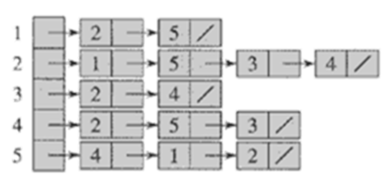
Graphs

- A graph G = (V, E)
  - V = set of vertices, E = set of edges
  - *Dense* graph: |E| close to $|V|^2$
  - *Sparse* graph: |E| much less than $|V|^2$
  - *Undirected graph:*
    - Edge (u,v) = Edge (v,u) and No self-loops
  - *Directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated u→v
  - A *weighted graph* associates weights with either the edges or the vertices

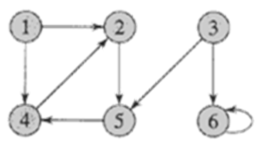
Representations of a Graph: Undirected Graphs

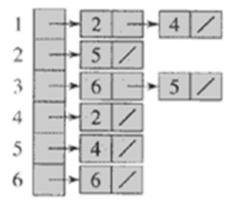


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	Adjacency List	**Adjacency Matrix**
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Representations of a Graph: Directed Graphs





Adjacency List

	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

Adjacency Matrix

	Adjacency List	Adjacency Matrix
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

3

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore systematically every vertex and every edge
- Discover much about structure of graph
- build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - might also build a *forest* if graph is not connected

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of discovered i.e. explored vertices across the *breadth* of the frontier
  - Discovers all vertices at distance k from s before discovering any vertices at distance k+1
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

Breadth-First Search

- To keep track progress BFS associate vertex "colors" to guide the algorithm
  - White vertices→ have not been discovered
    - All vertices start out white
  - Grey vertices → discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices→ discovered and fully explored
    - They are adjacent only to black and gray vertices
- All vertices start out white and may later become gray and then black
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

- Completely explore the vertices in order of their distance from u
- implemented using a queue
  - BFS algorithm uses first-in, first-out queue Q to manage the set of grey vertices

BFS Procedure

BFS search procedure assumes that input graph G(V,E) represented using adjacency List

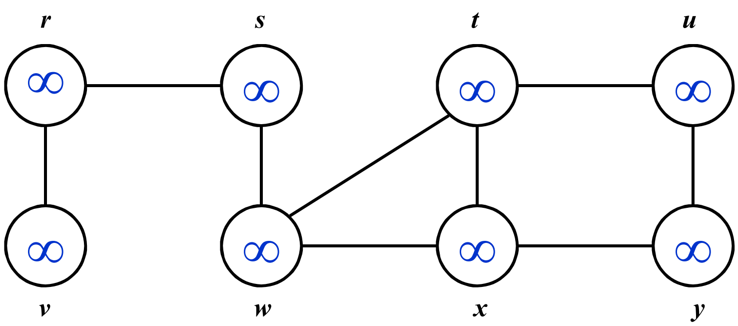
```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
      do color[u] ← WHITE      # store color of each vertex u
         d[u] ← ∞              # Store distance from s to u computed by algorithm
         π[u] ← NIL            # Store predecessor of u
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ Adj[u]
             do if color[v] = WHITE
                   then color[v] ← GRAY
                        d[v] ← d[u] + 1
                        π[v] ← u
                        ENQUEUE(Q, v)
         color[u] ← BLACK
```

Each vertex is enqueued at most once $\rightarrow O(V)$

Each entry in the adjacency lists is scanned at most once $\rightarrow O(E)$

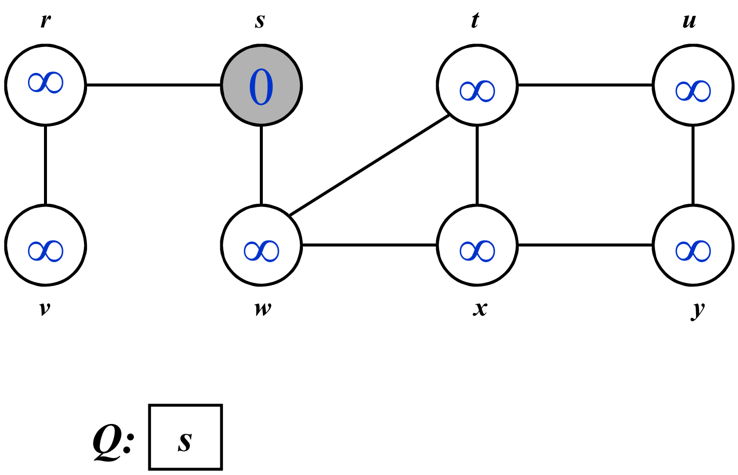
Thus run time is $O(V + E)$.

BFS: Example

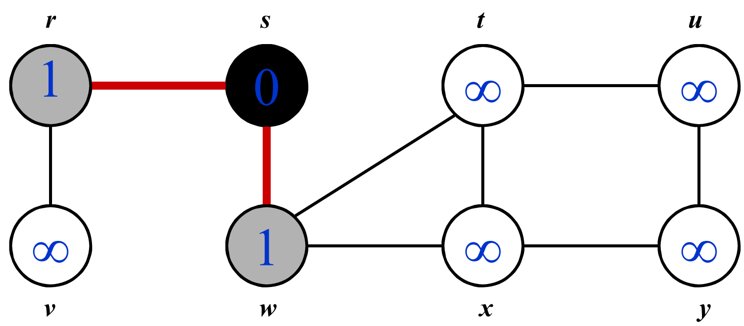


BFS search procedure assumes that input graph G(V,E) represented using adjacency List

