# CS 6150: HW 4 – Payman Behnam, U1078370
# Graph algorithms: shortest path

Submission date: Friday, November 1, 2019, 11:59 PM

This assignment has 6 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

| Question | Points | Score |
|---|---|---|
| Safest path | 5 | |
| Shortest and simplest path | 6 | |
| Going electric | 6 | |
| Rendezvous location | 6 | |
| Lucky paths | 12 | |
| All-Pairs Shortest Path | 15 | |
| Total: | 50 | |

Question 1: Safest path . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **[5]**

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.]

**Answer:** Since we want to pick the safest path, we should maximize the multiplications of consecutive $p_e s \in (0, 1)$ . However, we know that $\log AB = \log A + \log B$ . And if $0 < A < 1$ then $\log A < 0$. If we apply the $\log$ operation to the $p_e \in (0, 1)$ of every edge, we get some negative values. Low $p_e \in (0, 1)$ leads to larger negative number (magnitude value). Since we want to maximize $\prod_{i=1}^{k} p_i$, by applying log operation, we have $\log(\prod_{i=1}^{k} p_i) = (\sum_{i=1}^{k} \log p_i)$. Considering the fact that each log value is a negative number, we can store $-\log p_e$ and look for shortest path using Dijesktra Algorithm and $-\log p_e$ as weights . This is mainly due to the fact that, by having $-\log p_e$, the result of multiplication of value with high probabilities would be closer to 0 (low value), and the result of multiplication of value with low probabilities would be a high number.

**Algorithm 1** Safest Path

```
1: procedure SAFEST PATH
2:     Apply $-\log$ to each $p_e$ of the edge                         ▷ O(m)
3:     Call Dijkstra by considering the $-\log p_e$ as weights      ▷ O((m+n) log n)
```

According the above algorithm, we should apply -log into each edge that takes O(m) since each edge is visited once. Then, we need to apply Dijkstra that takes O((m+n) log n). We know that Dijkstra is correct. Apply log to take new weight is mathematically $-\log p_e$ correct. So, the algorithm is still correct.

As a separate approach we can consider the multiplication instead of addition in the Dijkstra's meaning that instead of looking for minimizing addition, maximizing the multiplication. We can use the following routine for maximizing:

https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

Question 2: Shortest and simplest path . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **[6]**

Let $G = (V, E)$ be a directed graph with **integer** edge lengths $\{w_e\}$. In class, we saw that Dijkstra's algorithm finds a 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., 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?]

**Answer:**

Solution1: We can add a very tiny value to the weight of each edge . So, the path with more hopes (that previously had equal additive weights) will have less cost. However, this value has to be carefully chosen such that is not change the desired result. For instance, suppose that there was a path that had a higher cost (longest path) but a lesser number of edges(hopes) before. After adding this tiny value, a previously shortest path can still remain the shorter path and the path with a higher cost (longest path) but a lesser number of edges (hopes) should end up having still higher cost.

What should be this tiny value $\epsilon$ ?

Let's say the graph has *m* edges and the shortest path (i.e. in the worst case) have *m-1* edges with the value of $S$. However, the shortest path value is slightly less than another path which has only 1 edge. (i.e. the wight of this path is $W + S$). So, after adding the tiny value (i.e., $\epsilon$), we have $(m-1) \times \epsilon + S$ for the shortest path and $W + S + \epsilon$ for another path. Still we should have $(m-1) \times \epsilon + S < W + S + \epsilon$ . Hence, $(m-2) \times \epsilon < W$ . Hence, $\epsilon < \frac{W}{m-2}$

**Algorithm 2** Shortest Path with minimum number of hopes

```
1: procedure SHORTEST PATH WITH MINIMUM NUMBER OF HOPES
2:     Compute $\epsilon$                                          ▷ O(1)
3:     Add $\epsilon$ to each edge weight                          ▷ O(m)
4:     Call Dijkstra by considering updated weights               ▷ O((m+n) log n)
```

According the above algorithm, we should compute $\epsilon$ with O(1), add $\epsilon$ into each edge that takes O(m) since each edge is visited once. Then, we need to apply Dijkstra that takes O((m+n) log n). We know that Dijkstra is correct and considering new weight is not changing Dijkstra function. Plus, new updated weight discriminates previous equal path. So, the algorithm is still correct.

To compute the $\epsilon$ we can compute the difference between two smallest edges and divide that obtained value by the number of edges as well.

