Algorithms[1]

1 Notations

- $G = (V, E)$
  - V : Set of vertices/nodes
  - E : Set of edges
- $G.Adj[u]$: Adjacency list of a vertex u of the graph G

2 Graph Representation

- Given an adjacency-list representation of a graph $G = (V, E)$, write an algorithm to obtain the adjacency-matrix representation of G.

Algorithm 1 ADJACENCY-LIST-TO-ADJACENCY-MATRIX

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of a graph $G = (V, E)$
Output: `Adj-Matrix`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of the graph G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Create an adjacency matrix `Adj-Matrix`$[1, 2, \ldots, n][1, 2, \ldots, n]$ of size $n \times n$
4: **for each** vertex $u \in G.V$ **do**
5: **for each** vertex $v \in G.V$ **do**
6: `Adj-Matrix`$[u][v] \leftarrow 0$ ▷ Initialize each element of the adjacency-matrix with 0
7: **end for**
8: **end for**
9: **for each** vertex $u \in G.V$ **do**
10: **for each** vertex $v \in$ `Adj-List`$[u]$ **do**
11: `Adj-Matrix`$[u][v] \leftarrow 1$ ▷ Edge from u to v
12: **end for**
13: **end for**
14: **return** `Adj-Matrix`

[1] Please verify the content.

- Given an adjacency-matrix representation of a graph $G = (V, E)$, write an algorithm to obtain the adjacency-list representation of G.

Algorithm 2 ADJACENCY-MATRIX-TO-ADJACENCY-LIST

Input: `Adj-Matrix`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of a graph $G = (V, E)$
Output: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of the graph G

```
1:  n ← |G.V|                                          ▷ Number of vertices in G
2:  m ← |G.E|                                          ▷ Number of edges in G
3:  Create an adjacency list Adj-List[1, 2, ..., n] of size n
4:  for each vertex u ∈ G.V do
5:      Adj-List[u] ← ∅                                ▷ Initialize an empty adjacency-list for vertex u
6:  end for
7:  for each vertex u ∈ G.V do
8:      for each vertex v ∈ G.V do
9:          if Adj-Matrix[u][v] = 1 then               ▷ Check for an edge from u to v
10:             Adj-List[u] ← Adj-List[u] ∪ {v}        ▷ Add v to the adjacency-list of u
11:         end if
12:     end for
13: end for
13: return Adj-List
```

3 Degree of Vertices in the Graph

- Given an adjacency-list representation of a directed graph $G = (V, E)$, write an algorithm to obtain the in-degree and out-degree of all the vertices.

Algorithm 3 IN-OUT-DEGREE-DIRECTED-GRAPH-ADJECANCY-LIST

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of a directed graph $G = (V, E)$
Output: In-degree and out-degree of all the vertices in the graph G

```
1:  n ← |G.V|                                          ▷ Number of vertices in G
2:  m ← |G.E|                                          ▷ Number of edges in G
3:  for each vertex u ∈ G.V do
4:      u.in-degree ← 0                                ▷ Initialize in-degree of vertex u to be 0
5:      u.out-degree ← 0                               ▷ Initialize out-degree of vertex u to be 0
6:  end for
7:  for each vertex u ∈ G.V do
8:      u.out-degree ← |Adj-List[u]|
9:      for each vertex v ∈ Adj-List[u] do
10:         v.in-degree ← v.in-degree + 1
11:     end for
12: end for
13: return in-degree and out-degree of vertices
```

- Given an adjacency-matrix representation of a directed graph $G = (V, E)$, write an algorithm to obtain the in-degree and out-degree of all the vertices.

Algorithm 4 IN-OUT-DEGREE-DIRECTED-GRAPH-ADJECANCY-MATRIX

Input: `Adj-Matrix`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of a directed graph $G = (V, E)$
Output: In-degree and out-degree of all the vertices in the graph G

```
1:  n ← |G.V|                                          ▷ Number of vertices in G
2:  m ← |G.E|                                          ▷ Number of edges in G
3:  for each vertex u ∈ G.V do
4:      u.in-degree ← 0                                ▷ Initialize in-degree of vertex u to be 0
5:      u.out-degree ← 0                               ▷ Initialize out-degree of vertex u to be 0
6:  end for
7:  for each vertex u ∈ G.V do
8:      for each vertex v ∈ G.V do
9:          if Adj-Matrix[u][v] = 1 then
10:             v.in-degree ← v.in-degree + 1
11:             u.out-degree ← u.out-degree + 1
12:         end if
13:     end for
14: end for
15: return in-degree and out-degree of vertices
```

- Given an adjacency-list representation of an undirected graph $G = (V, E)$, write an algorithm to obtain the degree of all the vertices.

Algorithm 5 DEGREE-UNDIRECTED-GRAPH-ADJECANCY-LIST

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of an undirected graph $G = (V, E)$
Output: Degree of all the vertices in the graph G

```
1: n ← |G.V|                                          ▷ Number of vertices in G
2: m ← |G.E|                                          ▷ Number of edges in G
3: for each vertex u ∈ G.V do
4:     u.degree ← 0                                   ▷ Initialize degree of vertex u to be 0
5: end for
6: for each vertex u ∈ G.V do
7:     u.degree ← |Adj-List[u]|
8: end for
9: return degree of vertices
```

- Given an adjacency-matrix representation of an undirected graph $G = (V, E)$, write an algorithm to obtain the degree of all the vertices.

Algorithm 6 DEGREE-UNDIRECTED-GRAPH-ADJECANCY-MATRIX

Input: `Adj-Matrix`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of an undirected graph $G = (V, E)$
Output: Degree of all the vertices in the graph G

```
1: n ← |G.V|                                          ▷ Number of vertices in G
2: m ← |G.E|                                          ▷ Number of edges in G
3: for each u ∈ G.V do
4:     u.degree ← 0                                   ▷ Initialize degree of vertex u to be 0
5: end for
6: for each vertex u ∈ G.V do
7:     for each vertex v ∈ G.V do
8:         if Adj-Matrix[u][v] = 1 then
9:             u.degree ← u.degree + 1
10:        end if
11:    end for
12: end for
13: return degree of vertices
```

4 Transpose of Graph

- Given an adjacency-list representation of a graph $G = (V, E)$, write an algorithm to obtain the adjacency-list representation of the transpose of G, *i.e.*, G^T.

Algorithm 7 TRANSPOSE-ADJECANCY-LIST

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of a graph $G = (V, E)$
Output: `Adj-List-Transpose`$[1, 2, \ldots, |G.V|]$: Adjacency-list representation of the transpose of the graph G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Create an adjacency list `Adj-List-Transpose`$[1, 2, \ldots, n]$ of size n for the transpose graph
4: **for each** vertex $u \in G.V$ **do**
5: `Adj-List-Transpose`$[u] \leftarrow \emptyset$ ▷ Initialize an empty adjacency-list for vertex u for the transpose graph
6: **end for**
7: **for each** vertex $u \in G.V$ **do**
8: **for each** vertex $v \in$ `Adj-List`$[u]$ **do**
9: `Adj-List-Transpose`$[v] \leftarrow$ `Adj-List-Transpose`$[v] \cup \{u\}$ ▷ Add u to the adjacency-list of v for the transpose graph
10: **end for**
11: **end for**
12: **return** Transpose of the graph G in the form of `Adj-List-Transpose`

- Given an adjacency-matrix representation of a graph $G = (V, E)$, write an algorithm to obtain the adjacency-matrix representation of the transpose of G, *i.e.*, G^T.

Algorithm 8 TRANSPOSE-ADJECANCY-MATRIX

Input: `Adj-Matrix`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of a graph $G = (V, E)$
Output: `Adj-Matrix-Transpose`$[1, 2, \ldots, |G.V|][1, 2, \ldots, |G.V|]$: Adjacency-matrix representation of the transpose of the graph G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Create an adjacency matrix `Adj-Matrix-Transpose`$[1, 2, \ldots, n][1, 2, \ldots, n]$ of size $n \times n$ for the transpose graph
4: **for each** vertex $u \in G.V$ **do**
5: **for each** vertex $v \in G.V$ **do**
6: `Adj-Matrix-Transpose`$[u][v] \leftarrow$ `Adj-Matrix`$[v][u]$
7: **end for**
8: **end for**
9: **return** Transpose of the graph G in the form of `Adj-Matrix-Transpose`

5 Breadth-First Search (BFS)

Given a graph $G = (V, E)$ and a source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is **reachable from** s.

- It computes the distance from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v.
- Breadth-first search produces a "breadth-first tree" with root s that contains all reachable vertices.
- For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G.
- The algorithm works on both directed and undirected graphs.

Algorithm 9 BFS-PRINT(s, G)

```
1:  for each vertex v ∈ V do
2:      u.visited ← FALSE                          ▷ Vertex u is not visited initially
3:  end for
4:  Q ← ∅                                          ▷ Create a queue to store the vertices
5:  s.visited ← TRUE                               ▷ Mark source vertex s as visited
6:  Q.ENQUEUE(s)                                   ▷ Insert source vertex s in the queue
7:  while Q ≠ ∅ do                                 ▷ Q is not empty
8:      u ← Q.DEQUEUE()                            ▷ Remove the vertex from the queue
9:      for each vertex v ∈ G.Adj[u] do            ▷ Explore all the adjacent vertices of vertex u
10:         if v.visited = FALSE then              ▷ Vertex v is not visited
11:             v.visited ← TRUE                   ▷ Mark vertex v as visited
12:             Q.ENQUEUE(v)                       ▷ Insert vertex v in the queue
13:         end if
14:     end for
15: end while
```