Breadth-First Search: Example

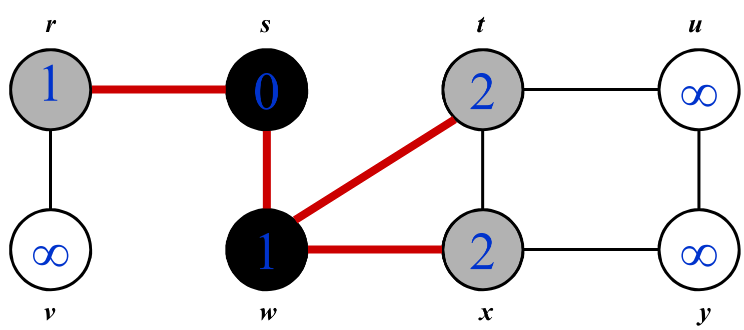


Breadth-First Search: Example



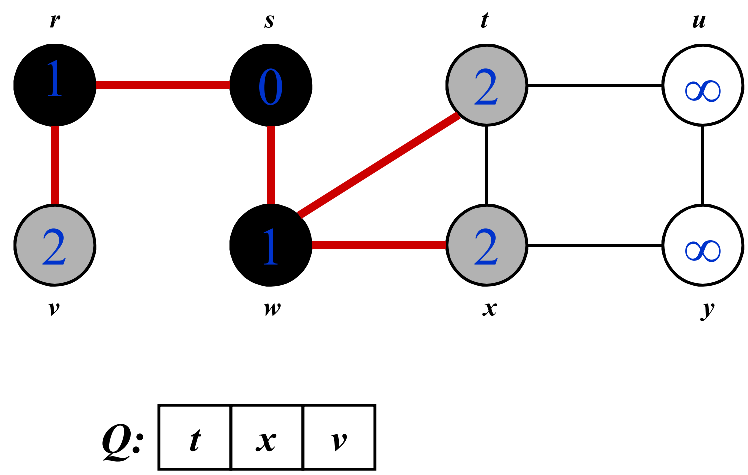
Q: | *w* | *r* |

Breadth-First Search: Example

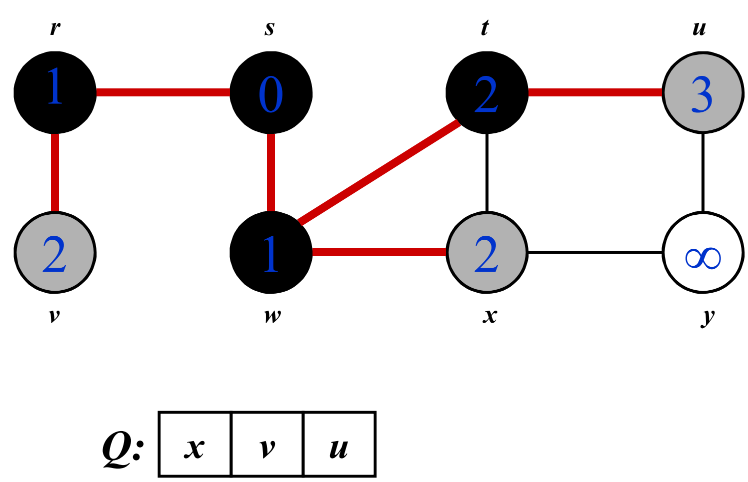


Q: | *r* | *t* | *x* |

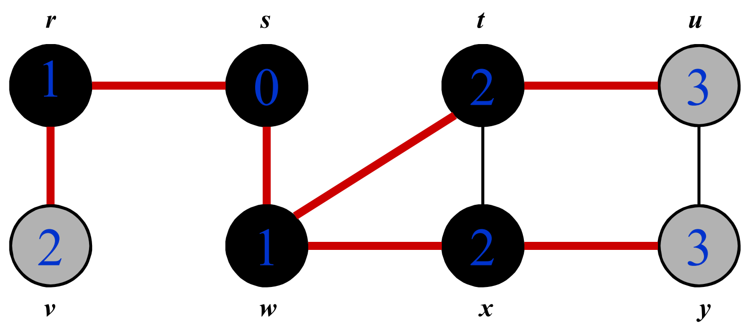
Breadth-First Search: Example



Breadth-First Search: Example



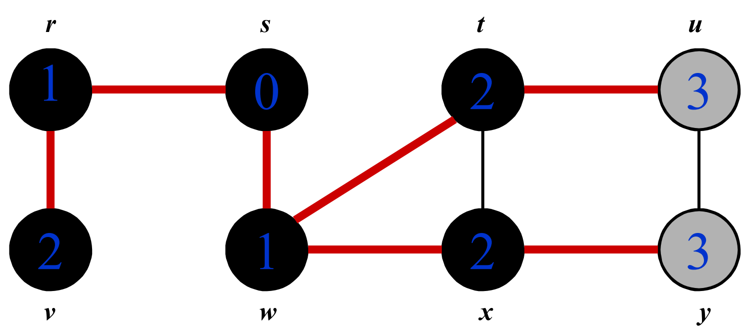
Breadth-First Search: Example



Q:

v	*u*	*y*

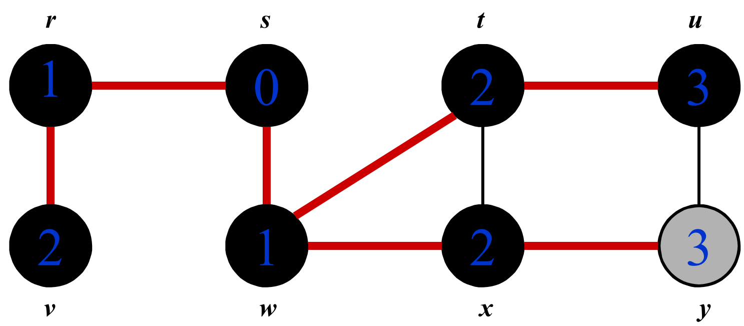
Breadth-First Search: Example



Q:

