CSC373 Worksheet 4 Solution

August 5, 2020

1.
   - Calculating out-degree

     Let $G = (V, E)$ be a directed graph. Let $[v_1, ..., v_n]$ 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 $i = 1, .., n$, 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 $\mathcal{O}(|V| + |E|)$.

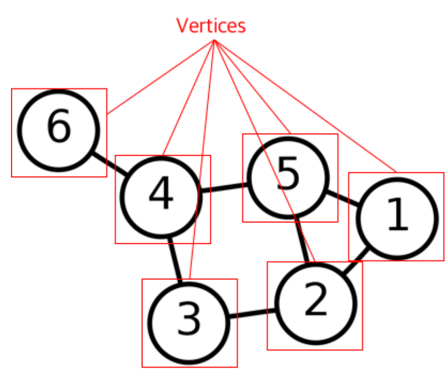
   - Calculating In-degree

     The outdegree of a vertex is indegree of another vertex.

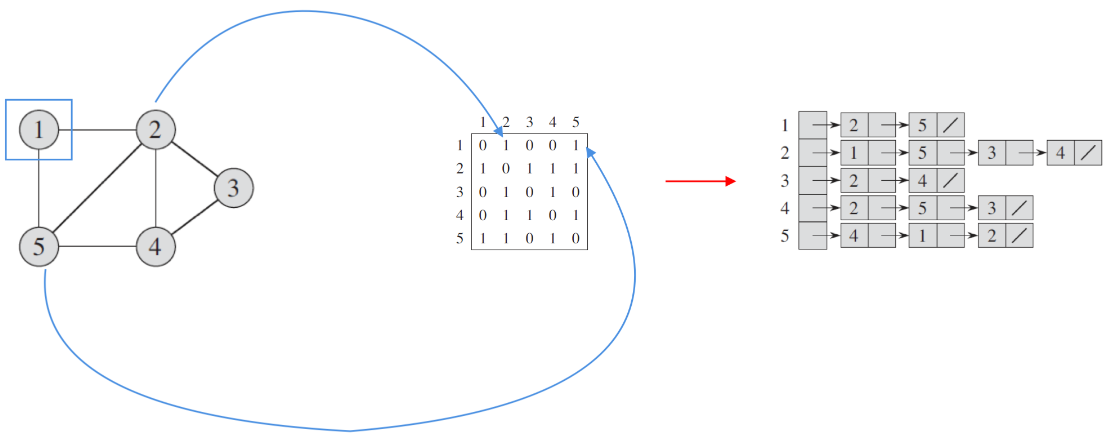
     Using this fact, we can conclude the running time of computing indegree of every vertax is $\mathcal{O}(|V| + |E|)$.

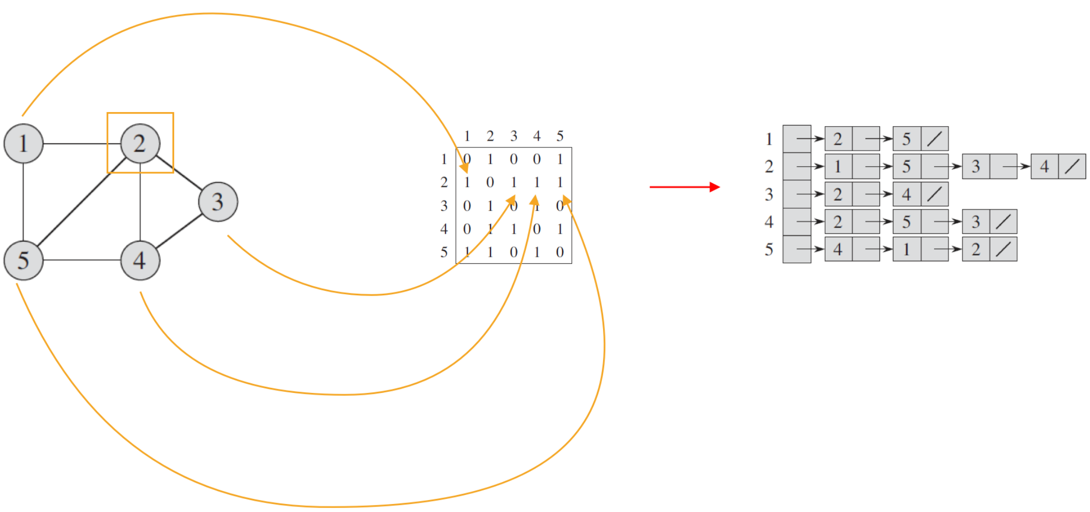
<u>Notes:</u>

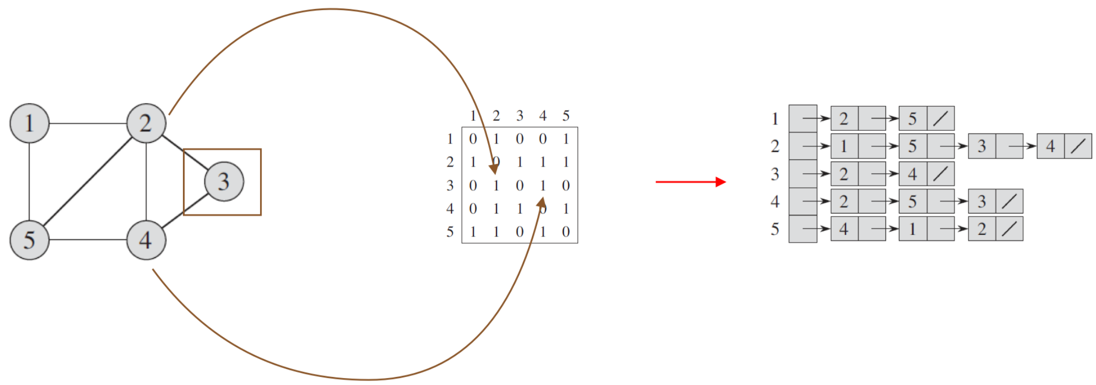
- **Vertex**
  - Is a fundamental unit of which graphs are formed
  - Also means node

