

# CSC373 Worksheet 4 Solution

August 7, 2020

1. • Calculating out-degree

Let  $G = (V, E)$  be a directed graph. Let  $[v_1, \dots, v_n]$  be a list of vertices in graph  $G$ .

I need to calculate the outdegree of every vertex using adjacency list.

We know that in addition to counting each  $v_i$  in adjacency list where  $i = 1, \dots, n$ , we are also counting  $|Adj[v_i]|$  edges.

Since there are  $|V| = n$  many vertices, we can write that the total count is  $|V| + \sum_{i=1}^n |Adj[v_i]| = |V| + |E|$ , which is  $\mathcal{O}(|V| + |E|)$ .

• Calculating In-degree

The outdegree of a vertex is indegree of another vertex.

Using this fact, we can conclude the running time of computing indegree of every vertex is  $\mathcal{O}(|V| + |E|)$ .

Notes:

• **Vertex**

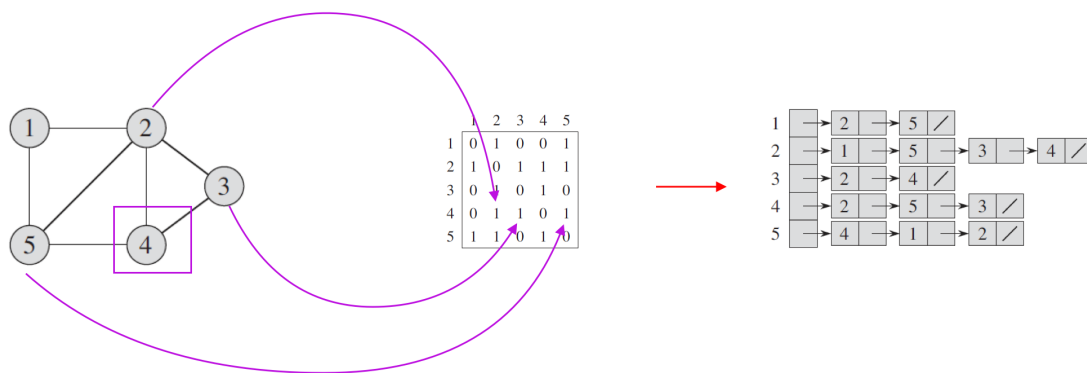
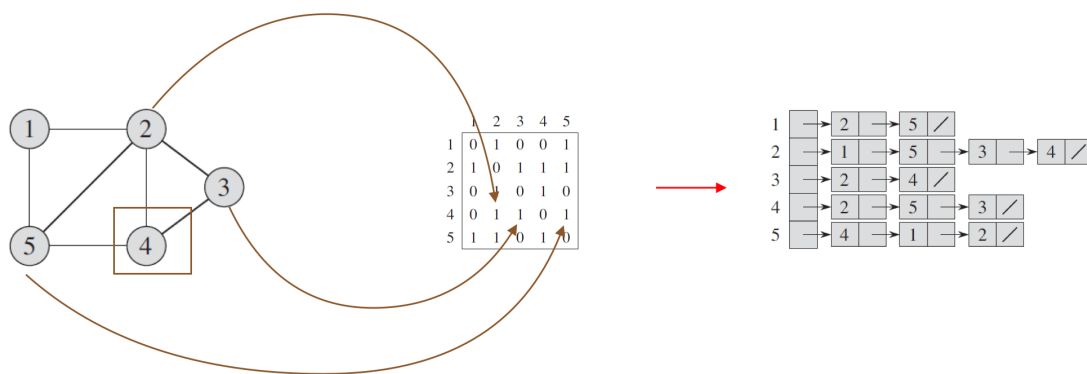
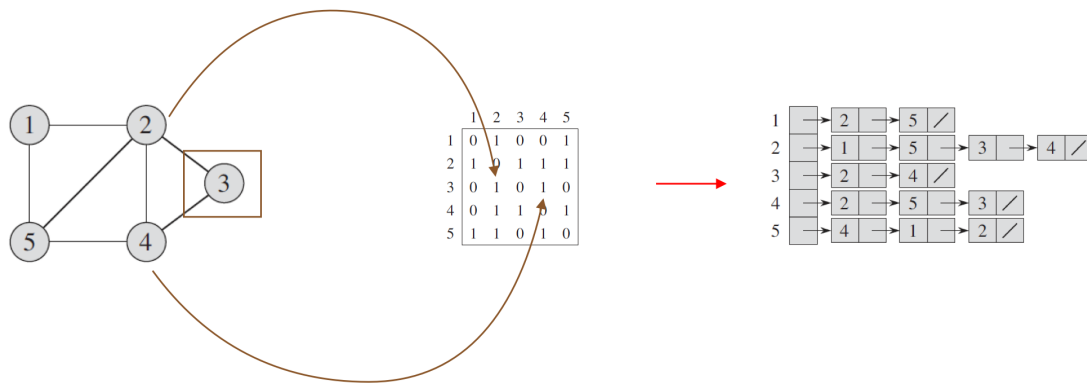
- Is a fundamental unit of which graphs are formed
- Also means node



### • Adjacency-list Representation

- Associates each vertex in a graph with the collection of its neighbouring vertices or edges
- Is represented by  $Adj[v]$ 
  - \* Means all vertices that are neighbour to vertex  $v$
  - \* In a directed graph,  $Adj[v]$  are all out-degree vertices of vertex  $v$
  - \*  $|Adj[v]|$  means the total number of outdegree of vertex  $v$







### • Directed graph

- Is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them



### • Out-degrees

- For a directed graph  $G = (V(G), E(G))$  and a vertex  $x_1 \in V(G)$ , the Out-Degree of  $x_1$  refers to the number of arcs incident from  $x_1$ . That is, the number of arcs directed away from the vertex  $x_1$ .



### • In-degrees

- For a directed graph  $G = (V(G), E(G))$  and a vertex  $x_1 \in V(G)$ , the In-Degree of  $x_1$  refers to the number of arcs incident to  $x_1$ . That is, the number of arcs directed towards the vertex  $x_1$ .