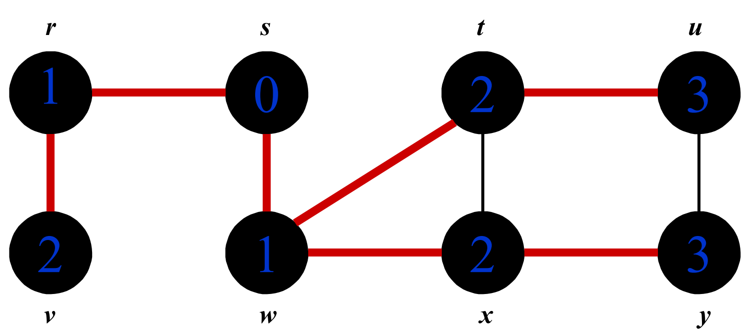
u	*y*

Breadth-First Search: Example



Q: *y*

Breadth-First Search: Example



Q: Ø

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

- DFS create depth first forest
- Maintains Two timestamps on each vertex
  - *d*[v] →records when v is first discovered (and grayed)
  - *f*[v] →records when search finishes examining v's adjacency list (and blackens v)
- Explore *every* edge, starting from different vertices if necessary
- As soon as vertex discovered, explore from it.
- Keep track of progress by colouring vertices:
  - Vertices initially colored white
  - Then colored gray when discovered (but not finished still exploring it)
  - Then black when finished (found everything that reachable from it)