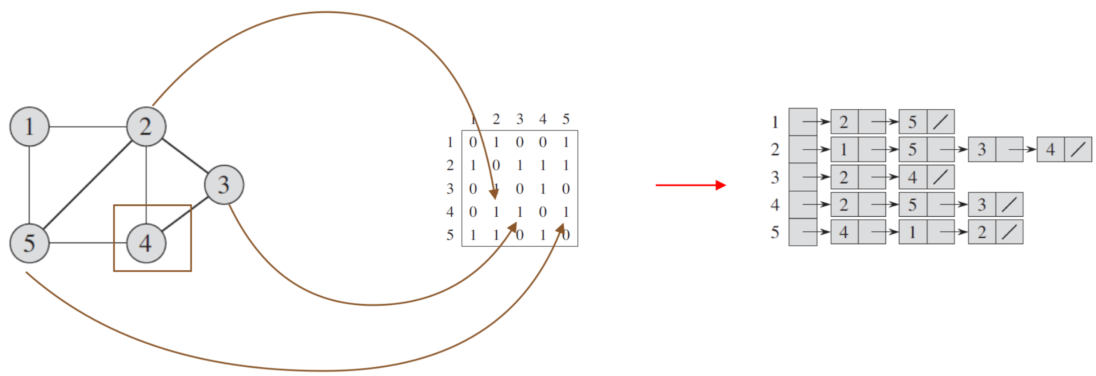


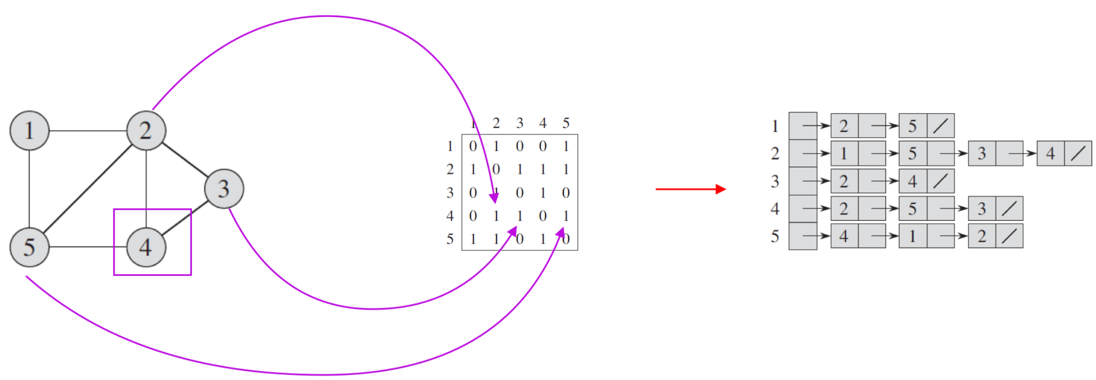
- **Adjacency-list Representation**
  - Associates each vertex in a graph with the collection of its neighbouring vertices or edges
  - Is represented by $Adj[v]$
    - Means all vertices that are neighbour to vertex v
    - In a directed graph, $Adj[v]$ are all out-degree vertices of vertax v
    -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

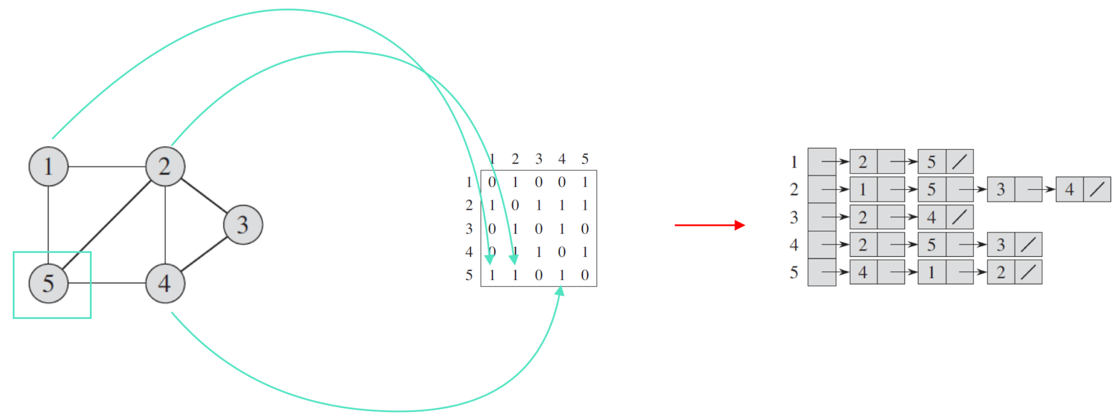




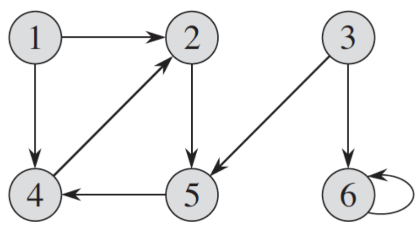

1 2 3 4 5
1 2 3 4 5
1 0 1 0 0 1
2 1 0 1 1 1
3 0 1 0 1 0
4 0 1 1 0 1
5 1 1 0 1 0
1 2 5
2 1 5 3 4
3 2 4
4 2 5 3
5 4 1 2


1 2 3 4 5
1 2 3 4 5
1 0 1 0 0 1
2 1 0 1 1 1
3 0 1 0 1 0
4 0 1 1 0 1
5 1 1 0 1 0
1 2 5
2 1 5 3 4
3 2 4
4 2 5 3
5 4 1 2

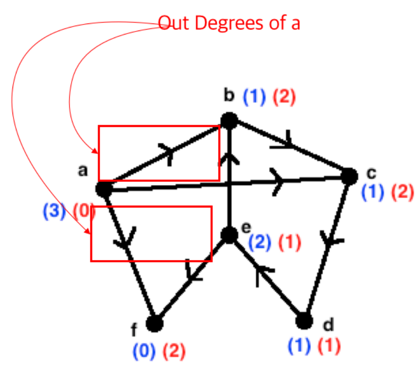

1 2 3 4 5
1 2 3 4 5
1 0 1 0 0 1
2 1 0 1 1 1
3 0 1 0 1 0
4 0 1 1 0 1
5 1 1 0 1 0
1 2 5
2 1 5 3 4
3 2 4
4 2 5 3
5 4 1 2



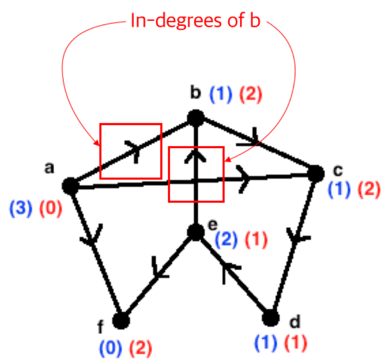
- **Directed graph**
  - Is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them



- **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 <ins>away</ins> from the vertex x_1.

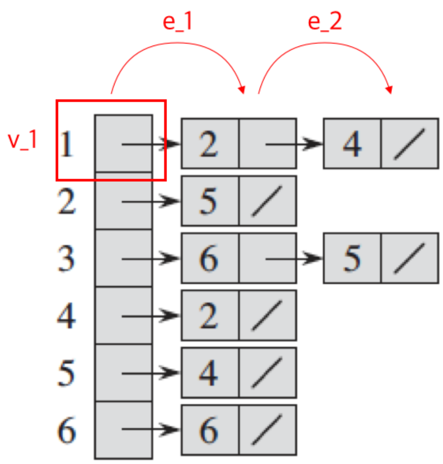


- **In-degrees**
  - For a directed graph $G = (V(G), E(G))$ and a vertex $x_1 \in V(G)$, the In-Degree of x_1 refers to the number of arcs incident to x_1. That is, the number of arcs directed <ins>towards</ins> the vertex x_1.



