ICT 5102

Lecture 1

Graphs

Dr. Hossen A Mustafa

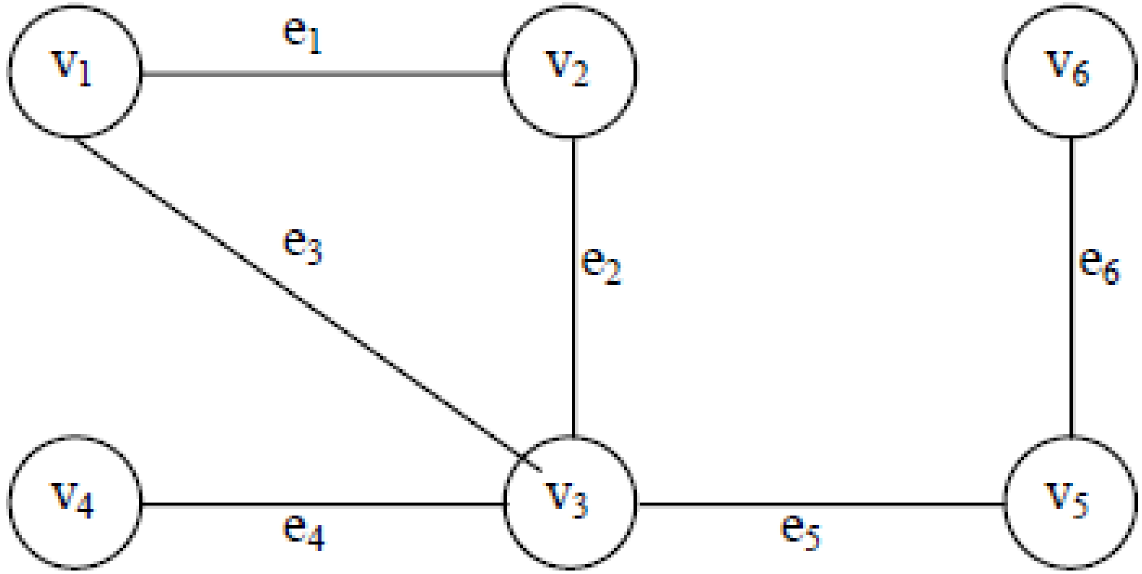
http://hossenmustafa.buet.ac.bd

https://learning.iict.buet.ac.bd

Graph

- A graph G consist of
  - Set of vertices V (called nodes), (V = {*v1, v2, v3, v4……}) and*
  - Set of edges E (*i.e., E {e1, e2, e3……cm}*
- A graph can be represents as G = (V, E), where
  - V is a finite and non empty set of vertices and
  - E is a set of pairs of vertices called edges.
  - Each edge 'e' *in E is identified with* a unique pair (*a, b) of nodes in V, denoted by e = [a, b].*

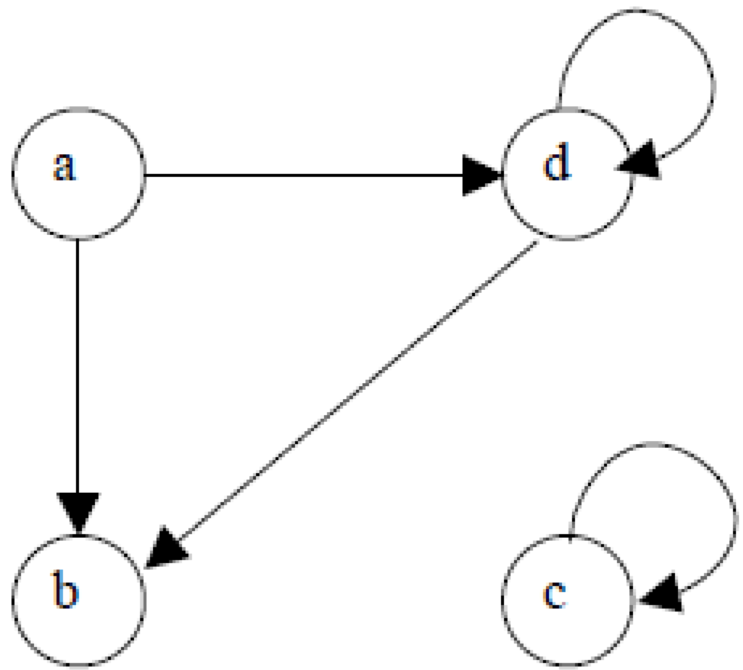
Graph Example



- V = {*v1, v2, v3, v4, v5, v6*}
- *E* = {*e1, e2, e3, e4, e5, e6*} OR *E* = {*(v1, v2) (v2, v3) (v1, v3) (v3, v4), (v3, v5) (v5, v6)*}.
- *This is an **undirected** graph*

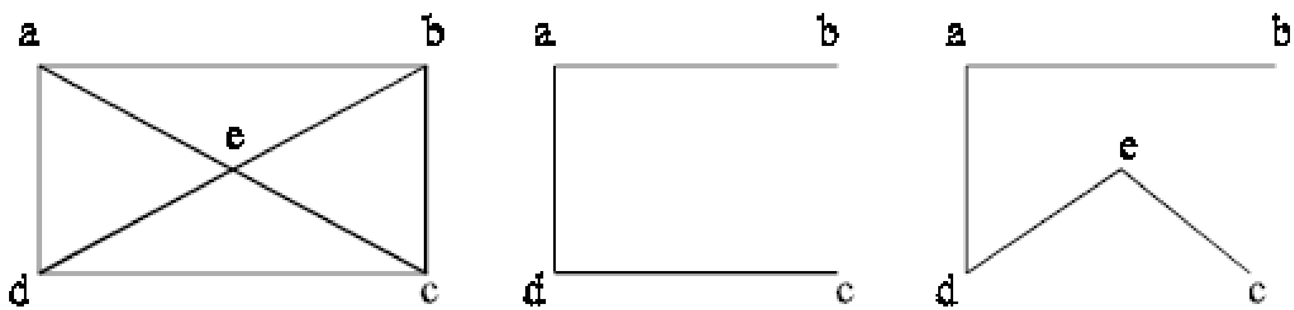
Directed Graph

- *A directed graph has direction for each edge*
- An edge (*a*, *b*) *is incident from a to b.*
  - *It means that we can go to b from a but b to a*
- *For bidirectional, there will be 2 edges, e.g., (a, b) and (b, a)*



Sub-Graph

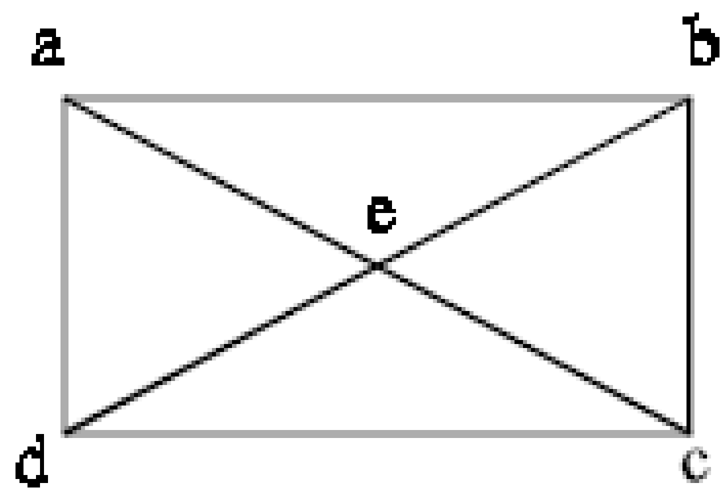
- A graph G1 = (V1, E1) is said to be a *sub-graph of G*
  - *if E1 is* a subset at E and V1 is a subset at V such that the edges in E1 are incident only with the vertices in V1.
  - (b) is a subgraph of (a)
- A sub-graph of G is a *spanning sub-graph if it contains all the vertices of G.*
  - *(c) shows* a spanning sub-graph of *(a)*



(a) (b) (c)

Degree

- *Degree is the* number of edges incident on a vertex.
  - *The degree of vertex a, is* written as degree (*a*).
  - *If the degree of vertex a is zero, then vertex a is called isolated vertex*
  - In figure
    - *degree(a) = 3*
    - *degree(e) = 4*

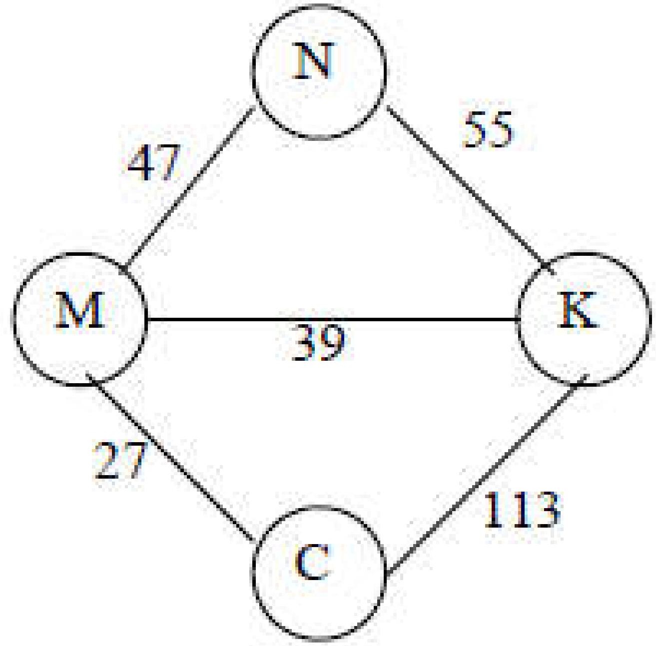


Weighted Graph

- A graph G is said to be *weighted graph if*
  - *every edge and/or vertices in the graph is* assigned with some weight or value.
- A weighted graph can be defined as G = (V, E, We, Wv) where
  - V is the set of vertices,
  - E is the set at edges
  - We *is a weights of the edges whose* domain is E
  - Wv *is a weight to the vertices whose domain is V.*

Weighted Graph

- In the figure
  - V = {N, K, M, C}
  - E = {(N, K), (N,M,), (M,K), (M,C), (K,C)}
  - We = {*55,47, 39, 27, 113*}
  - Wv = {*N, K, M, C*}



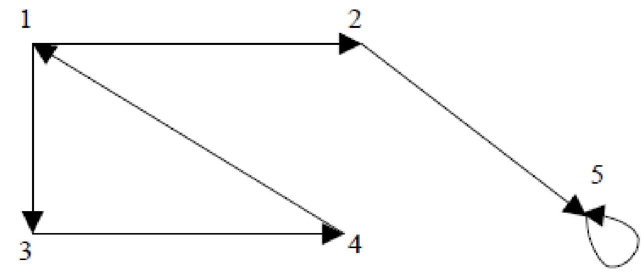
Definitions

- An undirected graph is said to be ***connected*** *if there exist a path from any vertex to* any other vertex. Otherwise, it is said to be ***disconnected*****.**
- graph G is said to **complete/fully connected/strongly connected** if there is a path from every vertex to every other vertex.
- A ***path*** *is a sequence of edges (e1, e2, e3, ... en) such that the* edges are connected with each other
  - *terminal vertex en can be reached with the initial* vertex e1

Graph Representation

- We need to represent a graph in formats so that programs can use it
- Several ways to represent a graph
  – Adjacency Matrix Representation

Adjacency Matrix Representation

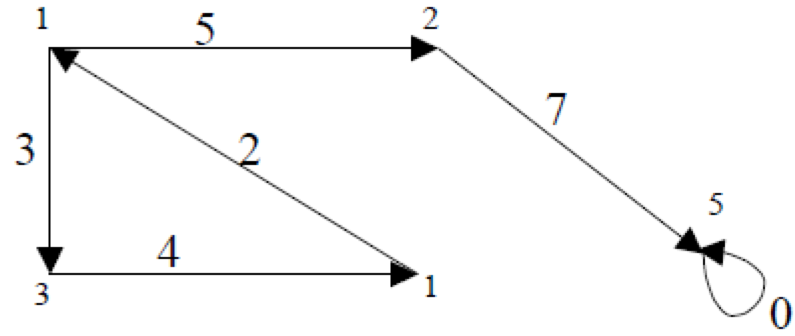


$A_{ij} = 1$ {if there is an edge from V_i to V_j or if the edge (i, j) is member of E.}

$A_{ij} = 0$ {if there is no edge from V_i to V_j}

$i \backslash j$	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	1

Weighted Graph in Adjacency Matrix

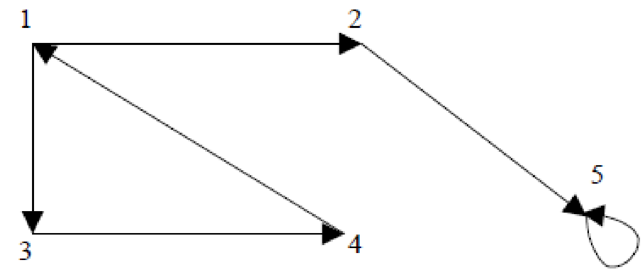


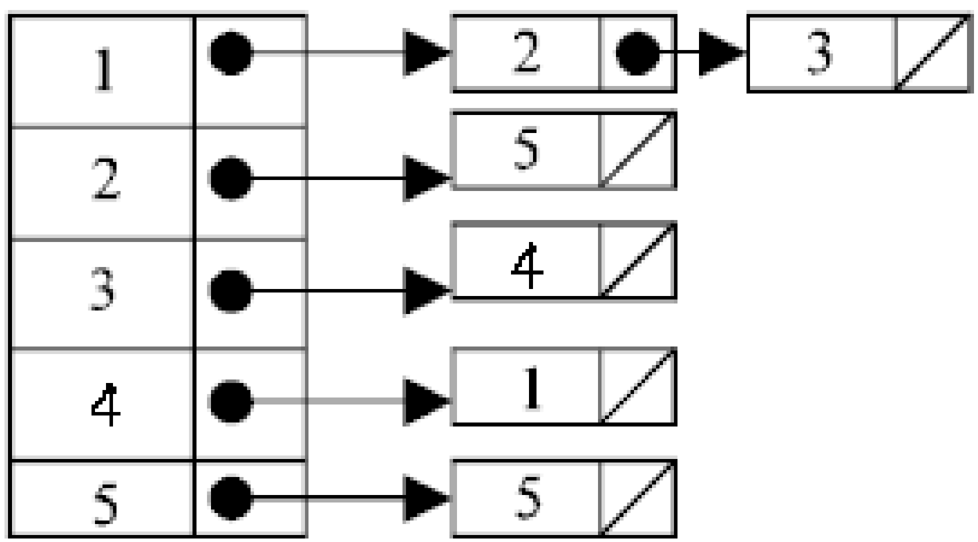
$A_{ij} = W_{ij}$ { if there is an edge from V_i to V_j then represent its weight W_{ij}.}

$A_{ij} = -1$ { if there is no edge from V_i to V_j}

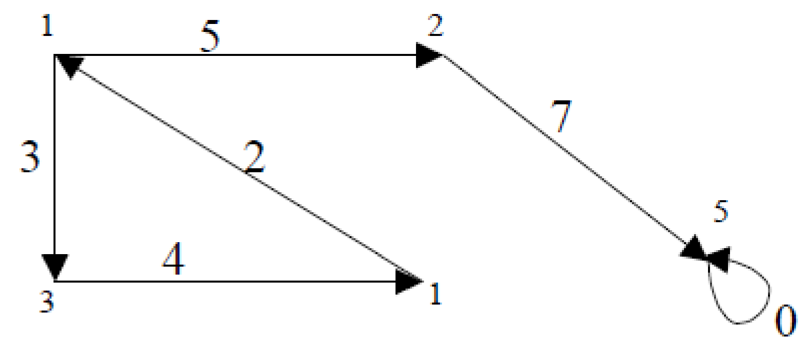
i \ j	1	2	3	4	5
1	– 1	5	3	– 1	– 1
2	– 1	– 1	– 1	– 1	7
3	– 1	– 1	– 1	4	– 1
4	2	– 1	– 1	– 1	– 1
5	– 1	– 1	– 1	– 1	0

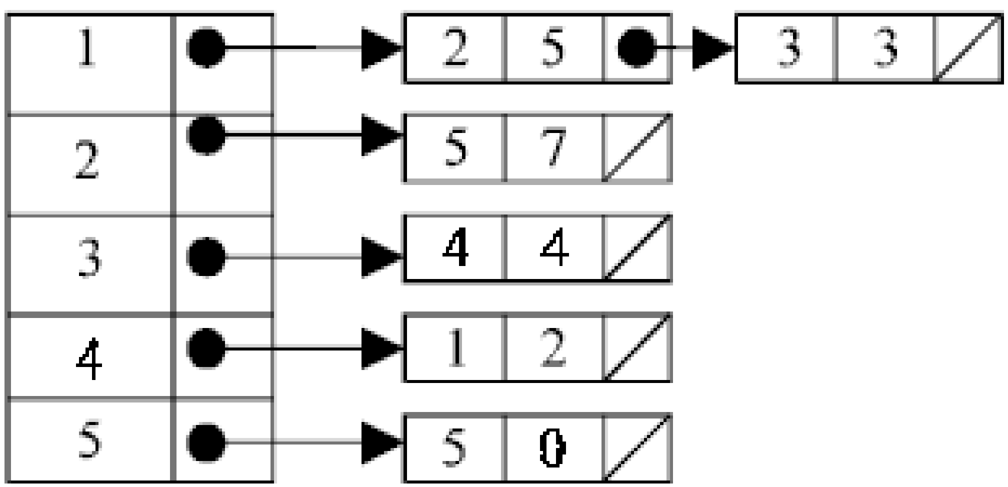
Link List Representation





Weighted Graph in Link List





Graph Traversal Algorithm

- Graph traversal means visiting all the nodes of the graph.
- There are two graph traversal methods
  - Breadth First Search (BFS)
  - *Depth First Search (DFS)*

Breadth-First Search

- Explores a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the breadth of the frontier
- Builds a tree over the graph
  - Pick a source vertex to be the root
  - Find its children, then their children, etc.

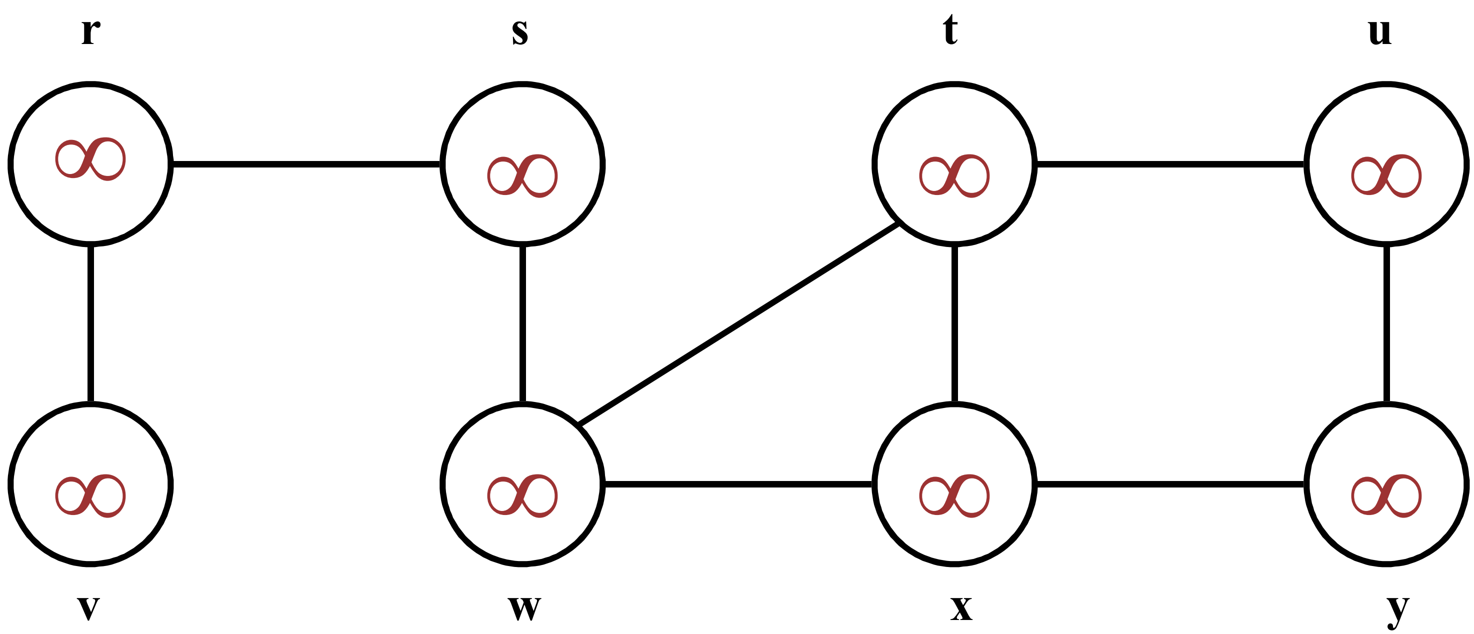
Breadth-First Search

- We will use vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

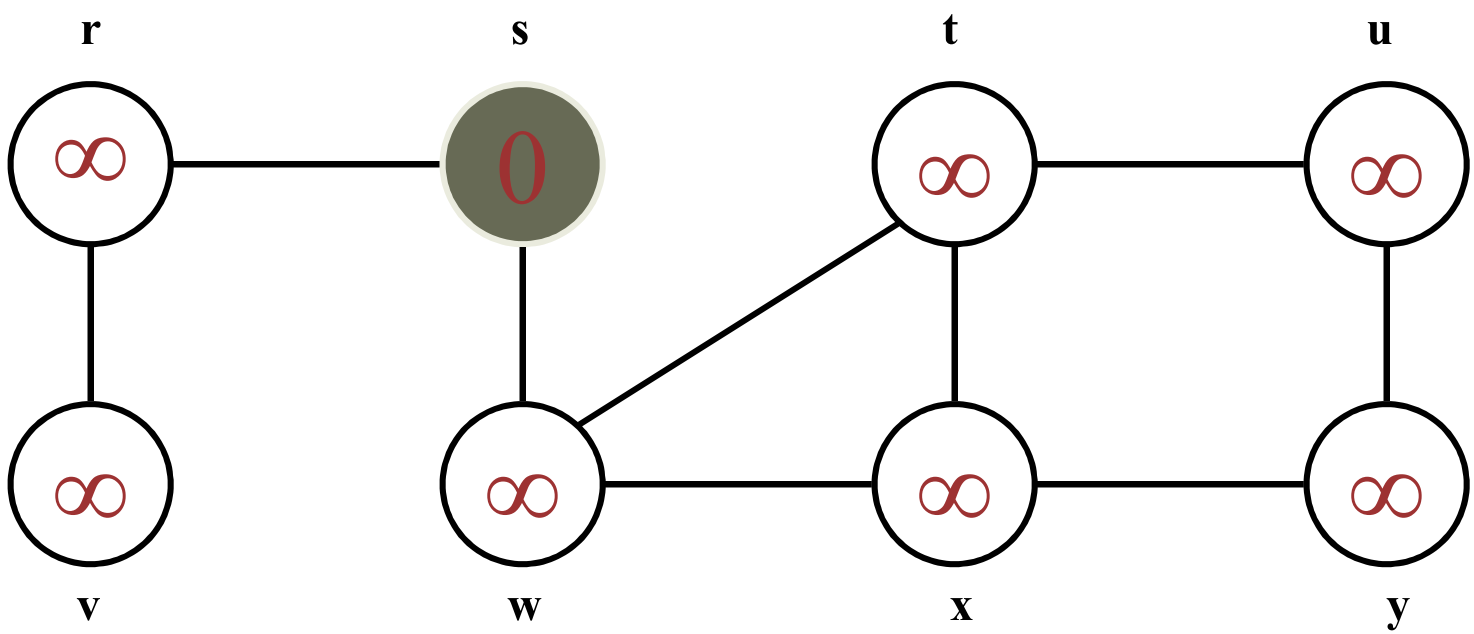
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};                    // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Breadth-First Search: Example