- Computing the outdegree of every vertex using adjacency list



3)



$$(v_1 + v_2 + v_3) + (e_1 + e_2 + e_3 + e_4 + e_5)$$

3)



$$(v_1 + v_2 + v_3) + (e_1 + e_2 + e_3 + e_4 + e_5)$$

4)



$$(v_1 + v_2 + v_3 + v_4) + (e_1 + e_2 + e_3 + e_4 + e_5)$$

4)



$$(v_1 + v_2 + v_3 + v_4) + (e_1 + e_2 + e_3 + e_4 + e_5)$$

6)



$$(v_1 + v_2 + v_3 + v_4 + v_5 + v_6) + (e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 + e_8)$$

So it has  $\mathcal{O}(V + E)$

- Computing the outdegree of every vertex using adjacency list

The outdegree of a vertex is indegree of another vertex.

Using this fact, we can conclude the running time of computing indegree of every vertex is  $\mathcal{O}(V + E)$ .

- Computing  $G^T$  from  $G$  in Adjacency List



```

1  COMPUTE-G-TRANSPOSE-ADJ-MATRIX( $Adj, V$ )
2    Let  $Adj'$  be a new adjacency list containing keys  $v_i \dots v_n$ 
3
4    for  $i = 1$  to  $|V|$ 
5      for every vertex  $w$  in  $Adj[v_i]$ 
6        Insert( $Adj'[w], v_i$ )
7
8    return  $Adj'$ 
9

```

- Computing  $G^T$  from  $G$  in Adjacency-Matrix





```

1  COMPUTE-G-TRANSPOSE-ADJ-MATRIX(A,V)
2      Let A'[1..|V|, 1..|V|] be a new adjacency matrix
3
4      for i = 1 to |V|
5          for j = 1 to |V|
6              A'[j,i] = A[i,j]
7
8      return A'
9

```

### Correct Solution:

- Computing  $G^T$  from  $G$  in Adjacency List



```

1  COMPUTE-G-TRANSPOSE-ADJ-MATRIX(Adj,V)
2      Let Adj' be a new adjacency list containing keys
3      v1...vn
4
5      for i = 1 to |V|
6          for every vertex w in Adj[vi]
7              Insert(Adj'[w], vi)
8
9      return Adj'

```

The running time is  $\mathcal{O}(|V| + |E|)$

- Computing  $G^T$  from  $G$  in Adjacency-Matrix



```

1  COMPUTE-G-TRANSPOSE-ADJ-MATRIX(A,V)
2      Let A'[1..|V|, 1..|V|] be a new adjacency matrix
3
4      for i = 1 to |V|
5          for j = 1 to |V|
6              A'[j,i] = A[i,j]
7
8      return A'
9

```

The running time is  $\mathcal{O}(|V|^2)$

```

31 Breadth-First-Search(V, v_i)
32     d = 0
33     for each v_i ∈ V
34         while performing BFS(V, v_i)
35             let w be the current node in BFS
36             if δ(v_i, w) > d
37                 d = δ(v_i, w)
38
39     return d
10

```

### Finding Runtime of Algorithm

Since the graph iterates  $\sum_{i=1}^n Adj[v_i] = |E|$  times for each  $v_i \in V$ , the algorithm iterates total of  $|V| \cdot |E|$  times, which is  $\mathcal{O}(|V||E|)$ .

### Notes:

- **Breadth First Search**

- Is an algorithm for searching or traversing a graph
- Is one of the simplest algorithm

- **Largest of All Shortest Path Distance**

- Means the shortest distance between two furthest apart nodes

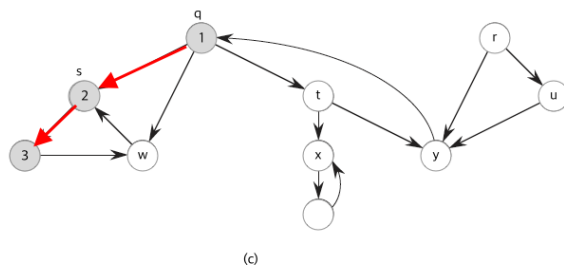
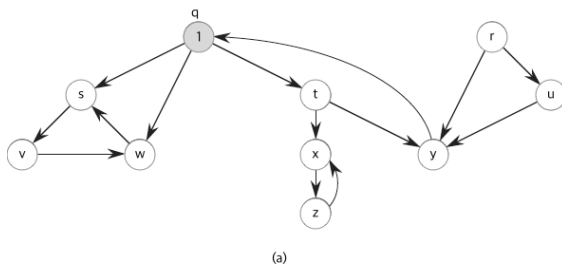


OR



## References

1) McGill University, 308-360 Tutorial, [link](#)



4.



(e)



(f)



(g)



(g)



(h)



(h)



(i)



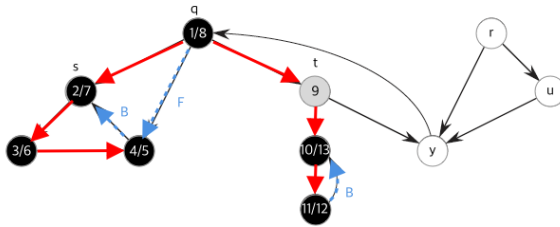
(i)



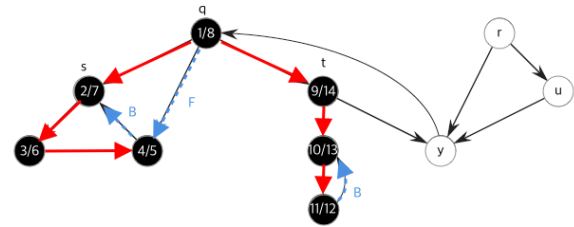
(m)