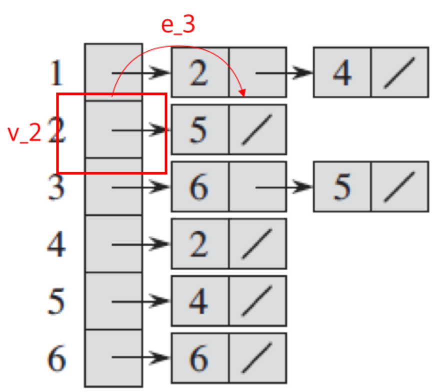
- Computing the outdegree of every vertex using adjacency list

1)



$(v_1) + (e_1 + e_2)$

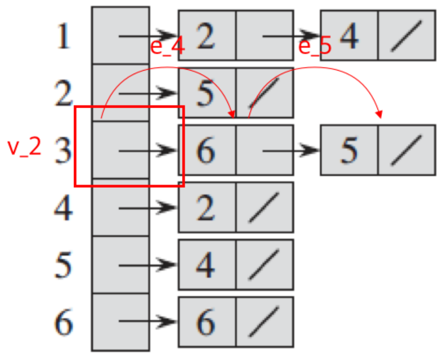
2)



$(v_1 + v_2) +$

$(e_1 + e_2 + e_3)$

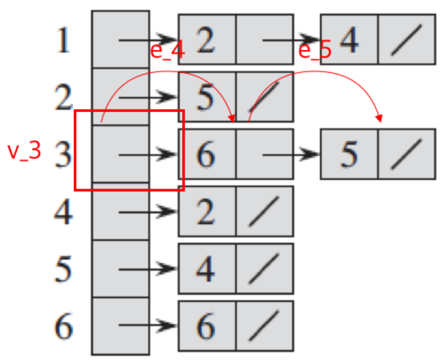
3)



$(v_1 + v_2 + v3) \quad +$

$(e_1 + e_2 + e_3 + e_4 + e_5)$

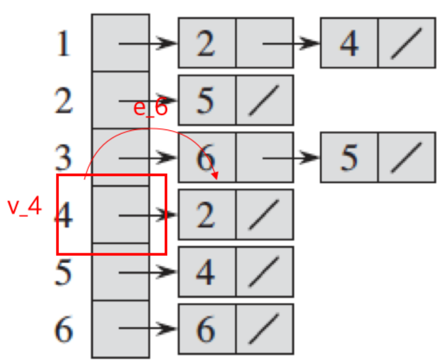
3)



$(v_1 + v_2 + v3) \quad +$

$(e_1 + e_2 + e_3 + e_4 + e_5)$

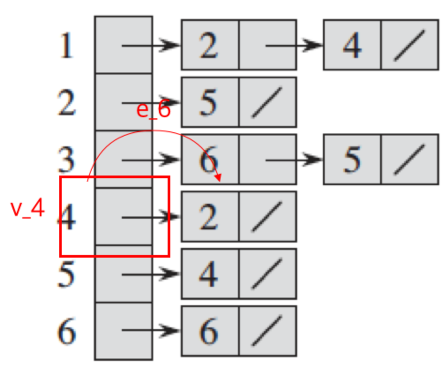
4)



$(v_1 + v_2 + v_3 + v_4) \; +$

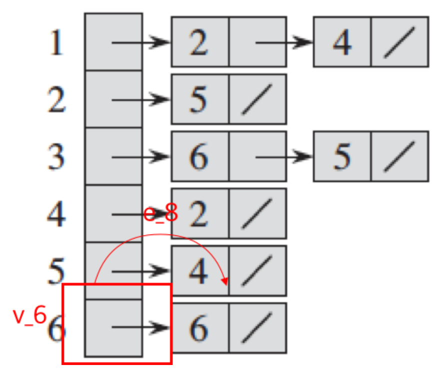
$(e_1 + e_2 + e_3 + e_4 + e_5)$

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)

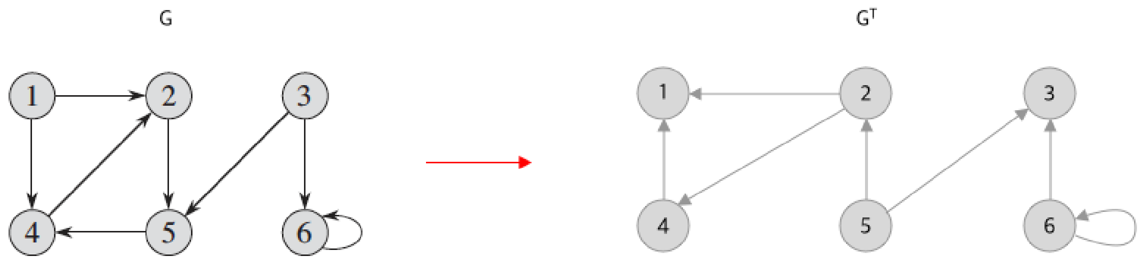
So it has $\mathcal{O}(V+E)$

- 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 vertax is $\mathcal{O}(V+E)$.

2. • Computing G^T from G in Adjacency List

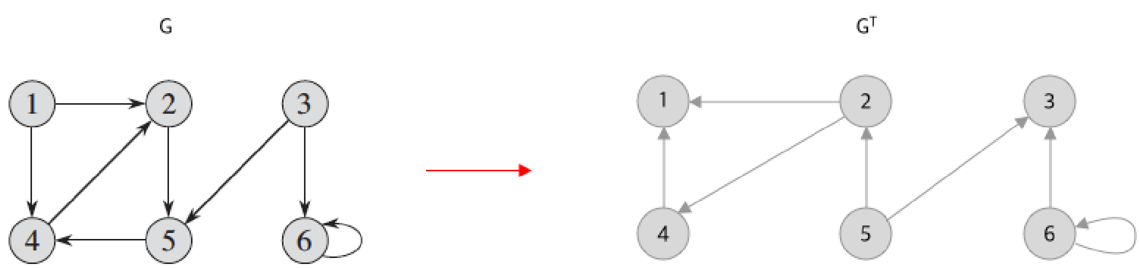


```
COMPUTE-G-TRANSPOSE-ADJ-MATRIX(Adj,V)
    Let Adj' be a new adjacency list containing keys v_i...v_n

    for i = 1 to |V|
        for every vertex w in Adj[v_i]
            Insert(Adj'[w], v_i)

    return Adj'
```

