CS 6150: HW 4 – Graph algorithms: shortest path

Submission date: Friday, November 3, 2023, 11:59 PM

This assignment has 6 questions, for a total of 50 point plus 5 bonus points (graded on a scale of 50). Unless otherwise specified, complete and reasoned arguments (proof that an algorithm is correct and of its running time) 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.]

Ans) We know that Dijkstra is applied to get the path with shortest length. So we will change the edge length. We will need 2 modifications.

1. We want to put all the edge lengths in such a function that if the probability is very high, the edge is very small. This can be achieved if we add log to the probabilities. Ex: if probability for a particular edge is 0.9 then the new edge length would be log(0.9) that is -0.04
2. Since we are getting answer in negative, we will add a -ve sign to this to get the right answer. Ex: p_1 the new edge length would be $-\log(p_1)$

We will do this for every edge.
Therefore the new edges would be:
$-\log(p_1)$
$\log(p_1)^{-1}$
$\log(1/p_1)$
This will happen for all edges.
Now we can apply Dijkstra on the new edge lengths.
Another issue that would be solved using log is that Dijkstra adds all the edges. Using log that can be converted in product form, which is asked in the question.
Lets consider that the probabilities for reaching from s to t would require to traverse through $p_1, p_2 and p_3$ edge weights, after applying Dijkstra.
Solution would be:
$-\log(p_1) - \log(p_2) - \log(p_3)$
$-\log(p_1 * p_2 * p_3)$
$\log(p_1 * p_2 * p_3)^{-1}$
$\log(\frac{1}{p_1 * p_2 * p_3})$

Algorithm 1 Dijkstra

```
procedure DIJKSTRA(graph, source)
    Create an empty priority queue $Q$
    Initialize distances for all vertices to $\infty$
    Set the distance of the source vertex to 0
    Add the source vertex to $Q$ with distance 0
    while $Q$ is not empty do
        $current\_vertex \leftarrow$ vertex in $Q$ with the smallest distance
        Remove $current\_vertex$ from $Q$
        for each neighbor of $current\_vertex$ do
            $tentative\_distance \leftarrow$ distance from source to $current\_vertex$ + distance from $current\_vertex$ to neighbor
            if $tentative\_distance <$ distance to neighbor then
                Set distance to neighbor = $tentative\_distance$
                Add neighbor to $Q$ with distance $tentative\_distance$
            end if
        end for
    end while
    return distances
end procedure
```

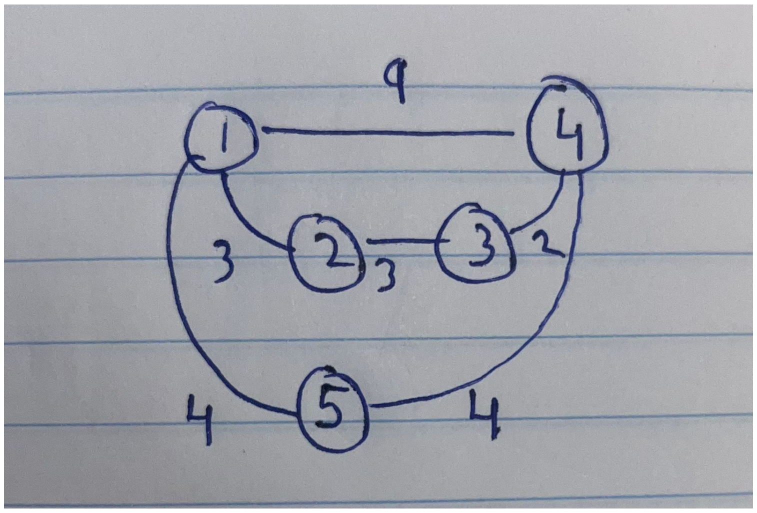
Time complexity would be $O((n+m)\log(n))$
We get minimum distance using priority queue, we go on each node in $O(\log n)$. For n nodes it would take $O(n * \log n)$
For the worst case it would take $O(m * \log m)$ for m nodes.
Therefore total time would be $O((n+m)\log(n))$

