

1. The goal of this problem is to prove that Dijkstra's algorithm correctly solves the Single Source Shortest Paths problem. Let x be the source node.
- 1.1. Call a path from x to i active if all nodes (except possibly the last) have been visited, and the nodes are ordered in increasing order of visitation.

Prove that for every $k \geq 0$, all the following are true after Dijkstra's algorithm visits k node:

- 1.1.1. The algorithm will have computed the shortest paths from x to each of the k visited nodes.
- 1.1.2. Every shortest path computed so far is active.
- 1.1.3. Every entry (d, i) in PQ represents some active path from x to i with distance d .
- 1.1.4. For every active path that is the shortest path to an unvisited node, there is a corresponding entry in P .

We prove by induction on nodes visited, k .

[Base Case: $k = 0$]

At $k = 0$, no nodes have been visited yet. PQ only contains $(0, x)$.

So,

- 1.1.1. is vacuously true since there are no visited nodes yet.
- 1.1.2. is vacuously true since there are no computed paths yet.
- 1.1.3. The empty path $[x]$ is active, and there are no “except last” nodes to check, represented by $(0, x)$,
- 1.1.4. is true since the only active path is the empty path to x , which has entry in PQ . Therefore, the base case holds.

[Inductive Step] We assume that it inductive hypothesis holds true for $k - 1$, and prove it for k .

Let (d, i) be the k th popped pair, node i to be visited.

[1.1.1.] We show that d is the shortest path to i .

Any shorter path P to i must have first unvisited node before reaching i . The first unvisited node would have a smaller distance in PQ by induction. But (d, i) was the smallest in PQ , which is a contradiction. Therefore, d is the shortest distance to i .

[1.1.2] Every computed shortest path comes from PQ entries, which represent active paths by induction. Therefore, all computed paths so far are active.

[1.1.3] Consider new entries $(d + c, j)$ for each neighbor j of i .

Their paths go from: $x \rightarrow \dots \rightarrow i \rightarrow j$. The subpath to i is active, i was just visited, and j is unvisited, so the order is preserved. Therefore, the total cost is $d + c$.

[1.1.4] Next, we show that every active shortest path to unvisited node has a PQ entry.

We consider the active shortest path p to an unvisited j . Let k be node before j . We consider the following cases regarding if $k = i$:

[Case 1: $k = i$] Then, the path is $x \rightarrow \dots \rightarrow i \rightarrow j$, with the entry $(d + c_{ij}, j)$ existing.

[Case 2: $k \neq i$] Then the path existed before visiting i , and by the inductive hypothesis, had the PQ entry before, and is still in PQ since j is unvisited.

Therefore, after all nodes are visited:

- All reachable nodes have been visited (otherwise shortest path to unvisited node would have a PQ entry, contradicting empty PQ)
- Unreachable nodes correctly have distance set to infinity.
- Dijkstra's algorithm correctly solves SSSP.

1.2. Using the previous result, prove that Dijkstra's algorithm is correct.

Using the results from 1.1, we prove Dijkstra's algorithm correctly solves the Single Source Shortest Paths problem. We prove the following claims:

“Claim 1: After the algorithm terminates with the PQ empty, all nodes reachable from source x have been visited.”

Suppose, for contradiction, that there exists a reachable but unvisited node y . Let P be the shortest path from x to y . Let z be the first unvisited node along P , which should exist because y exists.

Then the subpath $x \rightarrow \dots \rightarrow z$ is the shortest path to z , by the lemma “subpaths of shortest paths are shortest paths”. All nodes before z on this subpath are visited. By 1.1.4, there must be an entry (d, z) in PQ representing this active shortest path. However, PQ is empty, so we have a contradiction.

Therefore, all reachable nodes have been visited

“Claim 2: All unreachable nodes correctly have distance ∞ ”.

Note that only nodes with entries in PQ can be visited. Every PQ entry corresponds to some path from x. Therefore, the only reachable nodes can be visited. The unvisited nodes at termination must be unreachable. Since their distances remain as ∞ , this is correct.

“Claim 3: For each visited node i, $\text{dist}[i]$ is the shortest path distance.”

By 1.1.1., when the node i is visited, its distance is the shortest path from x to i. The distances never change after visitation since the algorithm assigns only once.