- Computing G^T from G in Adjacency-Matrix



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	1	1	0	0	0
6	0	0	1	0	0	1

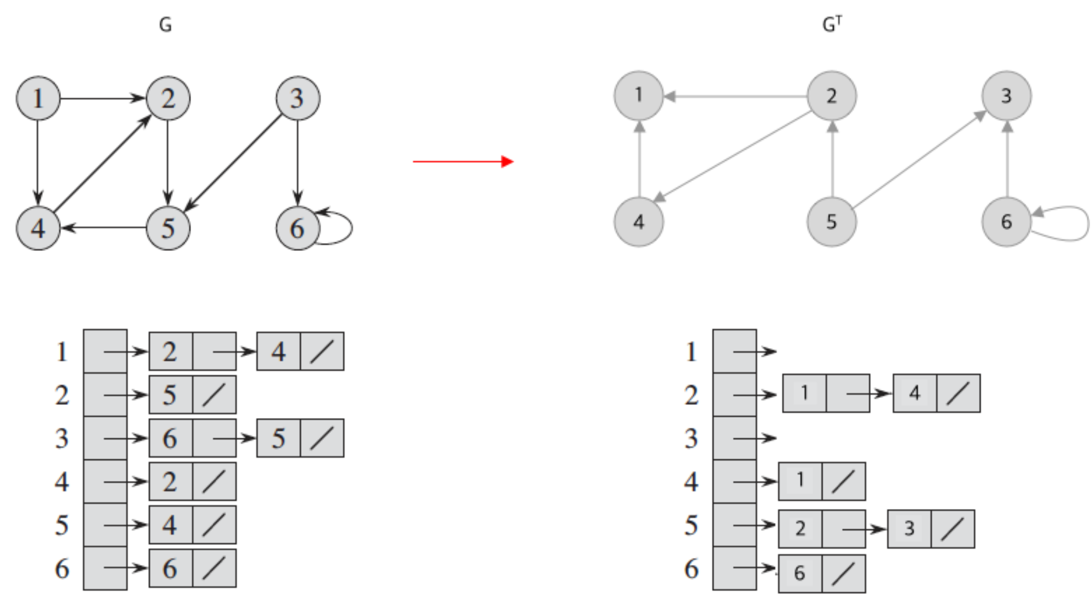
```
COMPUTE-G-TRANSPOSE-ADJ-MATRIX(A,V)
    Let A'[1..|V|, 1..|V|] be a new adjacency matrix

    for i = 1 to |V|
        for j = 1 to |V|
            A'[j,i] = A[i,j]

    return A'
```

Correct Solution:

- Computing G^T from G in Adjacency List



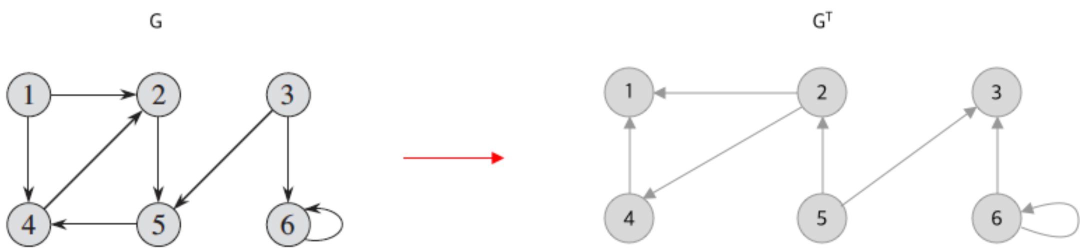

```
COMPUTE-G-TRANSPOSE-ADJ-MATRIX(Adj,V)
    Let Adj' be a new adjacency list containing keys
v_i...v_n

    for i = 1 to |V|
        for every vertax w in Adj[v_i]
            Insert(Adj'[w], v_i)

    return Adj'
```

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

- Computing G^T from G in Adjacency-Matrix



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	1	1	0	0	0
6	0	0	1	0	0	1

```
COMPUTE-G-TRANSPOSE-ADJ-MATRIX(A,V)
    Let A'[1..|V|, 1..|V|] be a new adjacency matrix

    for i = 1 to |V|
        for j = 1 to |V|
            A'[j,i] = A[i,j]

    return A'
```

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

3.

```
Breadth-First-Search(V,v_i)
    d = 0
    for each v_i ∈ V
        while performing BFS(V, v_i)
            let w be the current node in BFS
                if δ(v_i,w) > d
                    d = δ(v_i,w)

    return d
```

Finding Runtime of Algorithm

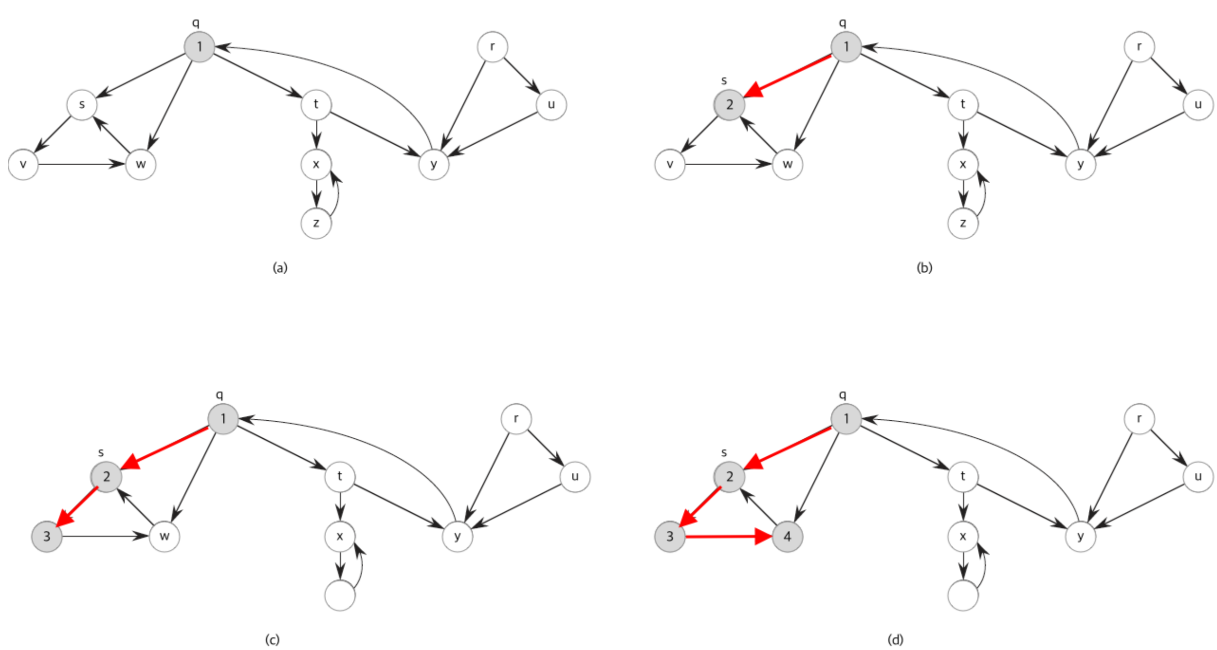
Since the graph iterates $\sum_{i=1}^{n} Adj[v_i] = |E|$ times for each $v_i \in V$, the algorithm iterates total of $|V| \cdot |E|$ times, which is $\mathcal{O}(|V||E|)$.

