k}} \leq e^{-\frac{k+1}{n}\cdot 2\ln n\frac{n}{k}} = e^{-(2\ln n)\cdot\frac{k+1}{k}} = \frac{1}{n^{2\cdot\frac{k+1}{k}}} \leq \frac{1}{n^2}$
With very little effort following this (exercise for the interested students), one can obtain an algorithm for APSP (on undirected, unweighted graphs) that runs in time $O(n^{2.75}\log n)$, which is considerably better than $O(n^3)$.