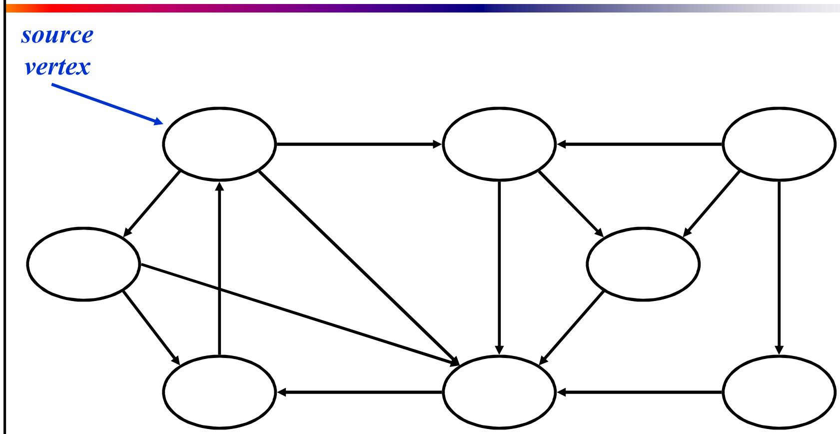
Depth-First Search Algorithm

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

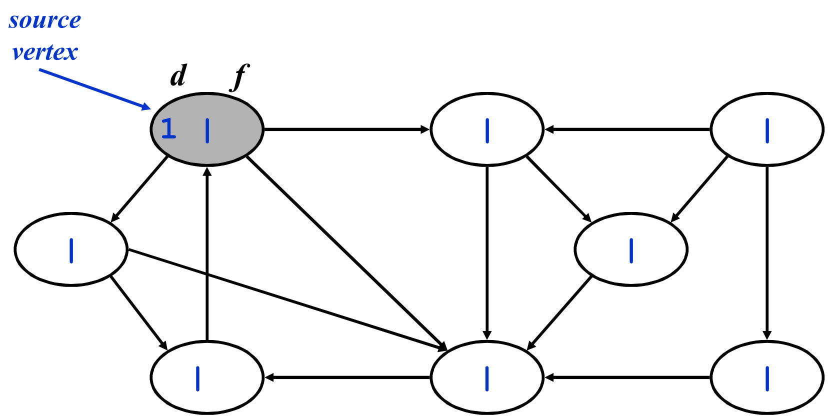
running time = O(V+E)

```
DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

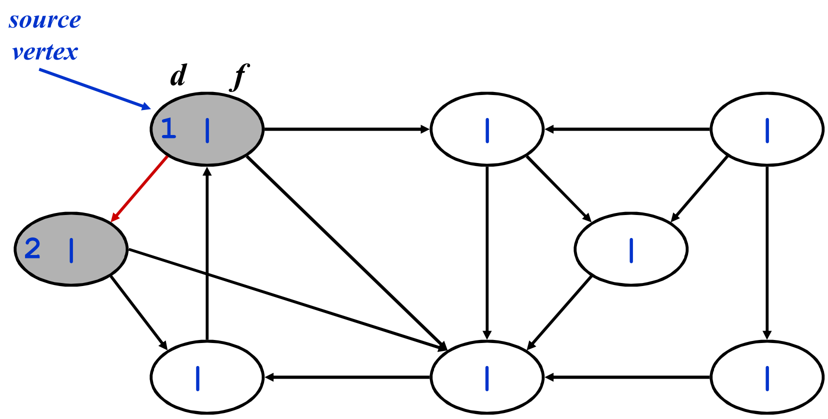
DFS Example



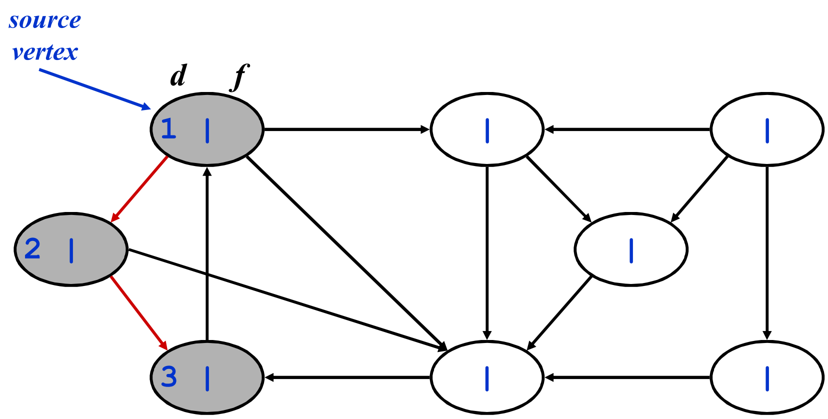
DFS Example



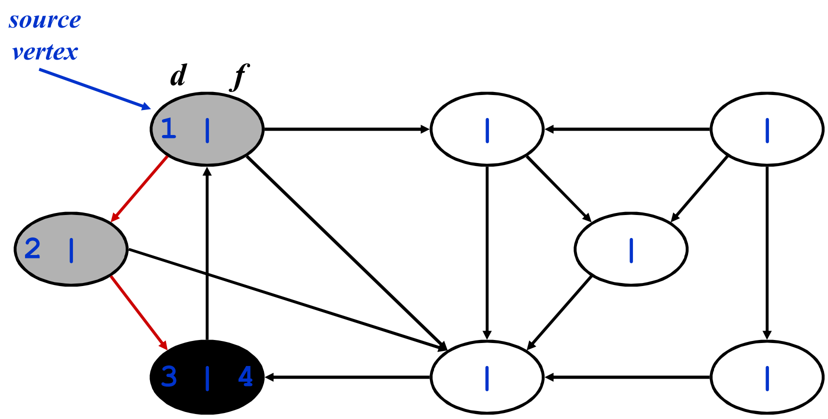
DFS Example



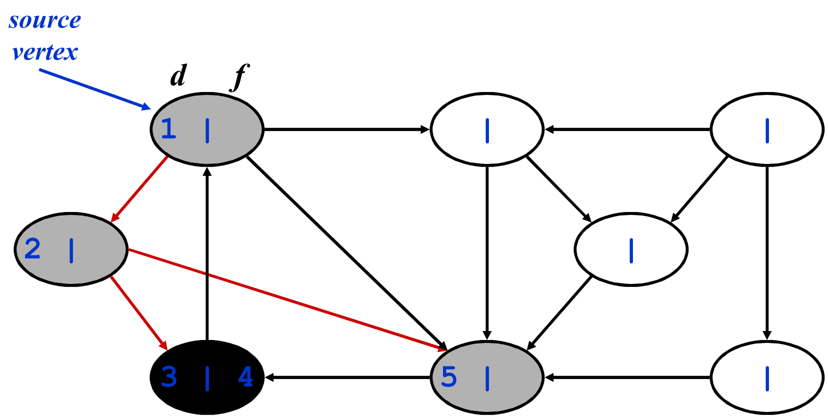
DFS Example



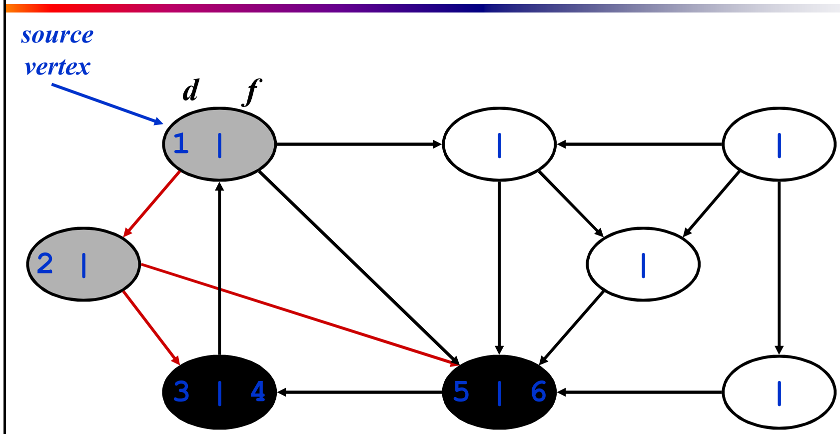
DFS Example



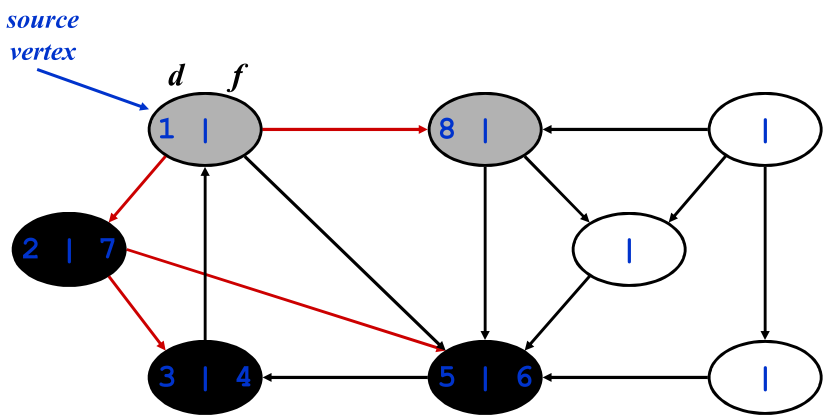
DFS Example



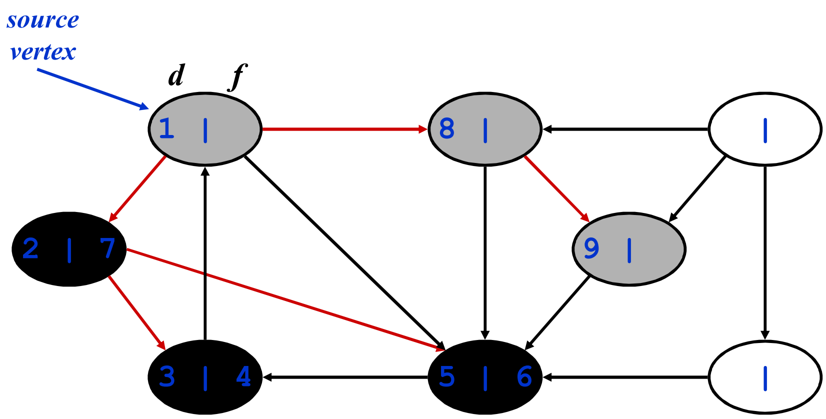
DFS Example



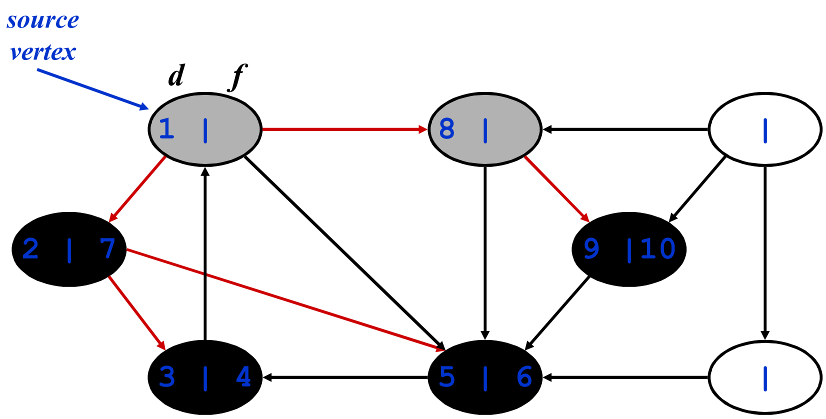
DFS Example



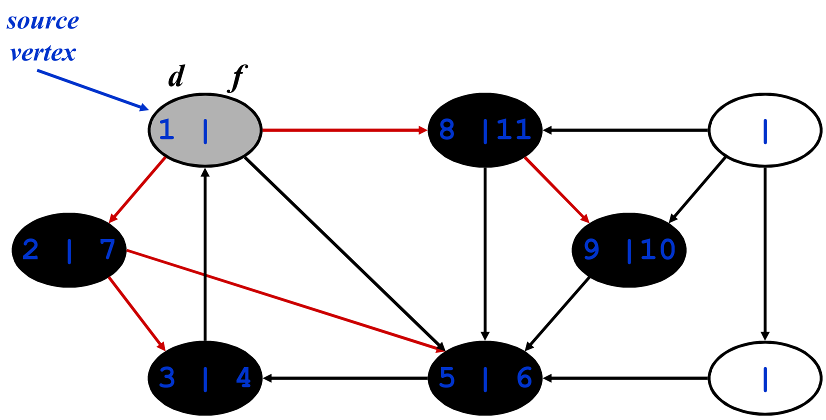
DFS Example



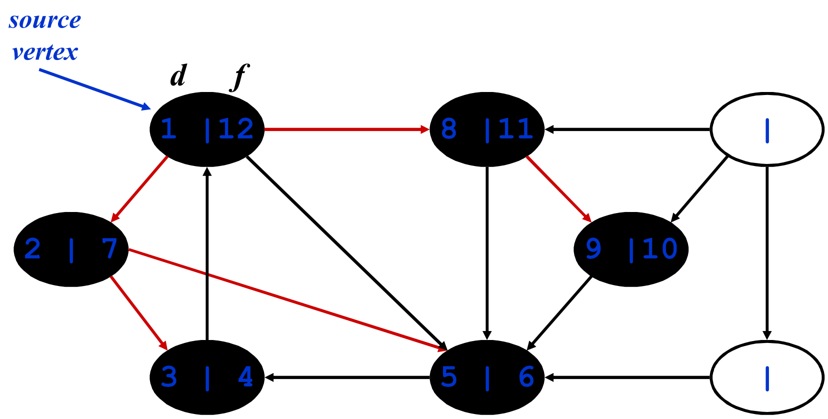
DFS Example



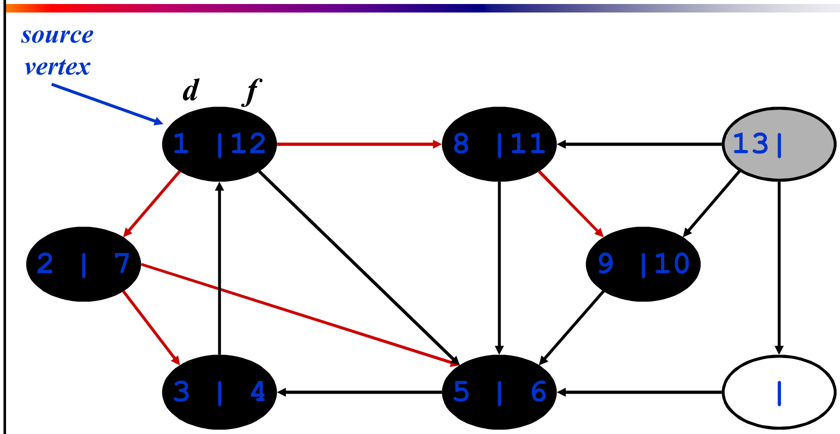
DFS Example



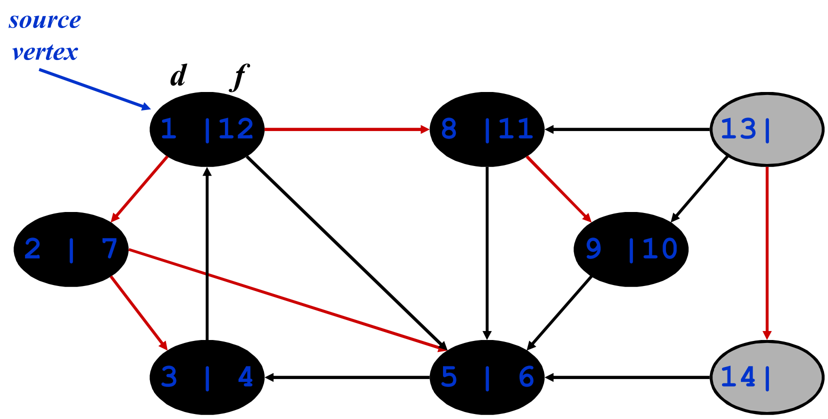
DFS Example



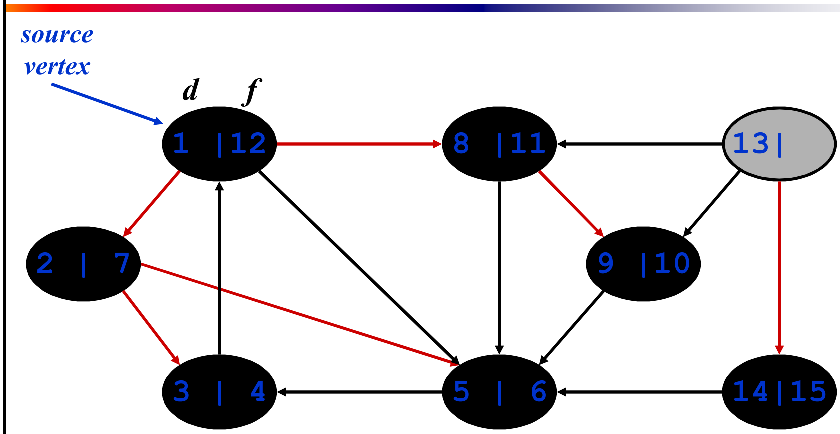
DFS Example



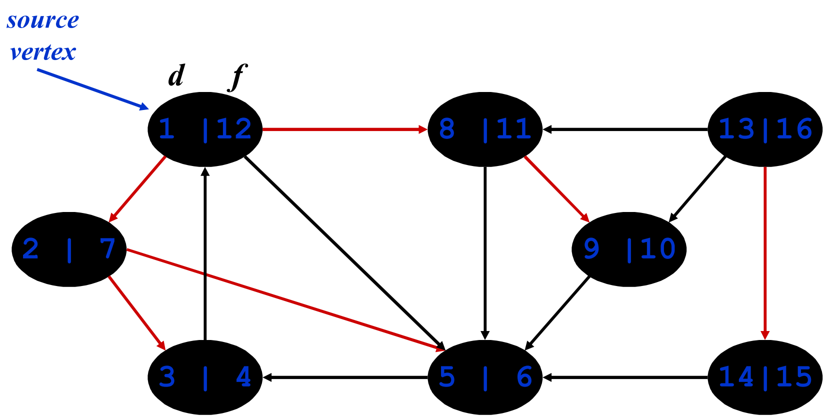
DFS Example



DFS Example



DFS Example



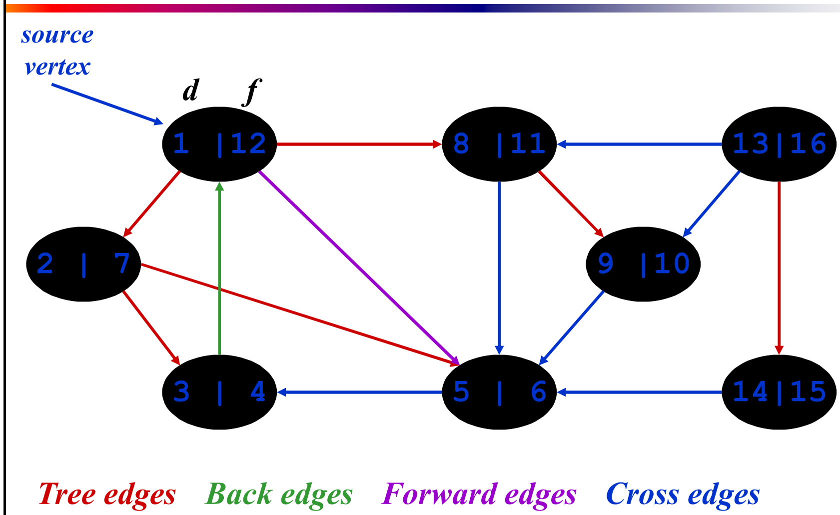
Depth-First Search

- Like BFS it does not recover shortest paths, but can be useful for extracting other properties of graph, e.g.,
  - Topological sorts