Algorithm 10 BFS-DISTANCE(s, G)

```
1:  for each vertex v ∈ V do
2:      u.d ← ∞                          ▷ Number of edges to reach vertex u from source vertex s is ∞
3:      u.π ← NIL                        ▷ No predecessor of vertex u, so NIL
4:      u.visited ← FALSE                ▷ Vertex u is not visited initially
5:  end for
6:  Q ← ∅                                ▷ Create a queue to store the vertices
7:  s.visited ← TRUE                     ▷ Mark source vertex s as visited
8:  s.d ← 0                              ▷ Number of edges to reach vertex s from source s is 0
9:  Q.ENQUEUE(s)                         ▷ Insert source vertex s in the queue
10: while Q ≠ ∅ do                       ▷ Q is not empty
11:     u ← Q.DEQUEUE()                  ▷ Remove the vertex from the queue
12:     for each vertex v ∈ G.Adj[u] do  ▷ Explore all the adjacent vertices of vertex u
13:         if v.visited = FALSE then    ▷ Vertex v is not visited
14:             v.visited ← TRUE         ▷ Mark vertex v as visited
15:             v.π ← u                  ▷ Make vertex u as a predecessor of vertex v
16:             v.d ← u.d + 1            ▷ Number of edges to reach vertex v from source vertex s
17:             Q.ENQUEUE(v)             ▷ Insert vertex v in the queue
18:         end if
19:     end for
20: end while
```

Applications:

- Shortest path (considering the number of edges) from source vertex
- Diameter of a graph
  - **Undirected Graph:** `Maximum distance in any BFS` $\leq$ `Diameter` $\leq 2 \times$ `Maximum distance in any BFS`
  - **Directed Graph:** ~~`Diameter` $\leq 2 \times$ `Maxim distance in any BFS`~~
- Bipartite graph

Diameter of a graph: Given a graph (**undirected/directed**) $G = (V, E)$ where the edge weight of all the edges are equal. The diameter of this graph G is defined as the largest shortest path distance in the graph, *i.e.*, diameter $= \max_{u,v \in V} \delta(u, v)$ where $\delta(u, v)$ is the shortest distance between vertices u and v.

Algorithm 11 DIAMETER

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency list representation of an undirected graph $G = (V, E)$
Output: Diameter of the graph G

$n \leftarrow |G.V|$ ▷ Number of vertices in G
$m \leftarrow |G.E|$ ▷ Number of edges in G
`diameter` $\leftarrow -\infty$
for each vertex $u \in G.V$ **do**
    Apply BFS(u) and find the maximum distance label. Let this distance label be d_u.
    if $d_u >$ `diameter` **then**
        `diameter` $\leftarrow d_u$
    end if
end for
return `diameter`

Check whether a given undirected graph $G = (V, E)$ is Bipartite or not. Assume that the graph is connected otherwise we have to check for bipartite for each component.

Algorithm 12 BIPARTITE

Input: `Adj-List`$[1, 2, \ldots, |G.V|]$: Adjacency list representation of an undirected graph $G = (V, E)$
Output: TRUE: If the graph is bipartite, FALSE: otherwise

$n \leftarrow |G.V|$ ▷ Number of vertices in G
$m \leftarrow |G.E|$ ▷ Number of edges in G
for each vertex $u \in G.V$ **do**
    u.`color` $\leftarrow$ NIL
end for
$s \leftarrow$ A random vertex from $G.V$ ▷ Select a source vertex randomly
s.`color` $\leftarrow$ RED ▷ Color source vertex s as RED
Q.ENQUEUE(s) ▷ Add s to the queue
while $Q \neq \emptyset$ **do**
    $u \leftarrow Q$.DEQUEUE() ▷ Dequeue from Q
    for each vertex $v \in$ `Adj-List`$[u]$ **do**
        if v.`color` $=$ NIL **then** ▷ Vertex v is not colored
            if u.`color` $=$ RED **then**
                v.`color` $\leftarrow$ BLUE ▷ Color vertex v as BLUE
            else
                v.`color` $\leftarrow$ RED ▷ Color vertex v as RED
            end if
            Q.ENQUEUE(v) ▷ Add v to the queue
        else
            if u.`color` $=$ v.`color` **then** ▷ Color of u and v is the same
                return FALSE ▷ Graph G is not bipartite
            end if
        end if
    end for
end while
return TRUE ▷ Graph G is bipartite

6 Depth-First Search (DFS)

Given a graph $G = (V, E)$, depth-first search searches "deeper" in the graph whenever possible.

- Discovery and finish times obtained from DFS traversal have parenthesis structure.
- Depth-first search produces a "breadth-first forest".
- The algorithm works on both directed and undirected graphs.

Algorithm 13 DFS-ITERATIVE(s, G)

```
1:  for each vertex v ∈ V do
2:      u.visited ← FALSE                          ▷ Vertex u is not visited initially
3:  end for
4:  S ← ∅                                          ▷ Create a stack to store the vertices
5:  s.visited ← TRUE                               ▷ Mark source vertex s as visited
6:  S.PUSH(s)                                      ▷ Push source vertex s into the stack
7:  while S ≠ ∅ do                                 ▷ S is not empty
8:      u ← Q.POP()                                ▷ Pop the vertex from the stack
9:      for each vertex v ∈ G.Adj[u] do            ▷ Explore all the adjacent vertices of vertex u
10:         if v.visited = FALSE then              ▷ Vertex v is not visited
11:             v.visited ← TRUE                   ▷ Mark vertex v as visited
12:             S.PUSH(v)                          ▷ Push vertex v into the stack
13:         end if
14:     end for
15: end while
```

Algorithm 14 DFS-RECURSIVE(s, G)

```
1:  for each vertex v ∈ V do
2:      u.visited ← FALSE                          ▷ Vertex u is not visited initially
3:  end for
4:  DFS-VISIT(G, s)
```

Algorithm 15 DFS-VISIT(u, G)

```
1:  u.visited ← TRUE                               ▷ Mark vertex u as visited
2:  for each vertex v ∈ G.Adj[u] do                ▷ Explore all the adjacent vertices of vertex u
3:      if v.visited = FALSE then                  ▷ Vertex v is not visited
4:          DFS-VISIT(G, v)
5:      end if
6:  end for
```

Applications:

- Directed Graph
  - Obtain the strongly connected components in a directed graph $G = (V, E)$
  - Check whether a directed graph $G = (V, E)$ contains a cycle or not?
  - Topological sorting in a directed graph $G = (V, E)$
- Undirected Graph
  - Check whether an undirected graph $G = (V, E)$ contains a cycle or not?
  - Given an undirected graph $G = (V, E)$. Any edge whose removal results in a disconnected graph is known as Bridge. A graph is 2-edge connected if it contains no bridges. Check whether the graph G is 2-edge connected or not.

Algorithm 16 2-Edge-Connected-Naive

Input: An undirected graph $G = (V, E)$
Output: True: If the graph is 2-edge connected, False: otherwise

$n \leftarrow |G.V|$ ▷ Number of vertices in G
$m \leftarrow |G.E|$ ▷ Number of edges in G
for each edge $(u, v) \in G.E$ **do**
  $G' \leftarrow G \setminus (u, v)$ ▷ Obtain the graph G' after removing edge (u, v) from G
  if G' is not connected **then**
    return False ▷ Graph G is not 2-edge connected
  end if
end for
return True ▷ Graph G is 2-edge connected

Algorithm 17 Find-All-Bridges(G)

Input: An undirected graph $G = (V, E)$
Output: Bridges in the graph G if exists

for each vertex $u \in G.V$ **do**
  $u.\texttt{color} \leftarrow$ White
  $u.\pi \leftarrow$ Nil
end for
$\texttt{time} \leftarrow 0$
for each vertex $u \in G.V$ **do**
  if $u.\texttt{color} =$ White **then** ▷ Vertex u is not visited yet
    DFS-Visit(u, G)
  end if
end for
/* – – – – – Check for the bridges – – – – – */
for each vertex $v \in G.V$ **do**
  $u \leftarrow v.\pi$ ▷ Edge (u, v) is a tree edge
  if $u \neq$ Nil **AND** $v.d = v.\texttt{low}$ **then** ▷ There is no back edge within v's subtree that leads to a vertex whose discovery time is strictly smaller than v's discovery time
    Edge (u, v) is a bridge
  end if
end for

Algorithm 18 DFS-Visit(u, G)

$\texttt{time} \leftarrow \texttt{time} + 1$
$u.d \leftarrow \texttt{time}$
$u.\texttt{color} \leftarrow$ Gray
$u.\texttt{low} \leftarrow u.d$ ▷ Initialize low
for each vertex $v \in \texttt{Adj-List}[u]$ **do**
  if $v.\texttt{color} =$ White **then** ▷ Edge (u, v) is a tree edge
    $v.\pi \leftarrow u$
    DFS-Visit(v)
    $u.\texttt{low} \leftarrow \min(u.\texttt{low}, v.\texttt{low})$
  else if $v \neq u.\pi$ **then** ▷ Edge (u, v) is a back edge
    $u.\texttt{low} \leftarrow \min(u.\texttt{low}, v.d)$
  end if
end for

7 Connected Graph

- Given an undirected graph $G = (V, E)$. Write an algorithm to check whether the graph G is connected or not.

Algorithm 19 CONNECTED

Input: An undirected graph $G = (V, E)$
Output: TRUE: If the graph is connected, FALSE: otherwise

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: **for each** vertex $u \in G.V$ **do**
4: $L_u \leftarrow \text{DFS}(u, G)$ ▷ Apply DFS/BFS considering vertex u as a source and store all the vertices reachable from u (including u) in L_u
5: **if** $|L_u| < n$ **then** ▷ All the vertices are not reachable from u
6: **return** FALSE ▷ Graph G is not connected
7: **end if**
8: **end for**
9: **return** TRUE ▷ Graph G is connected

Algorithm 20 CONNECTED-BETTER

Input: An undirected graph $G = (V, E)$
Output: TRUE: If the graph is connected, FALSE: otherwise

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Consider a vertex $u \in G.V$
4: $L_u \leftarrow \text{DFS}(u, G)$ ▷ Apply DFS/BFS considering vertex u as a source and store all the vertices reachable from u (including u) in L_u
5: **if** $|L_u| = n$ **then** ▷ All the vertices are reachable from u
6: **return** TRUE ▷ Graph G is connected
7: **else** ▷ All the vertices are not reachable from u
8: **return** FALSE ▷ Graph G is not connected
9: **end if**

- Given a directed graph $G = (V, E)$. Write an algorithm to check whether the graph G is strongly connected or not.

