All Pairs Shortest Path Algorithm

1 Difficult Problem 2

1.1 Brief Description

We assume that the Graph given to us is in the form of an adjacency matrix. However, if we are given an adjacency list, we can always convert it into the adjacency matrix representation. adj[i][j] represents an entry of the adjacency matrix where adj[i][j] = $w(i, j)$ if there is an edge from i to j and 0 otherwise.
We further assume that the nodes are labeled from 1 to n and each node is assigned a unique label.
We have an array D[i][j] which denotes the shortest distance between two nodes i and j. In the beginning, if $i = j$ we assign $D[i][j] = 0$. Else if we do not have an edge between the nodes i and j, we assign it to ∞. Else, we assign it to be the weight of the edge between them.
After doing the above initialization, on each iteration from vertex 1 to n, we either choose that vertex to act or not act as an intermediate vertex and accordingly change the value of D[i][j] of all vertex pairs (i, j) such that k either becomes or does not become an intermediate vertex in the path from i to j.

In the 2nd algorithm, we have a $Par[i][j]$ matrix. This matrix can be understood as follows -

- Consider a shortest path in the given graph from i to j.
- Par[i][j] denotes the vertex visited just before the vertex j was visited when traversing the shortest path from i to j.
- If we have $i = j$, then the we do not have any intermediate vertices and we trivially assign $Par[i][j] = i$
- If the shortest path from i to j is the edge between i and j, then also we assign $Par[i][j] = i$ since vertex i was visited before the vertex j.

1.2 PseudoCode

Algorithm 1 Algorithm to compute All Pairs Shortest Distances

```
1:  procedure $APSD(Adj[n+1][n+1])$        ▷ Adj is the adjacency matrix representation of the graph
2:      for $i$ from 1 to $n$ do
3:          for $j$ from 1 to $n$ do
4:              if $i == j$ then
5:                  D[i][j] ← 0
6:              else if adj[i][j] then
7:                  D[i][j] ← adj[i][j];
8:              else
9:                  D[i][j] ← $\infty$
10:     for $k$ in 1 to $n$ do
11:         for $i$ in 1 to $n$ do
12:             for $j$ in 1 to $n$ do
13:                 $D[i][j] \leftarrow min(D[i][j], D[i][k] + D[k][j])$
```

Algorithm 2 Algorithm to compute All Pairs Shortest Paths

```
procedure $APSP(Adj[n+1][n+1], Par[n+1][n+1])$ ▷ Adj is the adjacency matrix representation of the graph
    for $i$ from 1 to $n$ do
        for $j$ from 1 to $n$ do
            if $i == j$ then
                D[i][j] ← 0
                $Par[i][j] \leftarrow i$
            else if adj[i][j] then
                D[i][j] ← adj[i][j]
                $Par[i][j] \leftarrow i$
            else
                D[i][j] ← ∞
                $Par[i][j] \leftarrow 0$
    for $k$ in 1 to $n$ do
        for $i$ in 1 to $n$ do
            for $j$ in 1 to $n$ do
                $dist \leftarrow D[i][k] + D[k][j]$
                if $D[i][j] > dist$ then
                    $D[i][j] \leftarrow dist$
                    $Par[i][j] \leftarrow Par[k][j]$
```

Algorithm 3 Algorithm to compute a Shortest Path from i to j

```
procedure $ShortestPath(i, j)$        ▷ $i$ is the starting vertex and $j$ is the destination vertex
    if $D[i][j] = \infty$ then
        return $\phi$
    Stack S
    $S.push(j)$
    while $i \neq Par[i][j]$ do
        $S.push(j)$
        $j \leftarrow Par[i][j]$
    if $i \neq j$ then
        $S.push(i)$
    Return S        ▷ $S$ contains all vertices in a shortest path from $i$ to $j$ in the same order.
```

1.3 Proof of correctness

First, we introduce some notations:
Let $G = (V, E)$ be the given graph having no negative weight cycles.
$P^k(i, j)$ = set of all paths from i to j whose intermediate vertices have label at most k.
$p^k(i, j)$ = shortest path from the set $P^k(i, j)$.
$\delta^k(i, j)$ = length of the shortest path $p^k(i, j)$.
$p(i, j)$ = A shortest path between i and j
$\delta(i, j)$ = length of the shortest path $p(i, j)$.