  - Detection of cycles
  - Extraction of strongly connected components

Classification of Edges

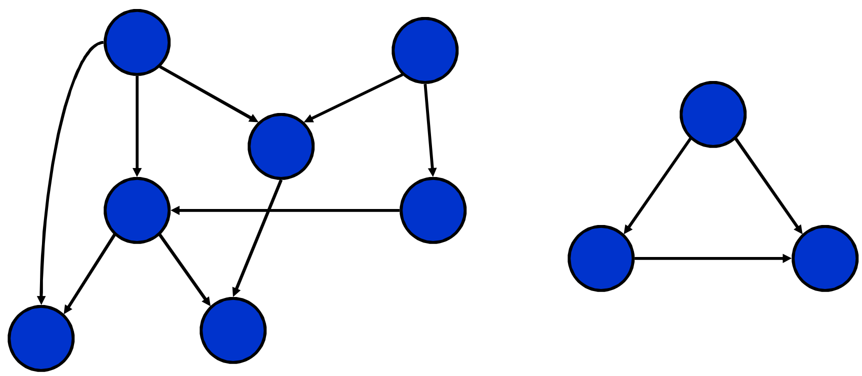
- Edge type for edge (u, v) can be identified when it is first explored by DFS.
- DFS algorithm can be modified to classify edges as it encounters them
- Classification is based on the **color of *v***
  - **White indicate a tree edge**: : encounter new (white) vertex
  - **Gray indicate a back edge**::from descendent to ancestor
    - Encounter a grey vertex (grey to grey)
  - **Black indicate a forward edge**:: from ancestor to descendent
    - non-tree edge
    - From grey node to black node
  - **Cross edge:** all other edges i.e. between a tree or subtrees
    - They can go between vertices in same/ different depth-first trees

DFS Example



Directed Acyclic Graphs

DAG is a directed graph with no directed cycles:



DAG Used in many applications to indicate precedence among events

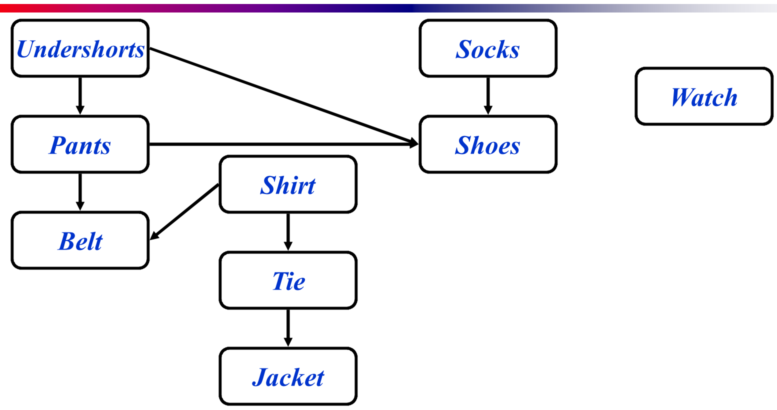
DFS and DAGs

- a directed graph G is acyclic if a DFS of G yields no back edges:
  - if G is acyclic, will be no back edges
    - But if a back edge it implies a cycle
  - if no back edges, G is acyclic

Topological Sort

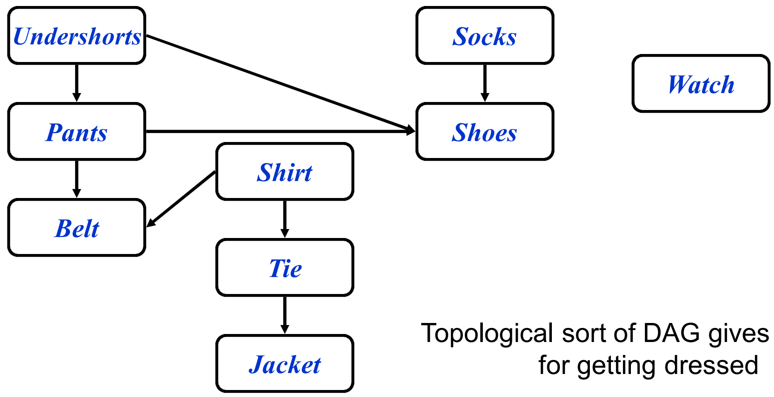
- *DFS used to perform topological sort of a DAG*
- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex u appears before vertex v *in the ordering* if edge $(u, v) \in G$
- Example: getting dressed

Getting Dressed

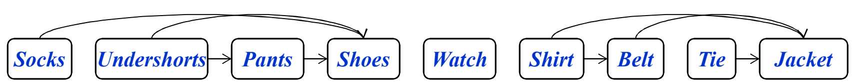


-directed edge (u,v) indicates that garment u must be put on before garment v
-certain garments put on before others
-other items may put on in any order

Getting Dressed



Topological sort of DAG gives an order for getting dressed

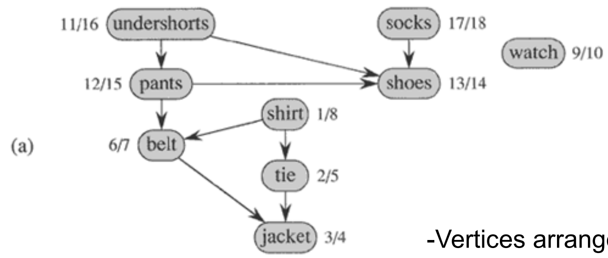


Topological Sort Algorithm

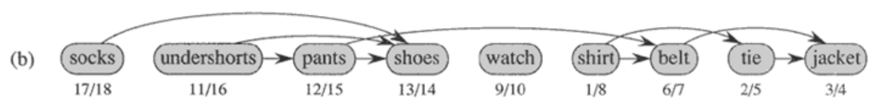
```
Topological-Sort(G)
{
1.Call DFS(G) to compute finishing times f(v)
  for each vertex v
2. As each vertex is finished, insert it onto
  the front of linked list
3.Return the linked list of vertices
}
```

Topological sort performed in Time: O(V+E)

Topologically sorted graph



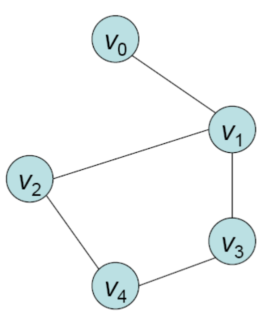
-Vertices arranged left to right
-All directed edges go from left to right



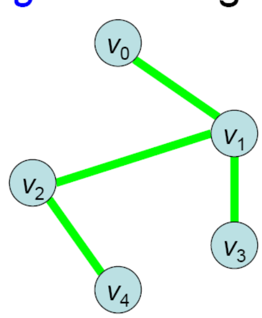
-topologically sorted vertices appears in reverse order of their finishing times

Spanning Tree

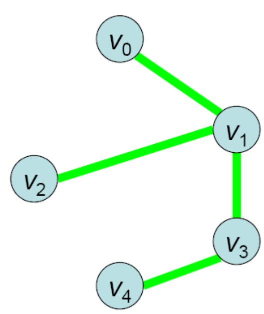
- A ***spanning tree*** (ST) of an *undirected* graph is a *tree* which contains *all vertices* and *some edges* of the graph.