Algorithm 21 STRONGLY-CONNECTED

Input: A directed graph $G = (V, E)$
Output: TRUE: If the graph is strongly connected, FALSE: otherwise

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: **for each** vertex $u \in G.V$ **do**
4: $L_u \leftarrow \text{DFS}(u, G)$ ▷ Apply DFS/BFS considering vertex u as a source and store all the vertices reachable from u (including u) in L_u
5: **if** $|L_u| < n$ **then** ▷ All the vertices are not reachable from u
6: **return** FALSE ▷ Graph G is not strongly connected
7: **end if**
8: **end for**
9: **return** TRUE ▷ Graph G is strongly connected

Algorithm 22 STRONGLY-CONNECTED-BETTER

Input: A directed graph $G = (V, E)$
Output: TRUE: If the graph is strongly connected, FALSE: otherwise

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: $u \leftarrow$ Select a random vertex
4: $L_u \leftarrow \text{DFS}(u, G)$ ▷ Apply DFS/BFS on graph G considering vertex u as a source and store all the vertices reachable from u (including u) in L_u
5: **if** $|L_u| < n$ **then** ▷ All the vertices in G are not reachable from u
6: **return** FALSE ▷ Graph G is not strongly connected
7: **else**
8: $G^T \leftarrow$ Transpose of the graph G ▷ Obtain the transpose graph by reversing the direction of all edges in G
9: $L_u^T \leftarrow \text{DFS}(u, G^T)$ ▷ Apply DFS/BFS on graph G^T considering vertex u as a source and store all the vertices reachable from u (including u) in L_u^T. The set of vertices reachable from u in G^T is basically the set of vertices in original graph G from where vertex u is reachable.
10: **if** $|L_u^T| < n$ **then** ▷ All the vertices in G^T are not reachable from u
11: **return** FALSE ▷ Graph G is not strongly connected
12: **end if**
13: **end if**
14: **return** TRUE ▷ Graph G is strongly connected

- Given a directed graph $G = (V, E)$. Write an algorithm to check whether the graph G is weakly connected or not.

Algorithm 23 WEAKLY-CONNECTED

Input: A directed graph $G = (V, E)$
Output: TRUE: If the graph is weakly connected, FALSE: otherwise

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Obtain the underlying undirected graph G' from the directed graph G
4: **for each** vertex $u \in G'.V$ **do**
5: $L_u \leftarrow \text{DFS}(u, G)$ ▷ Apply DFS/BFS on graph G' considering vertex u as a source and store all the vertices reachable from u (including u) in L_u
6: **if** $|L_u| < n$ **then** ▷ All the vertices are not reachable from u in G'
7: **return** FALSE ▷ Graph G is not weakly connected
8: **end if**
9: **end for**
10: **return** TRUE ▷ Graph G is weakly connected

8 Strongly Connected Components (SCCs)

Algorithm 24 SCC-Naive

Input: A directed graph $G = (V, E)$
Output: All the strongly connected components in the graph G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: **for each** vertex $u \in G.V$ **do**
4: $u.\texttt{inSCC} \leftarrow$ False ▷ Vertex u is not the part of any SCC
5: **end for**
6: $\texttt{SCComponents} \leftarrow \emptyset$ ▷ Initially there is no strongly connected component
7: **for each** vertex $u \in G.V$ **do**
8: **if** $u.\texttt{inSCC} =$ False **then** ▷ Vertex u is not the part of any SCC
/* – – – – – Obtain SCC where vertex u is one of the vertex – – – – – */
9: $L_{\rightarrow} \leftarrow$ Obtain the set of vertices reachable from u (including u)
10: $L_{\leftarrow} \leftarrow$ Obtain the set of vertices from where u can be reached (including u)
11: $\texttt{component} \leftarrow L_{\rightarrow} \cap L_{\leftarrow}$ ▷ SCC where vertex u is a member
12: $\texttt{SCComponents} \leftarrow \texttt{SCComponents} \cup \{\texttt{component}\}$ ▷ Add the obtained SCC to the set of SCC
13: **for each** vertex $v \in \texttt{component}$ **do**
14: $v.\texttt{inSCC} \leftarrow$ True ▷ Mark the vertices of SCC so that they do not take part again
15: **end for**
16: **end if**
17: **end for**
18: **return** SCComponents

9 Kruskal's Algorithm

Algorithm 25 KRUSKAL'S ALGORITHM

Input: A connected undirected graph $G = (V, E)$
Output: Minimum weight spanning tree T of G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: $T \leftarrow \emptyset$ ▷ Initialize minimum weight spanning tree
4: Sort the edges of $G.E$ in non-decreasing order of their weight
5: **for each** vertex $u \in G.V$ **do**
6: MAKE-SET(u) ▷ Create a separate component for each vertex
7: **end for**
8: **for each** edge $(u, v) \in G.E$ taken in non-decreasing order of their weight **do**
9: $\texttt{Rep}_u \leftarrow$ FIND-SET(u) ▷ Find the component where u belongs
10: $\texttt{Rep}_v \leftarrow$ FIND-SET(v) ▷ Find the component where v belongs
11: **if** $\texttt{Rep}_u \neq \texttt{Rep}_v$ **then** ▷ u and v are from different component
12: $T \leftarrow T \cup \{(u, v)\}$ ▷ Add edge (u, v) to the minimum weight spanning tree T
13: UNION($\texttt{Rep}_u, \texttt{Rep}_v$) ▷ Connect u and v
14: **else** ▷ u and v are from the same component
15: Do nothing ▷ Connecting u and v will give a cycle
16: **end if**
17: **end for**
18: **return** T

MAKE-SET()	FIND-SET()	UNION()
$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Table 1: Time complexity using tree representation for disjoint set data-structure.

- Line $1-3$: $\boldsymbol{\mathcal{O}(1)}$
- Line 4: $\boldsymbol{\mathcal{O}(m \log m)}$ as sorting m edges takes $\mathcal{O}(m \log m)$ time
- Line $5-7$: $\boldsymbol{\mathcal{O}(n)}$ as each call to MAKE-SET() takes $\boldsymbol{\mathcal{O}(1)}$ time
- Inside loop in line $8-17$
  - For each edge (u, v), we are calling FIND-SET() **twice** – FIND-SET(u) and FIND-SET(v)
  - Maximum number of call to FIND-SET(): $\boldsymbol{2m}$ as there are m edges
    - Total time taken by FIND-SET(): $2m \times \mathcal{O}(\log n) = \boldsymbol{\mathcal{O}(m \log n)}$
  - Total number of call to UNION(): $\boldsymbol{n - 1}$ as MST has $n - 1$ edges
    - Total time taken by UNION(): $(n - 1) \times \mathcal{O}(1) = \boldsymbol{\mathcal{O}(n)}$

$$
\begin{aligned}
T &= \underbrace{\mathcal{O}(1)}_{\text{Line } 1-3} + \underbrace{\mathcal{O}(m \log m)}_{\text{Line } 4} + \underbrace{\mathcal{O}(n)}_{\text{Line } 5-7} + \underbrace{\mathcal{O}(m \log n) + \mathcal{O}(n)}_{\text{Line } 8-17} \\
&= \mathcal{O}(m \log m) \qquad (1)
\end{aligned}
$$

Algorithm 26 SECOND-BEST MST

Input: A connected undirected graph $G = (V, E)$
Output: Second best minimum weight spanning tree T^* of G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: $T^* \leftarrow \emptyset$ ▷ Initialize second best minimum weight spanning tree
4: $T^*.\texttt{weight} \leftarrow \infty$ ▷ Initially, the sum of the weight of the edges of T^* is ∞
5: Find a MST T of graph G using Kruskal's Algorithm considering edge set $G.E$
6: **for each** edge $(u, v) \in T.E$ **do**
7: $E' \leftarrow G.E - (u, v)$ ▷ Remove edge (u, v) from edge set $G.E$
8: Find a MST T' of $G.E$ using Kruskal's Algorithm considering updated edge set E'
9: **if** $T'.\texttt{weight} < T^*.\texttt{weight}$ **then** ▷ Sum of the weight of the edges of T' is less than that of T^*
10: $T^* \leftarrow T'$ ▷ T' is the updated second best minimum weight spanning tree
11: **end if**
12: **end for**
13: **return** T^*

10 Prim's Algorithm

Given a graph $G = (V, E)$, a cut $C = \left(S, \overline{S}\right)$ is a partition of V into two subsets S and $\overline{S}$. The cut-set of a cut $C = \left(S, \overline{S}\right)$ is the set of edges that have one endpoint in S and the other endpoint in $\overline{S}$, *i.e.*, cut-set of a cut $C = \left\{(u, v) \in E \,|\, u \in S, v \in \overline{S}\right\}$.