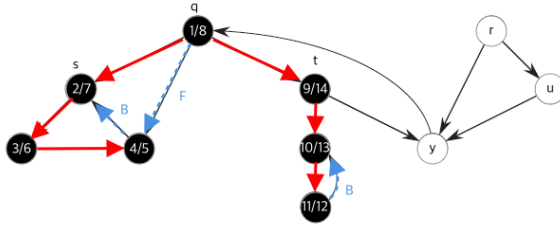
(n)



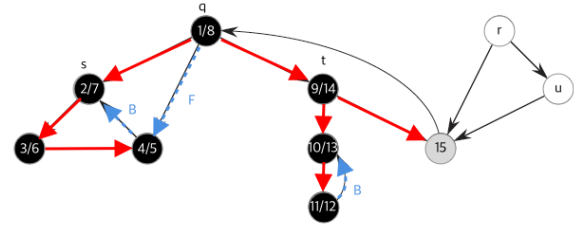
(o)



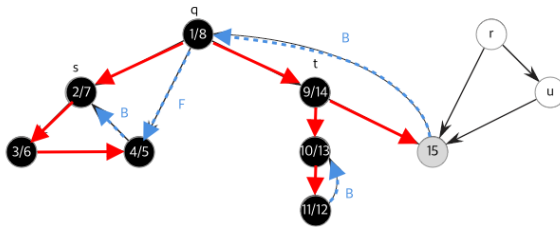
(p)



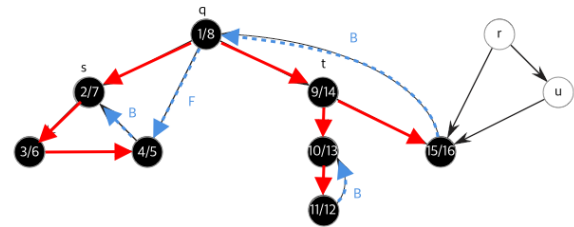
(r)



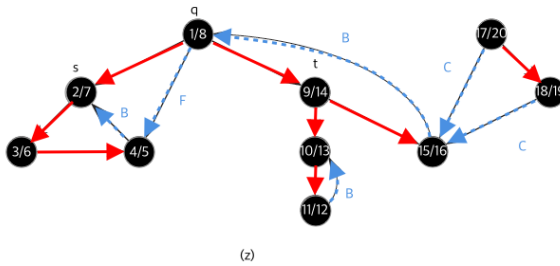
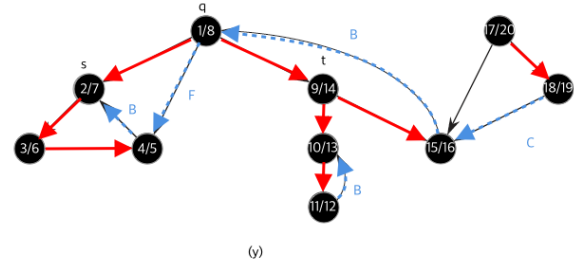
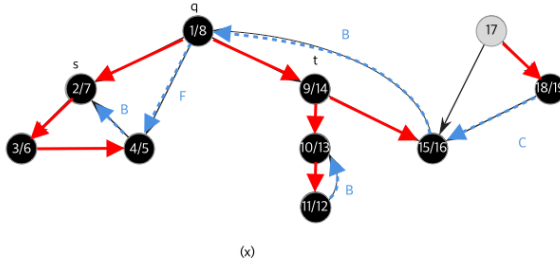
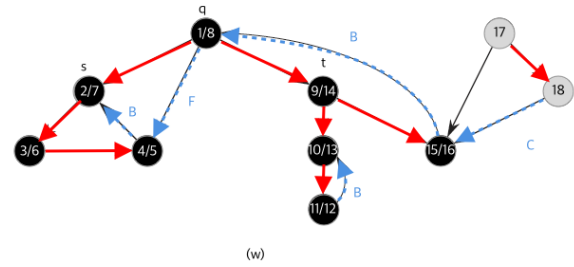
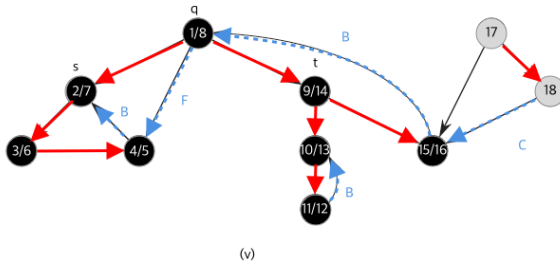
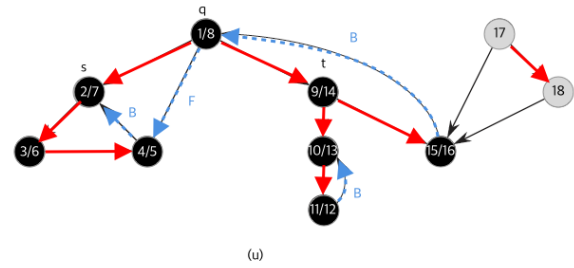
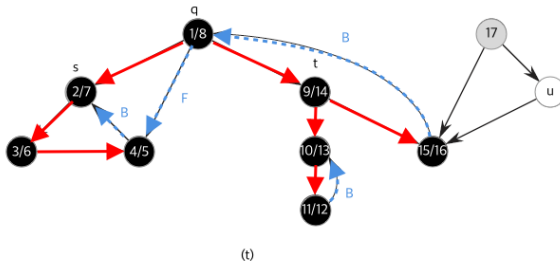
(q)



(s)



(t)



### Notes:

- **Depth First Search**

- Searches deeper in the graph whenever possible

- **Forward Edge**

- Is an edge  $(u, v)$  such that  $v$  is descendant but not part of the DFS tree. Edge  $1 \rightarrow 8$  is a forward edge



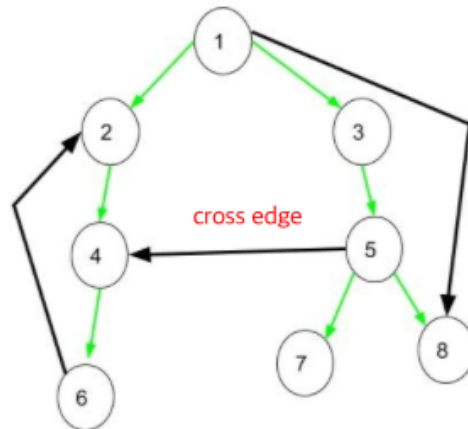
- **Back Edge**

- It is an edge  $(u, v)$  such that  $v$  is ancestor of edge  $u$  but not part of DFS tree. Edge from  $6 \rightarrow 2$  is a back edge.
- Indicates a cycle in a graph



- **Cross Edge**

- It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them. Edge from node  $5 \rightarrow 4$  is cross edge.