Notes:

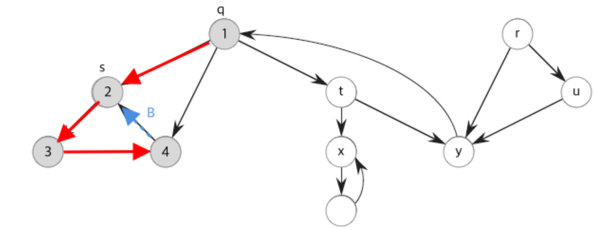
- **Breadth First Search**
  - Is an algorithm for searching or traversing a graph
  - Is one of the simplest algorithm
- **Largest of All Shortest Path Distance**
  - Means the shortest distance between two furthest apart nodes

References

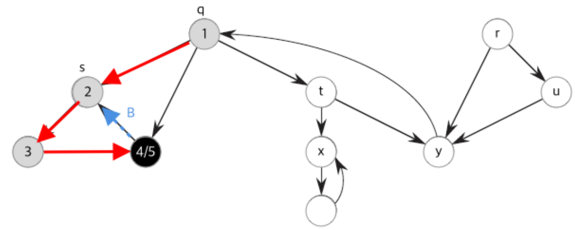
1) McGill University, 308-360 Tutorial, link



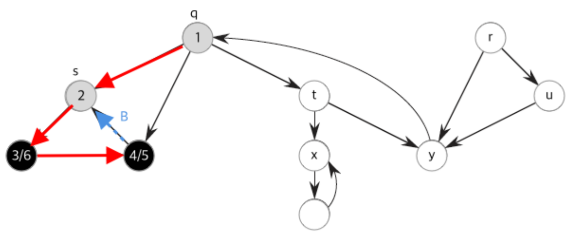
4.


q
1
s
2
B
3
4
t
x
y
r
u

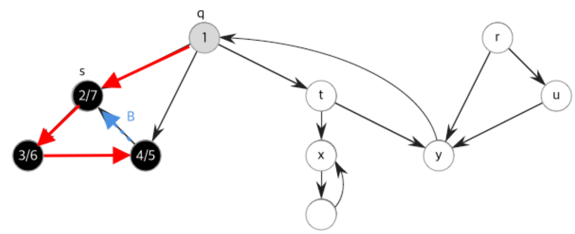
(e)


q
1
s
2
B
3
4/5
t
x
y
r
u

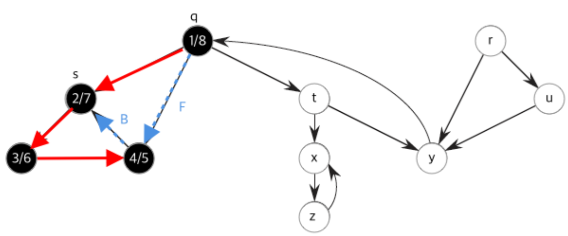
(f)


q
1
s
2
B
3/6
4/5
t
x
y
r
u

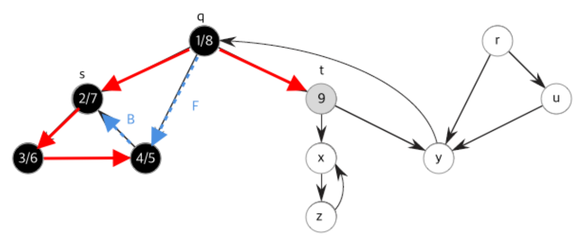
(g)


q
1
s
2/7
B
3/6
4/5
t
x
y
r
u

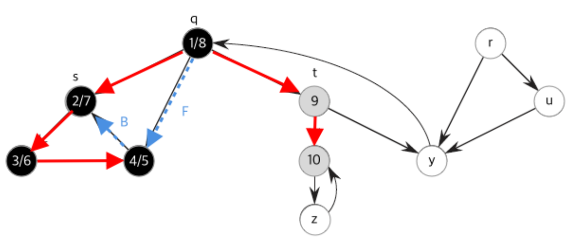
(g)


q
1/8
s
2/7
B
F
3/6
4/5
t
x
z
y
r
u

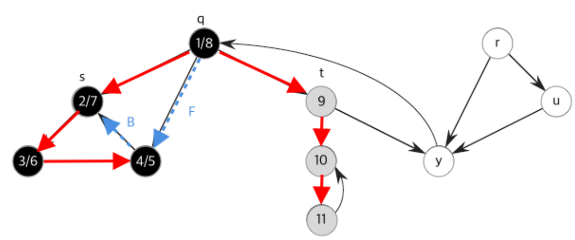
(i)


q
1/8
s
2/7
B
F
3/6
4/5
t
9
x
z
y
r
u

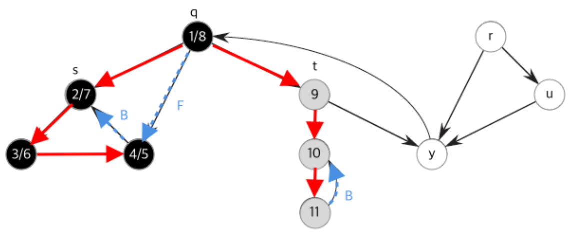
(j)


q
1/8
s
2/7
B
F
3/6
4/5
t
9
10
z
y
r
u

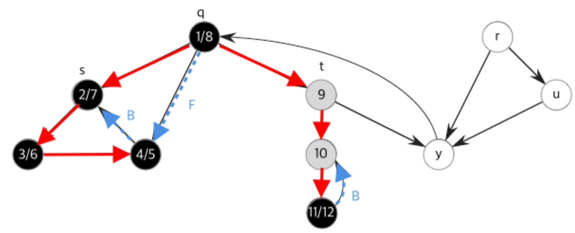
(k)


q
1/8
s
2/7
B
F
3/6
4/5
t
9
10
11
y
r
u

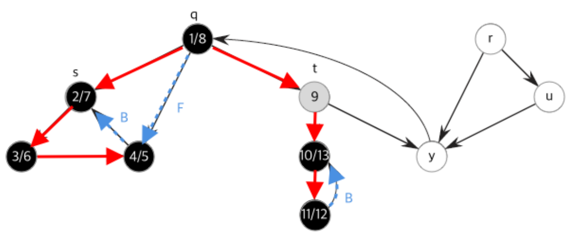
(l)