Algorithm 27 Prim's Algorithm

Input: A connected undirected graph $G = (V, E)$
Output: Minimum weight spanning tree T of G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: $T \leftarrow \emptyset$ ▷ Initialize minimum weight spanning tree
4: **for each** vertex $u \in G.V$ **do**
5: $u.\pi \leftarrow$ Nil
6: **end for**
7: $s \leftarrow$ A random vertex from $G.V$ ▷ Select a source vertex randomly
8: $S \leftarrow \emptyset$ ▷ Vertices in MST
9: $S \leftarrow S \cup \{s\}$ ▷ Add source vertex s to MST
10: **for** $i \leftarrow 1$ **to** $n - 1$ **do**
11: `edgeset` $\leftarrow \emptyset$ ▷ Initialize the set of edges in the cut
12: **for each** vertex $u \in S$ **do**
13: **for each** vertex $v \in$ `Adj-List`$[u]$ **do**
14: **if** $v \notin S$ **then** ▷ (u, v) is an edge of cut because $u \in S \wedge s \notin S$
15: `edgeset` $\leftarrow$ `edgeset` $\cup \{(u, v)\}$
16: **end if**
17: **end for**
18: **end for**
19: $(u, v) \leftarrow$ Edge with minimum weight from `edgeset` ▷ $u \in S$ and $v \notin S$
20: $T \leftarrow T \cup \{(u, v)\}$ ▷ Add edge (u, v) to the minimum weight spanning tree T
21: $S \leftarrow S \cup \{v\}$ ▷ Add vertex v to S
22: $v.\pi \leftarrow u$ ▷ Update the predecessor of v
23: **end for**
24: **return** T ▷ Minimum weight spanning tree can also be created using predecessor (π) of vertices

Algorithm 28 PRIM'S ALGORITHM-CLASS DISCUSSION

Input: A connected undirected graph $G = (V, E)$
Output: Minimum weight spanning tree T of G

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: $T \leftarrow \emptyset$ ▷ Initialize minimum weight spanning tree
4: **for each** vertex $u \in G.V$ **do**
5: $u.\pi \leftarrow$ NIL
6: **end for**
7: $s \leftarrow$ A random vertex from $G.V$ ▷ Select a source vertex randomly
8: $S_{\texttt{list}} \leftarrow \emptyset$ ▷ Vertices in MST; Implement S as a linked-list
9: $S_{\texttt{bool}}[1, 2, \ldots, n] \leftarrow \emptyset$ ▷ Vertices in MST; Implement S as a boolean array
10: **for each** vertex $u \in G.V$ **do**
11: $S_{\texttt{bool}}[u] \leftarrow$ FALSE ▷ Initially, no vertex is in S
12: **end for**
13: $S_{\texttt{list}} \leftarrow S_{\texttt{list}} \cup \{s\}$ ▷ Add source vertex s to MST
14: $S_{\texttt{bool}}[s] \leftarrow$ TRUE ▷ Add source vertex s to MST
15: **for** $i \leftarrow 1$ **to** $n - 1$ **do**
16: $e \leftarrow \emptyset$ ▷ Initialize the minimum weight edge in the cut
17: $e.\texttt{weight} \leftarrow \infty$ ▷ Initialize the weight of edge e to be ∞
18: **for each** vertex $u \in S_{\texttt{list}}$ **do** ▷ Linked-list representation of S will be better here
19: **for each** vertex $v \in \texttt{Adj-List}[u]$ **do**
20: **if** $v \notin S_{\texttt{list}} (i.e., S_{\texttt{bool}}[v] =$ FALSE$)$ **then** ▷ (u, v) is an edge of cut because $u \in S_{\texttt{list}} \wedge s \notin S_{\texttt{list}}$
21: **if** $(u, v).\texttt{weight} < e.\texttt{weight}$ **then** ▷ Weight of edge (u, v) is less than the minimum weight edge of the cut
22: $e \leftarrow (u, v)$ ▷ Update minimum weight edge of the cut
23: **end if**
24: **end if**
25: **end for**
26: **end for**
Here, edge e is the minimum weight edge of the cut
27: $T \leftarrow T \cup \{e\}$ ▷ Add edge e to the minimum weight spanning tree T
28: $S_{\texttt{list}} \leftarrow S_{\texttt{list}} \cup \{v\}$ ▷ Add vertex v to $S_{\texttt{list}}$
29: $S_{\texttt{bool}}[v] \leftarrow$ TRUE ▷ Add vertex v to $S_{\texttt{bool}}$
30: $v.\pi \leftarrow u$ ▷ Update the predecessor of v
31: **end for**
32: **return** T ▷ Minimum weight spanning tree can also be created using predecessor (π) of vertices

Let the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$. Let the number of vertices in the adjacency list of vertex v_k is n_k. Thus, $\sum_{k=1}^{n} |v_k| = 2m$.

- Line $1 - 3$: $\boldsymbol{\mathcal{O}(1)}$
- Line $4 - 6$: $\boldsymbol{\mathcal{O}(n)}$
- Line $7 - 9$: $\boldsymbol{\mathcal{O}(1)}$
- Line $10 - 12$: $\boldsymbol{\mathcal{O}(n)}$
- Line $13 - 14$: $\boldsymbol{\mathcal{O}(1)}$
- Let the order of vertices in which they are added to set S be $\{v_1, v_2, \ldots, v_n\}$.
- Inside loop in line $15 - 31$
  - At the start of the first iteration of the loop (when $i = 1$), S has one vertex
  - At the start of the second iteration of the loop (when $i = 2$), S has two vertices
  - At the start of the third iteration of the loop (when $i = 3$), S has three vertices
  - In general, At the start of the i^{th} iteration of the loop, S has i vertices
  - In the first iteration of the loop, all the vertices in the adjacency list of v_1 are explored as S has only v_1.
  - In the second iteration of the loop, all the vertices in the adjacency list of v_1, v_2 are explored as S has two vertices v_1, v_2.
  - In the third iteration of the loop, all the vertices in the adjacency list of v_1, v_2, v_3 are explored as S has

three vertices v_1, v_2, v_3.

- In general, in i^{th} iteration of the loop, all the vertices in the adjacency list of $v_1, v_2, \ldots, v_i$ are explored as S has i vertices $v_1, v_2, \ldots, v_i$.
- Hence, the total time taken to explore the adjacency list of vertices in the loop is as follows

$$
\begin{aligned}
l &= \underbrace{n_1}_{i=1} + \underbrace{(n_1+n_2)}_{i=2} + \underbrace{(n_1+n_2+n_3)}_{i=3} + \cdots + \underbrace{(n_1+n_2+\cdots+n_{n-1})}_{i=n-1} \\
l &= (n-1)n_1 + (n-2)n_2 + (n-3)n_3 + \cdots + 2n_{n-2} + n_{n-1} \\
l &\leq (n)n_1 + (n)n_2 + (n)n_3 + \cdots + nn_{n-2} + nn_{n-1} \\
l &\leq n\left(n_1 + n_2 + n_3 + \cdots + n_{n-2} + n_{n-1}\right) \\
l &\leq n\left(2m - n_n\right) \\
l &= \mathcal{O}(mn) \qquad (2)
\end{aligned}
$$

$$
\begin{aligned}
T = &\underbrace{\mathcal{O}(1)}_{\text{Line } 1-3} + \underbrace{\mathcal{O}(n)}_{\text{Line } 4-6} + \underbrace{\mathcal{O}(1)}_{\text{Line } 7-9} + \underbrace{\mathcal{O}(n)}_{\text{Line } 10-12} + \underbrace{\mathcal{O}(1)}_{\text{Line } 13-14} + \underbrace{\mathcal{O}(mn)}_{\text{Line } 15-31} \\
&= \mathcal{O}(mn) \qquad (3)
\end{aligned}
$$

Subpath of shortest path is shortest path: Let p be a shortest path from u to v. Let q be a subpath of p from x to y. Then q is a shortest path from x to y.

Proof. Starts here...

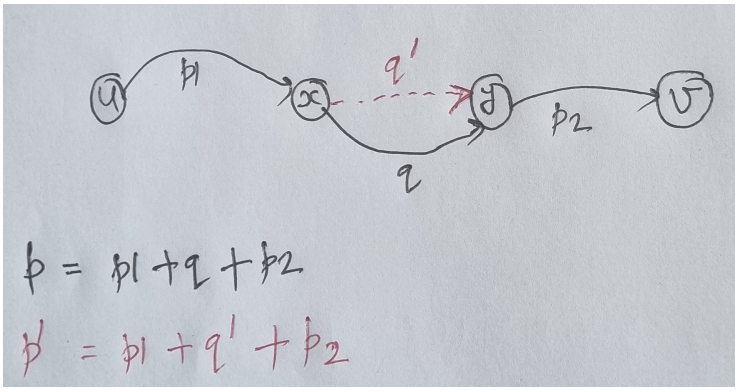
- Let p_1 be the subpath of p from u to x.
- Let p_2 be the subpath of p from y to v.

Therefore,

$$p = \underbrace{p_1}_{\text{Path from } u \text{ to } x} + \underbrace{q}_{\text{Path from } x \text{ to } y} + \underbrace{p_2}_{\text{Path from } y \text{ to } v} \tag{4}$$

and hence

$$w(p) = \underbrace{w(p_1)}_{\text{Path length from } u \text{ to } x} + \underbrace{w(q)}_{\text{Path length from } x \text{ to } y} + \underbrace{w(p_2)}_{\text{Path length from } y \text{ to } v} \tag{5}$$



Let q' be a shortest path from x to y. Thus,

$$w(q') \leq w(q) \tag{6}$$

Let p' be another path from u to v which consists of q'. Therefore,

$$p' = \underbrace{p_1}_{\text{Path from } u \text{ to } x} + \underbrace{q'}_{\text{Path from } x \text{ to } y} + \underbrace{p_2}_{\text{Path from } y \text{ to } v} \tag{7}$$

and hence

$$w(p') = w(p_1) + w(q') + w(p_2) \tag{8}$$

Subtract Eq. (5) from (8),

$$w(p') - w(p) = w(q') - w(q) \tag{9}$$

Since p is a shortest path from u to v, $w(p) \leq w(p') \Rightarrow w(q) \leq w(q')$.
Therefore, $w(q) \leq w(q')$, which means that q is a shortest path from x to y. □

11 Dijkstra's Algorithm

Algorithm 29 Dijkstra's Algorithm

Input: A directed graph $G = (V, E)$ and a source vertex $s \in G.V$
Output: Shortest distance between s to all other vertices

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: **for each** vertex $u \in G.V$ **do**
4: $u.\texttt{key} \leftarrow \infty$ ▷ Length of the shortest path from source vertex s
5: $u.\pi \leftarrow$ Nil
6: **end for**
7: $s.\texttt{key} \leftarrow 0$ ▷ Shortest distance from source vertex to itself is 0
8: $S \leftarrow \emptyset$ ▷ Set of the vertices for which we have the shortest path from source vertex s
9: $\overline{S} \leftarrow G.V$ ▷ Set of the vertices for which we have the shortest path from source vertex s which includes only vertices of S
10: **while** $\overline{S} \neq \emptyset$ **do**
11: $u \leftarrow$ Select a vertex with minimum key from $\overline{S}$
12: $S \leftarrow S \cup \{u\}$ ▷ Add u to S as we have the shortest path from s to u
13: $\overline{S} \leftarrow S \setminus \{u\}$ ▷ Remove u from $\overline{S}$ as we have the shortest path from s to u
14: **for each** vertex $v \in \texttt{Adj-List}[u]$ **do**
15: **if** $v.\texttt{key} > u.\texttt{key} + w(u, v)$ **then** ▷ There exists a better path to reach v from s which goes through u
16: $v.\texttt{key} \leftarrow u.\texttt{key} + w(u, v)$
17: $v.\pi \leftarrow u$ ▷ Shortest path to v has been explored through u
18: **end if**
19: **end for**
20: **end while**
/* $- - - - - - - - - \forall u \in G.V, u.\texttt{key}$ is the shortest path length from s to $u - - - - - - - - -$*/

12 Bellman-Ford Algorithm

- If the graph $G = (V, E)$ contains no negative-weight cycles **reachable from the source** s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph $G = (V, E)$ contains a negative-weight cycle **reachable from source** s, however, shortest-path weights are **not well defined**.
- No path from s to a vertex on the cycle can be a shortest path – we can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle.
- If there is a negative-weight cycle on some path from source s to v, we define $\delta(s, v) = -\infty$.