### References

- 1) Geeks For Geeks, Tree, Back, Edge and Cross Edges in DFS of Graph, link
5. *Proof.* Let  $G$  be a connected graph. Let  $A$  be a subset of  $E$  that is always included in some minimum spanning tree for  $G$ .

Assume for the sake of contradiction that  $(u, v)$  is not contained in some minimum spanning tree of  $G$ .

We know from the minimum-spanning tree algorithm that a light edge crossings in a cut of  $G$  that respects  $A$  is always chosen (since this is a safe edge and the algorithm always picks the safe edge).

Since the edge  $(u, v)$  is not a part of minimum spanning tree,  $(u, v)$  is not chosen in a cut.

Since  $(u, v)$  is not chosen, the edge  $(u', v')$  with smaller weight is chosen.

Then this violates the assumption that  $(u, v)$  is the edge with the smallest weight.

Thus, by contradiction,  $(u, v)$  belongs to some minimum spanning tree of  $G$ .

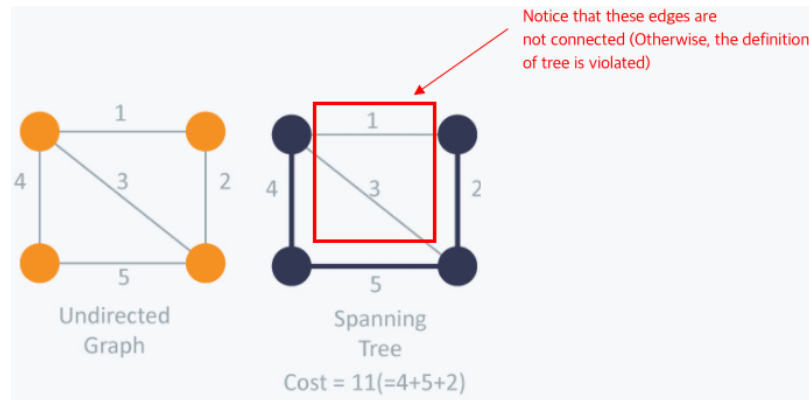
□

### Notes:



## • Spanning Tree

Given an undirected and connected graph  $G = (V, E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ )



## • Minimum Spanning Tree

- Is the spanning tree where the cost is minimum among all the spanning trees.
  - \* The cost of the spanning tree is the sum of the weights of all the edges in the tree.
- There can be many minimum spanning trees.



- Is used in
  1. Network design (Telephone, electrical, hydraulic, TV cable, computer, road)
  2. Approximation algorithm for NP-hard problems
  3. Learning salient features for real-time face verification
  4. Reducing data storage in sequencing amino acids in a protein

1)



2)



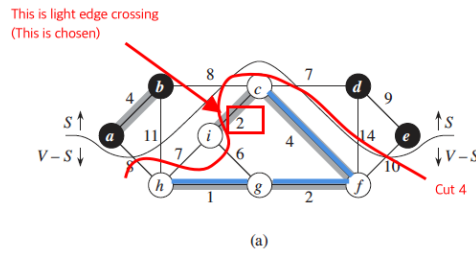
3)



4)



5)

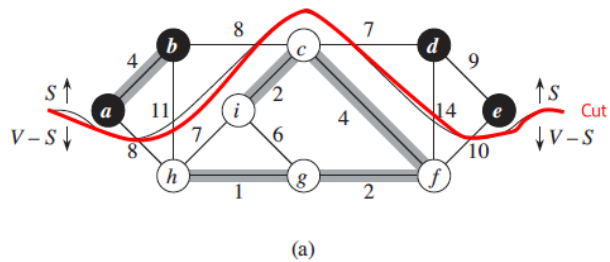


6)



### • Cut

- A cut of an undirected graph  $G = (V, E)$  is denoted  $(S, V - S)$
- Is a partition of  $V$



### • Light Edge Crossing

- An edge is a light edge crossing if its weight is the minimum of any edge crossing the cut



- **Safe Edge**

- Is an edge  $(u, v)$  that may be added to  $A$  without violating the invariant that  $A \cup (u, v)$  is a subset of some minimum spanning tree.

- **Cut respects a set  $A$  of edges**

- Means no edges in  $A$  crosses the crossing cut



### References:

- 1) Princeton University, Minimum Spanning Tree, [link](#)
- 2) McGill University, 308-360 Tutorial, [link](#)
- 3) Hacker Earth, Minimum Spanning Tree, [link](#)

```

6.1  MST-PRIM-ADJACENCY-LIST(Adj, w, r)
      Let Q be an empty list
      Q = Extract node objects from Adj
      for i = 1 to |Q|

```

```

6      u.key =  $\infty$ 
7      u. $\pi$  = NIL
8
9      r.key = 0
10
11     while Q  $\neq$  0
12         u = EXTRACT-MIN(Q)
13         for each v  $\in$  G.Adj[u]
14             if v  $\in$  Q and w(u,v) < v.key
15                 v. $\pi$  = u
16                 v.key = w(u,v)
17

```

### Notes:

- Kruskal Algorithm

- is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

- Prim's Algorithm

- Is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph
- Is very similar to Dijkstra's algorithm for finding shortest path