Therefore, with all three claims proven, Dijkstra’s algorithm correctly computes the shortest path distances to all reachable nodes, and marks unreachable nodes with distance ∞ .

2.Modify Dijkstra’s algorithm so that the shortest path from the source node x to any node i can be reconstructed.

- The modified algorithm should have the same time complexity ($O(e\log n)$, $O(e\log n)$ or $O(n \log n + e)$ depending on the variant and space complexity ($O(n+e)$)). Also, reconstructing the path from x to i should run in $O(l(i))$ time, where $l(i)$ is the number of edges in the shortest path from x to i.

...

```
def sssp_dijjkstra(g, x):
    adj = adjacency list
    mark all nodes as not visited
    pq = empty min-heap of triples (dist, node, predecessor)
    dist = array of length n, initially  $\infty$ 
    pred = array of length n, initially -1

    push(0, x, x) to pq

    while pq is not empty:
        (d,i,p) = pop pq
        if i visited: continue
        mark i as visited
        dist[i] = d
```

```

pred[i] = p
for (j, c) in adj[i]:
    push (d + c, j, i) to pq

return (dist, pred)

```

Here, $\text{pred}[i]$ stores the node immediately before i on some shortest path from x to i . For $i = x$, $\text{pred}[x] = x$. Then the path reconstruction can be done by running this function:

...

```

def construct_path(i, x, pred):
    path = [i]
    while path[-1] != x:
        path.append(pred[path[-1]])
    reverse path
    return path

```

We analyze the time complexity of the modified algorithm. Depending on the heap implementation, it remains to be $O(e\log e)$, $O(e\log n)$, or $O(n\log n + e)$. Then, adding the predecessor tracking adds $O(1)$ work per node, no asymptotic change.

Space complexity remains $O(n + e) = O(n + e) + O(n) = O(n + e)$.

The path reconstruction time runs $l(i)$ times, once per edge in path. Each append is $O(1)$ amortized with a dynamic array, reverse is $O(l(i))$ so the total is $O(l(i))$.

By 1.1.1, when node i is visited, $\text{dist}[i]$ is the shortest path distance. The triple (d, i, p) that visited i came from some neighbor p where $\text{dist}[p] + c = \text{dist}[i]$. Therefore, following pred pointers back to x yields a valid shortest path.

3. In this task, we will prove that the Floyd-Warshall algorithm is correct.

3.1. Prove that the three-parameter DP solution `apsp_dp` correctly solves the All-Pair Shortest Paths problem.

...

```

def apsp_dp(G):
    let s be an  $(n+1) \times n \times n$  matrix

    # Initialize
    for i, j:  $s[0][i][j] = \infty$ 
     $s[0][i][i] = 0$ 

```

```

for each edge (i, j, c):  $s[0][i][j] = \min(s[0][i][j], c)$ 

# Main loop
for k from 0 to n-1:
    for i from 0 to n-1:
        for j from 0 to n-1:
             $s[k+1][i][j] = \min(s[k][i][j], s[k][i][k] + s[k][k][j])$ 

return  $s[n]$ 
...

```

We prove this by induction on k, and hold that $s[k][i][j] = \text{length of the shortest path from } i \text{ to } j \text{ using only intermediate nodes from } \{0, 1, \dots, k-1\}$.

[Base Case: k = 0/Initialization]

No intermediate nodes allowed since $k - 1 = -1$. If $i = j$, then the empty path has length 0.
If $i \neq j$, then the path must be a single edge.

If the edge exists, $s[0][i][j]$ is the minimum cost among all such edges.

If no edge exists, then $s[0][i][j] = \text{inf}$

So the base case holds.

[Inductive Step/Maintenance] We assume the invariant holds for k and prove it for $k + 1$.

Consider the shortest path from i to j using intermediates $\{0, \dots, k\}$.

[Case 1: Path does not use node k as intermediate]

Then it only uses intermediates $\{0, \dots, k-1\}$. By induction, $\text{length} = s[k][i][j]$.

[Case 2: Path uses node k as an intermediate]

Then the path = $i \rightarrow \dots \rightarrow k \rightarrow \dots \rightarrow j$. The subpaths are shortest paths, assuming no negative cycles exist. so

$i \rightarrow k$ uses intermediates $\{0, \dots, k-1\}$ leading to $\text{length} = s[k][i][k]$.