Algorithm 30 BELLMAN-FORD ALGORITHM

Input: A directed graph $G = (V, E)$ and a source vertex $s \in G.V$
Output: Shortest distance between s to all other vertices

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: **for each** vertex $u \in G.V$ **do**
4: $u.\texttt{key} \leftarrow \infty$ ▷ Length of the shortest path from source vertex s
5: $u.\pi \leftarrow$ NIL
6: **end for**
7: $s.\texttt{key} \leftarrow 0$ ▷ Shortest distance from source vertex to itself is 0
8: **for** $i \leftarrow 1$ **to** $n - 1$ **do**
9: **for each** edge $(u, v) \in G.E$ **do**
10: **if** $v.\texttt{key} > u.\texttt{key} + w(u, v)$ **then** ▷ There exists a better path to reach v from s which goes through u
11: $v.\texttt{key} \leftarrow u.\texttt{key} + w(u, v)$
12: $v.\pi \leftarrow u$ ▷ Shortest path to v has been explored through u
13: **end if**
14: **end for**
15: **end for**
/* – – – – – – – – – Check for Negative weight cycle reachable from s – – – – – – – – –*/
16: **for each** edge $(u, v) \in G.E$ **do**
17: **if** $v.\texttt{key} > u.\texttt{key} + w(u, v)$ **then**
18: **return** FALSE ▷ Negative weight cycle reachable from s
19: **end if**
20: **end for**
21: **return** TRUE ▷ No negative weight cycle reachable from s

13 All-Pair Shortest Path

Let $D_{uv}^{(r)}$ be the weight of a shortest path from vertex u to vertex v that contains at most r edges, *i.e.*, weight of a shortest path from vertex u to vertex v using $\leq r$ edges. Thus, $D_{uv}^{(0)}$ is defined using Equation (10).

$$D_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } u \neq v \end{cases} \tag{10}$$

$D_{uv}^{(1)}$ is obtained using Equation (11).

$$D_{uv}^{(1)} = \begin{cases} 0 & \text{if } u = v \\ w(u,v) & \text{if } (u,v) \in G.E \\ \infty & \text{otherwise} \end{cases} \tag{11}$$

$D_{uv}^{(r)}$ where $r \geq 1$ is defined recursively using Equation (12).

$$D_{uv}^{(r)} = \min \left\{ \underbrace{D_{uv}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } v \text{ with atmost } r-1 \text{ edges}}}, \min \left\{ \underbrace{D_{uk}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } k \text{ with atmost } r-1 \text{ edges}}} + \underbrace{w(k,v)}_{\text{Weight of edge } (k,v)} : k \in \text{All predecessors of } v \right\} \right\} \tag{12}$$

k belongs to the set of vertices for which there is an edge going to v, *i.e.*, all predecessors of v. In a complete graph, all the vertices are predecessor of v. So $k \in G.V$ in case of complete graph. Considering $k \in G.V$ also makes the formulation simple. Doing so will not make any change to the shortest path because if there is no edge from k to v, the edge weight is ∞ and this ∞ value will not affect the minimum calculation. Thus, the updated formulation is given in Equation (13).

$$\begin{aligned} D_{uv}^{(r)} &= \min \left\{ \underbrace{D_{uv}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } v \text{ with atmost } r-1 \text{ edges}}}, \min \left\{ \underbrace{D_{uk}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } k \text{ with atmost } r-1 \text{ edges}}} + \underbrace{w(k,v)}_{\text{Weight of edge } (k,v)} : k \in G.V \right\} \right\} \\ &= \min \left\{ \underbrace{D_{uk}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } k \text{ with atmost } r-1 \text{ edges}}} + \underbrace{w(k,v)}_{\text{Weight of edge } (k,v)} : k \in G.V \right\} \end{aligned} \tag{13}$$

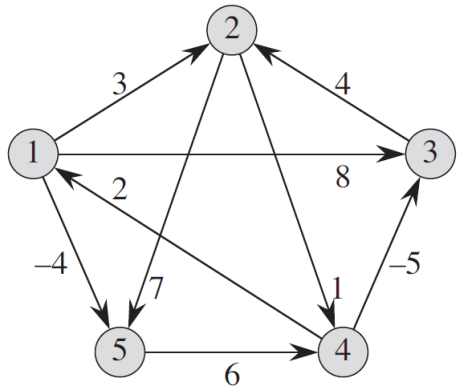


Figure 1: A directed graph.

Algorithm 31 APSP-Recursive

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Create a distance matrix $D[1, 2, \ldots, n][1, 2, \ldots, n]$ of size $n \times n$
4: **for each** vertex $u \in G.V$ **do**
5: **for each** vertex $v \in G.V$ **do**
6: $D[u][v] \leftarrow$ APSP-Helper$(n-1, u, v)$ ▷ Weight of a shortest path from vertex u to vertex v that contains at most $n-1$ edges
7: **end for**
8: **end for**
9: **return** D

Algorithm 32 APSP-Helper(r, u, v)

1: **if** $r = 0$ **then**
2: **if** $u = v$ **then**
3: **return** 0 ▷ Weight of a shortest path from vertex u to vertex v that contains at most 0 edge is 0 because vertex u and vertex v are the same
4: **else**
5: **return** ∞ ▷ Weight of a shortest path from vertex u to vertex v that contains at most 0 edge is ∞ because vertex u and vertex v are different
6: **end if**
7: **else**

Follow the formulation $$D_{uv}^{(r)} = \min \left\{ \underbrace{D_{uk}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } k \text{ with atmost } r-1 \text{ edges}}} + \underbrace{w(k, v)}_{\text{Weight of edge } (k,v)} : k \in G.V \right\}$$

8: `minDistance` $\leftarrow \infty$
9: **for each** vertex $k \in G.V$ **do**
10: `dist` $\leftarrow$ APSP-Helper$(r-1, u, k) + w(k, v)$
11: **if** `dist` $<$ `minDistance` **then**
12: `minDistance` $\leftarrow$ `dist`
13: **end if**
14: **end for**
15: **return** `minDistance`
16: **end if**

Recurrence Relation to Obtain the Shortest Path From Vertex u to Vertex v: Let $N = n$

$$T(n-1) = \underbrace{N}_{\substack{\text{No. of times APSP-Helper} \\ \text{is called in line 10}}} \left[\underbrace{T(n-2)}_{\substack{\text{Time by APSP-Helper} \\ \text{in line 10}}} + \underbrace{\mathcal{O}(1)}_{\substack{\text{Weight addition} \\ \text{in line 10}}} \right] + \mathcal{O}(1)$$

$$\begin{aligned}
&= NT(n-2) + \mathcal{O}(N) \\
&= NT(n-2) + N \\
&= N\left[NT(n-3) + N\right] + N \\
&= N^2T(n-3) + N^2 + N \\
&= N^2\left[NT(n-4) + N\right] + N^2 + N \\
&= N^3T(n-4) + N^3 + N^2 + N \\
&= N^3\left[NT(n-5) + N\right] + N^3 + N^2 + N \\
&= N^4T(n-5) + N^4 + N^3 + N^2 + N
\end{aligned}$$

$$
\begin{aligned}
&= \vdots \\
&= N^{N-1}T(n-n) + N^{N-1} + N^{N-2} + \cdots + N^4 + N^3 + N^2 + N \\
&= N^{N-1}T(0) + N^{N-1} + N^{N-2} + \cdots + N^4 + N^3 + N^2 + N \\
&= N^{N-1} + N^{N-1} + N^{N-2} + \cdots + N^4 + N^3 + N^2 + N \\
&= N^{N-1} + N\frac{N^{N-1}-1}{N-1} \\
&= \mathcal{O}(N^{N-1}) \\
&= \mathcal{O}(n^{n-1})
\end{aligned} \tag{14}
$$

- Time complexity to obtain the shortest path from vertex u to vertex v
  - $T(n) = \mathcal{O}(n^{n-1})$
- Need to find the shortest path between each pair of vertices
- Time complexity to obtain the shortest path between each pair of vertices
  - Number of pairs $\times T(n)$
  - $n^2 \times \mathcal{O}(n^{n-1})$
  - $\mathcal{O}(n^2 n^{n-1})$

Algorithm 33 APSP-Recursive-Memoization

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

$n \leftarrow |G.V|$ ▷ Number of vertices in G
$m \leftarrow |G.E|$ ▷ Number of edges in G
Create n distance matrices $D^r[1, 2, \ldots, n][1, 2, \ldots, n]$ of size $n \times n$ where $0 \leq r \leq n - 1$
for $r \leftarrow 0$ **to** $n - 1$ **do**
    for each vertex $u \in G.V$ **do**
        for each vertex $v \in G.V$ **do**
            $D^r[u][v] \leftarrow -\infty$
        end for
    end for
end for
for each vertex $u \in G.V$ **do**
    for each vertex $v \in G.V$ **do**
        $D^{n-1}[u][v] \leftarrow$ APSP-Helper-Memoization$(n - 1, u, v)$ ▷ Weight of a shortest path from vertex u to vertex v that contains at most $n - 1$ edges
    end for
end for
return D