### References:

- 1) Wikipedia, Kruskal Algorithm, link

```

71  BELLMAN-FORD( $G, w, s$ )
72      INITIALIZE-SINGLE-SOURCE( $G, s$ )
73      active = true
74
75      while(active)
76          for each edge  $(u, v) \in G.E$ 
77              RELAX( $u, v, w$ )
78
79              if  $v.d > u.d + w(u, v)$ 

```

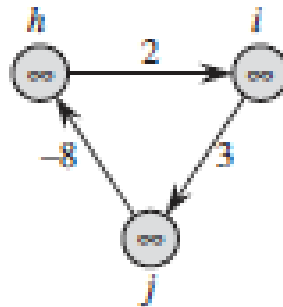
```

10         return False
11
12     return True
13
14
15     RELAX(u,v,w)
16         if v.d > u.d + w(u,v)
17             v.d = u.d + w(u,v)
18             v.π = u
19         else
20             active = false
21

```

### Notes:

- $m$  here represents the total number of edges in shortest path between  $u$  and  $v$
- **Negative-Weight Cycle**
  - Is a cycle with weights that sum to a negative number



- **Bellman-Ford Algorithm**

- Solves the single-source shortest-paths problem in the general case in which edge weights may be negative.



- Returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source

```

81  BELLMAN-FORD(G,w,s)
82      INITIALIZE-SINGLE-SOURCE(G,s)
83      for i = 1 to |G.V| - 1

```

```

4       for each edge  $(u,v) \in G.E$ 
5           RELAX( $u,v,w$ )
6
7       for each edge  $(u,v) \in G.E$ 
8           if  $v.d > u.d + w(u,v)$ 
9               return False
10
11       return True
12
13  RELAX( $u,v,w$ )
14      if  $v.d > u.d + w(u,v)$ 
15           $v.d = u.d + w(u,v)$ 
16           $v.\pi = u$ 
17      else
18           $v.d = -\infty$ 
19

```

9. I need to design algorithm that returns the most reliable path between two vertices.

```

1  DIJKSTRA( $G,w,s$ )
2      INITIALIZE-SINGLE-SOURCE( $G,s$ )
3       $S = \emptyset$ 
4       $Q = G.V$ 
5
6      while  $Q \neq \emptyset$ 
7           $u = \text{EXTRACT-MAX}(Q)$ 
8           $S = S \cup \{u\}$ 
9          for each vertex  $v \in G.\text{Adj}[u]$ 
10              RELAX( $u,v,w$ )
11
12  INITIALIZE-SINGLE-SOURCE( $G,s$ )
13      for each vertex  $v \in G.V$ 
14           $v.d = -\infty$ 
15           $v.\pi = \text{NIL}$ 
16       $s.d = 1$ 
17
18  RELAX( $u,v,w$ )
19      if  $u.d * w(u,v) > v.d$ 
20           $v.d = u.d * w(u,v)$ 
21           $v.\pi = u$ 
22

```

### Notes:

- Dijkstra Algorithm
  - Finds shortest path from one node to all nodes
  - Solves the single-source shortest-paths problem on a weighted directed graph  $G = (V, E)$

- Cannot have negative weight edges



10. The output of the Floyd-Warshall algorithm can detect the presence of a negative weight cycle by looking for a negative value in the matrix  $D^{(n)}$ .

### Notes:

- Floyd-Warshall Algorithm
  - Is an algorithm for finding shortest paths between all pairs of vertices in a weighted graph.
  - Negative edges allowed
  - Cannot have negative weight cycles
  - Is useful in graphs with dense number of edges

```

FLOYD-WARSHALL( $W$ )
1   $n = W.rows$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4    let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5    for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7         $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 

```

- \*  $n$ : the number of vertices
- \*  $W$ : the  $n \times n$  matrix of edge weights of an  $n$ -vertex directed graph  $G = (V, E)$

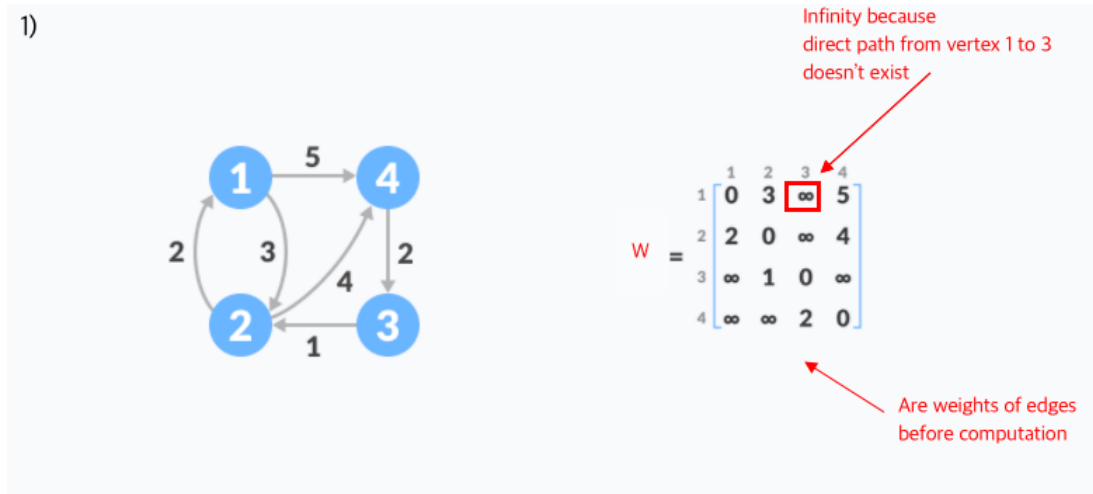
$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases} \quad (1)$$



- \*  $D^{(k)}$ : the  $n \times n$  matrix of edge weights of shortest path where all intermediate vertices are in the set  $1, 2, \dots, k$ .