q
1/8
s
2/7
t
9
r
u
B
F
3/6
4/5
10
y
B
11

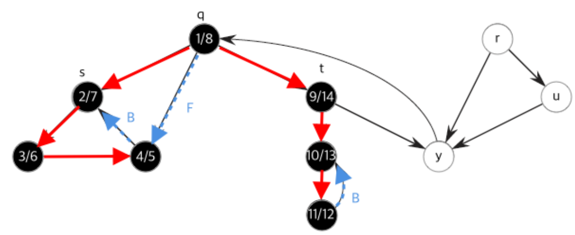
(m)


q
1/8
s
2/7
t
9
r
u
B
F
3/6
4/5
10
y
B
11/12

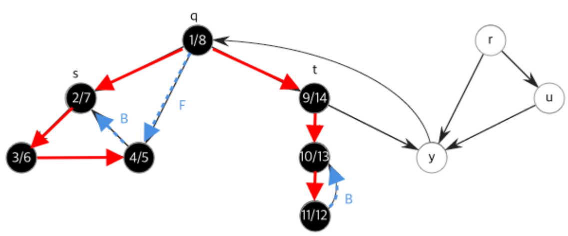
(n)


q
1/8
s
2/7
t
9
r
u
B
F
3/6
4/5
10/13
y
B
11/12

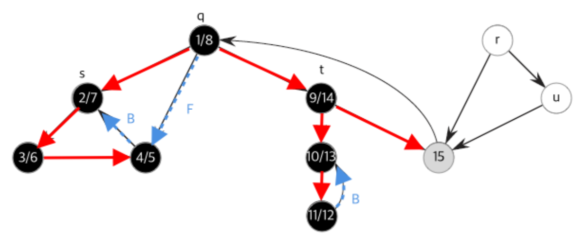
(o)


q
1/8
s
2/7
t
9/14
r
u
B
F
3/6
4/5
10/13
y
B
11/12

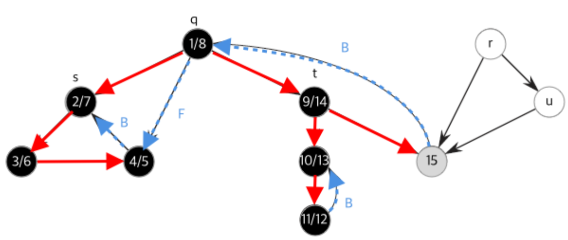
(p)


q
1/8
s
2/7
t
9/14
r
u
B
F
3/6
4/5
10/13
y
B
11/12

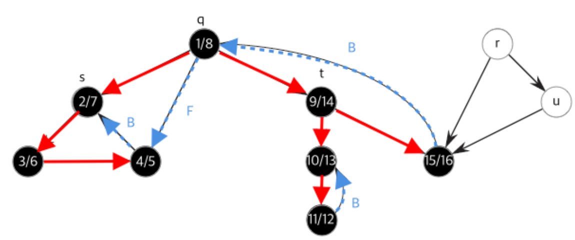
(p)


q
1/8
s
2/7
t
9/14
r
u
B
F
3/6
4/5
10/13
15
B
11/12

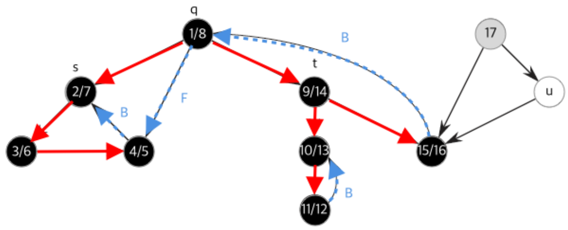
(q)


q
1/8
B
s
2/7
t
9/14
r
u
B
F
3/6
4/5
10/13
15
B
11/12

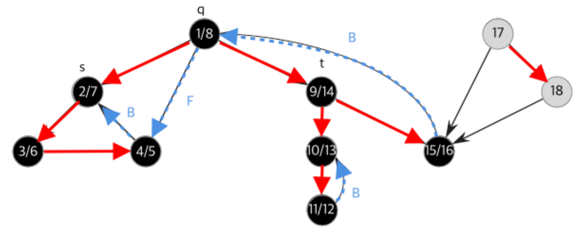
(r)


q
1/8
B
s
2/7
t
9/14
r
u
B
F
3/6
4/5
10/13
15/16
B
11/12

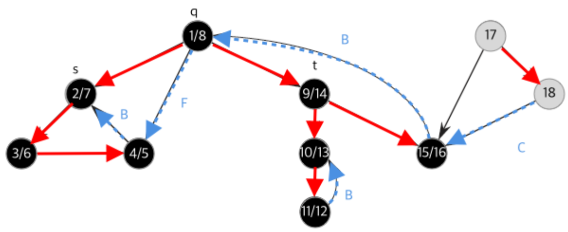
(s)



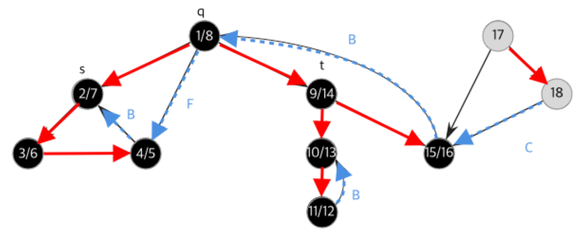
(t)



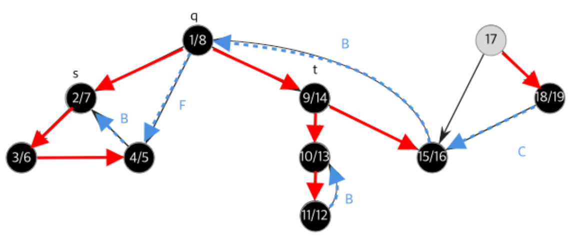
(u)



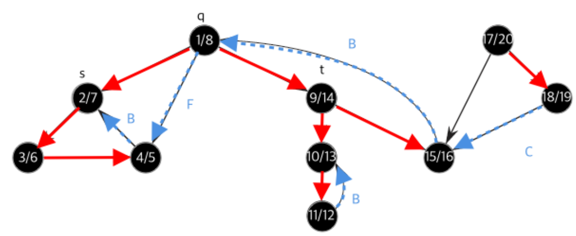
(v)



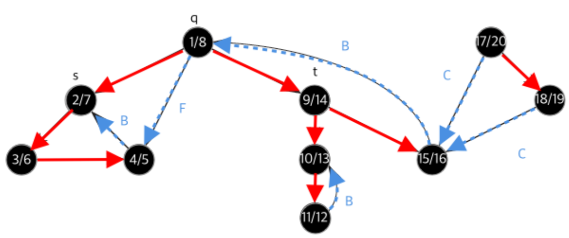
(w)



(x)