Algorithm 34 APSP-Helper-Memoization(r, u, v)

if $D^r[u][v] \neq -\infty$ **then** ▷ Weight of a shortest path from vertex u to vertex v that contains at most r edges is already computed
    return $D^r[u][v]$
end if
if $r = 0$ **then**
    if $u = v$ **then**
        return 0 ▷ Weight of a shortest path from vertex u to vertex v that contains at most 0 edge is 0 because vertex u and vertex v are the same
    else
        return ∞ ▷ Weight of a shortest path from vertex u to vertex v that contains at most 0 edge is ∞ because vertex u and vertex v are different
    end if
else

Follow the formulation

$$D_{uv}^{(r)} = \min \left\{ \underbrace{D_{uk}^{(r-1)}}_{\substack{\text{Weight of a shortest path from} \\ u \text{ to } k \text{ with atmost } r-1 \text{ edges}}} + \underbrace{w(k, v)}_{\text{Weight of edge } (k,v)} : k \in G.V \right\} \tag{15}$$

    `minDistance` $\leftarrow \infty$
    for each vertex $k \in G.V$ **do**
        `dist` $\leftarrow$ APSP-Helper$(r - 1, u, k) + w(k, v)$
        if `dist` $<$ `minDistance` **then**
            `minDistance` $\leftarrow$ `dist`
        end if
    end for
    $D^r[u][v] \leftarrow$ `minDistance`
    return `minDistance`
end if

Algorithm 35 APSP-Bottom-Up

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

```
1: n ← |G.V|                                          ▷ Number of vertices in G
2: m ← |G.E|                                          ▷ Number of edges in G
3: Create n distance matrices D^r[1, 2, ..., n][1, 2, ..., n] of size n × n where 0 ≤ r ≤ n − 1
4: for each vertex u ∈ G.V do
5:     for each vertex v ∈ G.V do
6:         if u = v then
7:             D^0[u][v] ← 0
8:         else
9:             D^0[u][v] ← ∞
10:        end if
11:    end for
12: end for
13: for r ← 1 to n − 1 do
14:    for each vertex u ∈ G.V do
15:        for each vertex v ∈ G.V do
16:            minDistance ← ∞
17:            for each vertex k ∈ G.V do
18:                dist ← D^{r−1}[u][v] + w(k, v)
19:                if dist < minDistance then
20:                    minDistance ← dist
21:                end if
22:            end for
23:            D^r[u][v] ← minDistance
24:        end for
25:    end for
26: end for
27: return D^{n−1}
```

Obtain $D^{(0)}$:

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \tag{16}$$

Obtain $D^{(1)}$:

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{17}$$

Obtain $D^{(2)}$:

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & \mathbf{2} & -4 \\ \mathbf{3} & 0 & \mathbf{-4} & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & \mathbf{-1} & -5 & 0 & \mathbf{-2} \\ \mathbf{8} & \infty & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{18}$$

$D_{14}^{(2)} = \min \begin{cases} D_{11}^{(1)} + w(1,4) = 0 + \infty & = \infty \\ D_{12}^{(1)} + w(2,4) = 3 + 1 & = 4 \\ D_{13}^{(1)} + w(3,4) = 8 + \infty & = \infty \\ D_{14}^{(1)} + w(4,4) = \infty + 0 & = \infty \\ D_{15}^{(1)} + w(5,4) = -4 + 6 & = 2 \end{cases} = 2$	$D_{21}^{(2)} = \min \begin{cases} D_{21}^{(1)} + w(1,1) = \infty + 0 & = \infty \\ D_{22}^{(1)} + w(2,1) = 0 + \infty & = \infty \\ D_{23}^{(1)} + w(3,1) = \infty + \infty & = \infty \\ D_{24}^{(1)} + w(4,1) = 1 + 2 & = 3 \\ D_{25}^{(1)} + w(5,1) = 7 + \infty & = \infty \end{cases} = 3$
$D_{23}^{(2)} = \min \begin{cases} D_{21}^{(1)} + w(1,3) = \infty + 8 & = \infty \\ D_{22}^{(1)} + w(2,3) = 0 + \infty & = \infty \\ D_{23}^{(1)} + w(3,3) = \infty + 0 & = \infty \\ D_{24}^{(1)} + w(4,3) = 1 - 5 & = -4 \\ D_{25}^{(1)} + w(5,3) = 7 + \infty & = \infty \end{cases} = -4$	$D_{34}^{(2)} = \min \begin{cases} D_{31}^{(1)} + w(1,4) = \infty + \infty & = \infty \\ D_{32}^{(1)} + w(2,4) = 4 + 1 & = 5 \\ D_{33}^{(1)} + w(3,4) = 0 + \infty & = \infty \\ D_{34}^{(1)} + w(4,4) = \infty + 0 & = \infty \\ D_{35}^{(1)} + w(5,4) = \infty + 6 & = \infty \end{cases} = 5$
$D_{35}^{(2)} = \min \begin{cases} D_{31}^{(1)} + w(1,5) = \infty - 4 & = \infty \\ D_{32}^{(1)} + w(2,5) = 4 + 7 & = 11 \\ D_{33}^{(1)} + w(3,5) = 0 + \infty & = \infty \\ D_{34}^{(1)} + w(4,5) = \infty + \infty & = \infty \\ D_{35}^{(1)} + w(5,5) = \infty + 0 & = \infty \end{cases} = 11$	$D_{42}^{(2)} = \min \begin{cases} D_{41}^{(1)} + w(1,2) = 2 + 3 & = 5 \\ D_{42}^{(1)} + w(2,2) = \infty + 0 & = \infty \\ D_{43}^{(1)} + w(3,2) = -5 + 4 & = -1 \\ D_{44}^{(1)} + w(4,2) = 0 + \infty & = \infty \\ D_{45}^{(1)} + w(5,2) = \infty + \infty & = \infty \end{cases} = -1$
$D_{45}^{(2)} = \min \begin{cases} D_{41}^{(1)} + w(1,5) = 2 - 4 & = -2 \\ D_{42}^{(1)} + w(2,5) = \infty + 7 & = \infty \\ D_{43}^{(1)} + w(3,5) = -5 + \infty & = \infty \\ D_{44}^{(1)} + w(4,5) = 0 + \infty & = \infty \\ D_{45}^{(1)} + w(5,5) = \infty + 0 & = \infty \end{cases} = -2$	$D_{51}^{(2)} = \min \begin{cases} D_{51}^{(1)} + w(1,1) = \infty + 0 & = \infty \\ D_{52}^{(1)} + w(2,1) = \infty + \infty & = \infty \\ D_{53}^{(1)} + w(3,1) = \infty + \infty & = \infty \\ D_{54}^{(1)} + w(4,1) = 6 + 2 & = 8 \\ D_{55}^{(1)} + w(5,1) = 0 + \infty & = \infty \end{cases} = 8$
$D_{53}^{(2)} = \min \begin{cases} D_{51}^{(1)} + w(1,3) = \infty + 8 & = \infty \\ D_{52}^{(1)} + w(2,3) = \infty + \infty & = \infty \\ D_{53}^{(1)} + w(3,3) = \infty + 0 & = \infty \\ D_{54}^{(1)} + w(4,3) = 6 - 5 & = 1 \\ D_{55}^{(1)} + w(5,3) = 0 + \infty & = \infty \end{cases} = 1$	

Obtain $D^{(3)}$:

$$D^{(3)} = \begin{bmatrix} 0 & 3 & \mathbf{-3} & 2 & -4 \\ 3 & 0 & -4 & 1 & \mathbf{-1} \\ \mathbf{7} & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \mathbf{5} & 1 & 6 & 0 \end{bmatrix} \tag{19}$$

$$D_{13}^{(3)} = \min \begin{cases} D_{11}^{(2)} + w(1,3) = 0 + 8 & = 8 \\ D_{12}^{(2)} + w(2,3) = 3 + \infty & = \infty \\ D_{13}^{(2)} + w(3,3) = 8 + 0 & = 8 \\ D_{14}^{(2)} + w(4,3) = 2 - 5 & = -3 \\ D_{15}^{(2)} + w(5,3) = -4 + \infty & = \infty \end{cases} = -3$$

$$D_{25}^{(3)} = \min \begin{cases} D_{21}^{(2)} + w(1,5) = 3 - 4 & = -1 \\ D_{22}^{(2)} + w(2,5) = 0 + 7 & = 7 \\ D_{23}^{(2)} + w(3,5) = -4 + \infty & = \infty \\ D_{24}^{(2)} + w(4,5) = 1 + \infty & = \infty \\ D_{25}^{(2)} + w(5,5) = 7 + 0 & = 7 \end{cases} = -1$$

$$D_{31}^{(3)} = \min \begin{cases} D_{31}^{(2)} + w(1,1) = \infty + 0 & = \infty \\ D_{32}^{(2)} + w(2,1) = 4 + \infty & = \infty \\ D_{33}^{(2)} + w(3,1) = 0 + \infty & = \infty \\ D_{34}^{(2)} + w(4,1) = 5 + 2 & = 7 \\ D_{35}^{(2)} + w(5,1) = 11 + \infty & = \infty \end{cases} = 7$$

$$D_{52}^{(3)} = \min \begin{cases} D_{51}^{(2)} + w(1,2) = 8 + 3 & = 11 \\ D_{52}^{(2)} + w(2,2) = \infty + 0 & = \infty \\ D_{53}^{(2)} + w(3,2) = 1 + 4 & = 5 \\ D_{54}^{(2)} + w(4,2) = 6 + \infty & = \infty \\ D_{55}^{(2)} + w(5,2) = 0 + \infty & = \infty \end{cases} = 5$$

Obtain $D^{(4)}$:

$$D^{(4)} = \begin{bmatrix} 0 & \mathbf{1} & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{20}$$

$$D_{12}^{(4)} = \min \begin{cases} D_{11}^{(3)} + w(1,2) = 0 + 3 & = 3 \\ D_{12}^{(3)} + w(2,2) = 3 + 0 & = 3 \\ D_{13}^{(3)} + w(3,2) = -3 + 4 & = 1 \\ D_{14}^{(3)} + w(4,2) = 2 + \infty & = \infty \\ D_{15}^{(3)} + w(5,2) = -4 + \infty & = \infty \end{cases} = 1$$

$$D_{35}^{(4)} = \min \begin{cases} D_{31}^{(3)} + w(1,5) = 0 + 3 & = 3 \\ D_{32}^{(3)} + w(2,5) = 3 + 0 & = 3 \\ D_{33}^{(3)} + w(3,5) = -3 + 4 & = 1 \\ D_{34}^{(3)} + w(4,5) = 2 + \infty & = \infty \\ D_{35}^{(3)} + w(5,5) = -4 + \infty & = \infty \end{cases} = 3$$