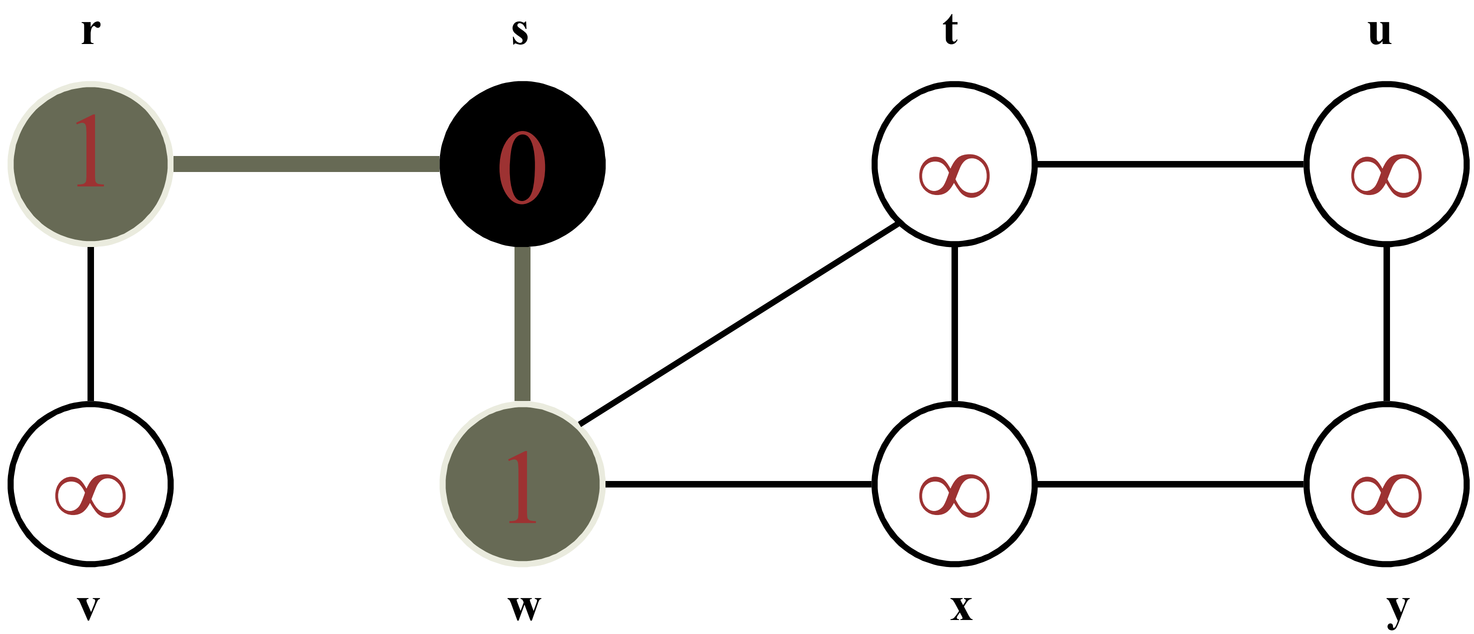


Breadth-First Search: Example



Q: s

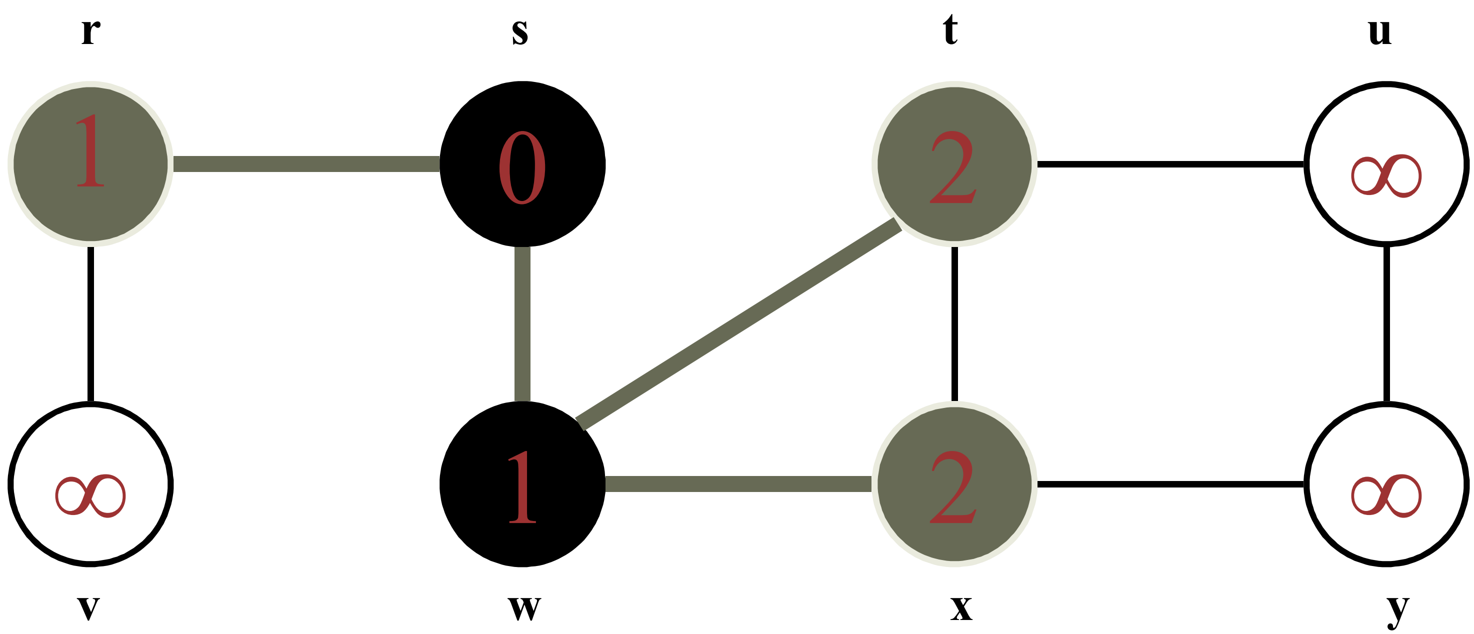
Breadth-First Search: Example

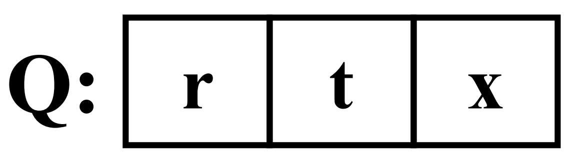


Q:

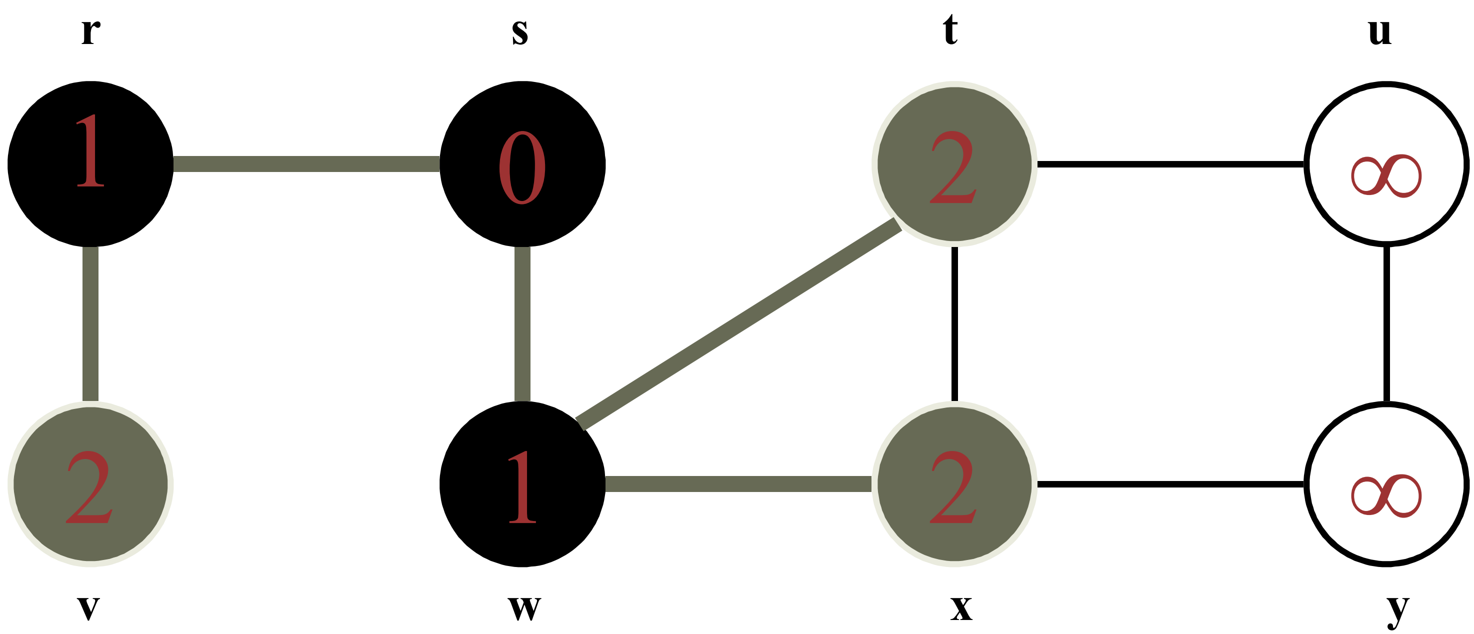
w	r

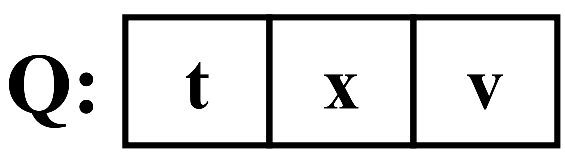
Breadth-First Search: Example



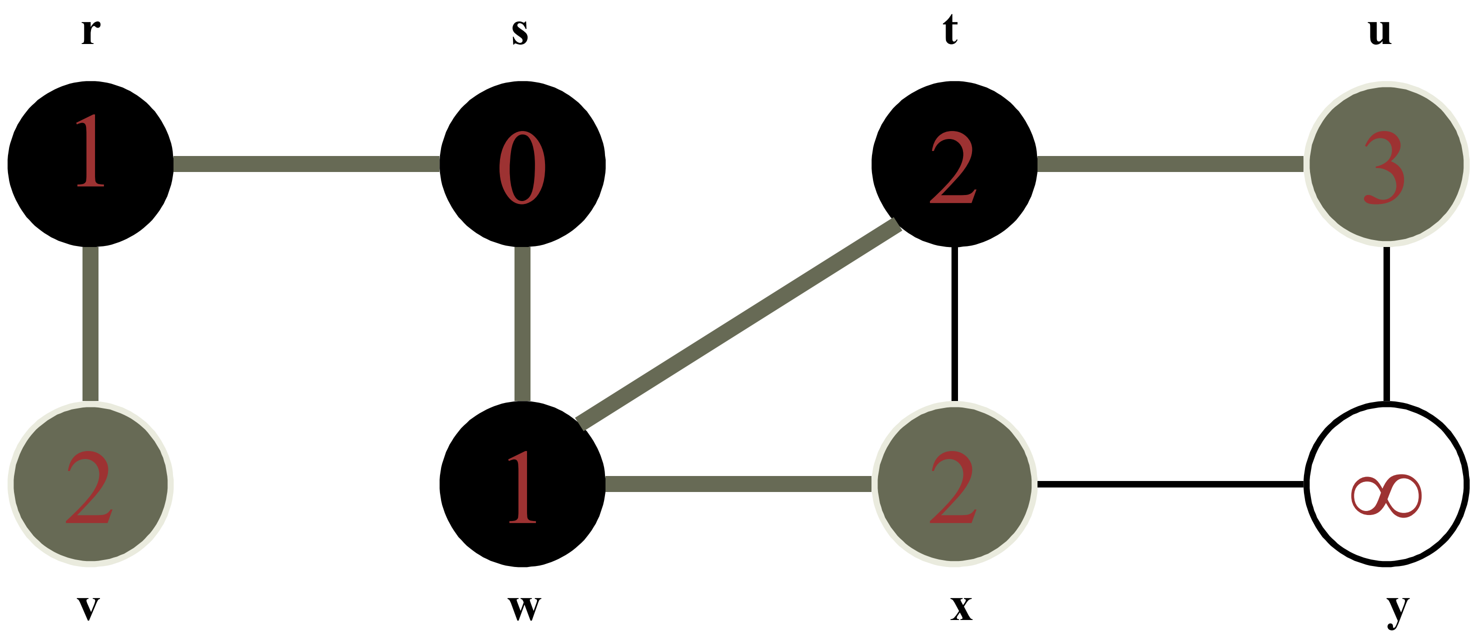


Breadth-First Search: Example



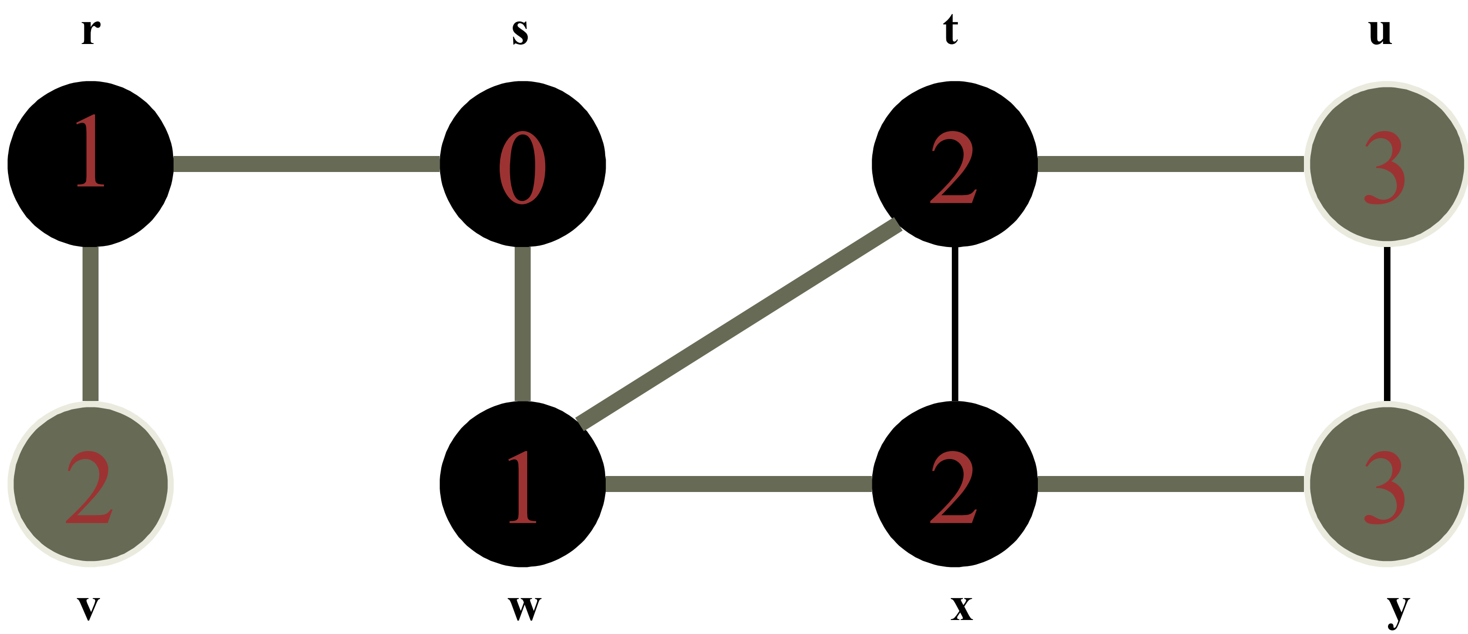


Breadth-First Search: Example



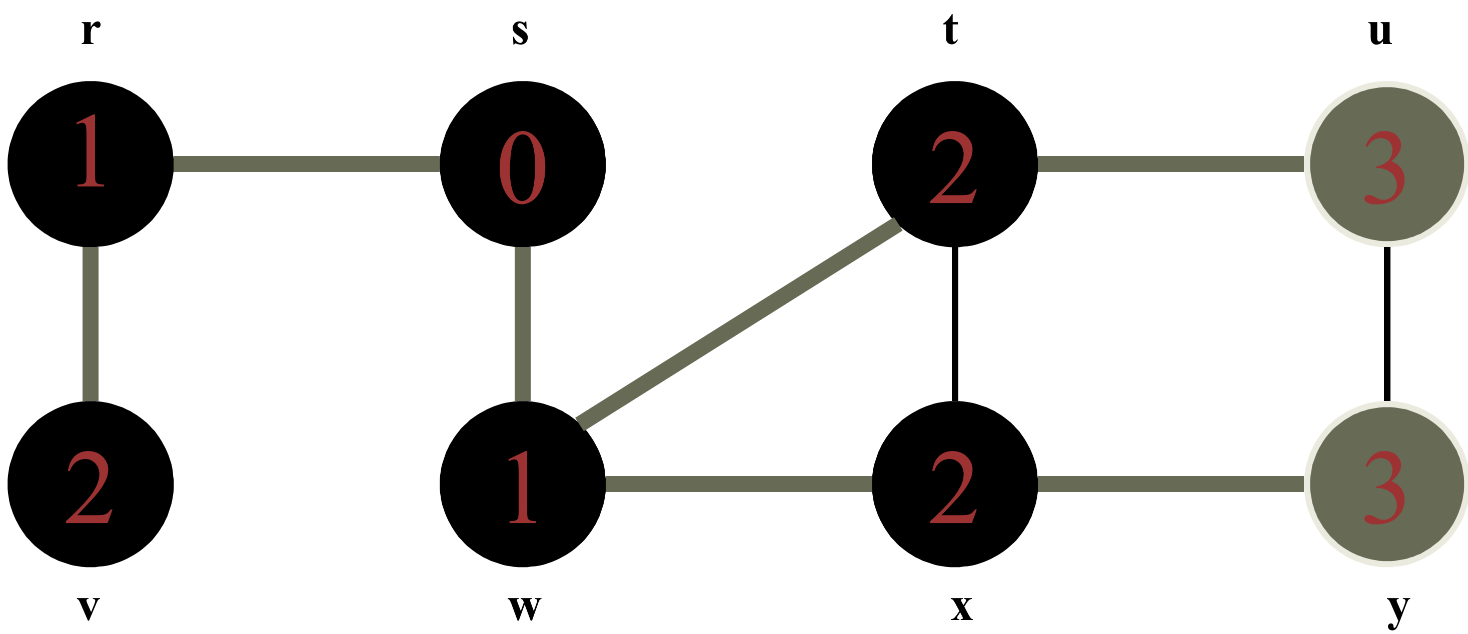
Q: | x | v | u |

Breadth-First Search: Example



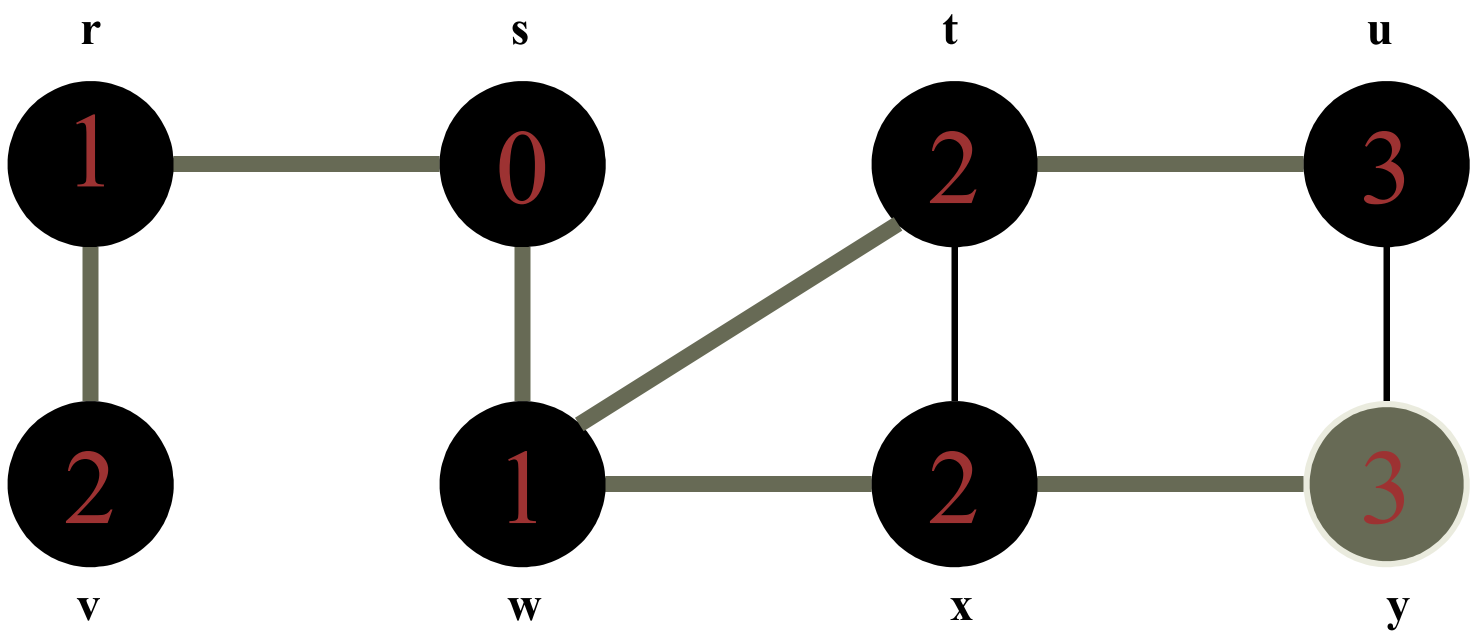
Q: | v | u | y |

Breadth-First Search: Example



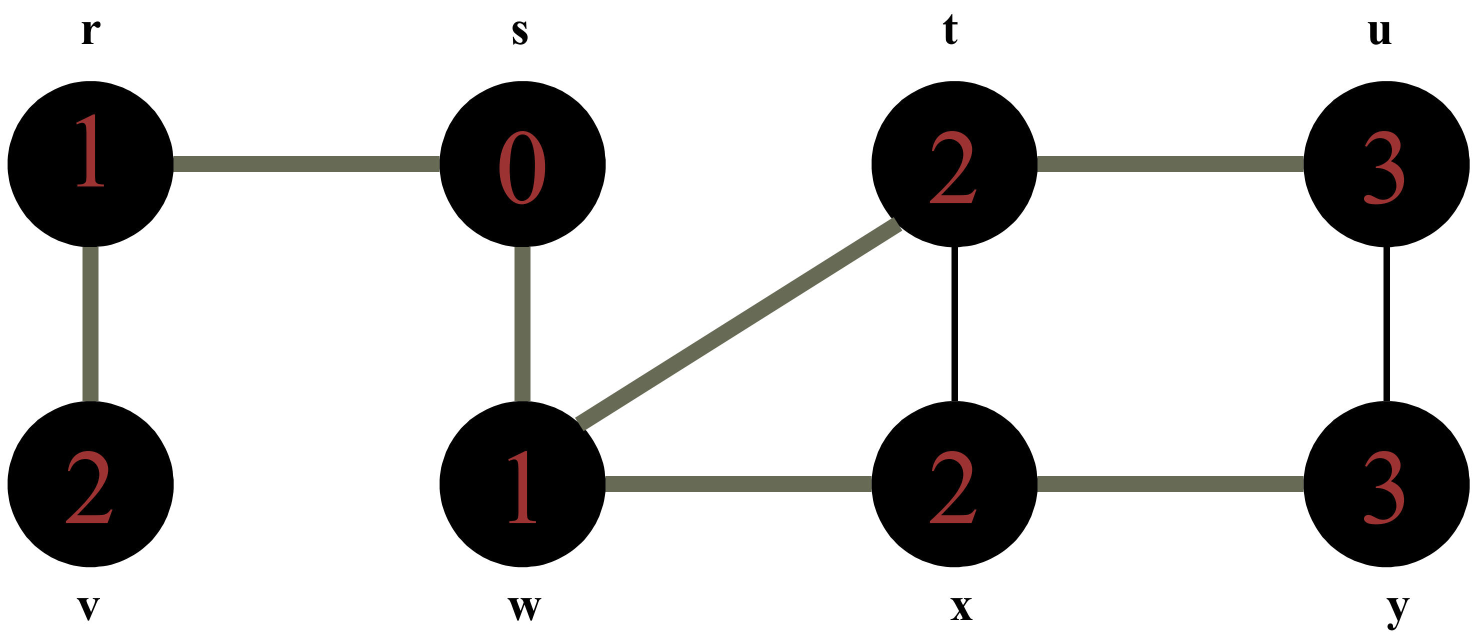
Q: | u | y |

Breadth-First Search: Example



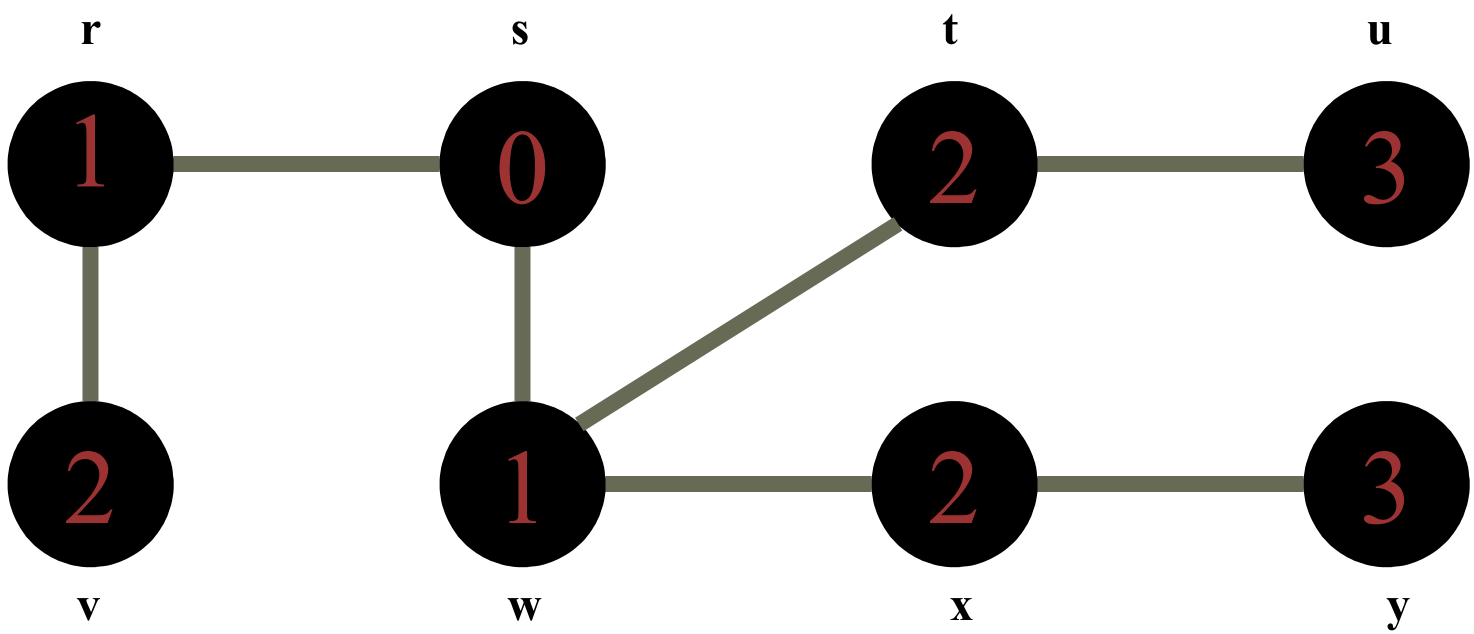
Q: y

Breadth-First Search: Example



Q: Ø

Breadth-First Search: Example



Q: Ø

Breadth-First Search: Properties

- BFS calculates the shortest-path distance to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
- BFS builds breadth-first tree, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time

Depth-First Search

- Depth-first search is another strategy for exploring a graph
  - Explore "deeper" in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex v that still has unexplored edges
  - When all of v's edges have been explored, backtrack to the vertex from which v was discovered

Depth-First Search

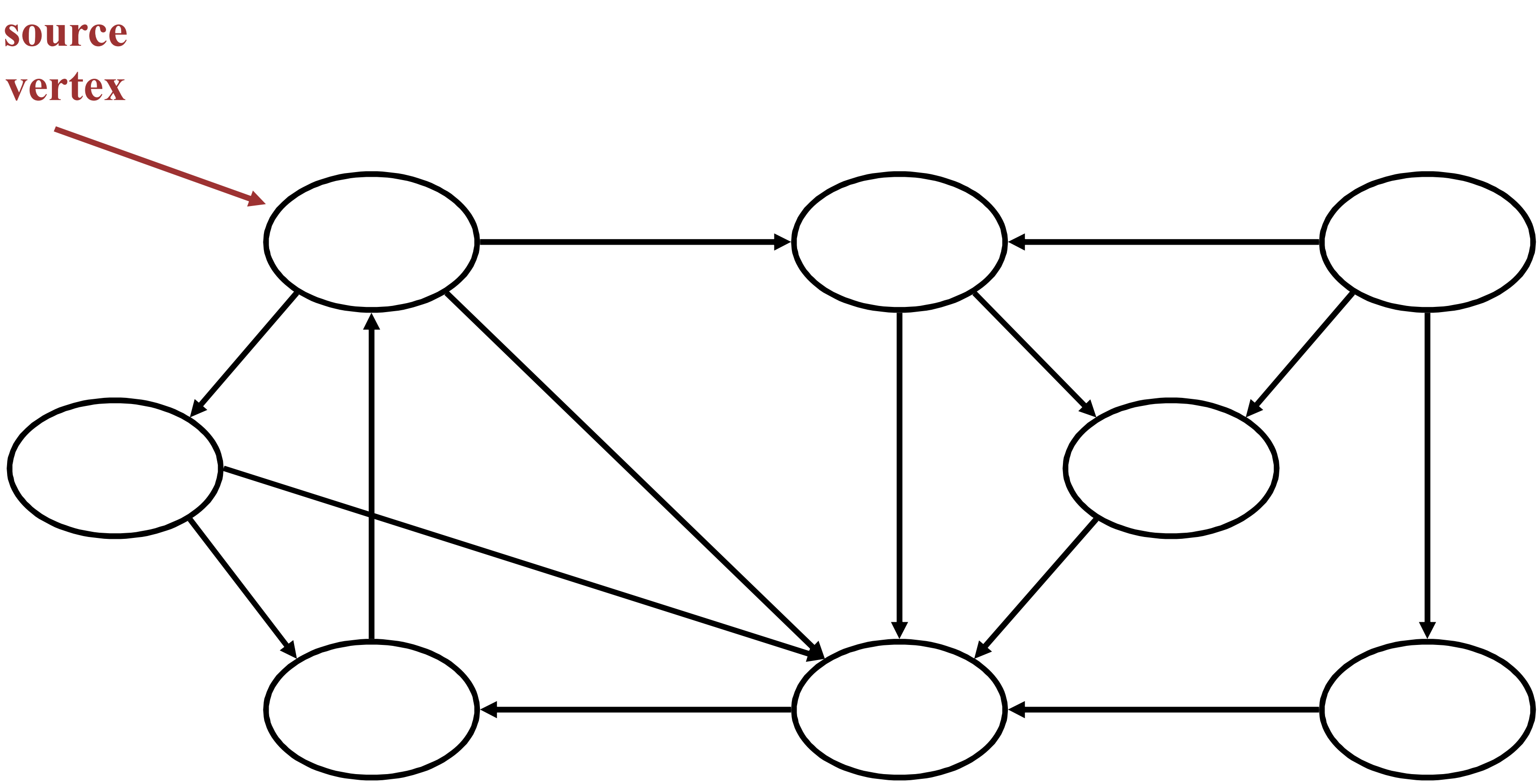
- We will use vertex "colors" to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices

Depth-First Search: The Code

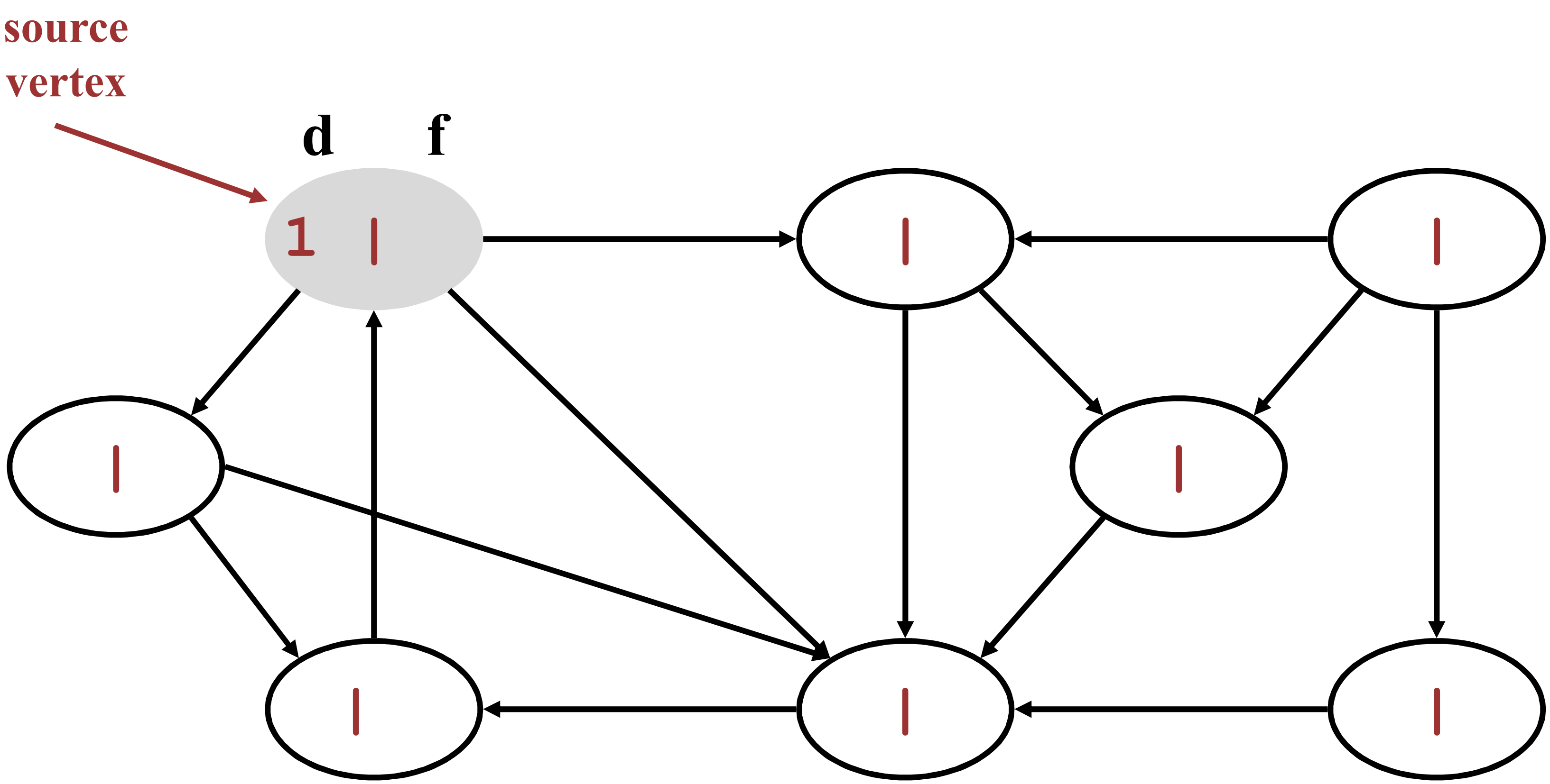
```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

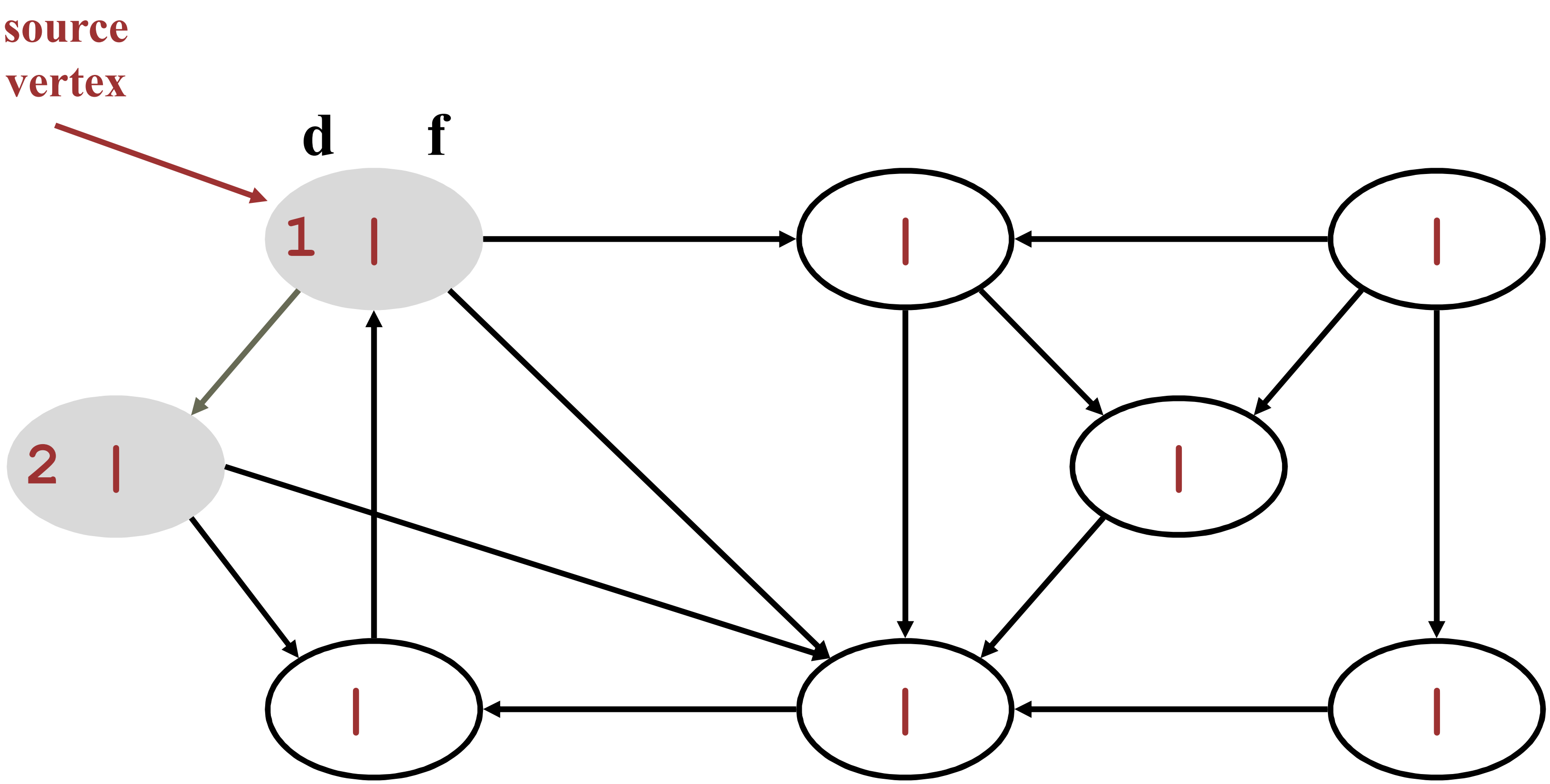
DFS Example



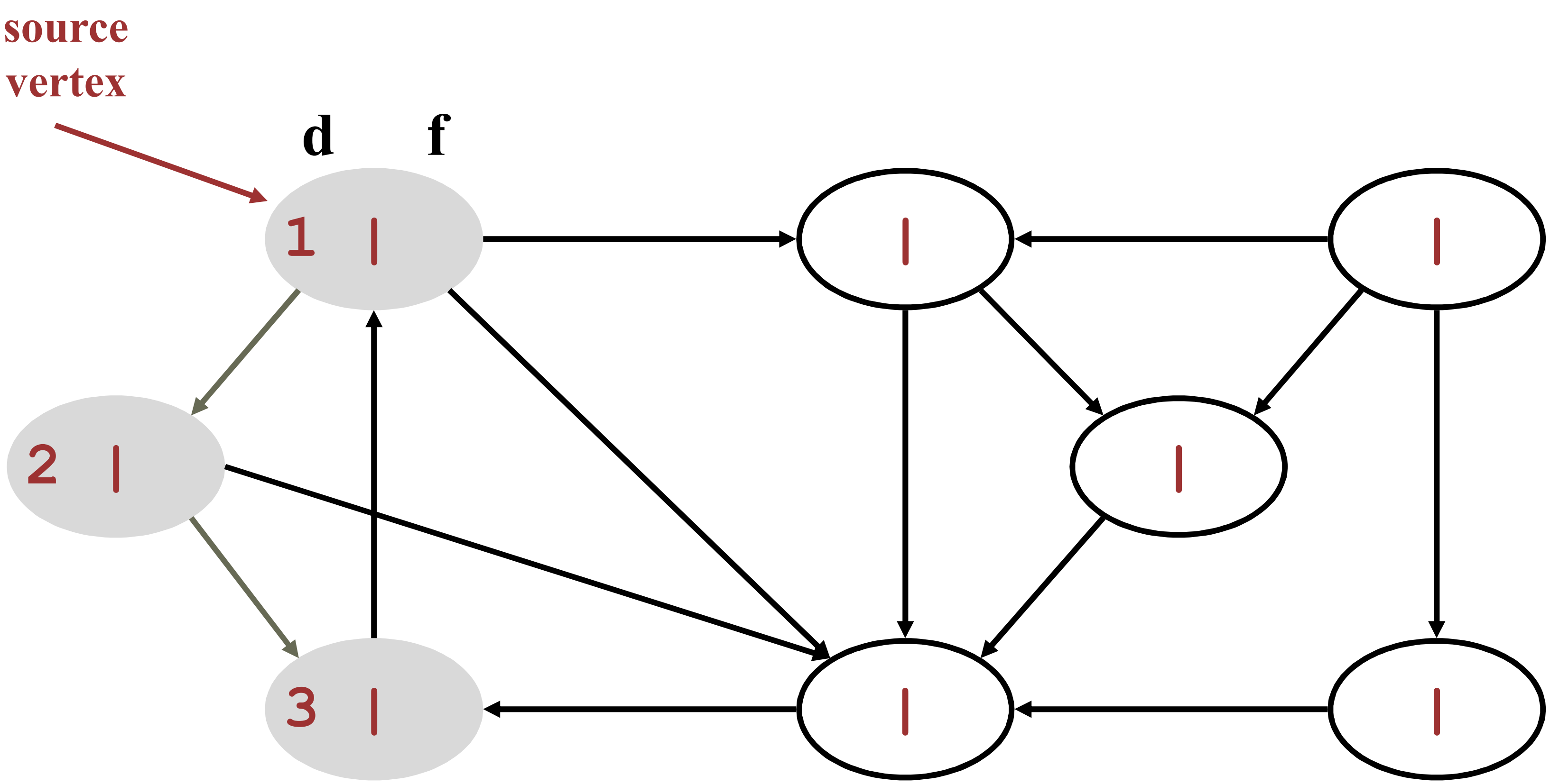
DFS Example



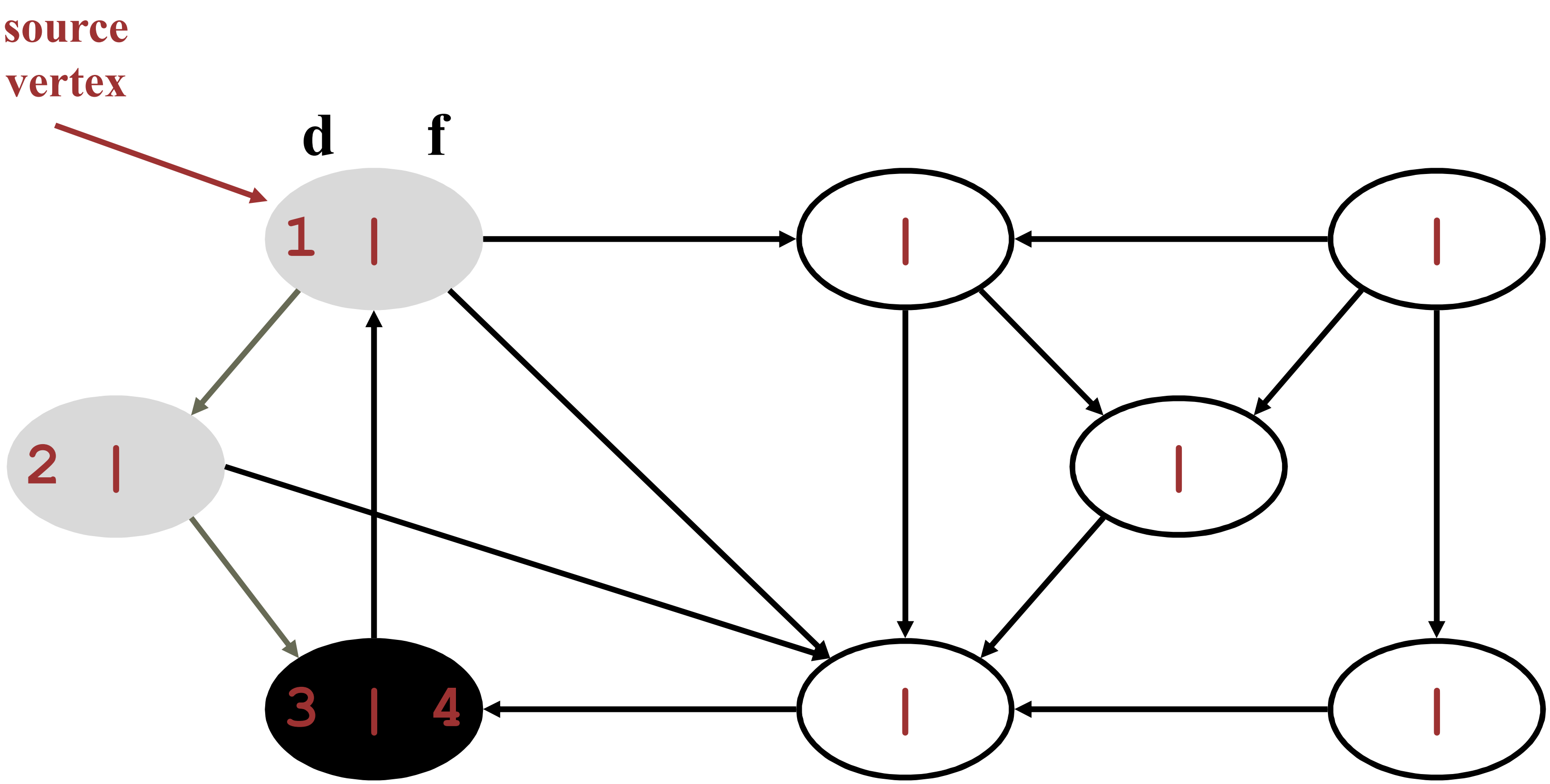
DFS Example



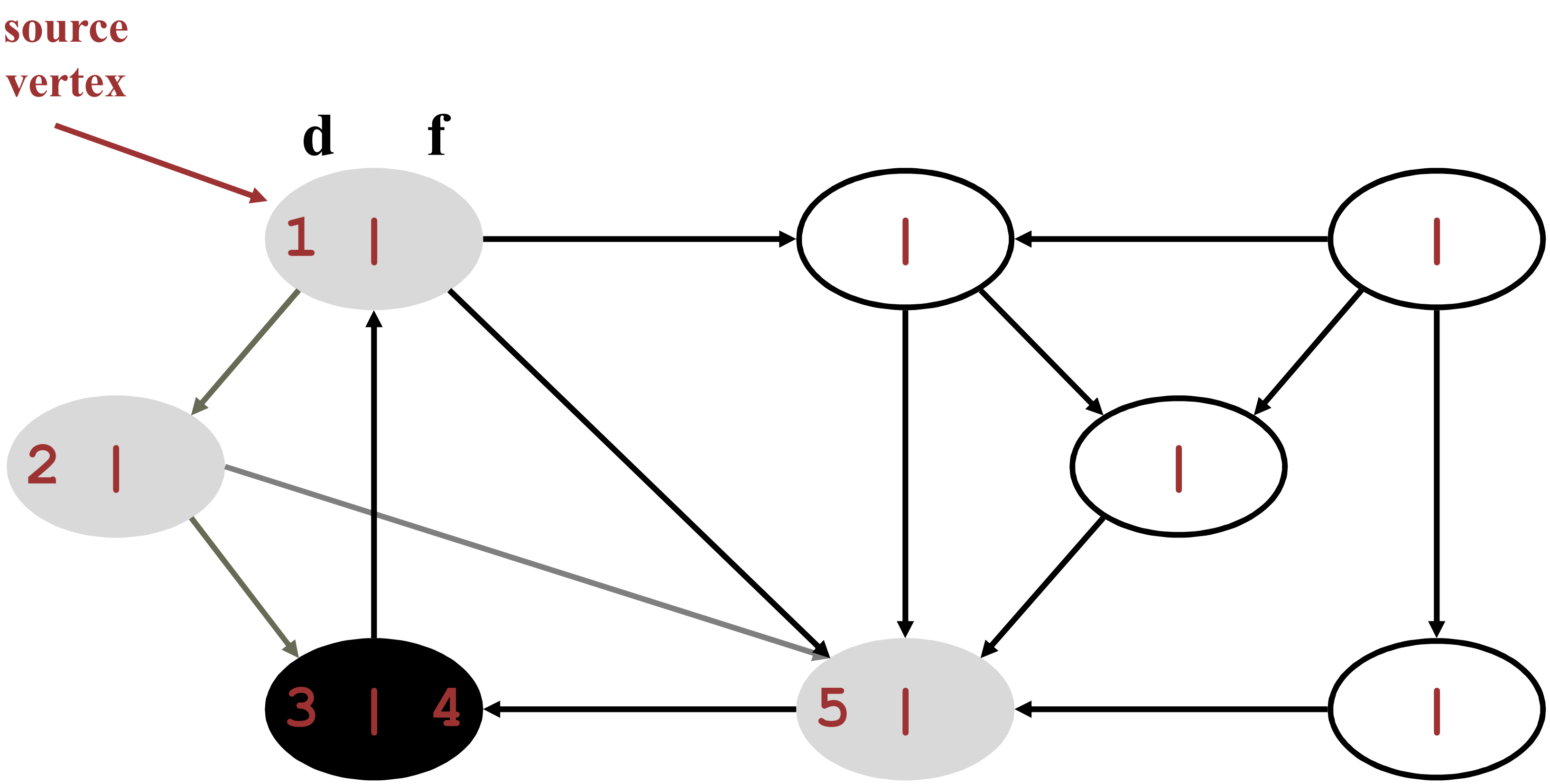
DFS Example



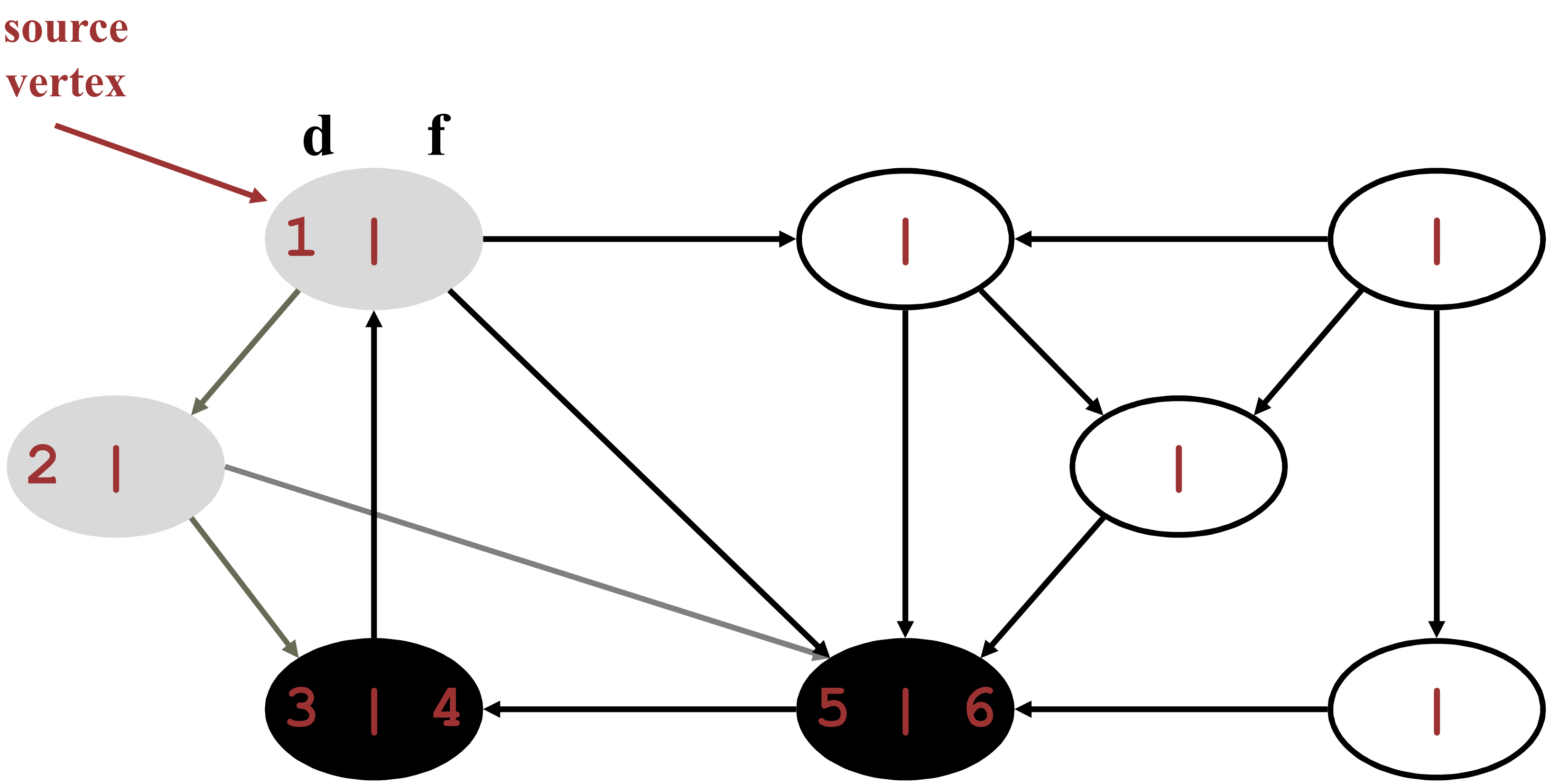
DFS Example



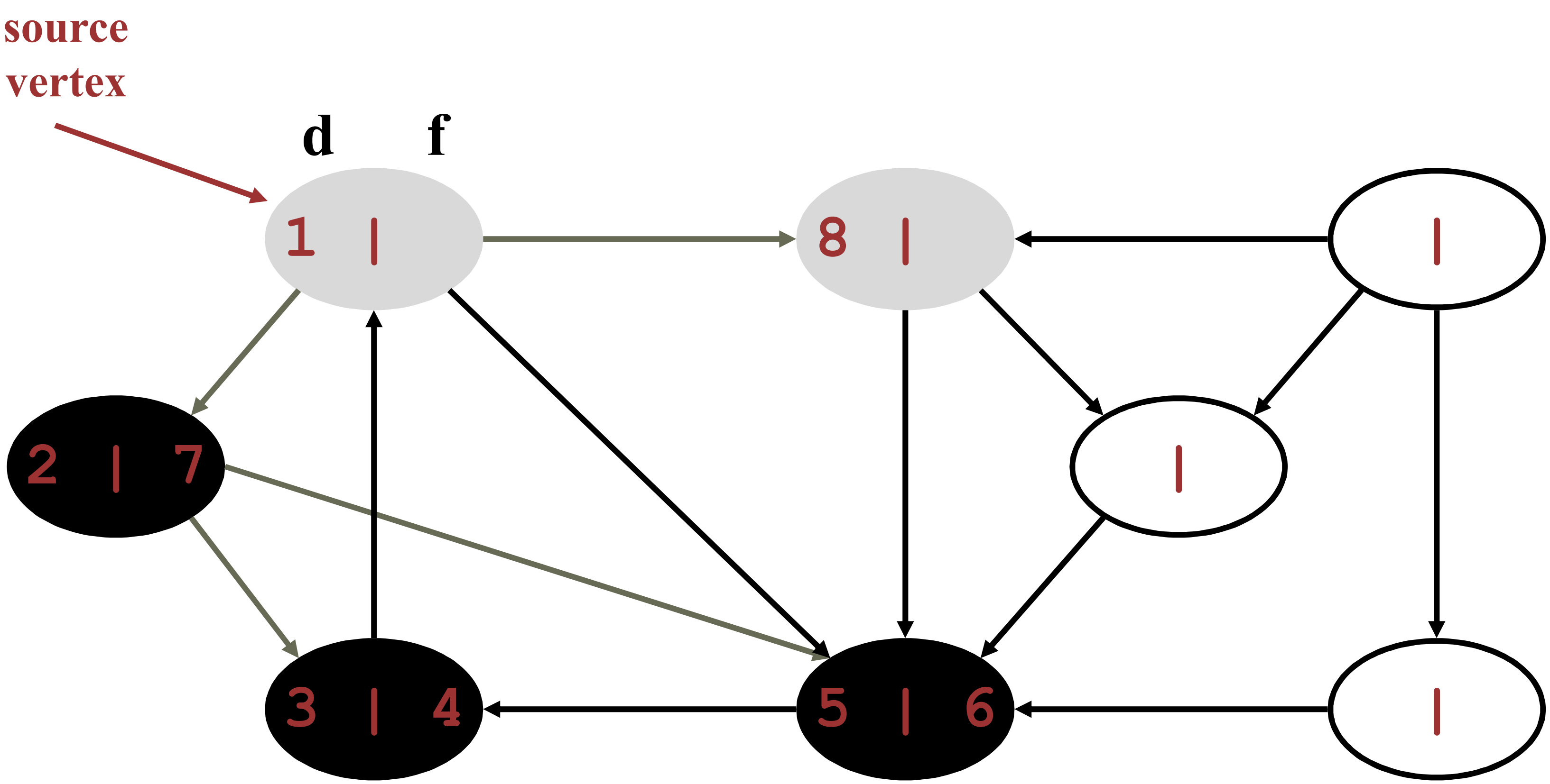
DFS Example



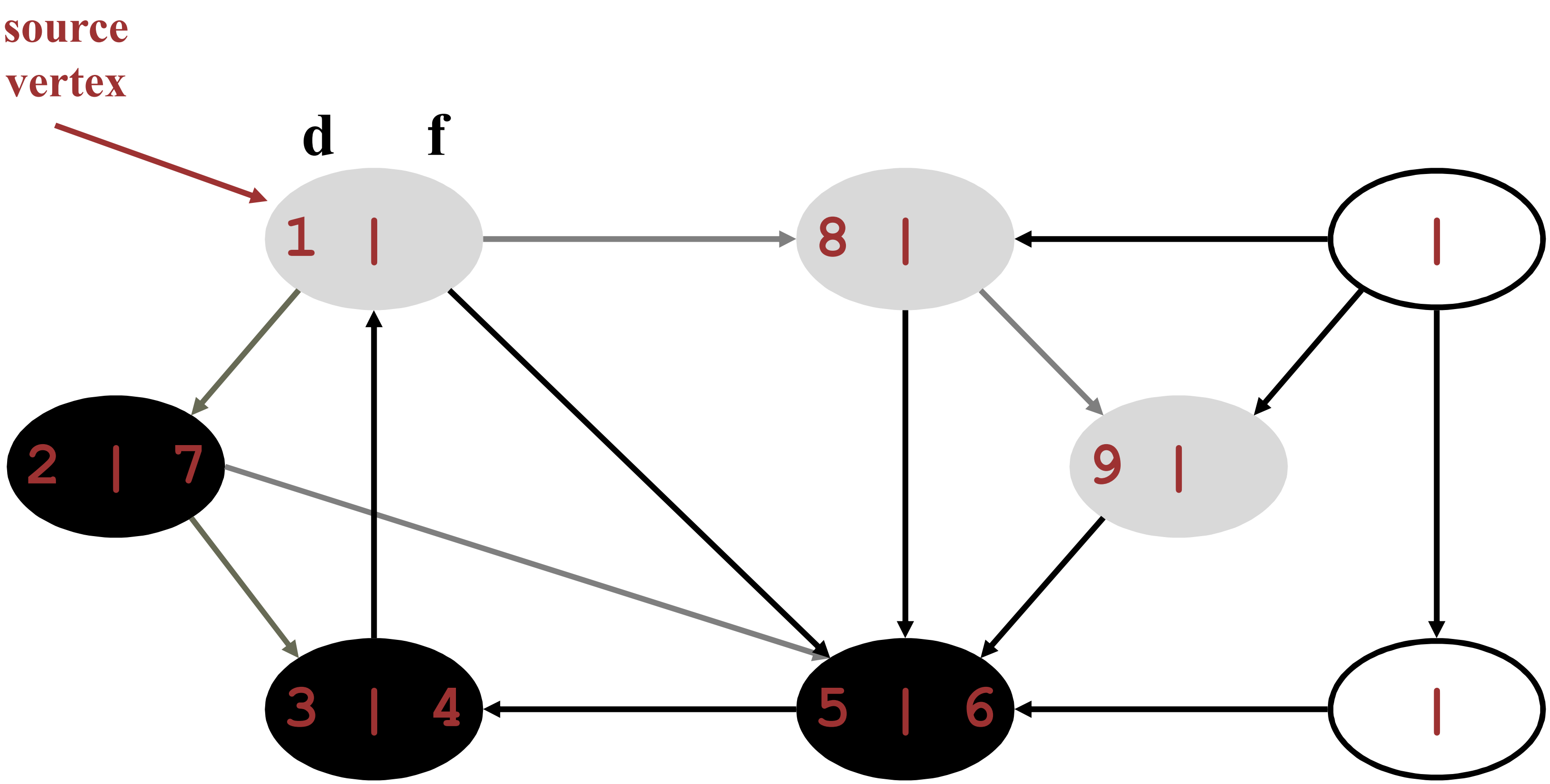
DFS Example



DFS Example

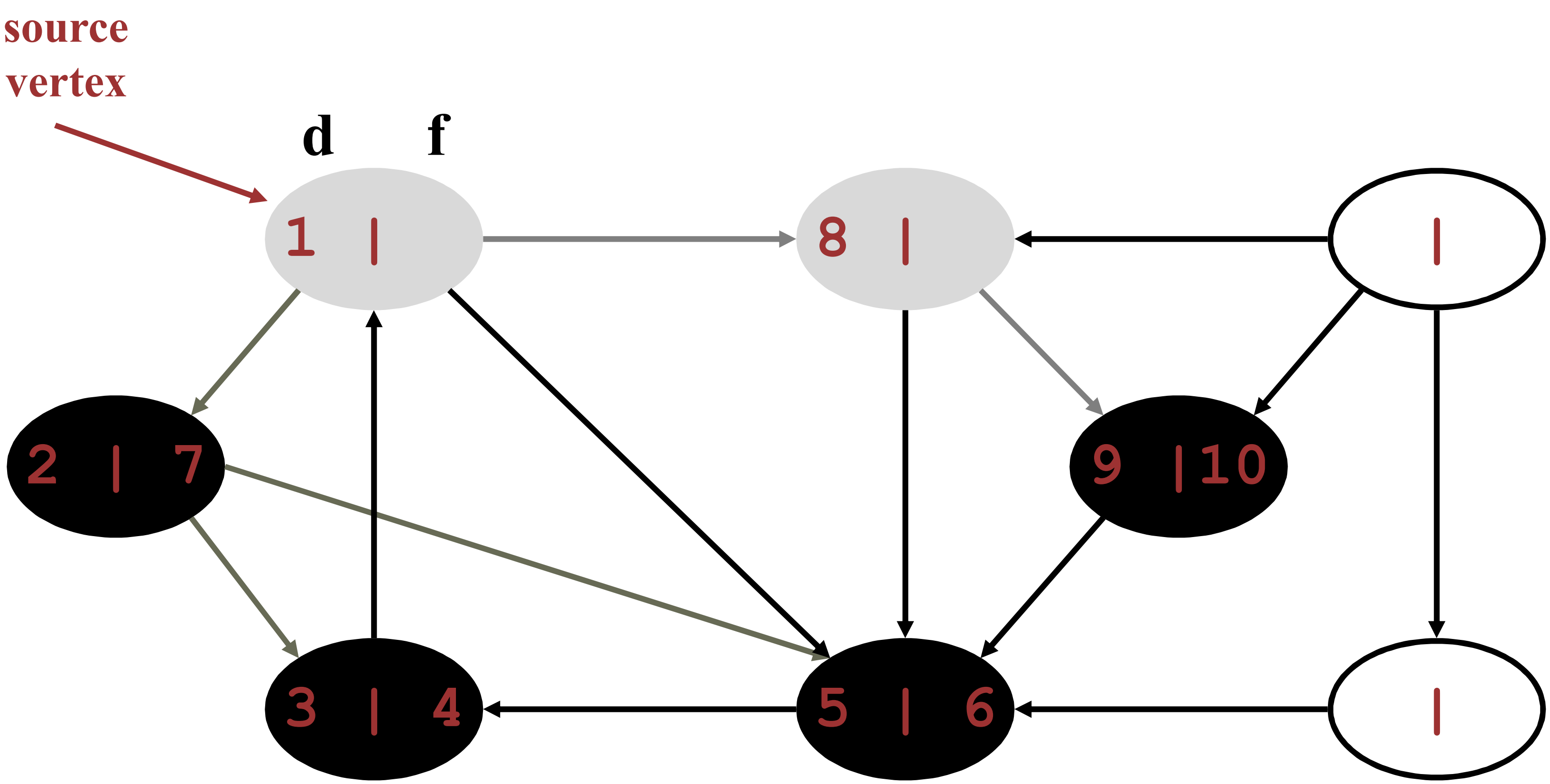


DFS Example

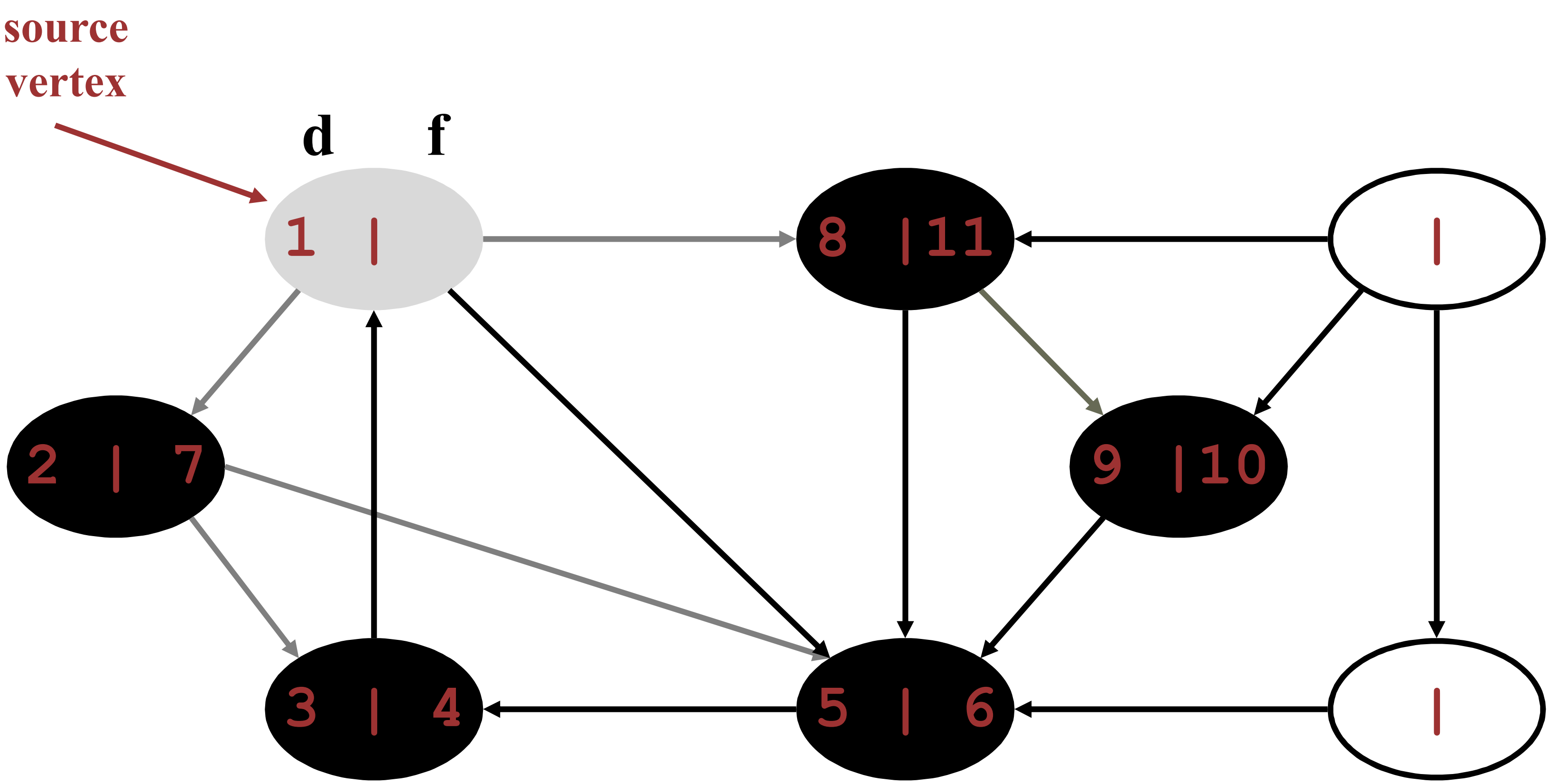


What is the structure of the grey vertices?
What do they represent?

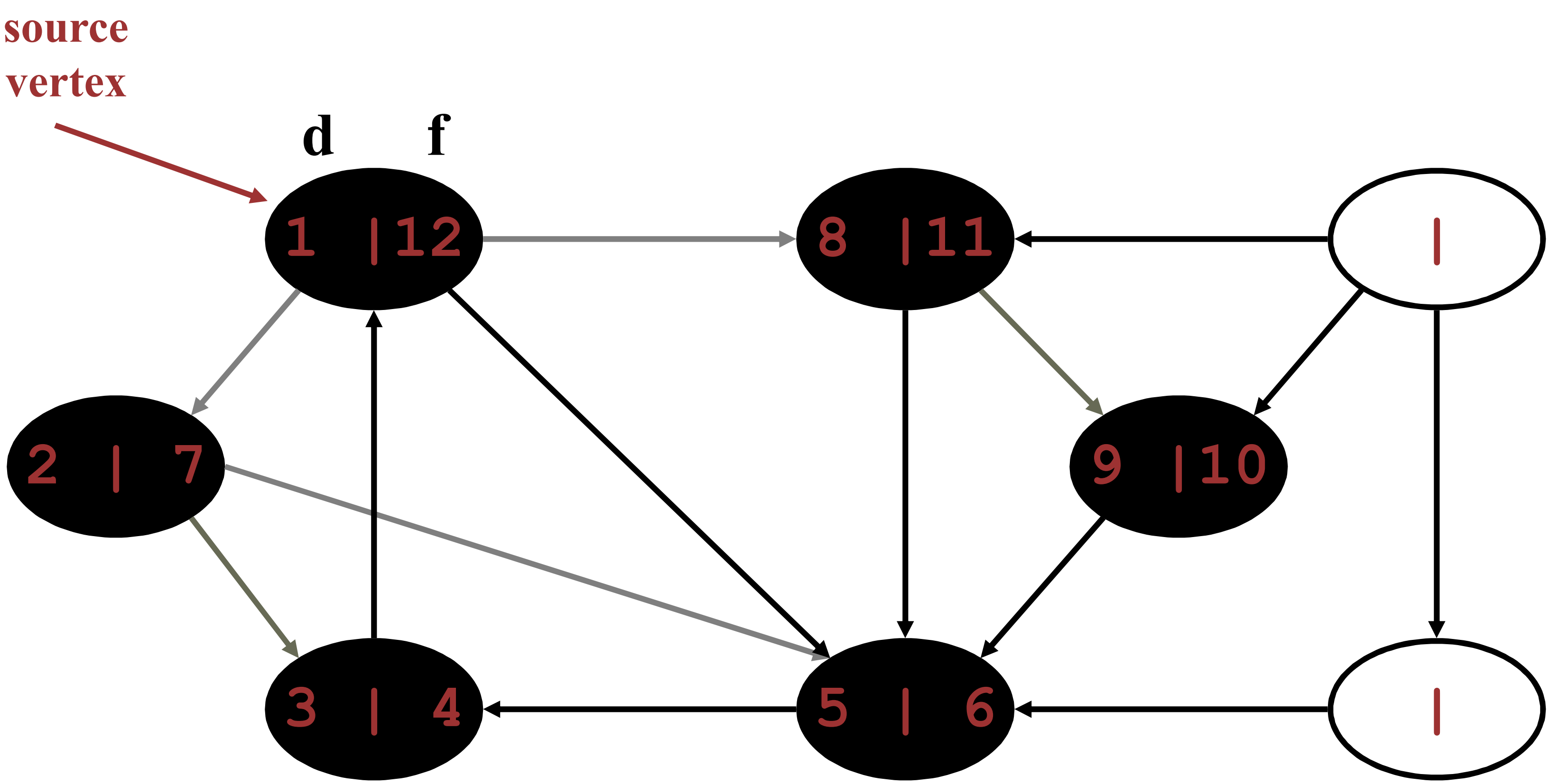
DFS Example



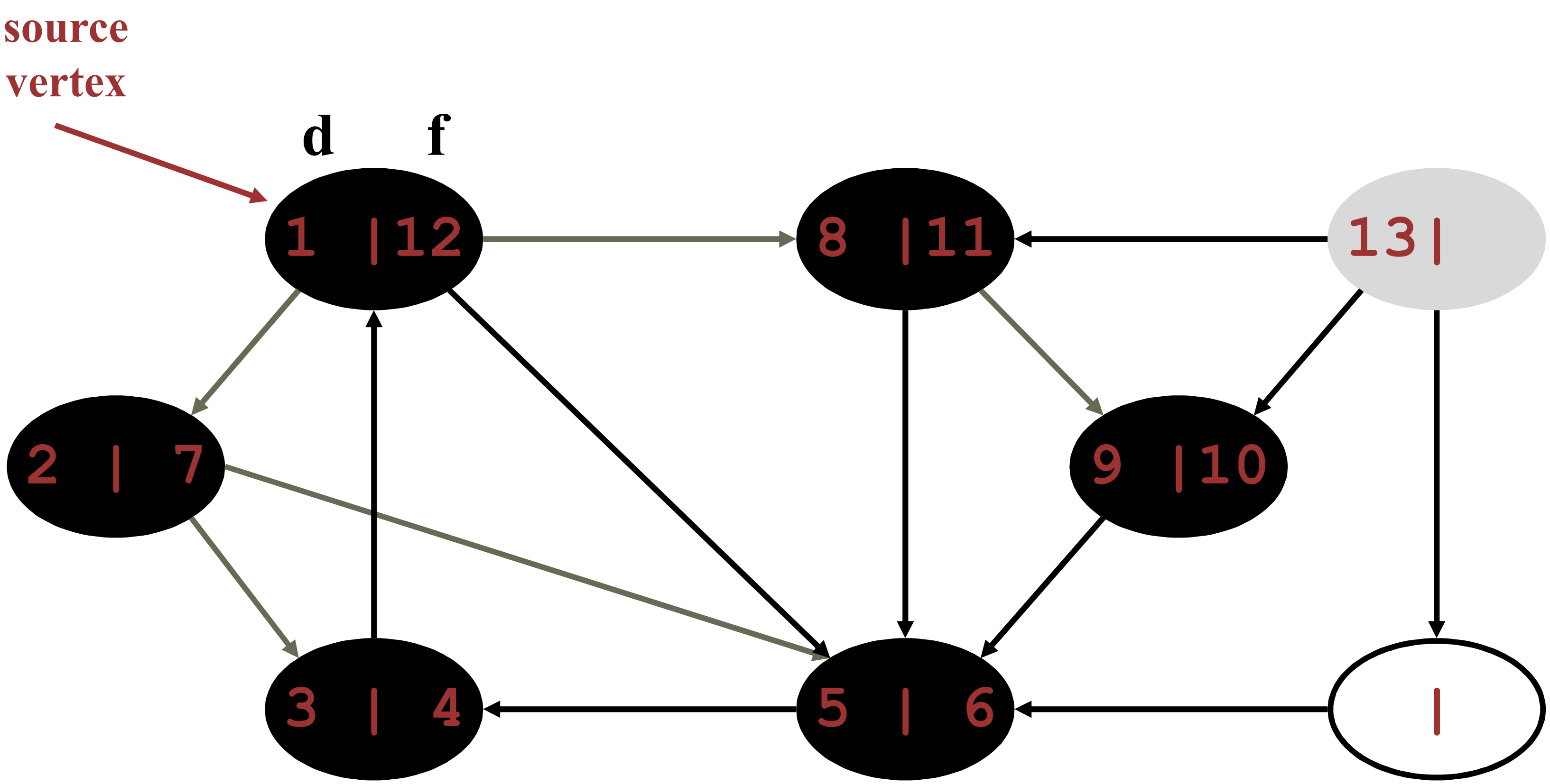
DFS Example



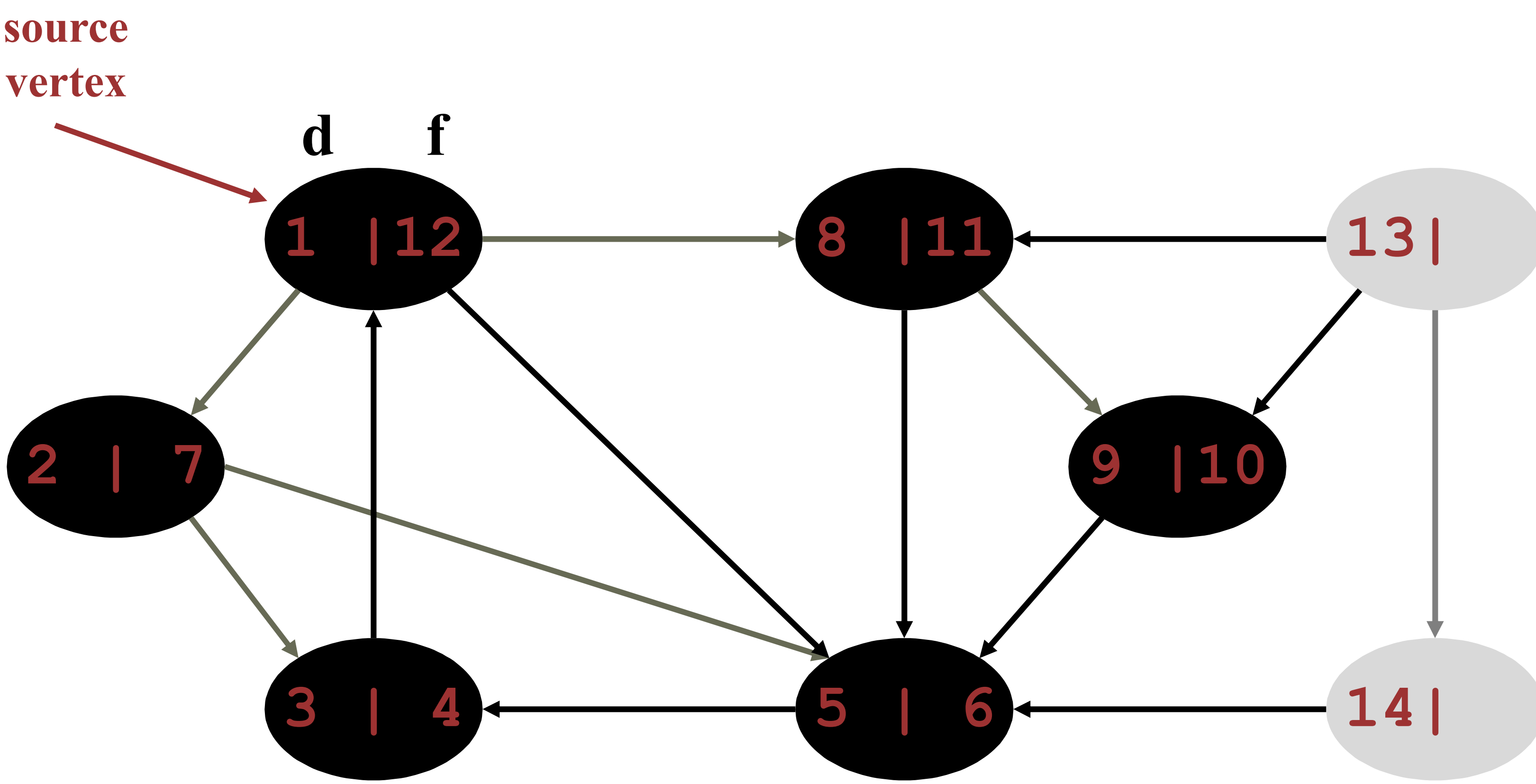
DFS Example



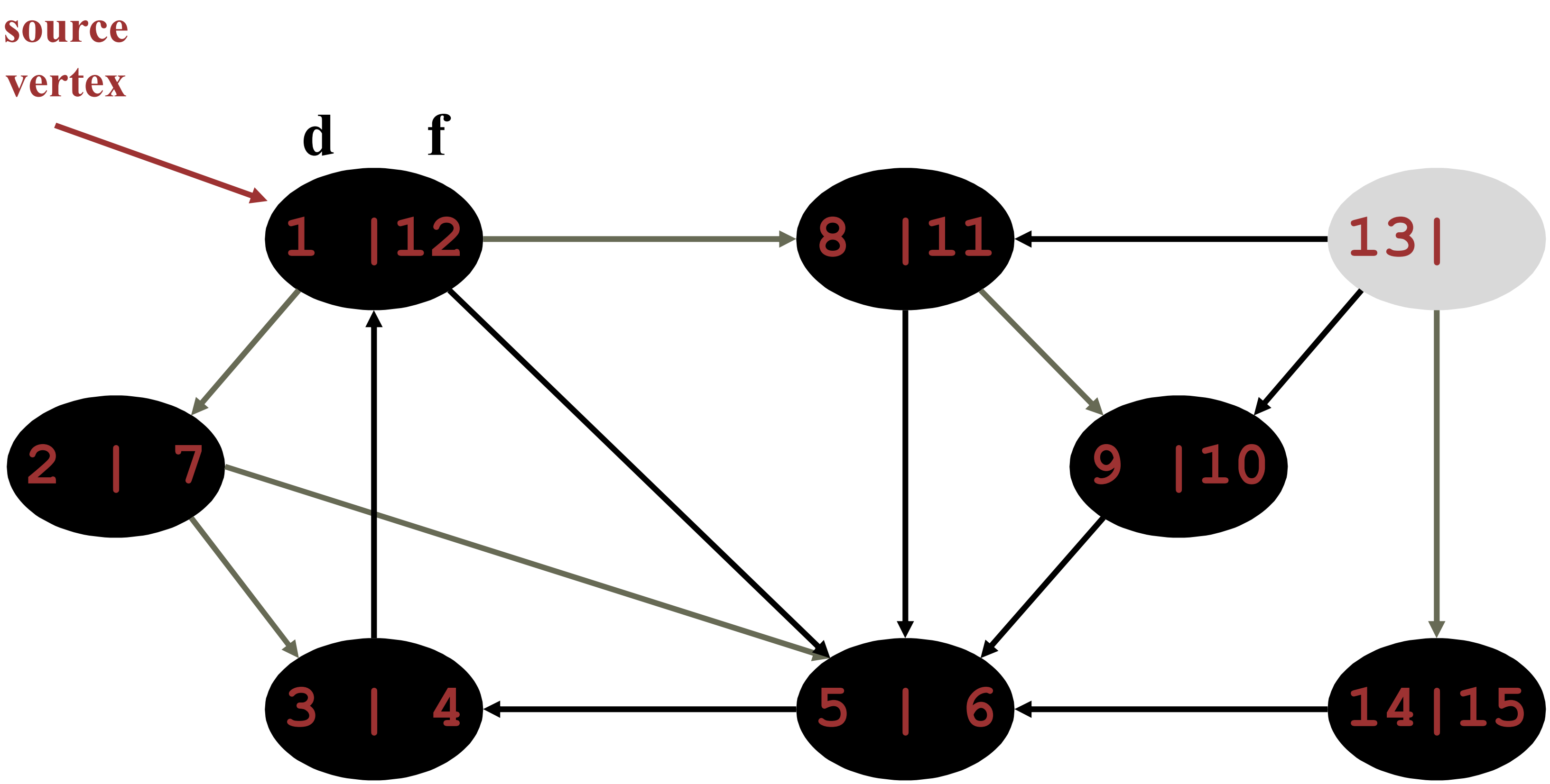
DFS Example



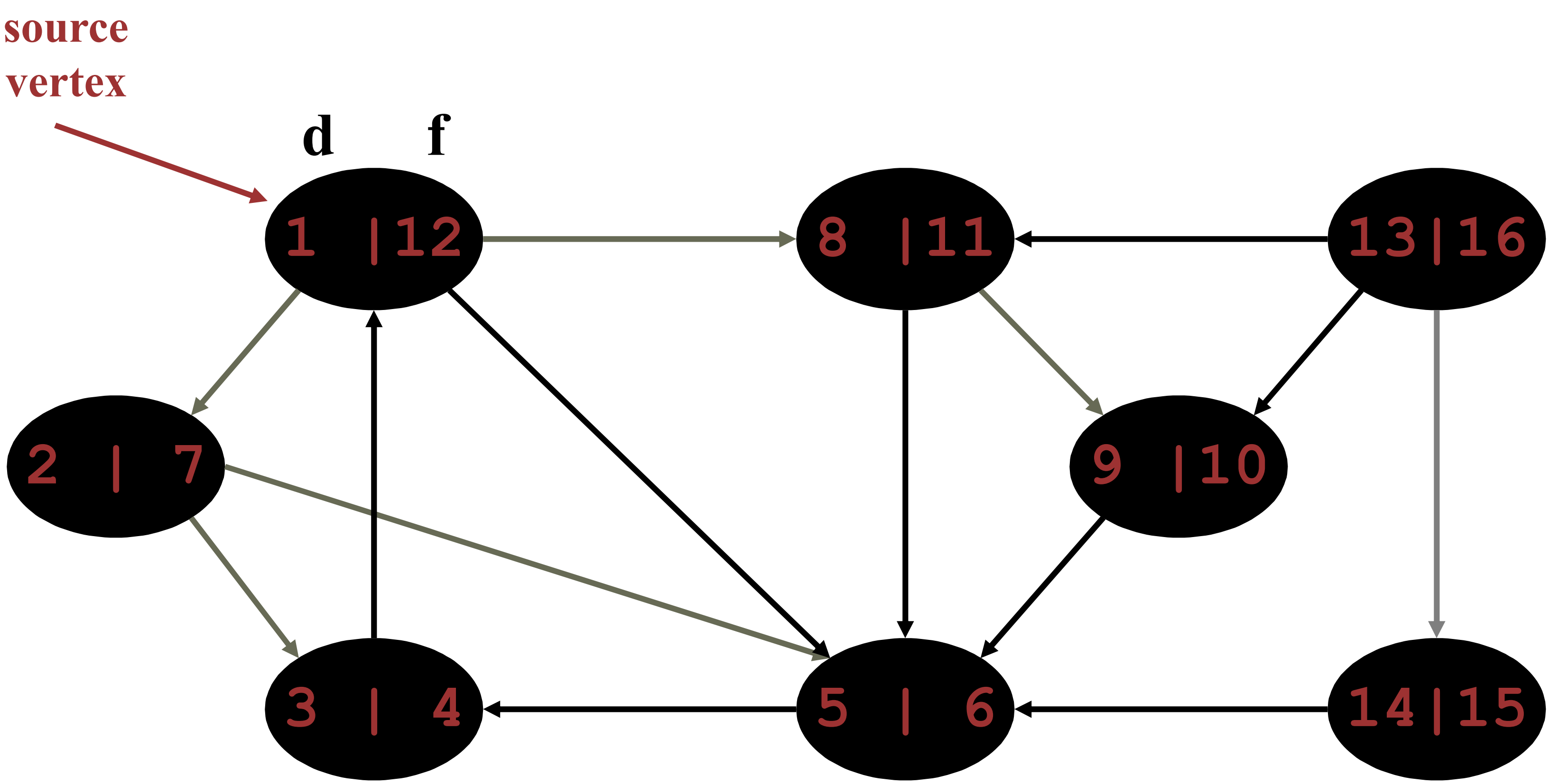
DFS Example



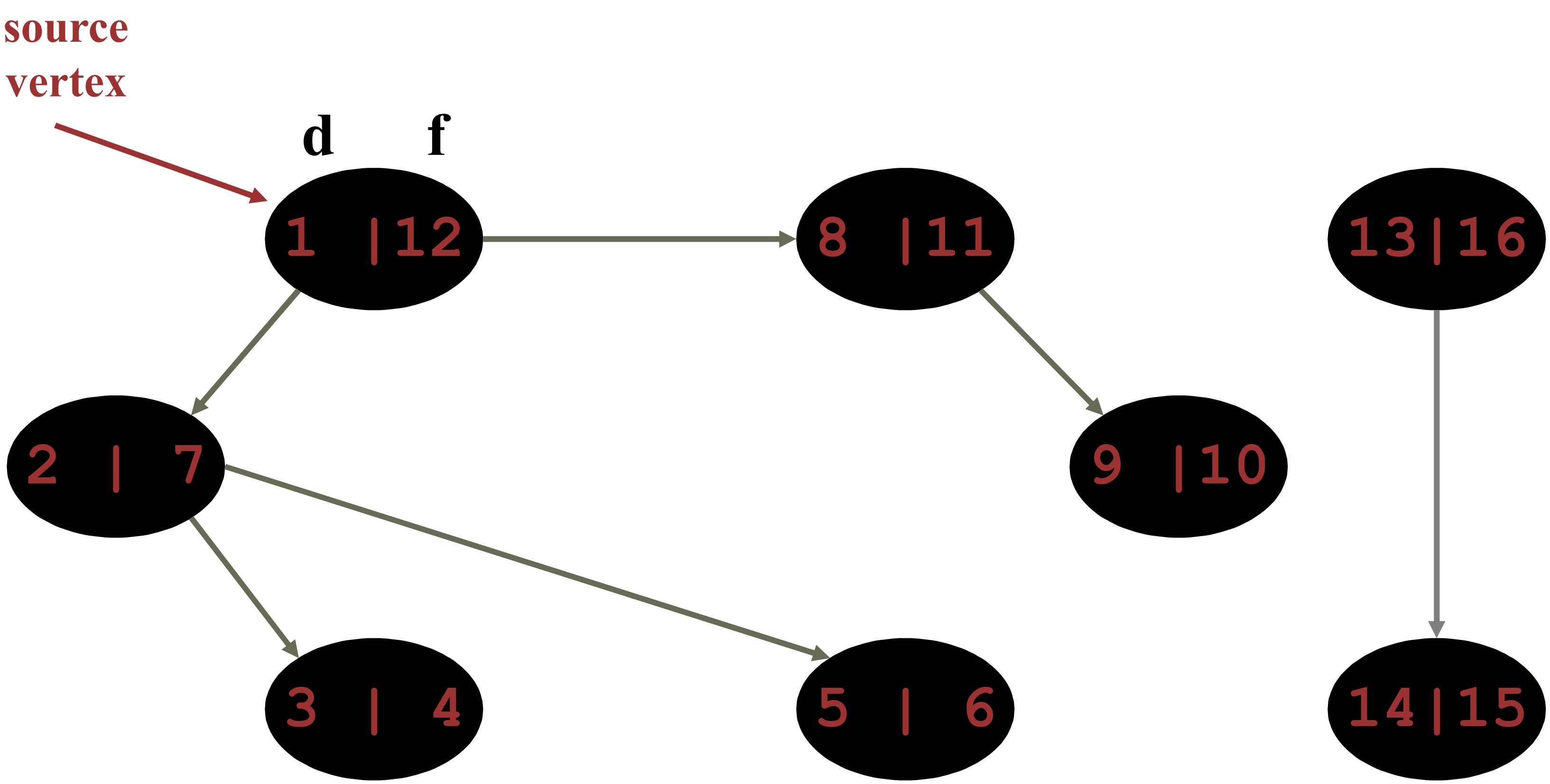
DFS Example



DFS Example



DFS Example



Minimum Spanning Tree

- A minimum spanning tree (MST) for a graph G = (V, E) is a sub graph G1 = (V1, E1) of G contains all the vertices of G.
- A MST fulfills the following properties:
  - The vertex set V1 is same as that at graph G.