$$d_{ij} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases} \quad (2)$$

### Example:



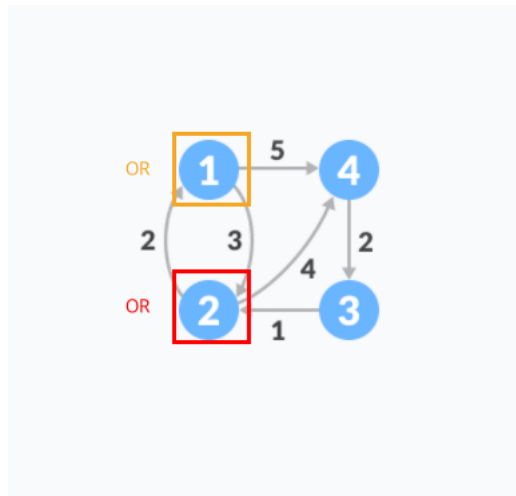
- \*  $k = 1$



$$D^1 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \quad (3)$$

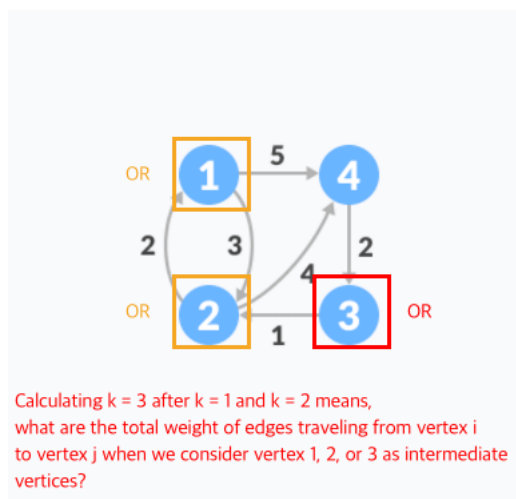
· Matrix calculated via line 3 to 7 in Floyd-Warshall algorithm

- \*  $k = 2$



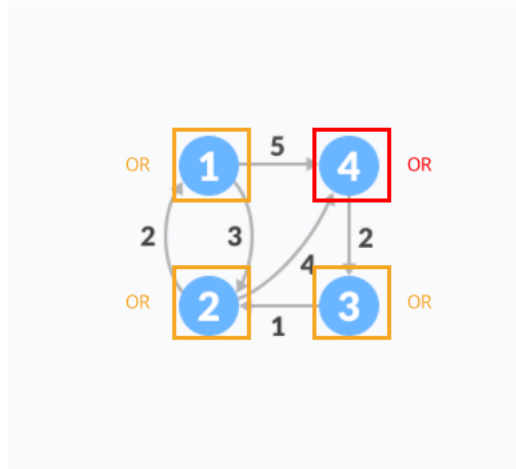
$$D^2 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \quad (4)$$

- 5 in  $D_{34}^2$  is the only value that's different
- \*  $k = 3$



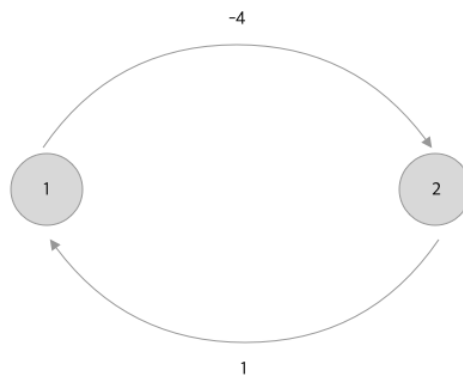
$$D^3 = \begin{bmatrix} 0 & 2 & \infty & 4 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & 5 \\ 7 & 3 & 2 & 0 \end{bmatrix} \quad (5)$$

- \*  $k = 4$



$$D^4 = \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 4 & 4 \\ 12 & 1 & 0 & 5 \\ 7 & 3 & 2 & 0 \end{bmatrix} \quad (6)$$

Example 2:



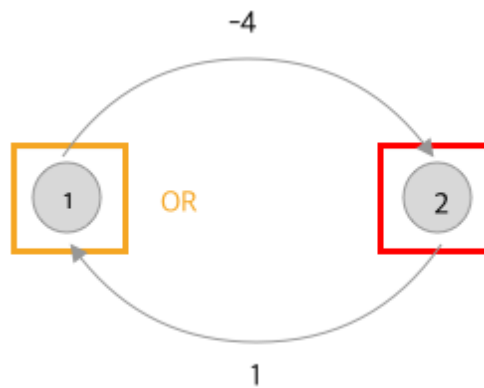
$$W = \begin{bmatrix} 0 & -4 \\ 1 & 0 \end{bmatrix} \quad (7)$$