Graph *G*



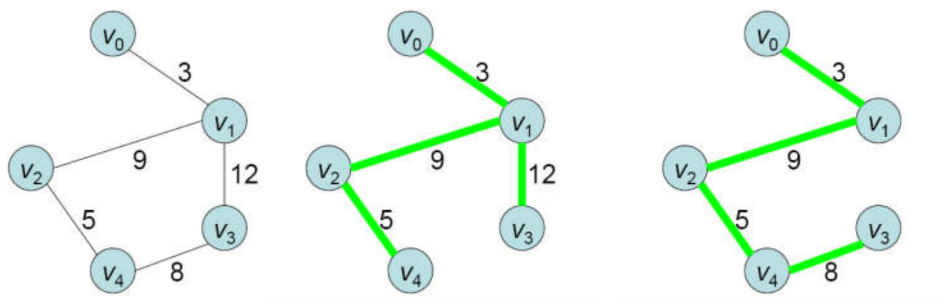
A spanning tree of *G*



Another spanning tree of *G*

Minimum Spanning Tree

- A ***minimum spanning tree*** (MST) of an *undirected weighted* graph is a spanning tree whose sum of all weights is minimum.

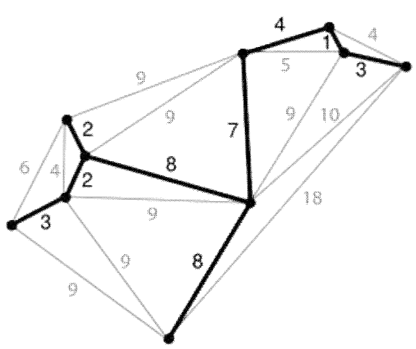


Graph G

A spanning tree of G (total weight = 3+9+12+5 = 29)

A MST of G (total weight = 3+9+5+8 = 25)

Real Life Application



- ISP company laying LAN cable
- graph represents which houses are connected by those LAN cables
- A *spanning tree* for this graph would be a subset of those paths that has no cycles but still connects to every house
- might be several spanning trees possible.
- *minimum spanning tree* would be one with the lowest total cost

MST Algorithm

- Two common algorithms for finding MSTs.
  - ***Kruskal's algorithm***
    - From "forest" to tree
  - ***Prim's algorithm***
    - Build tree to span all vertices

- Both uses a specific rule to determine a **safe edge**
- **Safe edge**: an edge which can be added to a subset of some minimum spanning tree without violating invariant i.e. determined edge is also a subset of a minimum spanning tree

Generic_MST(G, w)
{ A← { }
While A does not form a spanning tree