Claim: Shortest path between i and j with intermediate vertices having label atmost k must also follow the optimal subpath property i.e. $p^k(i, j)$ must follow the optimal subpath property

Proof: The optimal Subpath property was proved in the lectures while studying Bellman Ford algorithm. There, our graph had a particular start and destination point. Since the start and destination points were **arbitrary**, the optimal subpath property must hold for any pair of chosen start and destination points. Therefore, the shortest paths $p(i, j)$ must follow the optimal subpath property.
Now, we need to prove that the shortest paths such that the label of the intermediate vertices being atmost k also follow the optimal subpath property.
This can be seen very easily. Simply create another graph $H = (V, E')$. Let $J \subseteq E$ be the set of all edges which start from a vertex v and v has a label $> k$, $\forall v \in V$. Then $E' = E \backslash J$. This ensures that any path from two arbitrary vertices i and j do not have any intermediate vertex with label $> k$.
The only condition for optimal subpath property to hold is that there must not be any negative weight cycles present in the graph. It can be seen easily that **removing** edges from a graph can never create **new** negative weight cycles. Hence optimal subpath property must hold on this graph. Now the shortest path between i and j in the graph H is $p^k(i, j)$. Therefore, the shortest path between i and j with intermediate vertices having label atmost k must also follow the optimal subpath property i.e. the paths $p^k(i, j)$ follow the optimal subpath property.
Hence Proved.

Claim: Algorithm 1 correctly computes the shortest distance between all pairs of vertices.

Proof: We shall prove the above claim by induction on the maximum label of the intermediate vertices in a shortest path between two arbitrarily chosen vertices i and j.

We have the following implications -

- $\delta^0(i, j)$ = length of a shortest path between i and j having the highest label of intermediate vertex ≤ 0 $\implies$ we are not allowed to have any intermediate vertices. If $i = j$, then we do not have any intermediate vertices and hence $\delta(i, j) = 0$. If we have an edge between i and j, then trivially $\delta(i, j) = w(i, j)$. Else, we cannot have any possible path between i and j and hence $\delta(i, j) = \infty$

- $\delta^n(i, j)$ = length of shortest path between i and j having the highest label of intermediate vertices $\leq n$. Since the highest label in the given graph is n itself, then $\delta^n(i, j)$ represents a global shortest path from the vertex i to j i.e $\delta^n(i, j) = \delta(i, j)$

Recurrence relation:
For $k = 0$(Base case),

$$\delta^0(i, j) = 0 \qquad \text{if i = j.}$$
$$\delta^0(i, j) = w(i, j) \qquad \text{if (i,j) is an edge.}$$
$$\delta^0(i, j) = \infty \qquad \text{otherwise.}$$

Else

$$\delta^k(i,j) = min(\delta^{k-1}(i,j), \delta^{k-1}(i,k) + \delta^{k-1}(k,j))$$

Base case: After the initialization and before the start of any iteration of outer loop of k, D[i][j] = $\delta^0(i,j)$, since D[i][j] stores the length of the shortest path between nodes i and j, having 0 as the maximum label of intermediate vertices. Hence, base case is true.

Inductive Step:

To clarify the meaning of $\delta^k(i,j)$ in the proof,

$$\delta^k(i,j) = \text{value in D[i][j] after the } k^{th} \text{ iteration of the outermost loop.}$$

Let's assume that $\delta^{k-1}(i,j)$ correctly calculates the shortest distance between all pairs of vertices i and j having intermediate vertices with label at most $k-1$. Need to show, $\delta^k(i,j)$ also correctly calculates the shortest distance between all pairs of vertices i and j having intermediate vertices with label at most k. Consider the following three cases:

(i) There is no path from i to j with k as an intermediate vertex and such that label of all the remaining intermediate vertices are $< k$. In this case, we have $\delta^k(i,j) = \delta^{k-1}(i,j)$ trivially.