\*  $k = 1$



$$W = \begin{bmatrix} 0 & -4 \\ 1 & 0 \end{bmatrix} \quad (8)$$

\*  $k = 2$



$$W = \begin{bmatrix} 0 & -4 \\ 1 & 0 \end{bmatrix} \quad (9)$$

# 11. Notes:

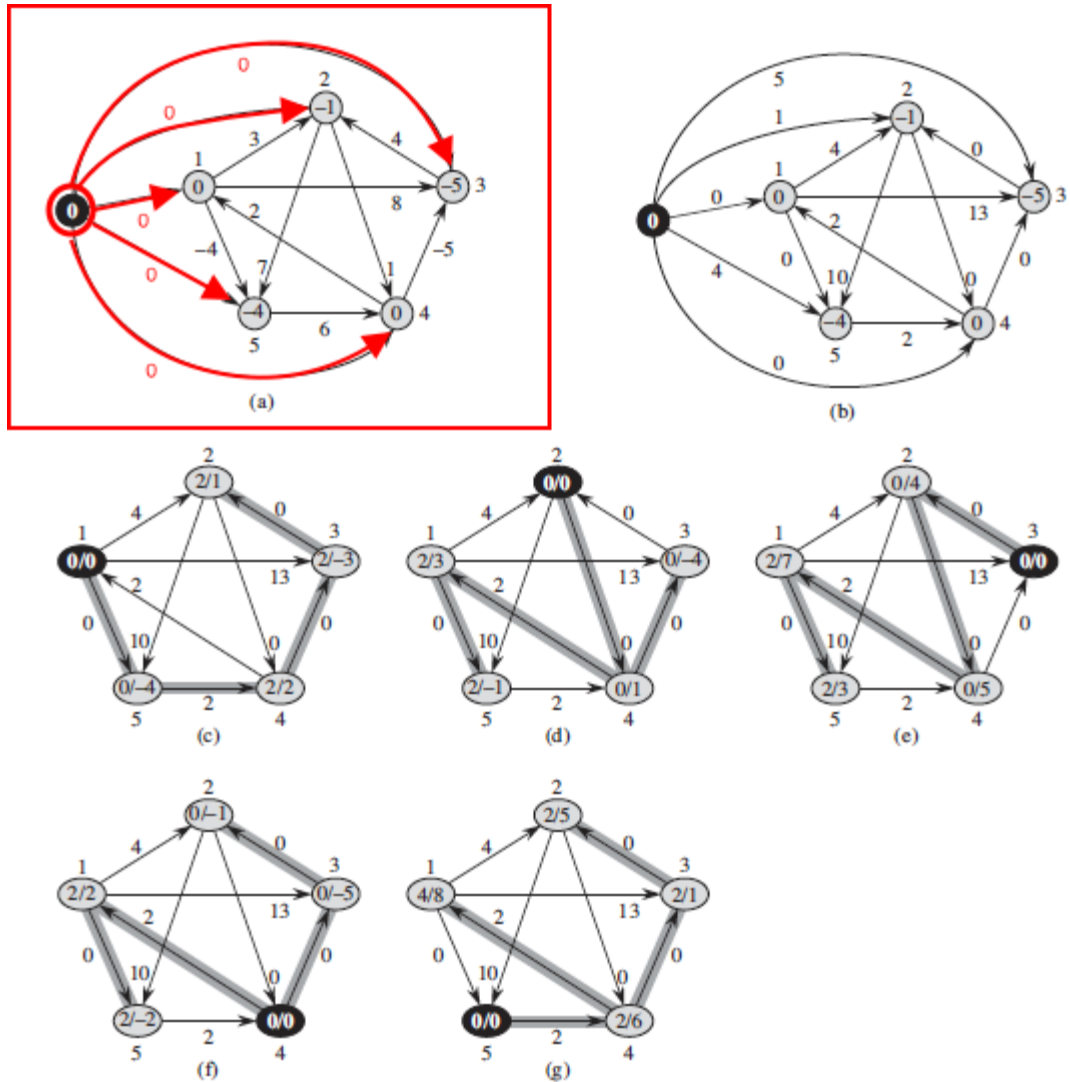
- Johnson's Algorithm

- Is a way to find the shortest paths between all pairs of vertices in an edge-weighted, directed graph
- Allows some edges to be negative-number
- No negative-cycles may exist
- Is similar to Floyd-Warshall Algorithm
- Is most effective in graph with sparse number of edges
- Works as a subroutine to Dijkstra and Bellman-Ford Algorithm

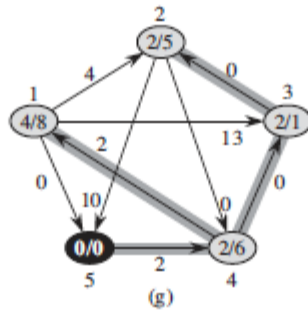
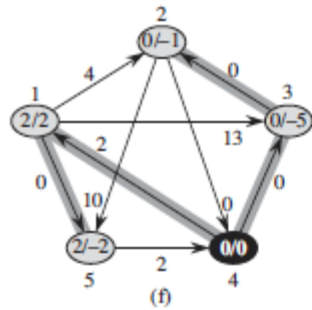
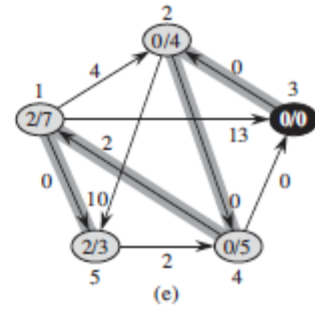
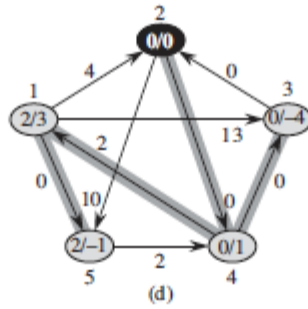
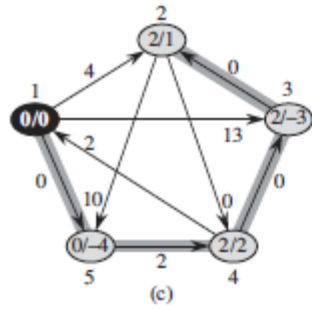
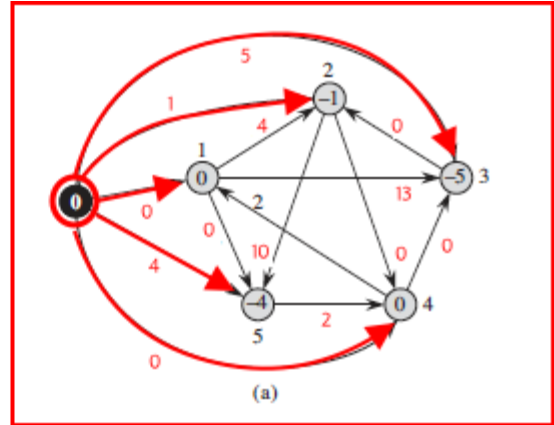
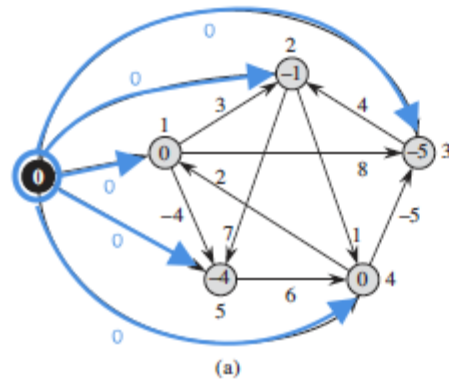
- Has running time of  $O(nm + n(m + n \log n)) = O(nm + n \log n)$
- Reweighting
  - Is a technique used in Johnson's Algorithm
  - Is so that all edges have non-negative weights <sup>[1]</sup>