  - The edge set E1 is a subset of G.
  - And there is no cycle.
- Three algorithms
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Sollin's Algorithm

Kruskal's Algorithm

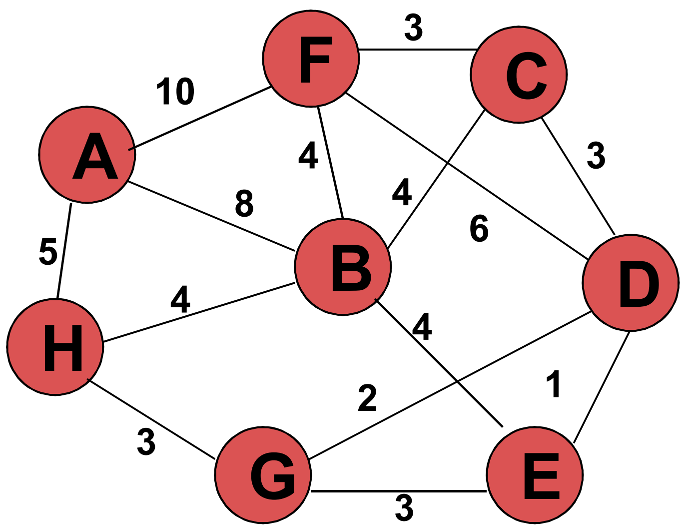
Work with edges, rather than nodes

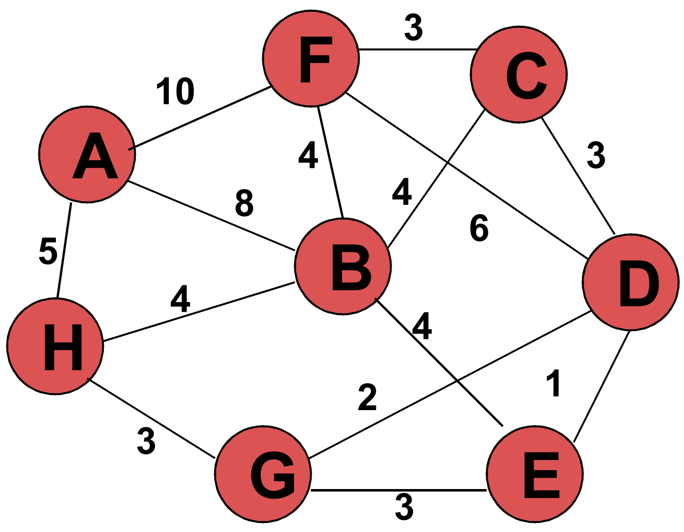
Two steps:

- Sort edges by increasing edge weight
- Select the first |V| – 1 edges that do not generate a cycle

Walk-Through

Consider an undirected, weight graph



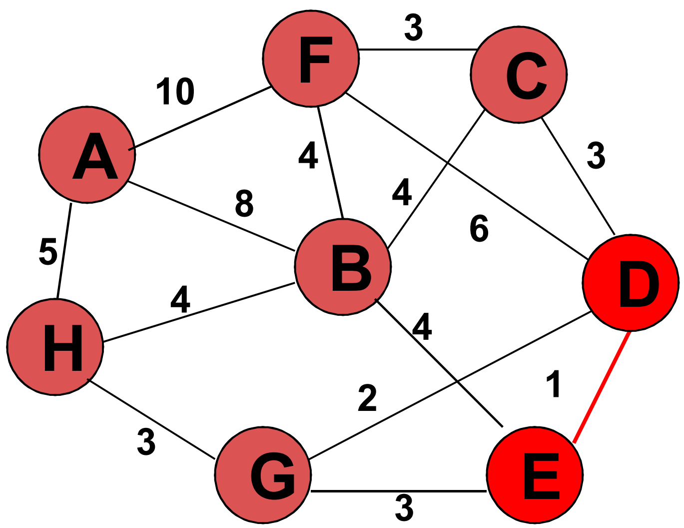


Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

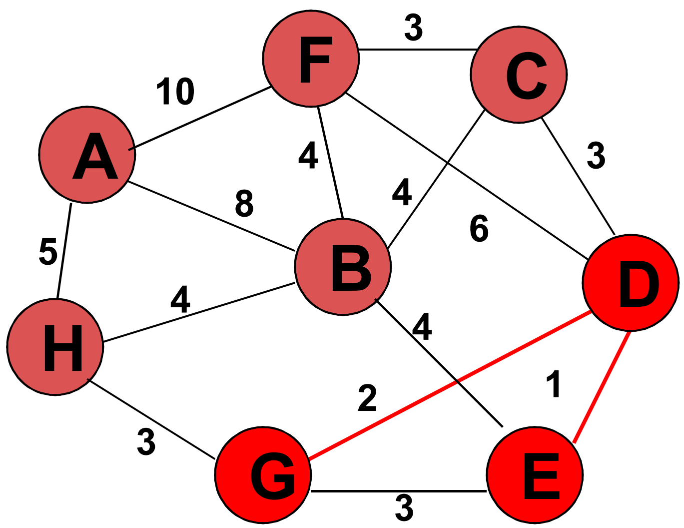
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

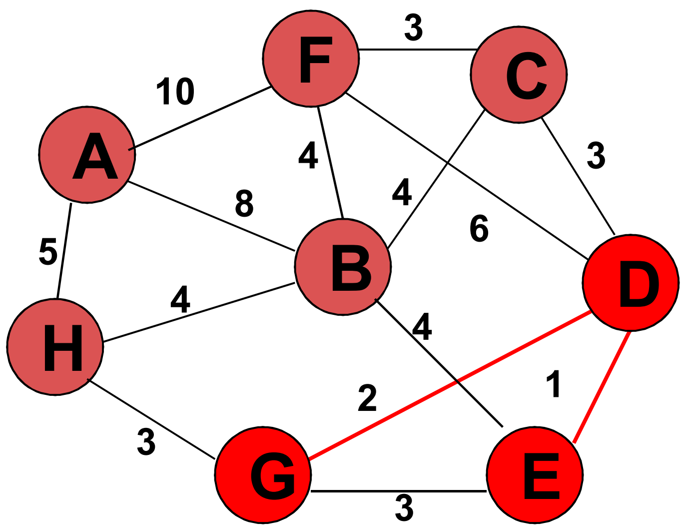
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



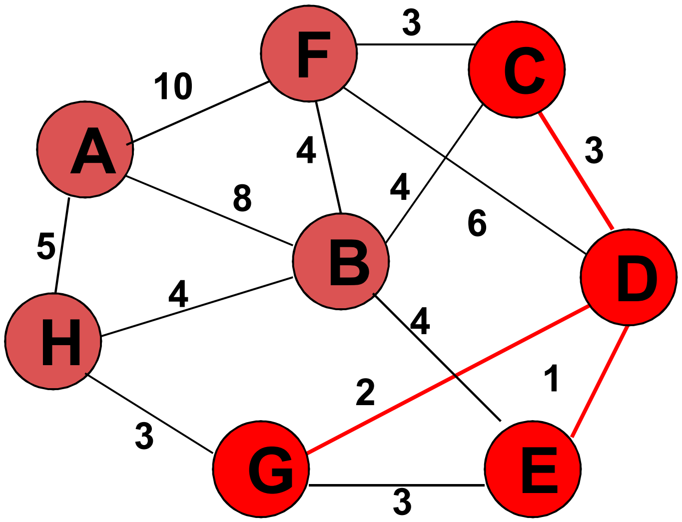
Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Accepting edge (E,G) would create a cycle

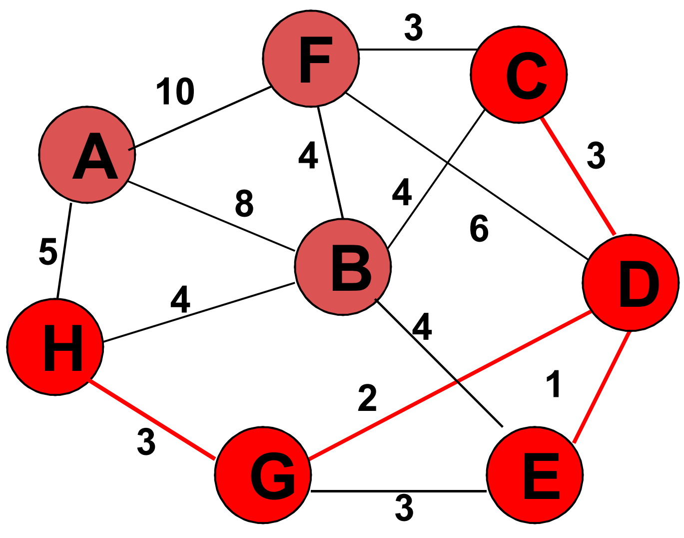
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

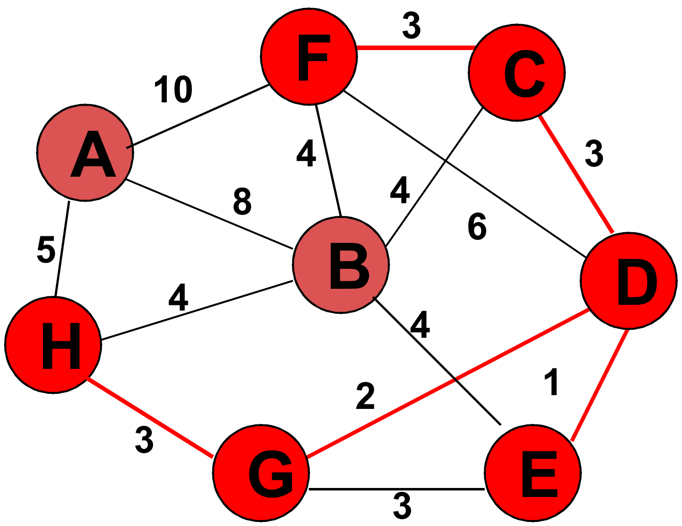
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

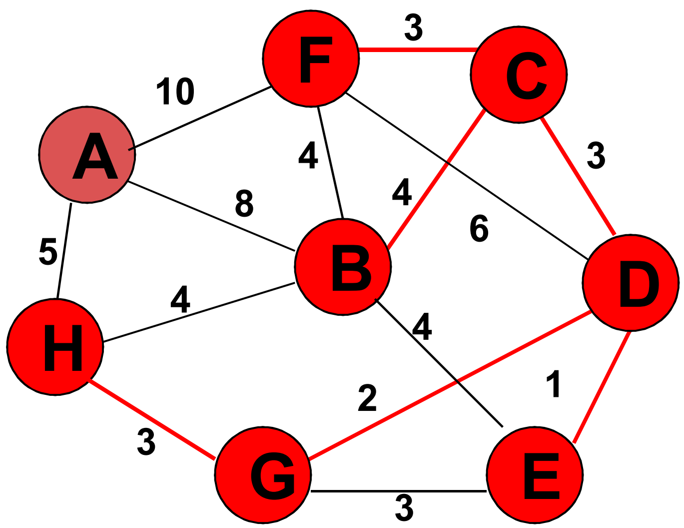
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

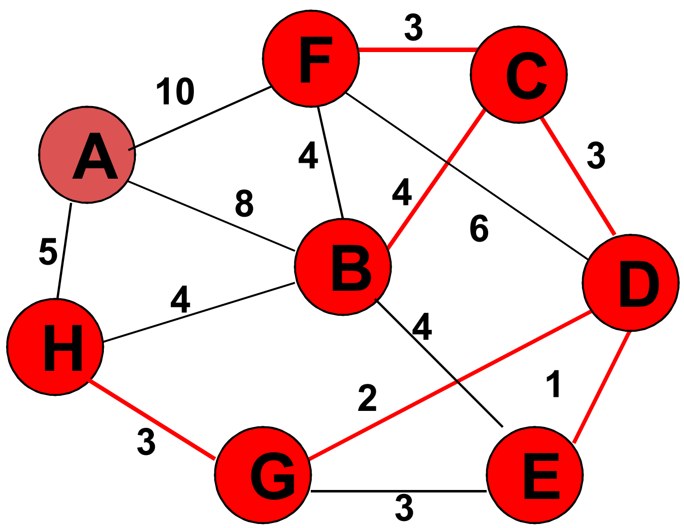
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

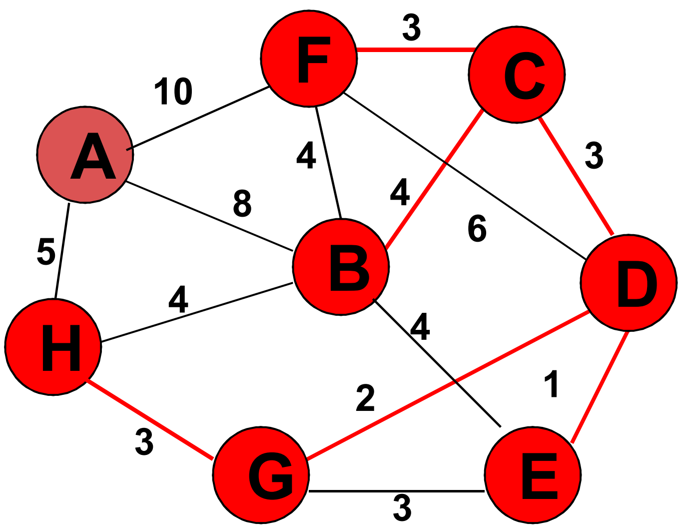