Reference https://stackoverflow.com/questions/20041931/find-the-shortest-path-with-the-least-number-of-edges

Solution2: The basic idea is making use of the Dijkstra's algorithm. However, we should keep track of number of passed hopes from the source vertex in a separate array. If we find some shorter paths with the same weight, then we will take the one with less number of hopes. In other words each node will keep (distance,hopes) instead of distance. For the terminal node the one among the nodes with minimum distance, the ones with minimum hopes are chosen.

Question 3: Going electric . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **[6]**

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 $A$.]

**Answer**

Solution1: We can make use of the DFS. We start from source $A$ , do DFS with the constraint that we should reach the supercharger nodes or $B$ with less than or equal depths of $R$. If we don't reach neither $B$ nor any superchargers, we are done with *No Found*. If we reach the $B$, we return *Found* as a solution. If we reach supercharger, we should mark these nodes again. Then, we should start DFS from the reached charger node and try if we can reach other superchargers or $B$ (if we reached multiple charger nodes, take one). We repeat this procedure until we find *Found/Not Found*.

**Algorithm 3** Going electric

```
1: procedure GOING ELECTRIC()
2:     Apply DFS from $A$                                                  ▷ O(m + n)
3:     if Reach the $B$ with the depth of $\leq R$ miles then
4:         Return Found
5:     else if Reach the $superchargers$ with the depth of $\leq R$ miles then
6:         A= a reached $supercharger$ node
7:         Going electric()
8:     else if Neither reach the $superchargers$ Nor $B$ with the depth of $\leq R$ miles then
9:         Return Not Found                                                ▷ O((m+n) log n)
```

Since we are calling the DFS at most $n$ times and time complexity for the DFS is $O(m+n)$, the complexity would be at most $O(n^3)$ considering the fact that m can be $O(n^2)$.

The algorithm is correct. If there is a path between $A$ and $B$, the DFS can detect that. Putting a constraint for the depth of DFS, and calling it multiple times will not change the nature of the DFS . However, this constraint will make sure that either we get full charger and we can continue traversing the path.

Solution2: In the next approach, we can make use of all pairs shortest path (APSP). We call APSP $O(n^3)$ and find a matrix of the shortest paths between all pairs in the graph. With this information, now we can build a new graph in $O(n^2)$. This graph contains only A,B and all the supercharger nodes while the weight would be the information that we obtain by calling APSP. Please note than we keep the edges for the nodes A,B and superchargers only if there is a path between them in the original graph and the shortest path is less than or equal to R . These information can be obtained from output of APSP. Now, we can run DFS to see if we can reach from A to the B which takes $O(m + n)$

**Algorithm 4** Going electric

```
1: procedure GOING ELECTRIC()
2:     Apply APSP from $A$                                                    ▷ $O(n^3)$
3:     Build new graph containing $A$ $B$ and $superchargers$ with constraint that SP between nodes are $\leq R$ miles
   by information of first line                                               ▷ $O(n^2)$
4:     Run DFS on the new graph                                               ▷ $O(n^2)$
```

The algorithm is correct since we are only looking for feasible paths. The new graph is built upon the information that we obtained by running the APSP. So, it will not change initial information. Removing some edges that the length is grater than $R$ is also fine since we shouldn't consider them for the feasible path between $A$ and $B$. Since in the final graph there is no constraint, running DFS for the reachability analysis is fine. The time complexity is mentioned during algorithm description.

Question 4: Rendezvous location . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **[6]**

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 $\max\{d(A, w), d(B, w)\}$ 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 basic idea is growing a ball around vertecis *Alice* and *Bob* such that the balls can meet each other. The point(s) that these ball meet each other is(are) the desired point(s). To implement this, we should run Dijkstra algorithm from *Alice* and *Bob* as source nodes separately. We know that a version of Dijkstra can find the shortest path from a source to all other nodes. The output would be the shortest distance from *Alice* to other nodes that we store them in the *Alice Array* and *Bob Array* respectively. The next step is to do the one to one comparison between the element of these two arrays (i.e., for each corresponding node $w$) and store the max result in the *Output Array*. Finally, we need to find the minimum element(s) of the *Output Array*. These element(s) is(are) desired point(s).