14 All-Pair Shortest Path – Floywd Warshal

Let $D_{uv}^{(k)}$ be the weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Thus, $D_{uv}^{(0)}$ is defined using Equation (21).

$$D_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v \\ w(u,v) & \text{if } (u,v) \in G.E \\ \infty & \text{otherwise} \end{cases} \tag{21}$$

$D_{uv}^{(k)}$ where $k \geq 1$ is defined recursively using Equation (22).

$$D_{uv}^{(k)} = \min \begin{cases} \underbrace{D_{uv}^{(k-1)}}_{\substack{\text{Weight of a shortest path from } u \text{ to } v \text{ for which} \\ \text{all intermediate vertices are in the set } \{1,2,\ldots,k-1\}}} \\ \underbrace{D_{uk}^{(k-1)}}_{\substack{\text{Weight of a shortest path from } u \text{ to } k \text{ for which} \\ \text{all intermediate vertices are in the set } \{1,2,\ldots,k-1\}}} + \underbrace{D_{kv}^{(k-1)}}_{\substack{\text{Weight of a shortest path from } k \text{ to } v \text{ for which} \\ \text{all intermediate vertices are in the set } \{1,2,\ldots,k-1\}}} \end{cases} \tag{22}$$

Algorithm 36 FLOYD-WARSHALL-RECURSIVE

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

```
n ← |G.V|                                   ▷ Number of vertices in G
m ← |G.E|                                   ▷ Number of edges in G
Create a distance matrix D[1, 2, ..., n][1, 2, ..., n] of size n × n
for each u ∈ G.V do
    for each v ∈ G.V do
        D[u][v] ← FW-HELPER(n, u, v)        ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set {1, 2, ..., n}
    end for
end for
return D
```

Algorithm 37 FW-HELPER

```
if k = 0 then
    if u = v then
        return 0        ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set {} (i.e., no intermediate vertex) is 0 because vertex u and vertex v are the same
    else if (u, v) ∈ G.E then
        return w(u, v)  ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set {} (i.e., no intermediate vertex) is w(u, v) because there is an edge from vertex u to vertex v
    else
        return ∞        ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set {} (i.e., no intermediate vertex) is ∞ because there is no edge from vertex u to vertex v
    end if
else
```

$$\textbf{return } \min \begin{cases} \text{FW-HELPER}(k-1, u, v) \\ \text{FW-HELPER}(k-1, u, k) + \text{FW-HELPER}(k-1, k, v) \end{cases}$$

```
end if
```

Recurrence Relation to Obtain the Shortest Path From Vertex u to Vertex v:

$$
\begin{aligned}
T(n) &= \underbrace{3}_{\text{No. of times FW-HELPER is called in line 10}} \times \underbrace{T(n-1)}_{\text{Time by FW-HELPER in line 10}} + \underbrace{\mathcal{O}(1)}_{\text{Other factors}} \\
&= 3T(n-1)+1 \\
&= 3\left[3T(n-2)+1\right]+1 \\
&= 3^2T(n-2)+[3+1] \\
&= 3^2\left[3T(n-3)+1\right]+[3+1] \\
&= 3^3T(n-3)+\left[3^2+3+1\right] \\
&= 3^3\left[3T(n-4)+1\right]+\left[3^2+3+1\right] \\
&= 3^4T(n-4)+\left[3^3+3^2+3+1\right] \\
&\vdots \\
&= 3^nT(n-n)+\left[3^{n-1}+2^{n-2}+\cdots+3^3+3^2+3+1\right] \\
&= 3^nT(0)+\left[3^{n-1}+2^{n-2}+\cdots+3^3+3^2+3+1\right] \\
&= 3^n+\left[3^{n-1}+2^{n-2}+\cdots+3^3+3^2+3+1\right] \\
&= \frac{1}{2}(3^{n+1}-1) \\
&= \mathcal{O}(3^{n+1})
\end{aligned} \tag{23}
$$

- Time complexity to obtain the shortest path from vertex u to vertex v
  - $T(n) = \mathcal{O}(3^{n+1})$
- Need to find the shortest path between each pair of vertices
- Time complexity to obtain the shortest path between each pair of vertices
  - Number of pairs $\times T(n)$
  - $n^2 \times \mathcal{O}(3^{n+1})$
  - $\mathcal{O}(n^2 3^{n+1})$

Algorithm 38 FW-RECURSIVE-MEMOIZATION

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

```
1:  n ← |G.V|                                          ▷ Number of vertices in G
2:  m ← |G.E|                                          ▷ Number of edges in G
3:  Create n + 1 distance matrices D^k[1, 2, ..., n][1, 2, ..., n] of size n × n where 0 ≤ k ≤ n
4:  for k ← 0 to n do
5:      for each vertex u ∈ G.V do
6:          for each vertex v ∈ G.V do
7:              D^k[u][v] ← −∞
8:          end for
9:      end for
10: end for
11: for each vertex u ∈ G.V do
12:     for each vertex v ∈ G.V do
13:         D^n[u][v] ← FW-HELPER-MEMOIZATION(n, u, v)   ▷ Weight of a shortest path from vertex u
            to vertex v for which all intermediate vertices are in the set {1, 2, ..., n}
14:     end for
15: end for
16: return D
```

Algorithm 39 FW-Helper-Memoization(k, u, v)

if $D^k[u][v] \neq -\infty$ **then** ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$ is already computed
  return $D^k[u][v]$
end if
if $k = 0$ **then**
  if $u = v$ **then**
    $D^k[u][v] \leftarrow 0$ ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set $\{\}$ (*i.e.*, no intermediate vertex) is 0 because vertex u and vertex v are the same
    return 0
  else if $(u, v) \in G.E$ **then**
    $D^k[u][v] \leftarrow w(u, v)$ ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set $\{\}$ (*i.e.*, no intermediate vertex) is $w(u, v)$ because there is an edge from vertex u to vertex v
    return $w(u, v)$
  else
    $D^k[u][v] \leftarrow \infty$ ▷ Weight of a shortest path from vertex u to vertex v for which all intermediate vertices are in the set $\{\}$ (*i.e.*, no intermediate vertex) is ∞ because there is no edge from vertex u to vertex v
    return ∞
  end if
else

$$D^k[u][v] = \min \begin{cases} \text{FW-Helper}(k-1, u, v) \\ \text{FW-Helper}(k-1, u, k) + \text{FW-Helper}(k-1, k, v) \end{cases}$$

  return $D^k[u][v]$
end if

Algorithm 40 FW-Bottom-Up

Input: A directed graph $G = (V, E)$
Output: Shortest distance between all pair of vertices

1: $n \leftarrow |G.V|$ ▷ Number of vertices in G
2: $m \leftarrow |G.E|$ ▷ Number of edges in G
3: Create n distance matrices $D^k[1, 2, \ldots, n][1, 2, \ldots, n]$ of size $n \times n$ where $0 \leq k \leq n$
4: **for each** vertex $u \in G.V$ **do**
5: **for each** vertex $v \in G.V$ **do**
6: **if** $u = v$ **then**
7: $D^0[u][v] \leftarrow 0$
8: **else if** $(u, v) \in G.E$ **then**
9: $D^0[u][v] \leftarrow w(u, v)$
10: **else**
11: $D^0[u][v] \leftarrow \infty$
12: **end if**
13: **end for**
14: **end for**
15: **for** $k \leftarrow 1$ **to** n **do**
16: **for each** vertex $u \in G.V$ **do**
17: **for each** vertex $v \in G.V$ **do**
18:

$$D^k[u][v] = \min \begin{cases} D^{k-1}[u][v] \\ D^{k-1}[u][k] + D^{k-1}[k][v] \end{cases}$$

19: **end for**
20: **end for**
21: **end for**
22: **return** D^n

Obtain $D^{(0)}$ and $\Pi^{(0)}$:

$$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{24}$$

$$\Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix} \tag{25}$$

Obtain $D^{(1)}$ and $\Pi^{(1)}$:

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{26}$$

$$\Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \mathbf{1} & 4 & \text{NIL} & \mathbf{1} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix} \tag{27}$$

$D_{42}^{(1)} = \min \begin{cases} D_{42}^{(0)} & = \infty \\ D_{41}^{(0)} + D_{12}^{(0)} & = 2+3 \end{cases} = 5$	$\Pi_{42}^{(1)} = \begin{cases} \Pi_{42}^{(0)} & \text{if } D_{42}^{(0)} \leq D_{41}^{(0)} + D_{12}^{(0)} \\ \Pi_{12}^{(0)} & \text{if } D_{42}^{(0)} > D_{41}^{(0)} + D_{12}^{(0)} \end{cases} = 1$
$D_{45}^{(1)} = \min \begin{cases} D_{45}^{(0)} & = \infty \\ D_{41}^{(0)} + D_{15}^{(0)} & = 2-4 \end{cases} = -2$	$\Pi_{45}^{(1)} = \begin{cases} \Pi_{45}^{(0)} & \text{if } D_{45}^{(0)} \leq D_{41}^{(0)} + D_{15}^{(0)} \\ \Pi_{15}^{(0)} & \text{if } D_{45}^{(0)} > D_{41}^{(0)} + D_{15}^{(0)} \end{cases} = 1$

Obtain $D^{(2)}$ and $\Pi^{(2)}$:

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & \mathbf{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{28}$$

$$\Pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \mathbf{2} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \mathbf{2} & \mathbf{2} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix} \tag{29}$$