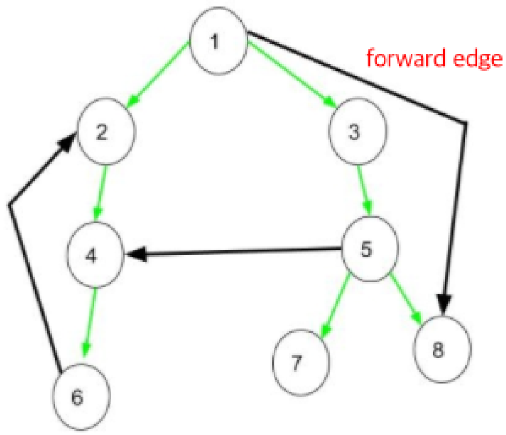
(y)



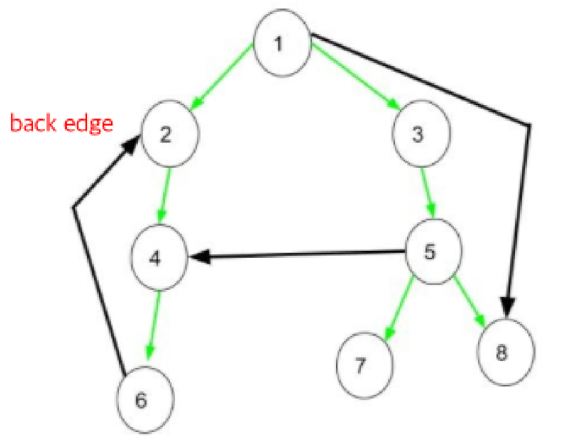
(z)

Notes:

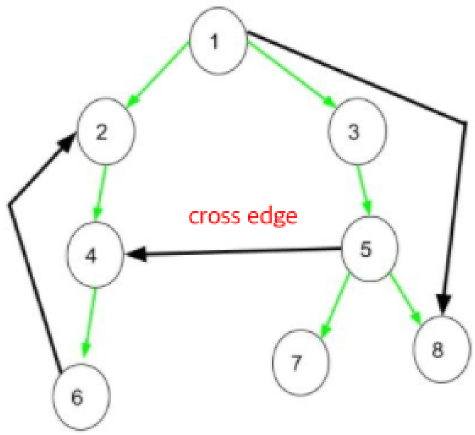
- **Depth First Search**
  - Searches deeper in the graph whenever possible
- **Forward Edge**
  - Is an edge (u, v) such that v is descendant but not part of the DFS tree. Edge $1 \rightarrow 8$ is a foward edge



- **Back Edge**
  - It is an edge (u, v) such that v is ancestor of edge u but not part of DFS tree. Edge from $6 \to 2$ 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 $5 \to 4$ 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 connectded graph. Let A be a subset of E that is always included in some minimum spanning treee 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.

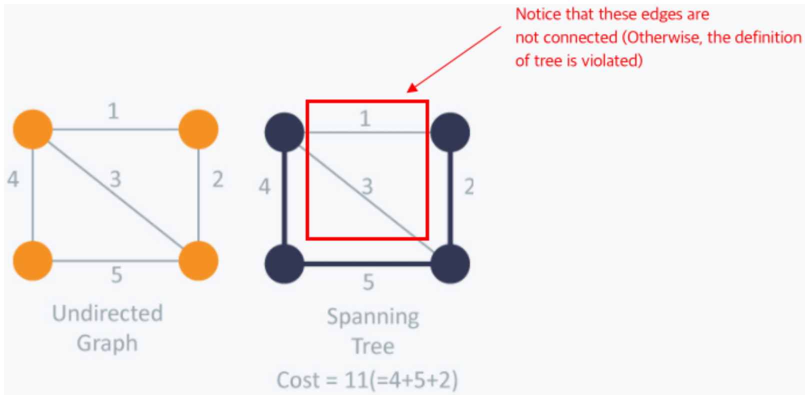
   Thus, by contradiction, (u, v) belongs to some minimum spanning tree of G.

   □

Notes:

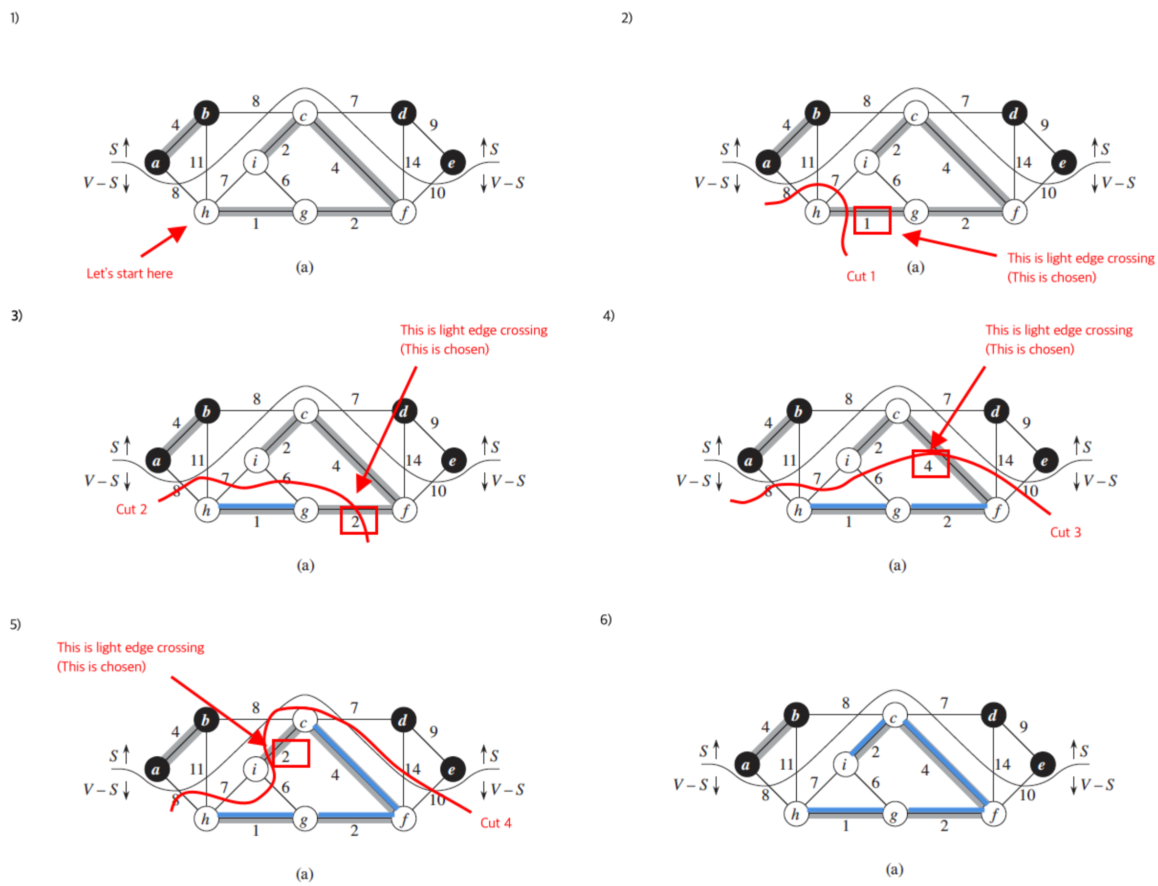
- **Spanning Tree**

  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)