$k \rightarrow j$ uses intermediates $\{0, \dots, k-1\}$ leading to $\text{length} = s[k][k][j]$

with the total length = $s[k][i][k] + s[k][k][j]$

So in both cases:

$$s[k+1][i][j] = \min(s[k][i][j], s[k][i][k] + s[k][k][j])$$

covers all possibilities and gives the shortest path using intermediates $\{0, \dots, k\}$.

[Termination: $k = n$]

Then $s[n][i][j]$ is the shortest path using intermediates $\{0, \dots, n-1\}$ with all possible nodes. Therefore, $s[n][i][j]$ is the true shortest path distance from i to j .

3.2. Prove that, during the main loop, the value of any entry of the d matrix in apsp_floyd_marshall never increases.

Claim: During the main loop of Floyd-Warshall, for any entry $d[i][j]$, its value never increases.

The only update that modifies any entry is:

$$d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

Let $d_{\text{old}}[i][j]$ be the value before this update, and $d_{\text{new}}[i][j]$ be the value after.

[Case 1: $d_{\text{old}}[i][j] \leq d[i][k] + d[k][j]$] Then the minimum is $d_{\text{old}}[i][j]$, and it remains unchanged.

[Case 2: $d_{\text{old}}[i][j] > d[i][k] + d[k][j]$] Then the minimum is $d[i][k] + d[k][j]$. So $d_{\text{new}}[i][j] = d[i][k] + d[k][j]$, which decreases.

In both cases, $d_{\text{new}}[i][j] \leq d_{\text{old}}[i][j]$. Therefore, values never increase, they either stay the same or decrease.

3.3. Let $d^{(k)}$ be the state of the d matrix in apsp_floyd_marshall after k iterations of the main loop. In particular, $d^{(0)}$ is the initial state before the loop, and $d^{(n)}$ is the final state after all n iterations. Let s be the s matrix in apsp_dp. **Prove that for all k, i, j such that $0 \leq k \leq n$, and $0 \leq i, j \leq n$,**

$$d^{(k)}[i][j] \leq s[k][i][j].$$

We prove this by induction on k .

[Base Case: $k = 0$]

$d^{(0)}$ and $s[0]$ are initialized identically. Note that the diagonal is 0, the edges are at its minimum edge cost, and for no edge, it is inf.

Therefore, $d^{(0)}[i][j] = s[0][i][j]$ for all i, j . Hence, $d^{(0)}[i][j] \leq s[0][i][j]$.

[Inductive Step] Assume that for all k , $d^{(k)}[i'][j'] \leq s[k][i'][j']$ for all i', j' .

By 3.2, values never increase during loop k , so at any point during iteration k ,

$$d[i'][j'] \leq s[k][i'][j']$$

At the end of iteration k , (the beginning of iteration $k+1$):

$$d^{(k+1)}[i][j] = \min(d[i][j], d[i][k] + d[k][j]).$$

Applying the inequality from IH:

$$d^{(k+1)}[i][j] \leq \min(s[k][i][j], s[k][i][k] + s[k][k][j])$$

But the RHS is exactly $s[k+1][i][j]$ by definition of s . Therefore,

$$d^{(k+1)}[i][j] \leq s[k+1][i][j]$$

By induction, $d^{(k)}[i][j] \leq s[k][i][j]$ holds for all k, i, j .

3.4. Prove that for all k, i, and j such that $0 \leq k \leq n$, and $0 \leq i, j \leq n$:

$$d^{(k)}[i][j] \geq s[n][i][j]$$

We prove the claim above by proving the lemma below:

Lemma: At any point after initialization, $d[i][j]$ contains the cost of some path from i to j (or inf).

[Base Case: Initialization]:

- If $i = j$: $d[i][j] = 0$, empty path
- If $i \neq j$ and an edge exists: $d[i][j] = \text{cost of that edge}$
- If no edge: $d[i][j] = \text{inf}$

[Inductive Case: During updates]

- Before update: $d[i][j]$, $d[i][k]$, $d[k][j]$ each represent some path.
- New value = $\min(d[i][j], d[i][k] + d[k][j])$
- Minimum of two path costs is also a valid path cost.