(ii) If there **exists** a path from i to j with k as an intermediate vertex and such that all the remaining intermediate vertices $< k$ **but** the shortest path does not actually pass through k. In this case too, $\delta^k(i,j) = \delta^{k-1}(i,j)$ holds.

(iii) If the shortest path from i to j actually passes through k, then we have

$$\delta^k(i,j) = \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$$

This is because the shortest path from i to j must pass through k. Hence the shortest path from i to k must be optimal. Similarly, the path between k to j must be optimal too. This follows from the optimal subpath property i.e. our first claim. The optimal shortest path distance from i to k such that the label of intermediate vertices from i to k is at most $k-1$ is denoted by $\delta^{k-1}(i,k)$. Similarly, the shortest path distance from k to j such that the label of the intermediate vertices is atmost $k-1$ is denoted by $\delta^{k-1}(k,j)$. Hence the shortest path distance from i to j via k is simply the sum of the above two expressions.

Note that the cases (i) and (ii) yield the same recursion. Hence we essentially only have two candidates for our shortest path between two vertices i and j such that the label of intermediate vertices is atmost k i.e.

$$\delta^k(i,j) = \delta^{k-1}(i,j) \text{ or } \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$$

Since we do not know which of the above two holds for some arbitrary i, j and k, we simply take the minimum i.e.

$$\delta^k(i,j) = min(\delta^{k-1}(i,j), \delta^{k-1}(i,k) + \delta^{k-1}(k,j))$$

Therefore, $\delta^k(i,j)$ correctly computes the shortest distance between i and j having the label of intermediate vertices at most k Hence, our inductive step is also true and all we are required to do is output the value of $\delta^n(i,j)$ which denotes the shortest path distance between i and j i.e. $\delta(i,j)$.

Hence Proved.

We do not prove the correctness of algorithm 2 as it is simply a copy of algorithm 1 with just one additional parameter i.e. $par[i][j]$. If the shortest path was not changed from $(k-1)^{th}$ to the k^{th} iteration i.e. $\delta^k(i,j) = \delta^{k-1}(i,j)$, then our par[i][j] also does not change. On the other hand, if the shortest path changes i.e. $\delta^k(i,j) = \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$, then our value of par[i][j] also changes. Since now we take a path from i to k and then k to j, the parent of j going from i to j becomes equal to the parent of j when we go from k to j i.e. $par[i][j] = par[k][j]$

In algorithm 3, we simply traverse the path from i to j in the reverse order. We start from j and recursively visit the parents until we reach the vertex i. Since we visit in the reverse direction, we maintain a stack whose top element will be the vertex i itself. Now, if we keep on popping the elements, we can print the whole shortest path from i to j inclusive.

1.4 Time and Memory Complexity

For both algorithms 1 and 2, Pre-processing takes only $O(n^2)$ time. The algorithm itself consists only of three nested **For** loops with each variable going from 1 all the way to n. Hence the time complexity of the algorithm is $O(n^3)$
In algorithm 3, we only visit the shortest path from i to j in the reverse order. Hence the number of iterations performed by the **While** loop is equal to the length of the shortest path from i to j. This is optimal and in the worst case i.e. a shortest path consisting of n vertices, it takes $O(n)$ time.
Algorithm 1 only stores an array D[i][j] which is an nxn matrix. Hence the space taken is $O(n^2)$.
Algorithm 2 stores an additional matrix $par[i][j]$ which is also an nxn matrix and our required data structure for reporting the shortest path from any pair of vertices i and j. This also takes a space of $O(n^2)$.
In Algorithm 3, we return a stack S whose size is the order of the number of vertices in the shortest path from i to j. Hence it is optimal, and in the worst case can take a space of $O(n)$.