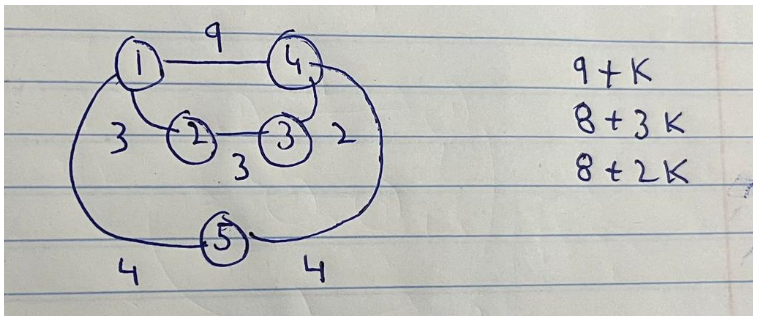
Question 2: Shortest and simplest path .... **[7]**

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?]

Ans) We know Dijkstra's algorithm doesn't see the number of hops to find the solution. We have to change the edge lengths in such a way that hops are included. Consider this graph.
We see that the last 2 paths have the same length but number of hops in the second is 3 whereas number of hops in third is 2.
So we add a constant k to each edge length.
It will be something like this.
We know Dijkstra uses minimum length to give results.
We want to ensure that k doesn't increase length more than the edge that is not in contest.
Ex: here 8+3k and 8+2k are in contest for the path with least hops, but we don't want 9+k to somehow be less than 8+3k because of k and gets chosen by Dijkstra. Ex: if we take k as 2 we can see 9+k would be selected as it would be the least.
Therefore
8+3k<9+k
2k<1
k<1/n
k=1/1+n
Where n is 1,2,3...





The algorithm we are using is Dijkstra using priority queue which will give us time complexity of $O(m + n)(\log(n))$

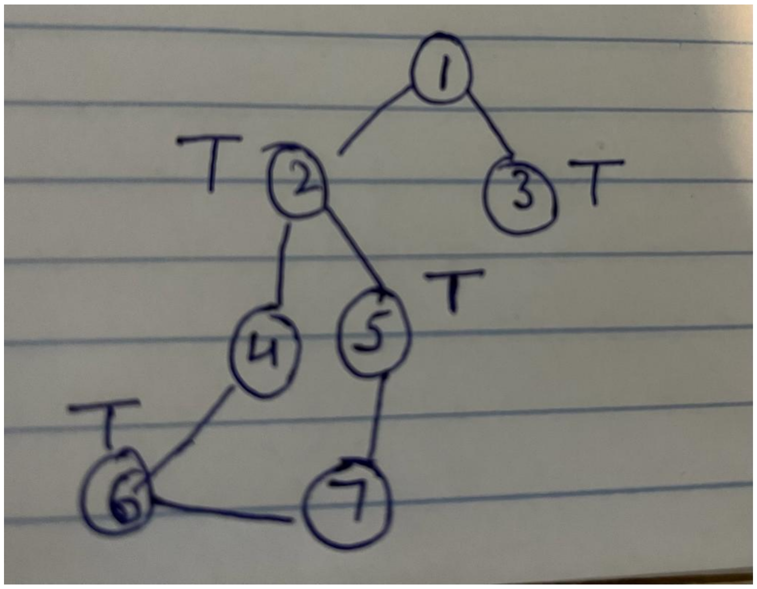
Algorithm 2 Dijkstra

```
1:  procedure DIJKSTRA(graph, source)
2:      Create an empty priority queue $Q$
3:      Initialize distances for all vertices to $\infty$
4:      Set the distance of the source vertex to 0
5:      Add the source vertex to $Q$ with distance 0
6:      while $Q$ is not empty do
7:          $current\_vertex \leftarrow$ vertex in $Q$ with the smallest distance
8:          Remove $current\_vertex$ from $Q$
9:          for each neighbor of $current\_vertex$ do
10:             $tentative\_distance \leftarrow$ distance from source to $current\_vertex$ + distance from $current\_vertex$ to neighbor
11:             if $tentative\_distance <$ distance to neighbor then
12:                 Set distance to neighbor = $tentative\_distance$
13:                 Add neighbor to $Q$ with distance $tentative\_distance$
14:             end if
15:         end for
16:     end while
17:     return distances
18: end procedure
```

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.]

Ans) We will apply Breadth First Search(BFS). This will help us to go to every node(neighbours).
Constraint here is that we would go to nodes which don't have superchargers as well. Say the edge length is x for a non-charger. Then the distance(edge length) would change to R-x.
But we won't traverse to the neighbours whose total edge length goes beyond R.