Therefore, at all times, $d[i][j]$ is the cost of some path from i to j (or inf).

By this Lemma, $d^{(k)}[i][j]$ is the cost of some path from i to j. Let this path be P with cost $c = d^{(k)}[i][j]$. By definition, $s[n][i][j]$ is the shortest possible path cost from i to j. Therefore, $c \geq s[n][i][j]$, $d^{(k)}[i][j] \geq s[n][i][j]$.

3.5. Using the previous results, prove/conclude that the Floyd–Warshall algorithm (apsp_floyd_marshall) is correct.

From 3.3 and 3.4, for all k, i, j :

$$s[n][i][j] \leq d^{(k)}[i][j] \leq s[k][i][j]$$

Setting $k = n$:

$$s[n][i][j] \leq d^{(n)}[i][j] \leq s[n][i][j]$$

Therefore,

$$d^{(n)}[i][j] = s[n][i][j]$$

From 3.1, $s[n][i][j]$ is the length of the shortest path from i to j. Hence, $d^{(n)}[i][j]$, the final state, equals the true shortest path distance for all pairs i, j. Thus, Floyd-Warshall correctly solves the All-Pairs Shortest Paths problem.

4. Consider the following algorithm:

```
def sdsp_dp(digraph G = (V, E), node x):
    let adj be the adjacency list of G

    Let _d be a (n + 1) × n matrix
    Let _v be a (n + 1) × n matrix, all initially False
    def d(t, i):
        if i == x:
            return 0

        if not _v[t][i]:
            _v[t][i] = True
            _d[t][i] = INF
            if t > 0:
                for j, c in adj[i]:
                    _d[t][i] = min(_d[t][i], c + d(t - 1, j))

    return _d[n][i]

return [d(n, i) for i from 0 to n-1]
```

4.1. Prove that it correctly solves the Single Destination Shortest Paths problem.

We prove by induction on t that $d(t,i)$ returns the length of the shortest path from i to destination x using at most t edges.

[Base Cases]

$i = x$: path of length 0 exists, no edges. any other path has positive weight, with positive edges. thus, we return 0.

$i \neq x$ and $t = 0$: no path possible without edges when $i \neq x$. we return inf.

[Inductive Step] We assume the invariant holds for $t - 1$ (all nodes). We prove for $t > 0$ and $i \neq x$.

Any path from i to x must start with some edge (i, j) with cost c , where $(j,c) \in \text{adj}[i]$. The remainder is a path from j to x using at most $t-1$ edges since we already used one edge.

By induction: the shortest such remainder has length $d(t-1, j)$. Therefore, for this specific first edge, the total path length = $c + d(t-1, j)$. The shortest path overall must be the minimum over all possible first edges:

$d(t,i) = \min_{(j,c) \in \text{adj}[i]} (c + d(t-1,j))$, which is exactly what the algorithm computes.

Therefore, $d(n,i)$ = shortest path from i to x using at most n edges. Since the shortest path in a graph with n nodes uses at most $n - 1$ edges (no cycles in shortest paths with positive weights), this equals the true shortest path distance. Thus, the algorithm correctly solves the Single Destination Shortest Paths problem.

4.2.What is the time complexity of the algorithm? (Briefly explain why.)

The time complexity of the algorithm is $O(n^2 + ne)$.

For each pair (t,i) where $0 \leq t \leq n$ and $0 \leq i < n$:

- work performed = $O(1 + |\text{adj}[i]|)$
 - constant work for initialization and return
 - loop over all outgoing edges of i (degree of i)

Summing over all t and i :

$$\sum_{t=0}^n \sum_{i=0}^{n-1} (1 + |\text{adj}[i]|) = (n+1) \sum_{i=0}^{n-1} |\text{adj}[i]|$$

Since $\sum_{i=0}^{n-1} |\text{adj}[i]| = e$:

$$= (n+1)(n+e) = O(n^2 + ne)$$

Considering also the matrix initialization and result array:

$$O(n^2) + O(n) + O(n^2 + ne) = O(n^2 + ne).$$

4.3.What is the space complexity of the algorithm? (Briefly explain why.)