$D_{14}^{(2)} = \min \begin{cases} D_{14}^{(1)} & = \infty \\ D_{12}^{(1)} + D_{24}^{(1)} & = 3+1 \end{cases} = 4$	$\Pi_{14}^{(2)} = \begin{cases} \Pi_{14}^{(1)} & \text{if } D_{42}^{(1)} \leq D_{12}^{(1)} + D_{24}^{(1)} \\ \Pi_{24}^{(1)} & \text{if } D_{42}^{(1)} > D_{12}^{(1)} + D_{24}^{(1)} \end{cases} = 2$
$D_{34}^{(2)} = \min \begin{cases} D_{34}^{(1)} & = \infty \\ D_{32}^{(1)} + D_{24}^{(1)} & = 4+1 \end{cases} = 5$	$\Pi_{34}^{(2)} = \begin{cases} \Pi_{34}^{(1)} & \text{if } D_{34}^{(1)} \leq D_{32}^{(1)} + D_{24}^{(1)} \\ \Pi_{24}^{(1)} & \text{if } D_{34}^{(1)} > D_{32}^{(1)} + D_{24}^{(1)} \end{cases} = 2$
$D_{35}^{(2)} = \min \begin{cases} D_{35}^{(1)} & = \infty \\ D_{32}^{(1)} + D_{25}^{(1)} & = 4+7 \end{cases} = 11$	$\Pi_{35}^{(2)} = \begin{cases} \Pi_{35}^{(1)} & \text{if } D_{35}^{(1)} \leq D_{32}^{(1)} + D_{25}^{(1)} \\ \Pi_{25}^{(1)} & \text{if } D_{35}^{(1)} > D_{32}^{(1)} + D_{25}^{(1)} \end{cases} = 2$

Obtain $D^{(3)}$ and $\Pi^{(3)}$:

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \mathbf{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{30}$$

$$\Pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & \mathbf{3} & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & NIL \end{bmatrix} \tag{31}$$

$D_{42}^{(3)} = \min \begin{cases} D_{42}^{(2)} & = 5 \\ D_{43}^{(2)} + D_{32}^{(2)} & = -5+4 \end{cases} = -1$	$\Pi_{42}^{(3)} = \begin{cases} \Pi_{42}^{(2)} & \text{if } D_{42}^{(2)} \leq D_{43}^{(2)} + D_{32}^{(2)} \\ \Pi_{32}^{(2)} & \text{if } D_{42}^{(2)} > D_{43}^{(2)} + D_{32}^{(2)} \end{cases} = 3$

Obtain $D^{(4)}$ and $\Pi^{(4)}$:

$$D^{(4)} = \begin{bmatrix} 0 & 3 & \mathbf{-1} & 4 & -4 \\ \mathbf{3} & 0 & \mathbf{-4} & 1 & \mathbf{-1} \\ \mathbf{7} & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & \mathbf{5} & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{32}$$

$$\Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & \mathbf{4} & 2 & 1 \\ \mathbf{4} & \text{NIL} & \mathbf{4} & 2 & \mathbf{1} \\ \mathbf{4} & 3 & \text{NIL} & 2 & \mathbf{1} \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \mathbf{4} & \mathbf{3} & \mathbf{4} & 5 & \text{NIL} \end{bmatrix} \tag{33}$$

$D^{(4)}_{13} = \min \begin{cases} D^{(3)}_{13} & = 8 \\ D^{(3)}_{14} + D^{(3)}_{43} & = 4 - 5 \end{cases} = -1$	$\Pi^{(4)}_{13} = \begin{cases} \Pi^{(3)}_{13} & \text{if } D^{(3)}_{13} \leq D^{(3)}_{14} + D^{(3)}_{43} \\ \Pi^{(3)}_{43} & \text{if } D^{(3)}_{13} > D^{(3)}_{14} + D^{(3)}_{43} \end{cases} = 4$
$D^{(4)}_{21} = \min \begin{cases} D^{(3)}_{21} & = \infty \\ D^{(3)}_{24} + D^{(3)}_{41} & = 1 + 2 \end{cases} = 3$	$\Pi^{(4)}_{21} = \begin{cases} \Pi^{(3)}_{21} & \text{if } D^{(3)}_{21} \leq D^{(3)}_{24} + D^{(3)}_{41} \\ \Pi^{(3)}_{41} & \text{if } D^{(3)}_{21} > D^{(3)}_{24} + D^{(3)}_{41} \end{cases} = 4$
$D^{(4)}_{23} = \min \begin{cases} D^{(3)}_{23} & = \infty \\ D^{(3)}_{24} + D^{(3)}_{43} & = 1 - 5 \end{cases} = -4$	$\Pi^{(4)}_{23} = \begin{cases} \Pi^{(3)}_{23} & \text{if } D^{(3)}_{23} \leq D^{(3)}_{24} + D^{(3)}_{43} \\ \Pi^{(3)}_{43} & \text{if } D^{(3)}_{23} > D^{(3)}_{24} + D^{(3)}_{43} \end{cases} = 4$
$D^{(4)}_{25} = \min \begin{cases} D^{(3)}_{25} & = 7 \\ D^{(3)}_{24} + D^{(3)}_{45} & = 1 - 2 \end{cases} = -1$	$\Pi^{(4)}_{25} = \begin{cases} \Pi^{(3)}_{25} & \text{if } D^{(3)}_{25} \leq D^{(3)}_{24} + D^{(3)}_{45} \\ \Pi^{(3)}_{45} & \text{if } D^{(3)}_{25} > D^{(3)}_{24} + D^{(3)}_{45} \end{cases} = 1$
$D^{(4)}_{31} = \min \begin{cases} D^{(3)}_{31} & = \infty \\ D^{(3)}_{34} + D^{(3)}_{41} & = 5 + 2 \end{cases} = 7$	$\Pi^{(4)}_{31} = \begin{cases} \Pi^{(3)}_{31} & \text{if } D^{(3)}_{31} \leq D^{(3)}_{34} + D^{(3)}_{41} \\ \Pi^{(3)}_{41} & \text{if } D^{(3)}_{31} > D^{(3)}_{34} + D^{(3)}_{41} \end{cases} = 4$
$D^{(4)}_{35} = \min \begin{cases} D^{(3)}_{35} & = 11 \\ D^{(3)}_{34} + D^{(3)}_{45} & = 5 - 2 \end{cases} = 3$	$\Pi^{(4)}_{35} = \begin{cases} \Pi^{(3)}_{35} & \text{if } D^{(3)}_{35} \leq D^{(3)}_{34} + D^{(3)}_{45} \\ \Pi^{(3)}_{45} & \text{if } D^{(3)}_{35} > D^{(3)}_{34} + D^{(3)}_{45} \end{cases} = 1$
$D^{(4)}_{51} = \min \begin{cases} D^{(3)}_{51} & = \infty \\ D^{(3)}_{54} + D^{(3)}_{41} & = 6 + 2 \end{cases} = 8$	$\Pi^{(4)}_{51} = \begin{cases} \Pi^{(3)}_{51} & \text{if } D^{(3)}_{51} \leq D^{(3)}_{54} + D^{(3)}_{41} \\ \Pi^{(3)}_{41} & \text{if } D^{(3)}_{51} > D^{(3)}_{54} + D^{(3)}_{41} \end{cases} = 4$
$D^{(4)}_{52} = \min \begin{cases} D^{(3)}_{52} & = \infty \\ D^{(3)}_{54} + D^{(3)}_{42} & = 6 - 1 \end{cases} = 5$	$\Pi^{(4)}_{52} = \begin{cases} \Pi^{(3)}_{52} & \text{if } D^{(3)}_{52} \leq D^{(3)}_{54} + D^{(3)}_{42} \\ \Pi^{(3)}_{42} & \text{if } D^{(3)}_{52} > D^{(3)}_{54} + D^{(3)}_{42} \end{cases} = 3$
$D^{(4)}_{53} = \min \begin{cases} D^{(3)}_{53} & = \infty \\ D^{(3)}_{54} + D^{(3)}_{43} & = 6 - 5 \end{cases} = 1$	$\Pi^{(4)}_{53} = \begin{cases} \Pi^{(3)}_{53} & \text{if } D^{(3)}_{53} \leq D^{(3)}_{54} + D^{(3)}_{43} \\ \Pi^{(3)}_{43} & \text{if } D^{(3)}_{53} > D^{(3)}_{54} + D^{(3)}_{43} \end{cases} = 4$

Obtain $D^{(5)}$ and $\Pi^{(5)}$:

$$D^{(5)} = \begin{bmatrix} 0 & \mathbf{1} & \mathbf{-3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{34}$$

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & \mathbf{3} & \mathbf{4} & \mathbf{5} & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix} \tag{35}$$

$D_{12}^{(5)} = \min \begin{cases} D_{12}^{(4)} & = 3 \\ D_{15}^{(4)} + D_{52}^{(4)} & = -4+5 \end{cases} = 1$	$\Pi_{12}^{(5)} = \begin{cases} \Pi_{12}^{(4)} & \text{if } D_{12}^{(4)} \le D_{15}^{(4)} + D_{52}^{(4)} \\ \Pi_{52}^{(4)} & \text{if } D_{12}^{(4)} > D_{15}^{(4)} + D_{52}^{(4)} \end{cases} = 3$
$D_{13}^{(5)} = \min \begin{cases} D_{13}^{(4)} & = -1 \\ D_{15}^{(4)} + D_{53}^{(4)} & = -4+1 \end{cases} = -3$	$\Pi_{13}^{(5)} = \begin{cases} \Pi_{13}^{(4)} & \text{if } D_{13}^{(4)} \le D_{15}^{(4)} + D_{53}^{(4)} \\ \Pi_{53}^{(4)} & \text{if } D_{13}^{(4)} > D_{15}^{(4)} + D_{53}^{(4)} \end{cases} = 4$
$D_{14}^{(5)} = \min \begin{cases} D_{14}^{(4)} & = 4 \\ D_{15}^{(4)} + D_{54}^{(4)} & = -4+6 \end{cases} = 2$	$\Pi_{14}^{(5)} = \begin{cases} \Pi_{14}^{(4)} & \text{if } D_{14}^{(4)} \le D_{15}^{(4)} + D_{54}^{(4)} \\ \Pi_{54}^{(4)} & \text{if } D_{14}^{(4)} > D_{15}^{(4)} + D_{54}^{(4)} \end{cases} = 5$