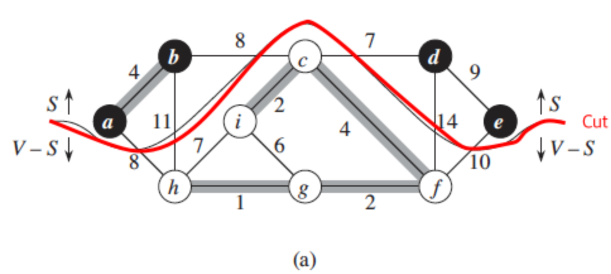


- **Minimum Spanning Tree**
  - Is the spanning tree where the <u>cost is minimum</u> among all the spanning trees.
    * The cost of the spanning tree is the sum of the weights of all the edges in the tree.
  - There can be many minimum spanning trees.

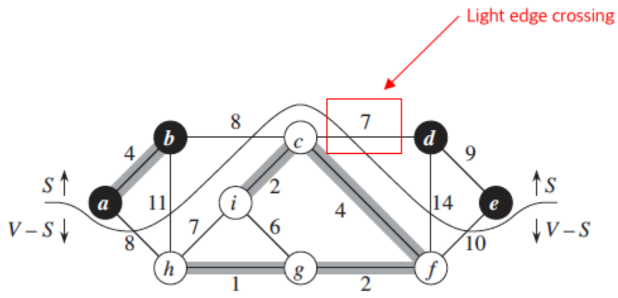
  - Is used in
    1. Network design (Telephone, electrical, hydraulic, TV cable, computer, road)
    2. Approximation algorithm for NP-hard problems
    3. Learning sailent features for real-time face verification
    4. Reducing data storage in sequencing amino acids in a protein



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

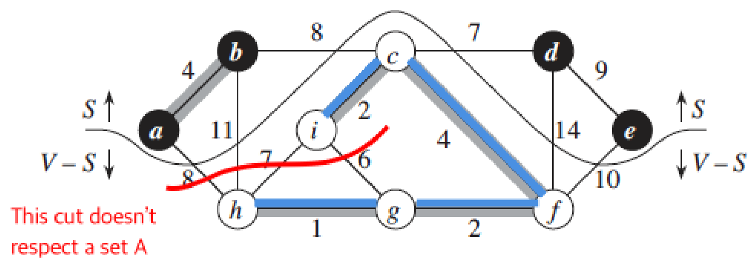


- **Light Edge Crossing**
  - An edge is a light edge crossing if its weight is the minimum of any edge crossing the cut



(a)

- **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



(a)

References:

1) Princeton University, Minimum Spanning Tree, link

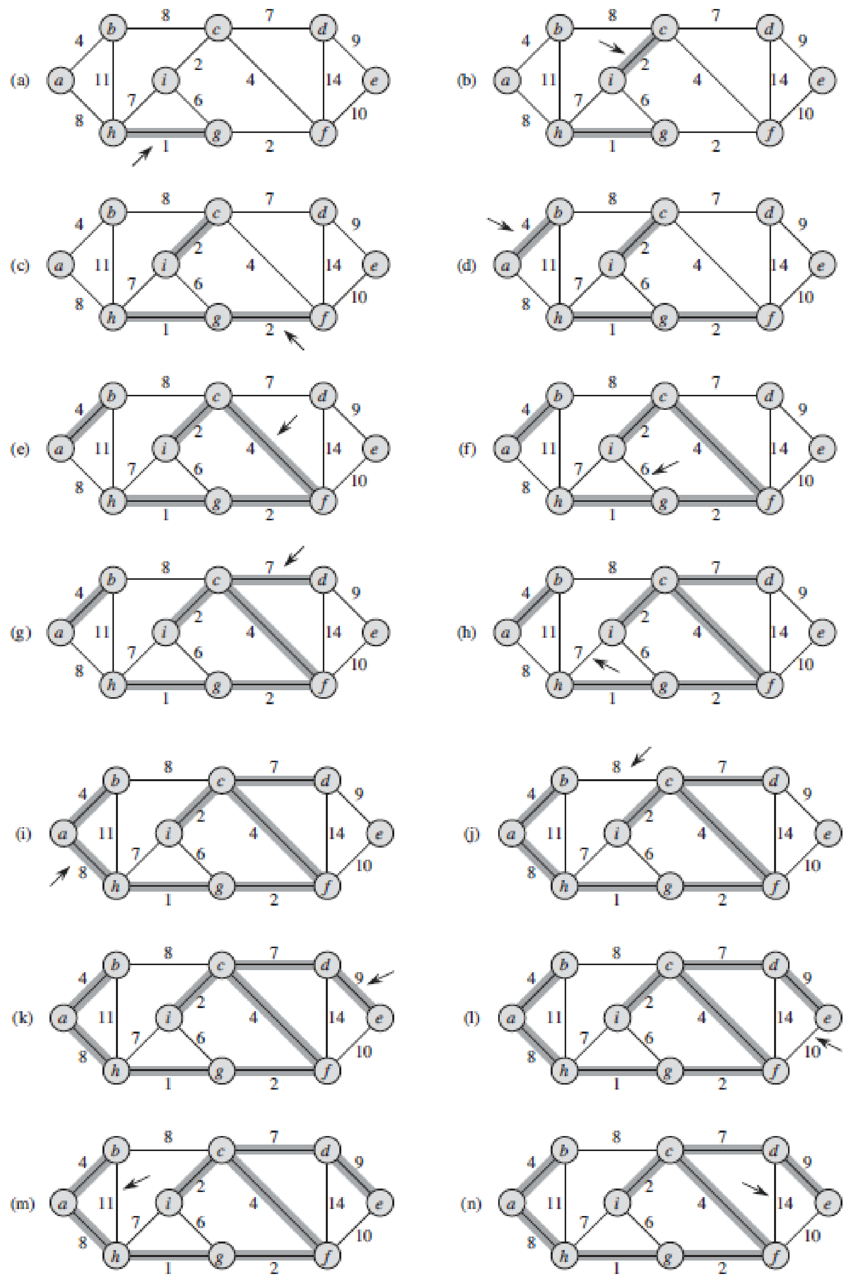
2) McGill University, 308-360 Tutorial, link

3) Hacker Earth, Minimum Spanning Tree, link

6. **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

Minimum weight edge connecting u to a vertax in the tree

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

References:

1) Wikipedia, Kruskal Algorithm, link