The algorithm only stores the adjacency list, which uses $\Theta(n + e)$ memory, as well as the matrices $_v$ and $_d$, which uses $\Theta(2n^2) = \Theta(n^2)$ memory. Therefore, the space complexity is $\Theta(n + e + n^2) = \Theta(n^2 + e)$.

4.4 Improve the space complexity of the algorithm to $O(n + e)$.

Observe that $d(t,i)$ only depends on $d(t-1, *)$: the previous layer of the DP table. Therefore, we only need to keep two rows at any time, not all $n + 1$ rows. We do bottom-up DP with two arrays:

...

```
def sdsp_dp(G, x):
    adj = adjacency list of G
    n = number of nodes

    d = [inf] * n
    d_new = [inf] * n
```

```

for t from 0 to n:
    for i from 0 to n-1:
        if i == x:
            d_new[i] = 0
        else:
            d_new[i] = inf
            if t > 0:
                for (j,c) in adj[i]:
                    d_new[i] = min(d_new[i], c + d[j])
    swap(d, d_new)

return d

```

Adjacency list + two arrays of length $n = O(n + e) + O(2n) = O(n + e)$, maintaining the same functionality while reducing the space from $O(n^2 + e)$ down to $O(n + e)$.

5. Consider the following problem:

Problem 1. There are n neighborhoods and e two-way highways in a city. Each highway connects two different neighborhoods, and there is at most one highway between every pair of neighborhoods. Each highway has a positive length. Some of the neighborhoods have an XM city mall, and some don't. You are given which neighborhoods have an XM city mall, and which don't.

For each neighborhood, find the distance to the nearest XM city mall, assuming you can only traverse the city using the highways. (If there is no reachable XM city mall from a neighborhood, we set the answer to be ∞ .)

5.1. Find an $O(n^3)$ solution to Problem 1.

We model the problem as a weighted directed graph. Each undirected highway (i,j) becomes two directed edges $(i,j), (j,i)$. Let the edge weight be the length of the highway.

Next, we compute all-pairs shortest paths via Floyd-Warshall's algorithm (which is $O(n^3)$) to compute $d[i][j] = \text{shortest path distance from } i \text{ to } j$.

Then, we compute the nearest mall distance for each node i : $\text{ans}[i] = \min_{(j \in M)} d[i][j]$. For each i , check all $|M|$ malls, $O(n)$ per node, totalling $O(n^2)$ work.

So the time complexity is $O(n^3 + n^2) = O(n^3)$. This is correct because $d[i][j]$ is the true shortest path distance, taking minimum over all malls gives distance to the nearest mall.

5.2. Find an $O(n^2 + neloge)$ solution to Problem 1.

We use the same answer as in 5.1, but use Dijkstra's algorithm for each possible source node instead of Floyd-Warshall. The complexity of Dijkstra's algorithm is $O(n + eloge)$, so doing it n times gives the time complexity: $O(n(n + eloge) + n^2) = O(n^2 + neloge + n^2) = O(n^2 + neloge)$.

5.3. Find an $O((n + e) \log (n + e))$ solution to Problem 1.

From 5.1, we first note that $d[i][j] = d[j][i]$ because every path can be reversed, with the same total cost. Thus, we can also write the distance to the nearest XM city mall to i as $\min_{j \in M} d[j][i]$.

Next, we modify the graph by adding a new node x that will act as the “mega-source”. For each node $j \in M$ corresponding to a node with an XM city mall, we add a directed edge $x \rightarrow j$ with cost 0.

We then perform Dijkstra's algorithm on this modified graph with source node x , giving us an array D where $D[i]$ represents the cost of the shortest path from x to i . Using the fundamental shortest path equation, any path from x to $i \neq x$ must start with an edge from x . The edges coming from x are precisely (x, j) for every $j \in M$, with each costing 0. The subsequent path must be a path from j to i in the original graph since there's no way to return to x . Because subpaths of shortest paths are shortest paths, the length of this part must be $d[j][i]$. Therefore,

$$D[i] = \min_{j \in M} (0 + d[j][i]),$$

which means D contains the answers that we're looking for.

The time complexity is dominated by Dijkstra's algorithm on the modified graph, which has $n+1$ nodes and $e + |M| \leq e + n$ edges, so the overall running time is

$$O((n+1) + (e+n)\log(e+n)) = O((n+e)\log(n+e)).$$