**Algorithm 5** Rendezvous location

```
1: procedure RENDEZVOUS LOCATION
2:     Run Dijkstra considering Alice as source and store the result in Alice Array        ▷ O((m+n) log n)
3:     Run Dijkstra considering Bob as source and store the result in Bob Array            ▷ O((m+n) log n)
4:     Do one to one comparison for the corresponding nodes and put the max value in the Output Array    ▷ O(n)
5:     Traverse the Output Array and select the min element(s) that corresponded to specific node(s).    ▷ O(n)
```

As it is clarified in the algorithm, the running time is still O((m+n) log n). The algorithm is correct since if there is a path between Alice and Bob, Dijkstra detect that and return it. So, the shortest path from Alice to all other nodes, and from Bob to all other nodes have some common nodes (since there is a path between Alice and Bob, otherwise they cannot meet each other). By doing comparison for the corresponding nodes and extract the max value we are doing $\max\{d(A,w), d(B,w)\}$ operation asked in the question. Since we are looking for location that $\max\{d(A,w), d(B,w)\}$ is as small as possible, we have to select minimum ones for the shared nodes which the values are stored in separate array.

For the Dijkstra algorithm, I use the following references.

Reference: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7

Just in case, I kept the algorithm here:

Algorithm with complexity $O((m+n)logn)$.

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn't include all vertices

….a) Pick a vertex u which is not there in sptSet and has minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

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.

**Answer:**

I am following the format of describing the algorithm in a way it is described in the question partially:

In the $G' = (V, E')$, with new edges that are multiple of 3, we can run BFS (or DFS) to figure it out whether we Calvin to reach from $u$ to $v$ while the length is a multiple of 3.. For each new edge $E'$ we can keep a linked list with 4 element: The first element would be the id of the new created edge (i.e., $E'_i or NE'_i$) and other elements in the linked list would be the id of the removed edges in the original graph$G = (V, E)$, which is distinguished using the the vertex numbers (i.e. $v_i v_j ORv_j v_i ORv_i v_i$) . We can have a big connected linked list for all the edges or one per each new created edge. After applying BFS and getting the path from $u$ to $v$, we know the $ID$ of the edges that are outputed by BFS in the $G' = (V, E')$. So, we will go the linked list and traverse it until we reach the new desired edge $ID$. Then, we output the edges (that we kept from the original ones) until reach the new edge $ID$. This process should happen for all edges outputted by BFS.

Creating new $G' = (V, E')$ can happen by building $A^3$ whereas A is the adjacency matrix. The complexity would be $O(n^3)$ (please refer to the last question for detailed information about adjacency matrix and its complexity). Traversing linked list can happen $N$ times and would be $O(N)$ each time. So, the complexity would be $O(n^2)$. Insert and delete to the unsorted linked list would be $O(1)$ and search would be $O(n)$.

The algorithm is correct. The graph is unweighted; so, BFS (DFS) would work to find the path from the source to the destination. Plus, we are merging some consecutive edges that are multiple of three to one edge. We are not changing the connectivity of paths. And the graph also is unweighted. So, if in the original graph $G = (V, E)$, we can traverse from $u$ tp $v$, in the $G' = (V, E')$ we can do the same. It also considers the path that are multiple of 3 since for every edge that BFS is printed in the output, we replace edge from in the $G' = (V, E')$ with 3 edges of the $G = (V, E)$ stored in the linked list. Please note that in the $G' = (V, E')$, only edges of length 3 from $i$ to $j$ in $G$ is kept.

(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.]

**Answer:**

The basic idea is instead of just remembering if a vertex is visited, remember the length of the path used to visit the vertex, modulo 3. If the length of the path used to visit the vertex, modulo 3 is zero, we are done; otherwise we should walk to the neighbors and do the same procedures.

**Algorithm 6** Lucky Path BFS

```
procedure LUCKY PATH BFS
    let Q be queue;
    luckyIndex = 0;
    Q. Enqueue(u);
    mark u as visited;
    while Q is not empty do
        luckyIndex ++;
        v=Q. Dequeue();
        Examine the node v;
        if (the destination is reached and luckyIndex mod 3 == 0) then
            Done and return;
        else
            For (all neighbors w of v in Graph G)
            if w is not visited then
                Q.enqueue( w );
                mark w as visited;
    Return "not found" ▷ queue is empty, every node on the graph has been examined – quit the search and return "not found"
```

We can use DFS and make use of stack instead of queue.

We are tweaking BFS by adding luckyIndex, increment it and check condition. So, the complexity of the algorithm would be still the complexity of BFS $O(m+n)$.

The algorithm is correct. Given the hint, first of allm by using the BFS we can reach from source to destination if it is feasible. Plus, BFS remember the length of the path used to visit the vertex, modulo 3. When the length path to destination is multiple of 3, result of modulo 3 is zero which is considered in the proposed algorithm. My reference: https://stackoverflow.com/questions/17286686/find-a-path-whose-length-can-be-divided-by-3

and https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

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 $A^k(i, j) > 0$.