In this picture, we would traverse from 2 to 6 by 4 even though 4 doesn't have supercharger. If distance between 2 and 6 is less than R, it goes through.

Algorithm 3 BFS

```
1:  procedure BFS(graph, start, $R$)
2:      Create an empty queue $Q$
3:      Create a set *visited*
4:      Enqueue *start* into $Q$
5:      Mark *start* as visited
6:      while $Q$ is not empty do
7:          *current_vertex* ← dequeue from $Q$
8:          Process *current_vertex*
9:          for each neighbor of *current_vertex* do
10:             if neighbor is not in *visited* and edge_length(*current_vertex*, neighbor) $\leq R$ then
11:                 Enqueue neighbor into $Q$
12:                 Mark neighbor as visited
13:             end if
14:         end for
15:     end while
16: end procedure
```

Time complexity would be O(n+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.]

Ans)

We will apply Dijkstra two times.
One from A side and second from B side.
Since we are not exactly finding the mid point between A and B.
At a particular point, A would have more length than B or vice versa.
Therefore we take maximum of (w,A),(w,B).
Whichever has larger edge length we would store that node in a queue.
Then we will find the minimum of those nodes. This will be our meeting point.

Algorithm 4 Dijkstra

```
1: procedure DIJKSTRA(graph, source1)
2:     Create an empty priority queue $Q$
3:     Create an empty priority queue $R$
4:     Initialize distances for all vertices to $\infty$
5:     Set the distance of the source vertex to 0
6:     Add the source vertex to $Q$ with distance 0
7:     while $Q$ is not empty do
8:         current_vertex ← vertex in $Q$ with the smallest distance
9:         Remove current_vertex from $Q$
10:        for each neighbor of current_vertex do
11:            tentative_distance ← distance from source to current_vertex + distance from current_vertex
   to neighbor
12:            store edge in R.
13:            if tentative_distance < distance to neighbor then
14:                Set distance to neighbor = tentative_distance
15:                Add neighbor to $Q$ with distance tentative_distance
16:            end if
17:        end for
18:    end while
19:    DIJKSTRA(graph,source2)
20:    $max(e_A, e_B)$ (e is edge, stored in R)
21:    min(R)
22:    return min(R)
23: end procedure
```

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.

Ans)

Graph G' = (V, E'), vertex set is V, there is an edge (ij) if there is directed path of length 3 from i to j in G.

We need to find shortest path from u to v in G'.

Algorithm 5 Construct Graph G'

```
function constructGraphGPrime($G$)
    $G' \leftarrow$ createEmptyGraph($V$)                    ▷ $V$ is the set of vertices in $G$
    for each vertex $i$ in $V$ do
        for each vertex $j$ in $V$ do
            if existsPathOfLengthThree($G, i, j$) then
                addDirectedEdge($G', i, j$)
            end if
        end for
    end for
    return $G'$
end function
```

Algorithm 6 Find Shortest Path in G'

```
function findShortestPathInGPrime($G', u, v$)
    $queue \leftarrow$ createEmptyQueue
    $predecessors \leftarrow \{\}$
    markAsVisited($u$)
    enqueue($queue, (u, [])$)
    while not isEmpty($queue$) do
        $(current, path) \leftarrow$ dequeue($queue$)
        if $current = v$ then
            return $path$
        end if
        for each $neighbor$ in neighborsOf($G', current$) do
            if not isVisited($neighbor$) then
                markAsVisited($neighbor$)
                enqueue($queue, (neighbor, path + [neighbor])$)
                $predecessors[neighbor] \leftarrow current$
            end if
        end for
    end while
    return []                                                ▷ No path found
end function
```

Algorithm 7 Return Shortest Path

```
function returnShortestPath($G, u, v$)
    $G' \leftarrow$ constructGraphGPrime($G$)
    $shortestPath \leftarrow$ findShortestPathInGPrime($G', u, v$)
    return $shortestPath$
end function
```

Running time:
O(Construct Graph G')+O(Find shortest path in G')+O(Return shortest path)
$O(m + n(n)) + O(n^3) + O(n)$
$O(n^3)$
Correctness: For every node V, up to 3 hops were done. SO to reach from i to j 3 hops had to be done. We are applying BFS to get G'. We are using queue to store nodes and dictionary is used to reconstruct the path.