### Steps:

1. Create a source vertex  $V' = V \cup \{s\}$  for some  $s \notin V$  and  $E' = E \cup \{(s, v) : v \in V\}$
2. Extend the weight function  $w$  so that  $w(s, v) = 0$  for all  $v \in V$



3. Reweight each edge  $(u, v)$  with weight function  $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$



### Sample Calculations

$$\hat{w}(s, u_1) = w(s, u_1) + h(s) - h(u_1) \quad (1)$$

$$= 0 + 0 - 0 \quad (2)$$

$$= 0 \quad (3)$$

$$\hat{w}(s, u_2) = w(s, u_2) + h(s) - h(u_2) \quad (4)$$

$$= 0 + 0 - (-1) \quad (5)$$

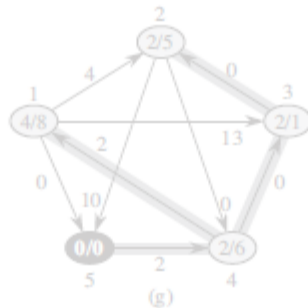
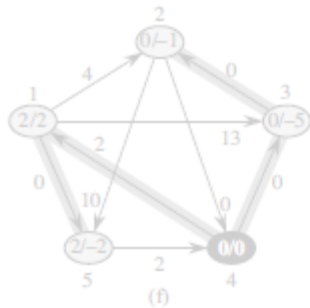
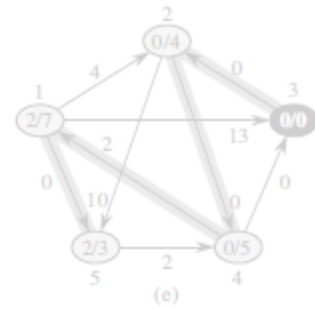
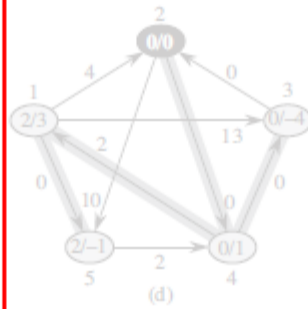
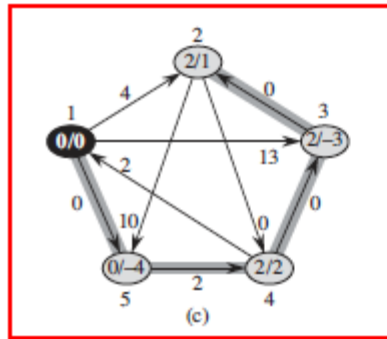
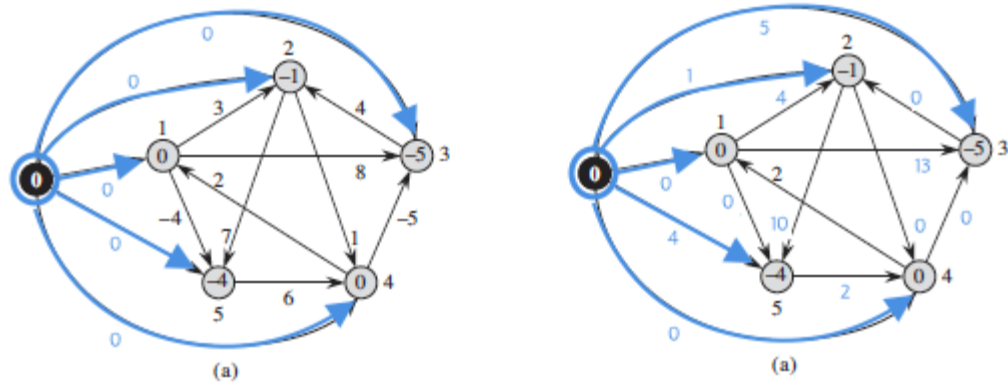
$$= 1 \quad (6)$$

$$\hat{w}(u_1, u_5) = w(u_1, u_5) + h(u_1) - h(u_5) \quad (7)$$

$$= 0 + 0 - (-4) \quad (8)$$

$$= 4 \quad (9)$$

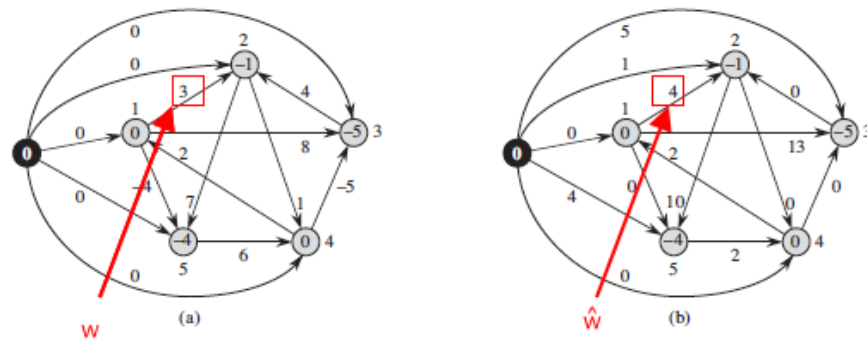
4. Remove vertex  $s$  and run Dijkstra's algorithm on every node in the graph



– Apply Dijkstra's algorithm to calculate  $\delta(\hat{u}, v)$

NOTE!!

- \*  $\delta$  means shortest-path weights ( $v.d$ ) derived from weight function  $w$
- \*  $\hat{\delta}$  means shortest-path weights ( $v.d$ ) derived from weight function  $\hat{w}$



- Convert each  $\delta(\hat{u}, v)$  to  $\delta(u, v)$

References

- 1) Columbia University, Johnson's Algorithm for All-Pairs Shortest Paths, link