**Answer:**

**Theorem:** Suppose that $A$ is the adjacency matrix. We prove that elements in $A^k$ show the number of paths with length $k$ between each $i, j$ in the graph. If the element is zero, it means that there is no path with length K between $i, j$. Accordingly, if we want to find the smallest distances between the $v_i, v_j$ , we should find the smallest K such that $A^k_{i,j} > 0$ or $A^k(i, j) > 0$ . Hence, for any two vertices $(i, j)$, the distance with length k in the graph between $i, j$ is the smallest $k \geq 0$ such that $A^k(i, j) > 0$.

I prove it using induction:

Proof by induction: Let $H^{(n)}_{ij}$ be the path with length $n$ between vertex $v_i$ and $v_j$. Let B(n) be the predicate that the above theorem is true for $n$. B(n) is then the predicate that $H^{(n)}_{ij} = A^n_{i,j}$.

Base step: B(1)

$H^{(1)}_{ij} = A^1_{i,j}$. $H^{(1)}_{ij}$ either is 0 or 1. Let's see what is the value of $A^1_{i,j}$ Based on the definition of the adjacency matrix:

If $v_i, v_j \in E$, so there is a path with length 1 between $v_i, v_j$ and hence $A^1_{i,j} = 1$

If $v_i, v_j \notin E$, so there is no path between $v_i, v_j$ and hence $A^1_{i,j} = 0$.

Any how, the $H^{(1)}_{ij} = A^1_{i,j}$ is true.

Inductive step: We assume B(n) is true and prove that B(n+1) is true.

A path of length n+1 from $v_i$ to $v_j$ is considered as path with length $n$ from $v_i$ to $v_k$ and a path with length 1 from $v_k$ to $v_j$.

So, the number of paths with (n+1)-length from $v_i$ to $v_j$ is the sum over all paths from $v_i$ to $v_k$ times the number of path with length 1 from $v_k$ to $v_j$.

Since we know $H^{(n)}_{ij} = A^n_{i,j}$

$H^{(n+1)}_{ij} = \sum_{l=1}^{|v|} H^{(n)}_{ik} A_{k,j} = \sum_{l=1}^{|v|} A^n_{ik} A_{k,j} A^{n+1}_{ij}$

The above is the formula that is used for matrix multiplication which essentially says that the $H^{(n+1)}_{ij} = A^{n+1}_{i,j}$

We can also prove that the path with *maximum* length k would be the $\sum_{t=1}^{k} A^t$

Ref:https://math.stackexchange.com/questions/2434064/proof-raising-adjacency-matrix-to-n-th-power-gives-n-length-walks-between

(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.

**Answer**

We know that $A^2 = A \times A$ and $A^3 = A^2 \times A$. Hence, $A^k = A \times A..... \times A$ (k As). Since the complexity of each multiplication is $O(n^{2.5})$ and it happens K-1 times, the complexity would be $k \times O(n^{2.5})$ which is equal to $O(kn^{2.5})$

(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.]

**Answer**

We can define random variable $X_i$ . $X_i$ =1 if the vertex is in the shortest path or it equals 0 otherwise. So the expectation value of the $X_i$ would be:

$$E[x_i] = 0 \times Pr[X_i = 0] + 1 \times Pr[X_i = 1] = Pr[X_i = 1] = \frac{k}{n}$$

.

We know that $d(i,j) \geq k$. Hence the probability of getting one vertex in the shortest path would be greater than or equal $\frac{k}{n}$. Accordingly, the probability of not getting a vertex in the shortest path would be less than or equal $(1-\frac{k}{n})$. Since, we are sampling $(2\ln n)\cdot\frac{n}{k}$, then we sample $(2\ln n)\cdot\frac{n}{k}$ vertices of the graph uniformly at random, we would have:

the probability of not getting a vertex in the shortest path $\leq (1-\frac{k}{n})^{(2\ln n)\cdot\frac{n}{k}}$. Using the formula, $1-x \leq e^{-x}$ and considering $x=\frac{k}{n}$ we would have $(1-\frac{k}{n})^{(2\ln n)\cdot\frac{n}{k}} \leq e^{(-1)(2\ln n)\cdot\frac{n}{k}\cdot\frac{k}{n}}$ = $e^{\ln n^{-2}} = \frac{1}{n^2}$

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 $O(n^3)$.

**Answer** I think this leads to Johnson's algorithm.

refhttps://www.geeksforgeeks.org/johnsons-algorithm/