(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.]

Ans) To achieve constant-time path length retrieval, add a variable in the queue to store path length.

Algorithm 8 Modified BFS

```
procedure ModifiedBFS($G, u, v$)
    Initialize three visited arrays: $\text{visited}_0$, $\text{visited}_1$, $\text{visited}_2$
    for each node $i$ in $V$ do
        Set $\text{visited}_0[i]$ to false
        Set $\text{visited}_1[i]$ to false
        Set $\text{visited}_2[i]$ to false
    end for
    Initialize queue with source node and empty path set: $queue = \{(u, \{\})\}$
    while queue is not empty do
        (currNode, currPathSet) ← dequeue(queue)
        currPathModulo ← len(currPathSet)  mod 3
        Update specific $\text{visited}_{\text{modulo}}$[currNode] based on currPathModulo
        if currNode == $v$ and currPathModulo == 0 then
            return currPathSet + currNode                ▷ Path in BFS order
        else
            for each neighbor of currNode do
                updatePathSet ← currPathSet + currNode
                updatePathModulo ← len(updatePathSet)  mod 3
                if $\text{visited}_{\text{updatePathModulo}}$[neighbor] is false then
                    enqueue(queue, (neighbor, updatePathSet))
                    Mark $\text{visited}_{\text{updatePathModulo}}$[neighbor] as true
                end if
            end for
        end if
    end while
end procedure
```

Modify BFS to track the number of times each node is visited and store the path taken from the source to that node. Maintain three visiting arrays for each modulo, preventing repetition in the path. Running time remains O(n + m), same as normal BFS, with only constant-time checks added. Return the destination only when the length of the path set is divisible by 3, ensuring a multiple of 3 path.

Correctness:

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 $A^k(i,j) > 0$.
Ans) We will use induction.
Base case:$A^0(i,i) \quad \forall i \in V$
For any $t \leq k$ distance for two vertices will be smallest for t $A^t(i,j) > 0$ will hold true.
Inductive step:
$A^t(i,j) = 0 \quad \forall t \leq k$ and $A^{k+1}(i,j) > 0$
Therefore
d(i,j)=k+1
But if $d(i,j) \leq k$
Therefore for some $t \leq k$ for which $A^t(i,j) > 0$
We get a contradiction from our consideration.
$A^{k+1} = A^k \times A$ and for $A^{k+1}(i,j) \geq 0$, there would be $u \in V$ for which $A * ki, u \geq 0$ and $A(u,j) > 0$
We can say, smallest distance $d(i,u) = k$ and $d(u,j) = 1$
Since the smallest distance between (i,u) is less than k, there exists a path from i to u and u to j, a possible path from i to j which takes less than k+1 distance.
Therefore there must exist some $t \leq k$ for $A^t(i,j) \geq 0$
This gives us contradiction.
Hence proved

.

(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.
Ans)
When $A^t(i,j) \geq 0$ we are storing $d(i,j) = 0$
We are performing 2 operations every time:

1. Matrix multiplication of $O(n^{2.5})$.
2. Iteration on a matrix of $O(n^2)$.

We are incrementing power of A^t.
For $A^t(i,j) \geq 0$ we are changing d(i,j) with smallest t.
We are doing this for k iterations, that means for matrices A^1 to A^k.
Therefore Running time: $k * n^{2.5} + k * n^2$
$O(n^2.5)$

(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.]

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 $O(n^3)$.
Ans)

Probability of n vertices of sample=$\frac{(2\ln n) * \frac{n}{k}}{n}$

Probability of not sampling in 1 iteration: $n - \frac{(2\ln n) * \frac{n}{k}}{n}$

Probability of not sampling in k iteration: $(n - \frac{(2\ln n) * \frac{n}{k}}{n})^k$

Simplifying: $(1 - \frac{(2\ln n)}{k})^k)$

Using the hint $1 - x \leq e^{-x}$

$(1-x)^k \leq e^{-kx}$

$x = \frac{(2\ln n)}{k}$

Probability of not sampling=$\leq e^{-kx}$

$\leq e^{-kx}$

$\leq e^{-k(\frac{(2\ln n)}{k})}$

$\leq e^{-2(\ln n)}$

$\leq e^{\ln n^{-2})}$

$\leq n^{-2(\ln e)}$

$\leq n^{-2}$

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