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

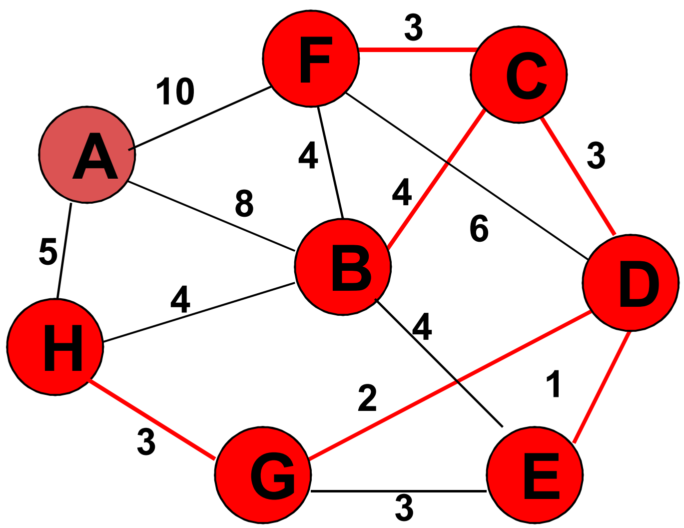


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

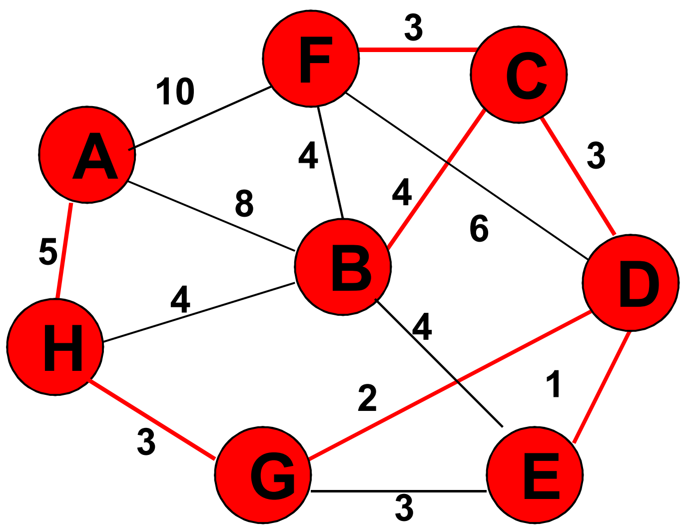
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

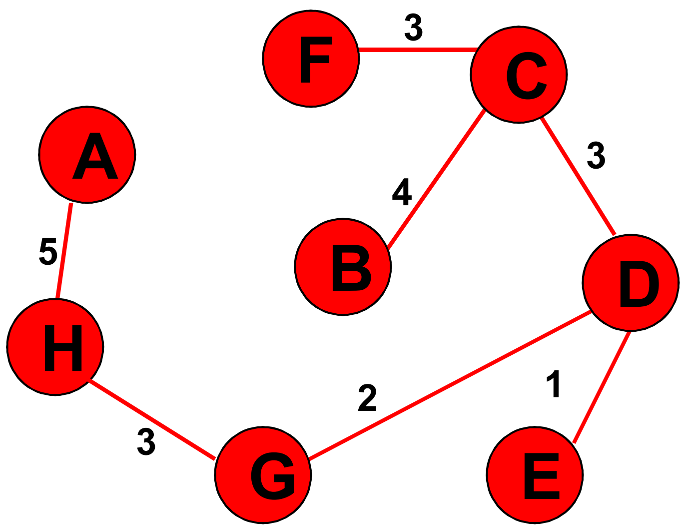
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

not considere d

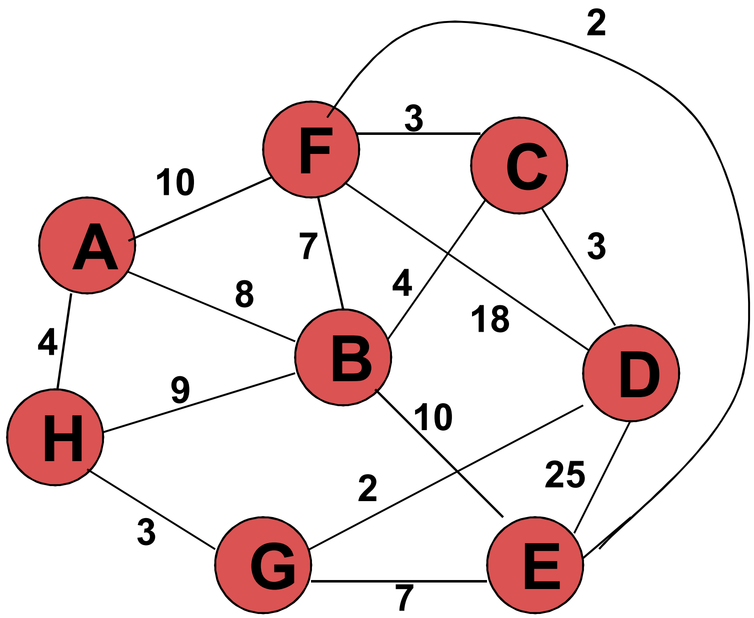
Done

Total Cost = $\Sigma\, d_v = 21$

Prim's Algorithm

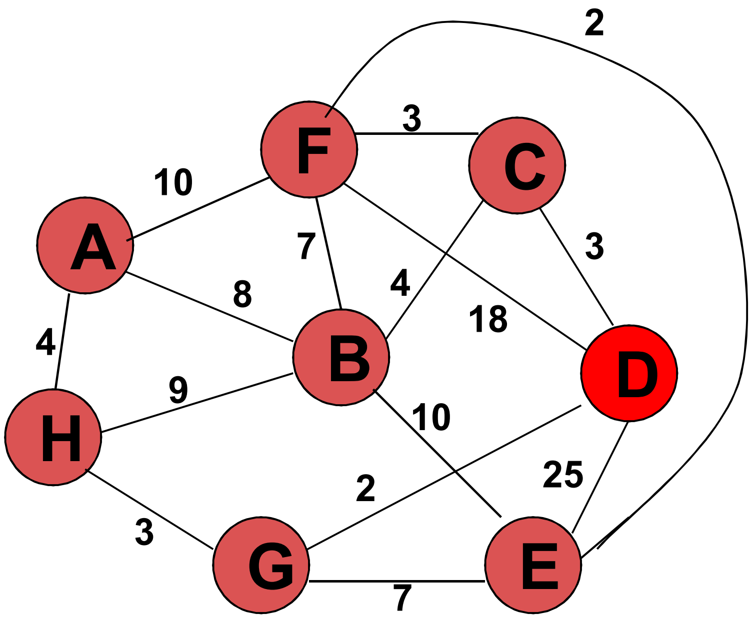
1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

Walk-Through



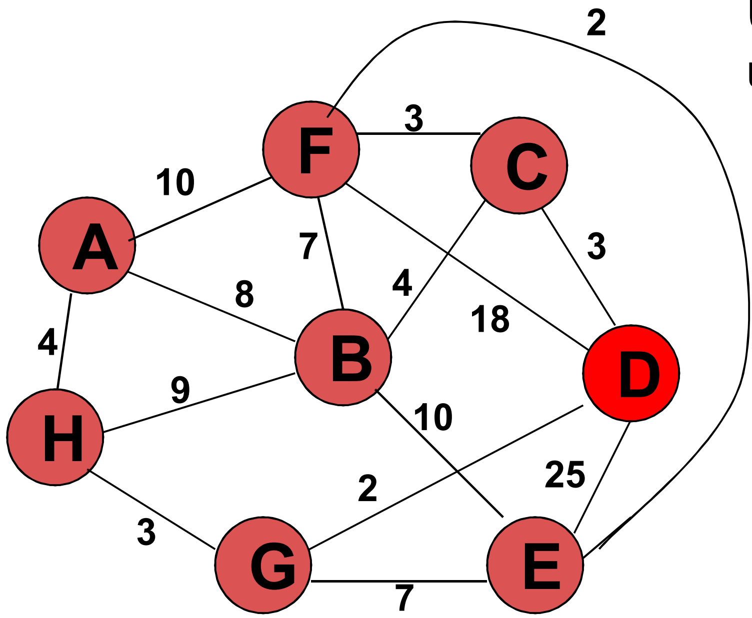
Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–



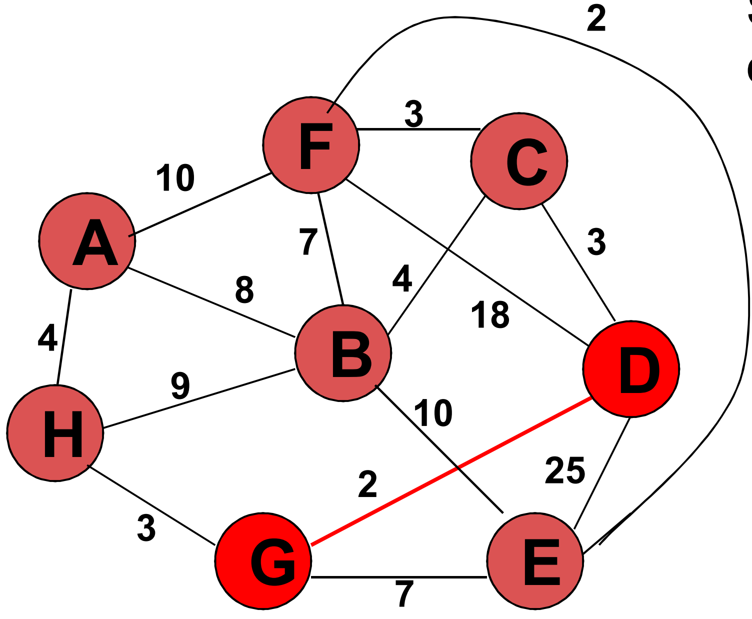
Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			



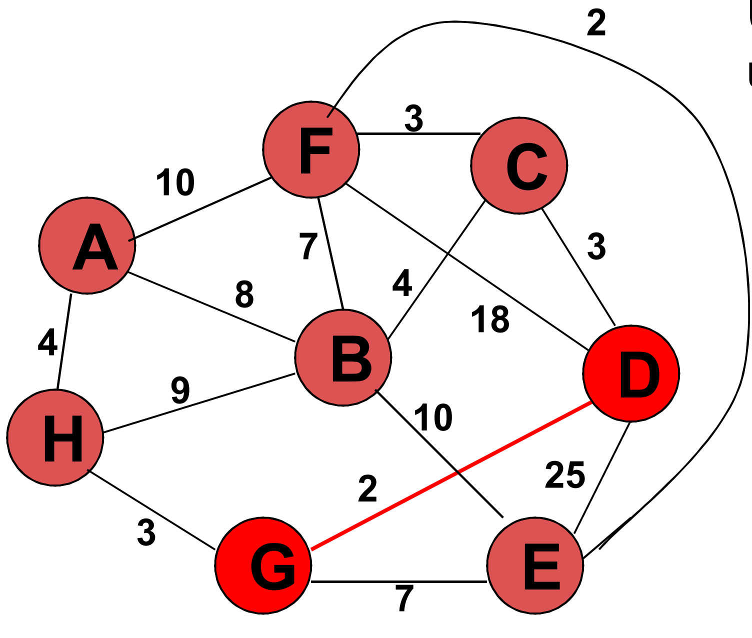
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			



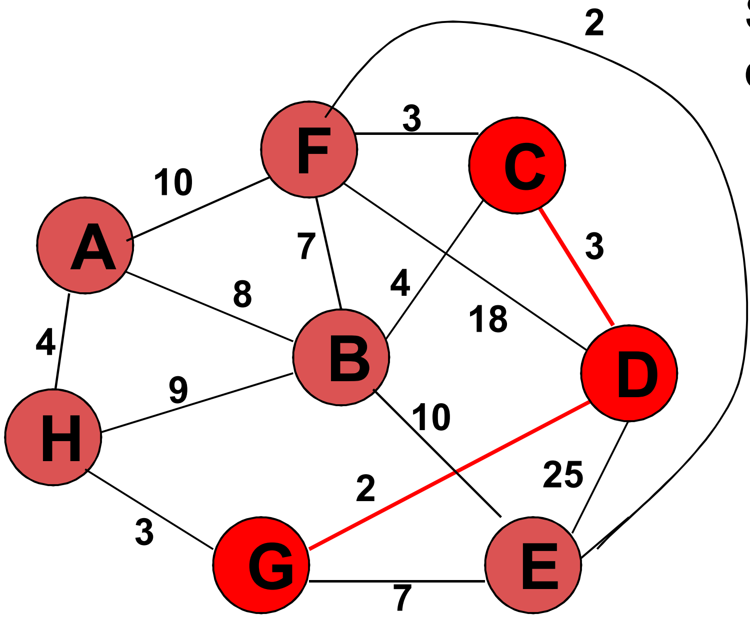
Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



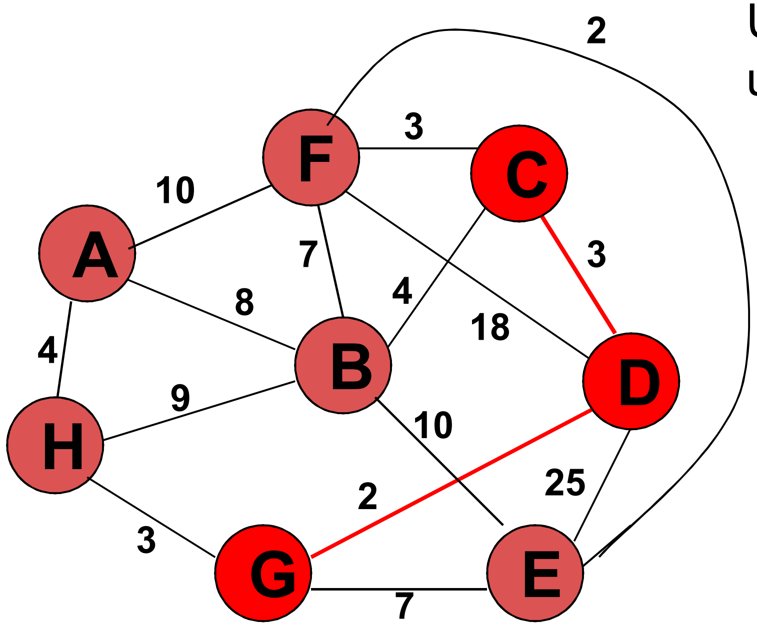
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



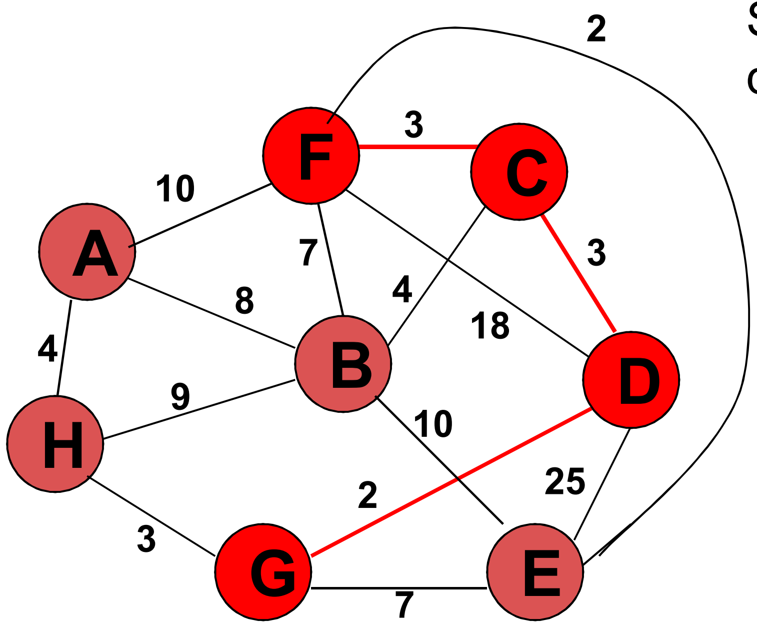
Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



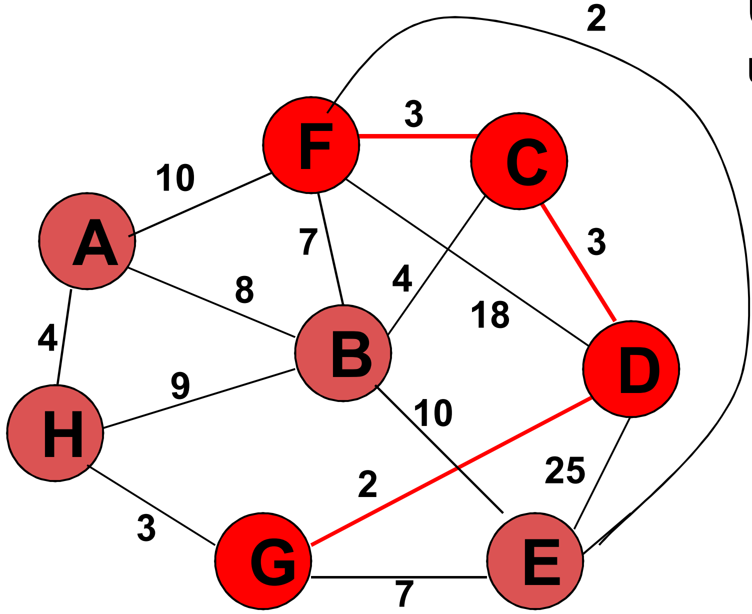
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



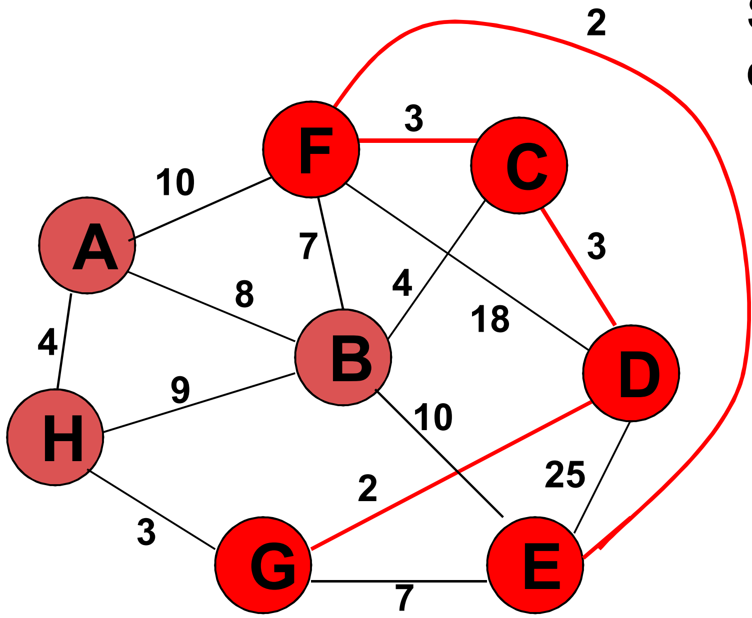
Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



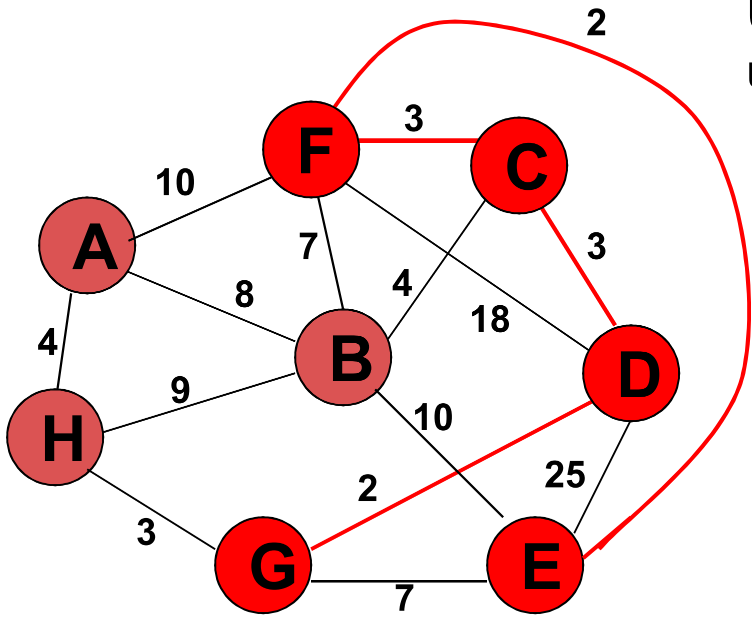
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

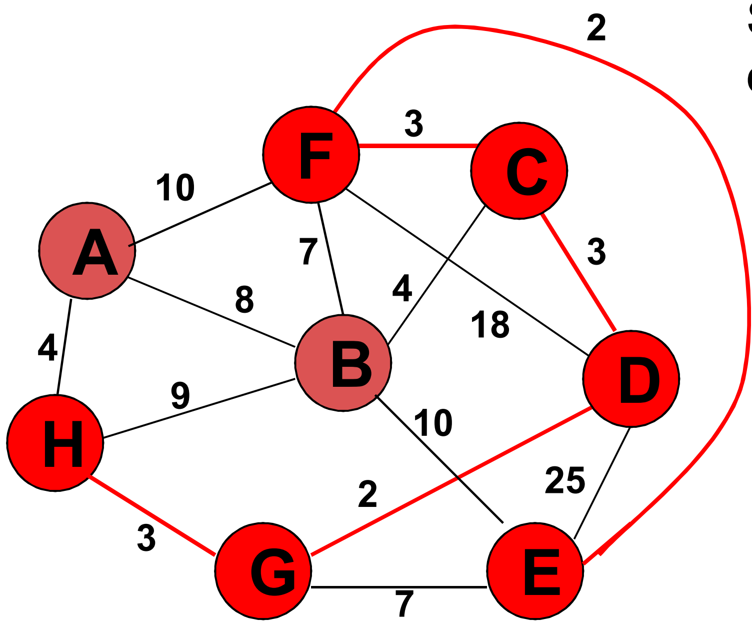
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

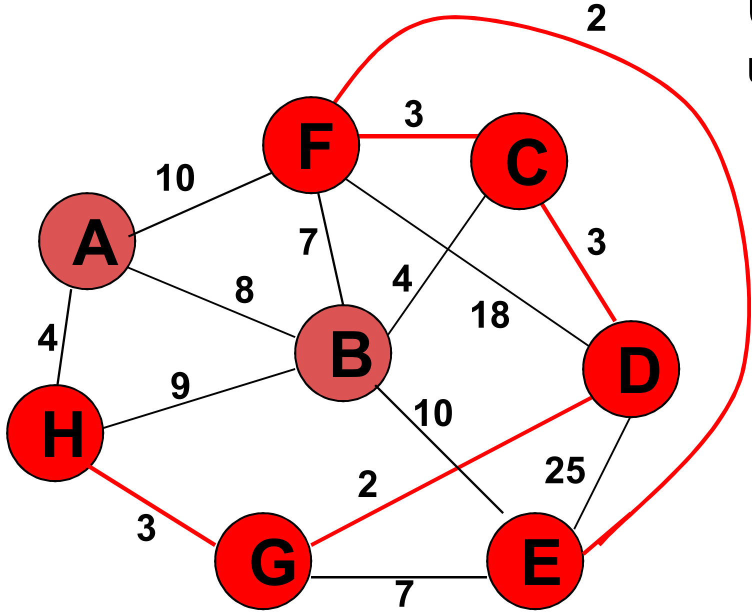
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



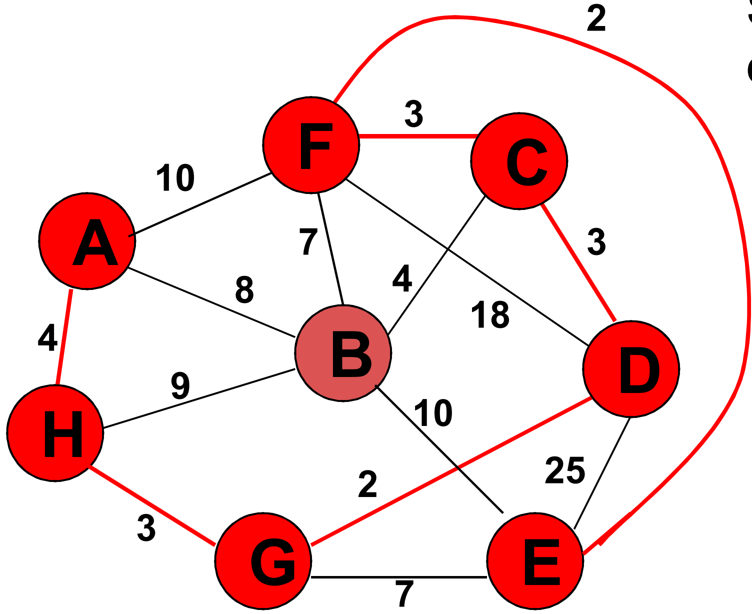
Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



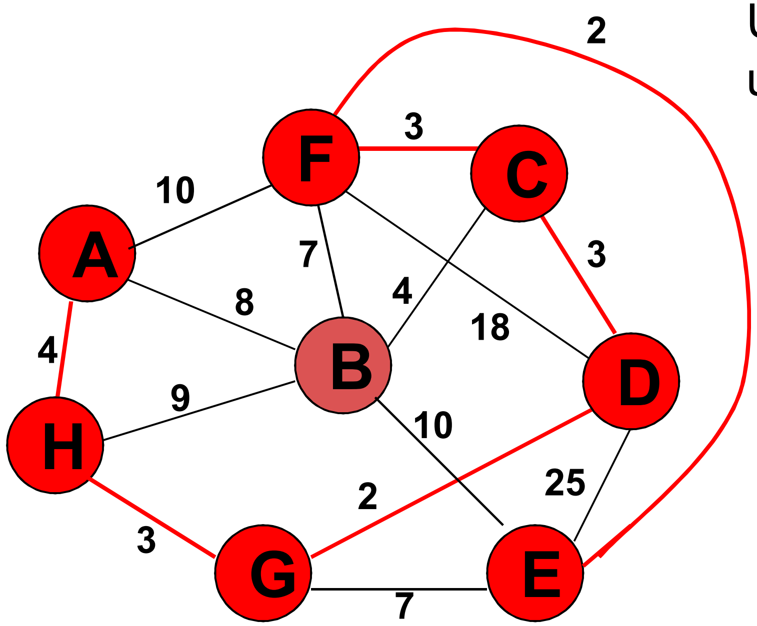
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

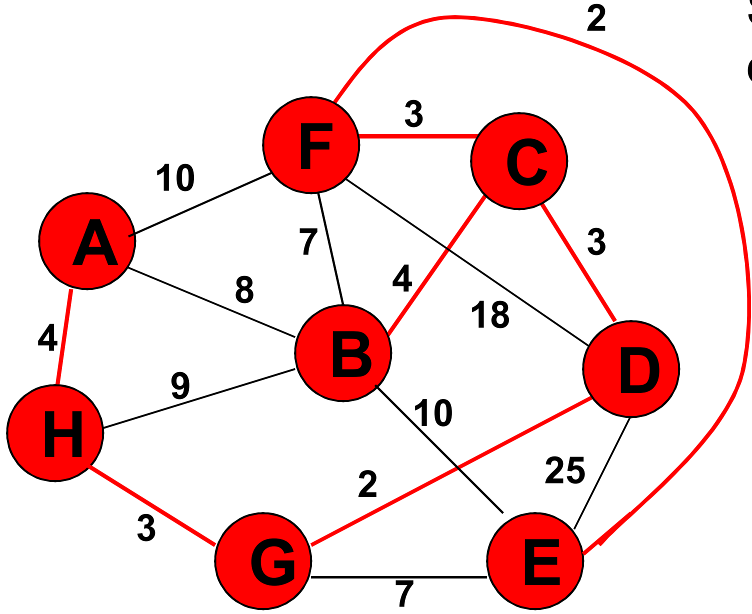
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

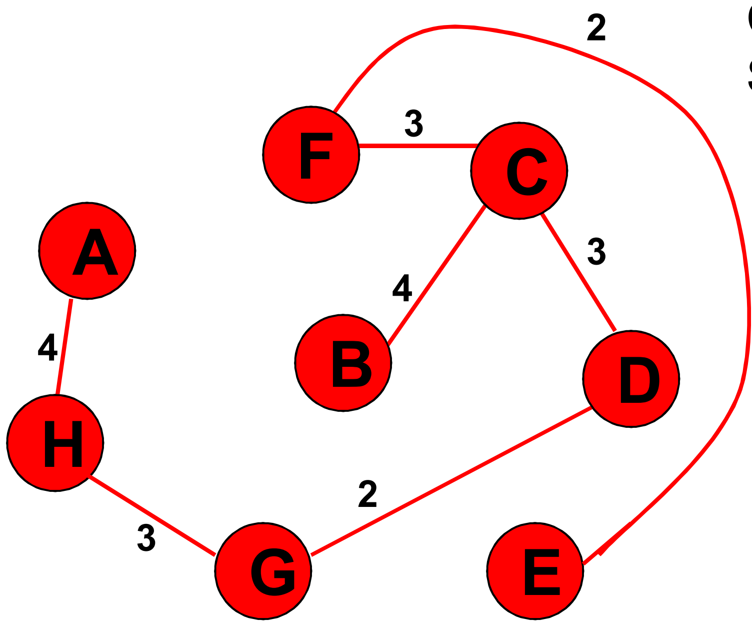
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\Sigma\, d_v$ = 21

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done

DIJKSTRA'S Algorithm

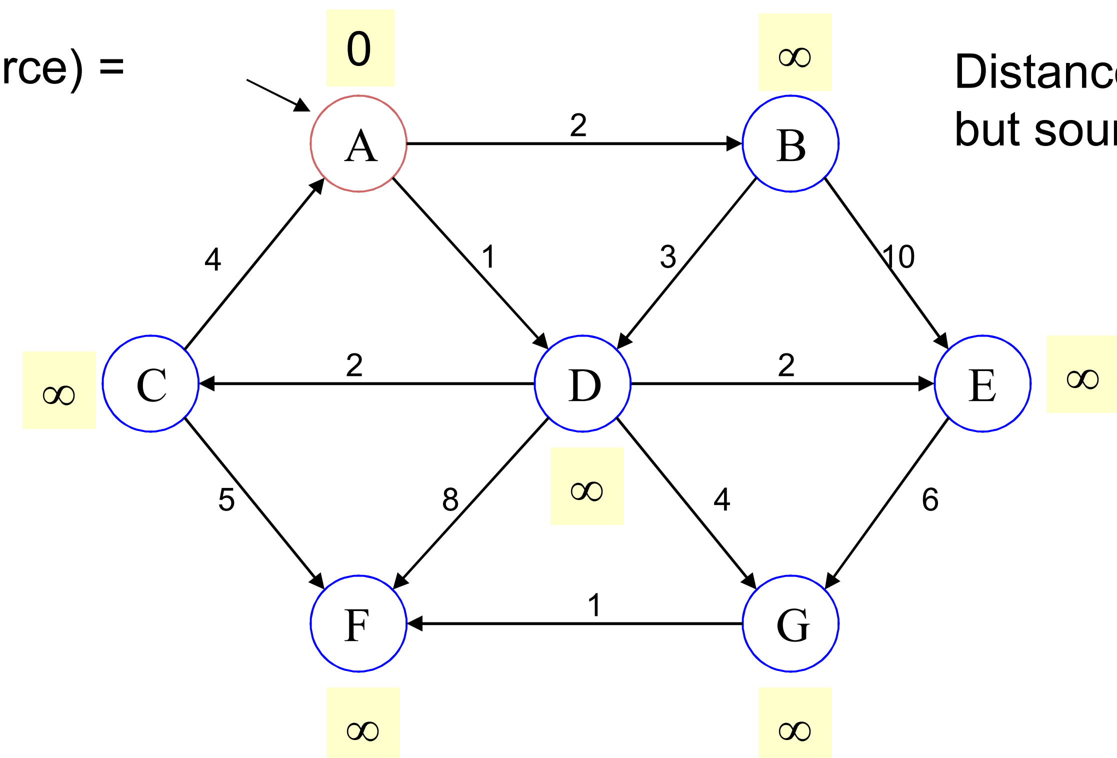
- Finds shortest (minimum weight) path between a particular pair of vertices in a weighted directed graph with nonnegative edge weights

- Dijkstra's algorithm is a greedy algorithm
  - make choices that currently seem the best
  - locally optimal does not always mean globally optimal

Example: Initialization

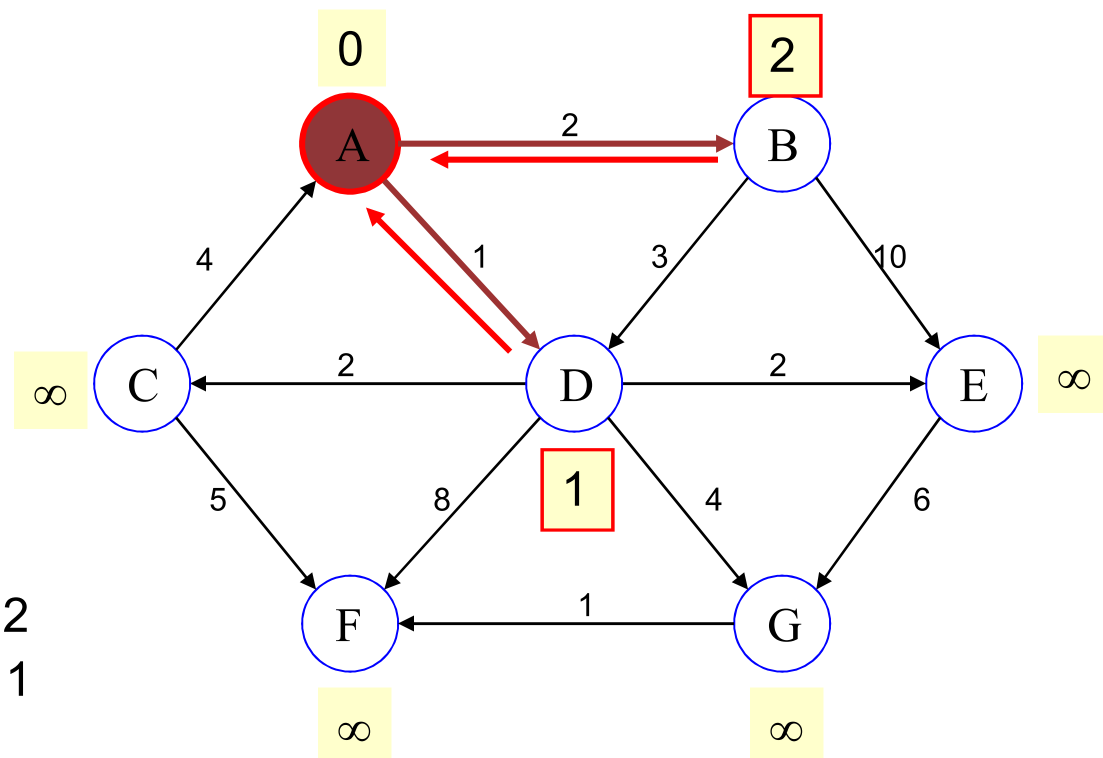
Distance(source) = 0

Distance (all vertices but source) = ∞



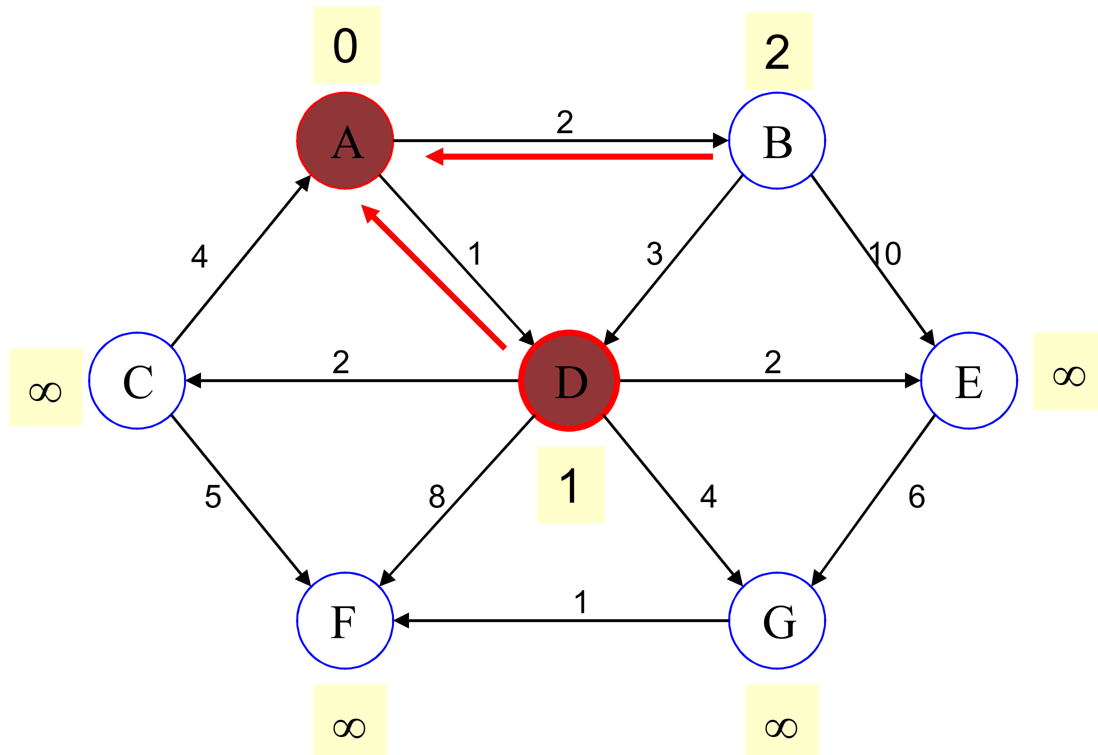
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



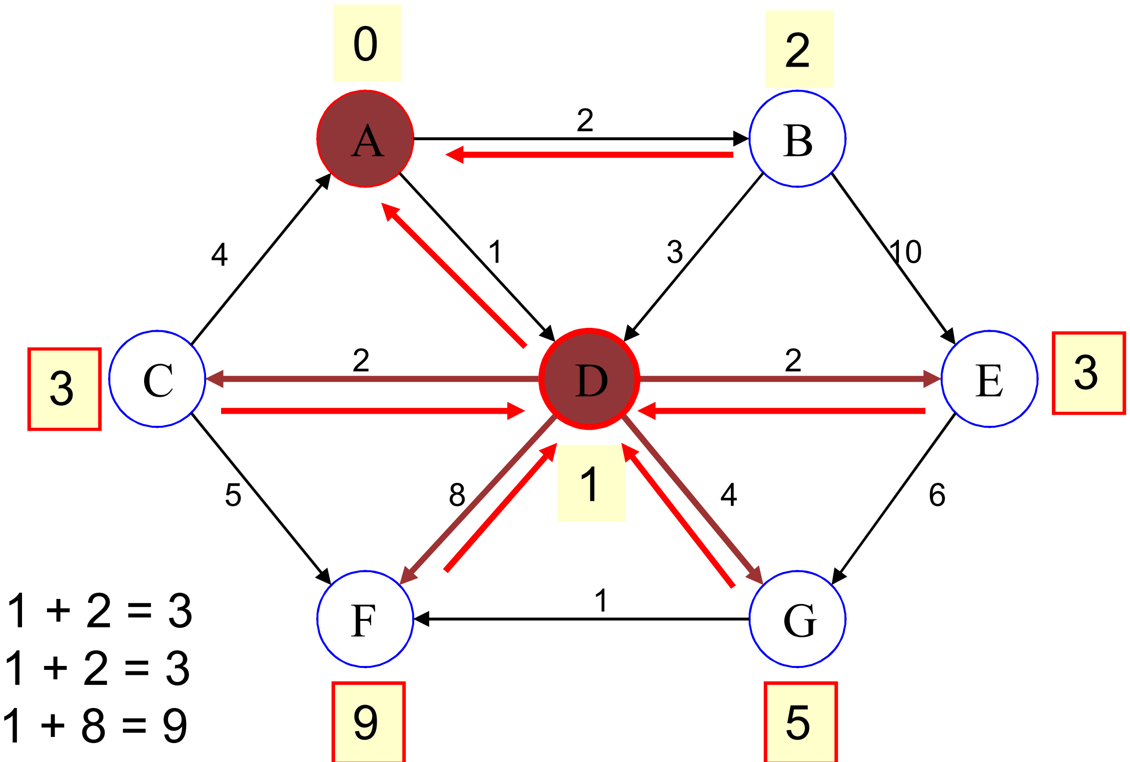
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

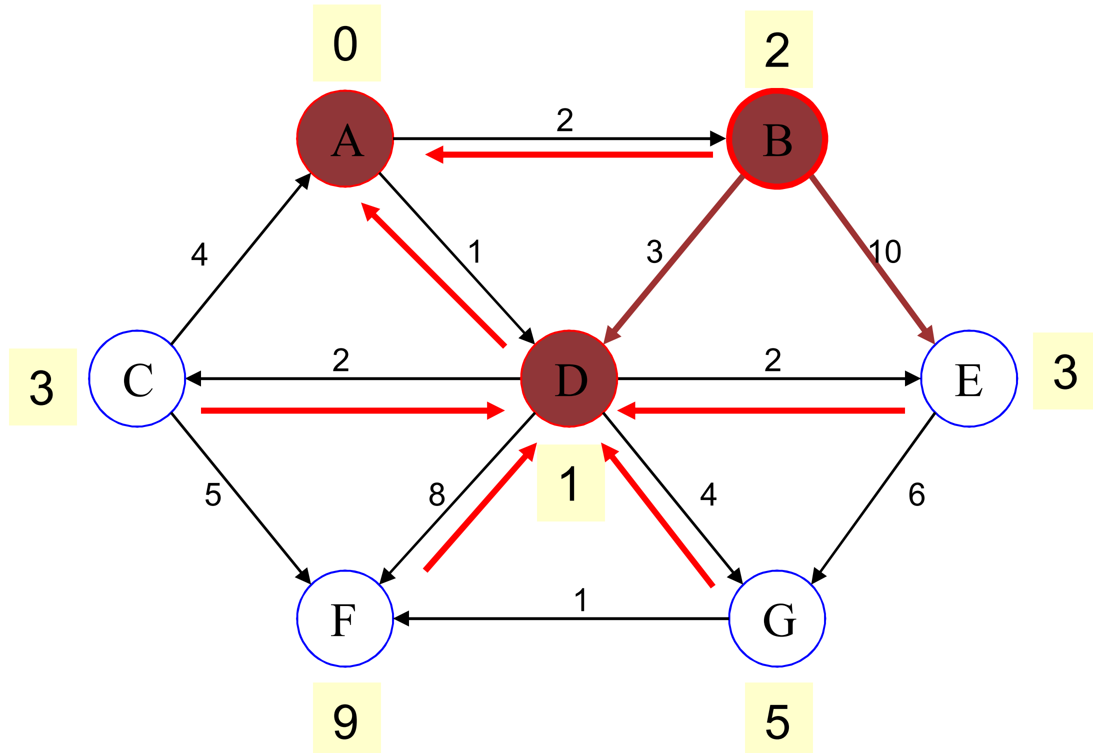
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

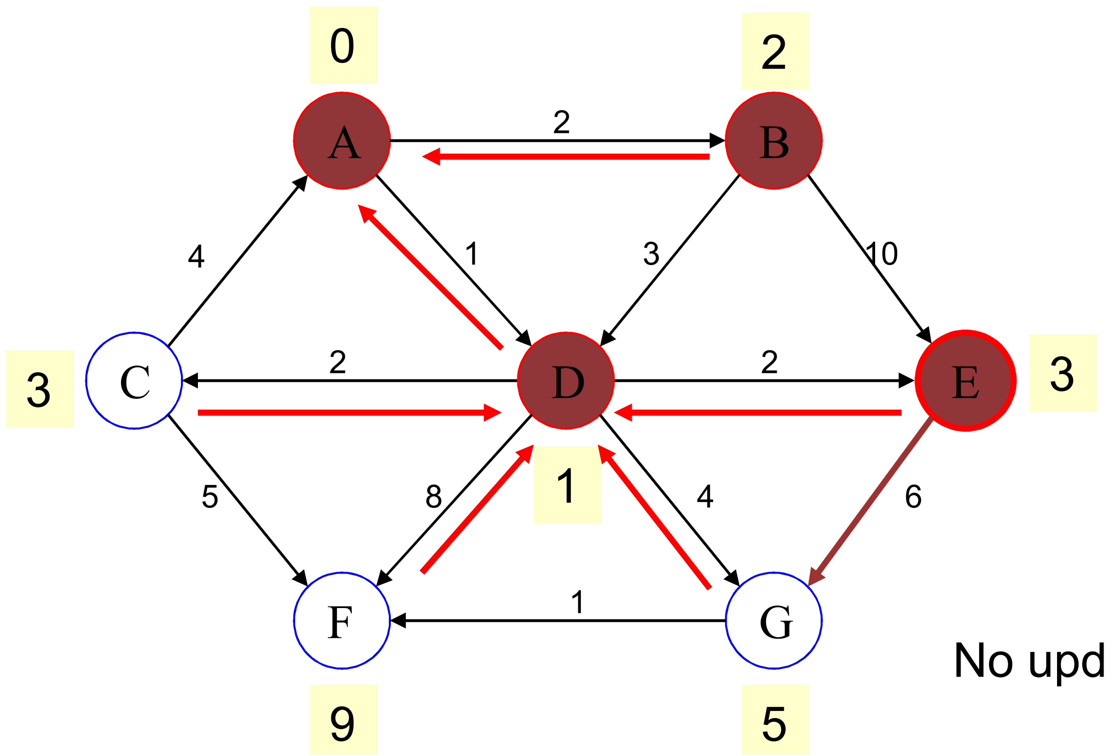
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

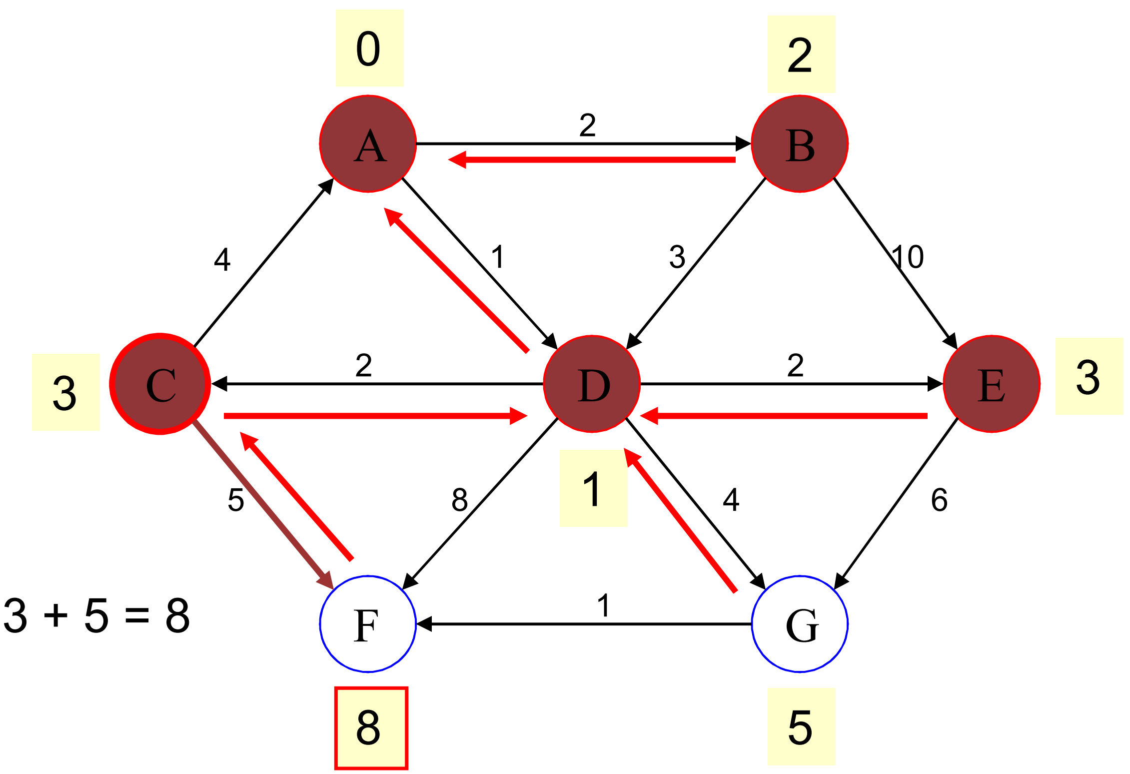
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

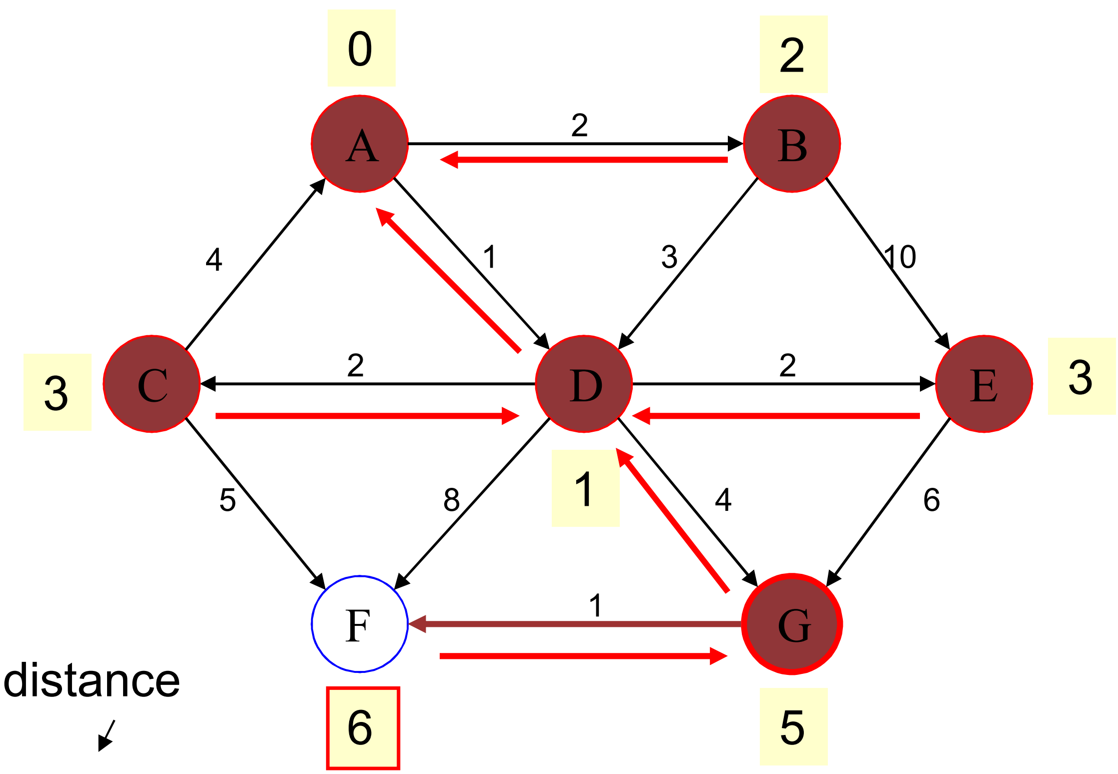
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

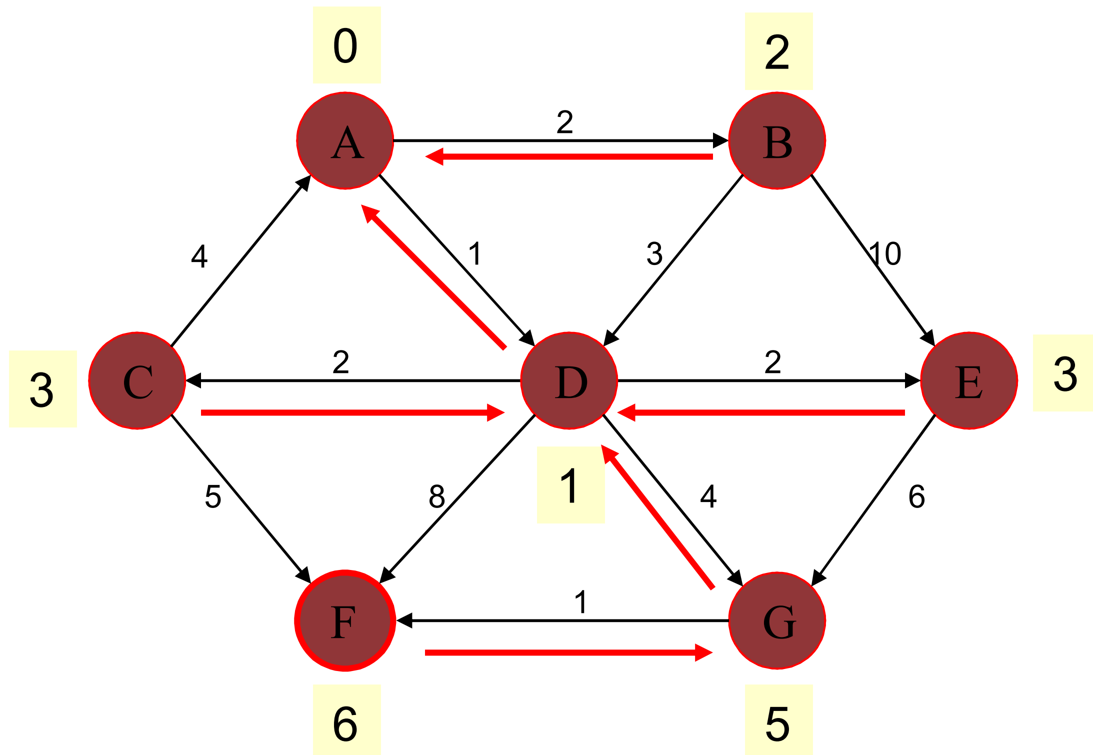
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

Distance(F) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (F) and update neighbors

Dijkstra Pseudocode

```
Dijkstra(v1, v2):
    for each vertex v:                          // Initialization
        v's distance := infinity.
        v's previous := none.
    v1's distance := 0.
    List := {all vertices}.

    while List is not empty:
        v := remove List vertex with minimum distance.
       mark v as known.
        for each unknown neighbor n of v:
            dist := v's distance + edge (v, n)'s weight.

            if dist is smaller than n's distance:
                n's distance := dist.
                n's previous := v.

    reconstruct path from v2 back to v1,
    following previous pointers.
```