DO find an *edge(u,v) that is safe for A*
A←A ∪{(u, v)}

Kruskal's Algorithm

- Increasingly sort the edges by weights.
- For each edge *e* in sorted order
  - If *e* does not form a cycle with the already picked edges, then
    - Pick *e*. (Done when *n* - 1 edges are picked.)
  - Else
    - Discard *e*.
- If fewer than *n* - 1 edges are picked, then
  - Graph is disconnected. No MST exists.

Kruskal's Algorithm

KRUSKAL(G(V, E), w)

A ← { }   → Set A will finally contains the edges of the MST
***for** each vertex v in V[G]*
  ***do** MAKE-SET(v)*
sort edges of E into nondecreasing order by weight w
***for** each edge (u, v) in E taken in nondecreasing order by weight from the sorted list*
  ***do if** FIND-SET(u) != FIND-SET(v) //determining whether u ,v belong to same tree*
    ***then** A ← A ∪{(u, v)}*
      UNION(u, v)
***return** A*

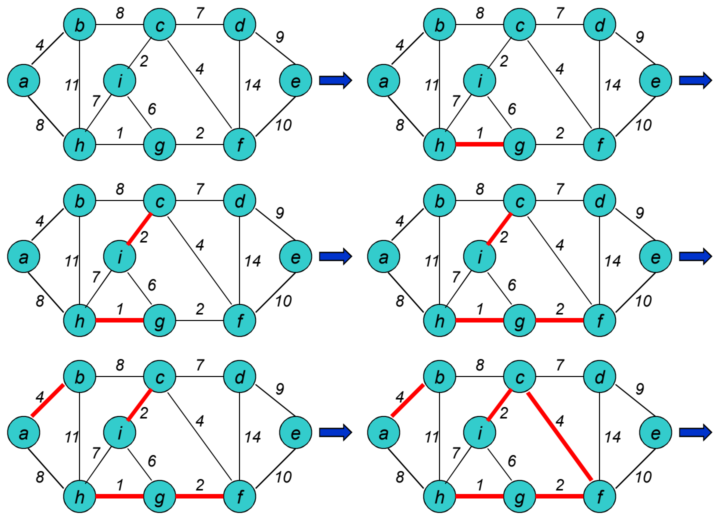
--G is a connected graph
--w is edge weights

Running Time: **O(E lg V)**

Make_SET(v): Create a new set whose only member is pointed to by **v**.
FIND_SET(v): Returns a representative element to the set

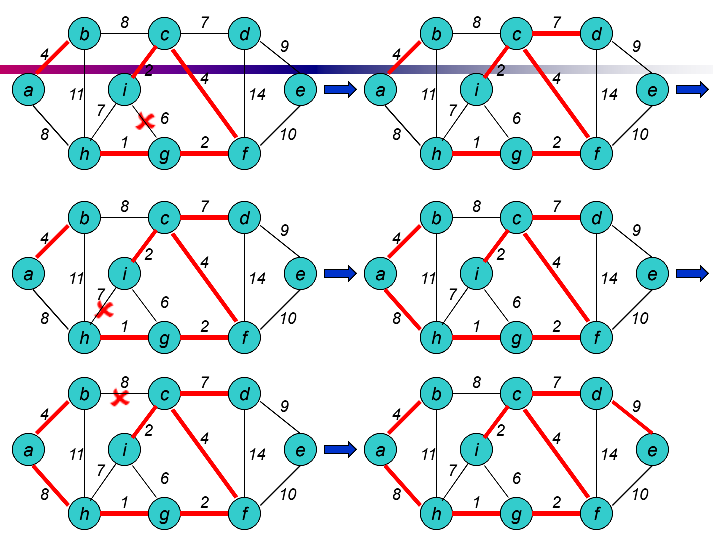
Kruskal's Algorithm

Edge	Weight	
<h, g>	1	✓
<c, i>	2	✓
<g, f>	2	✓
<a, b>	4	✓
<c, f>	4	✓
<g, i>	6	
<c, d>	7	
<h, i>	7	
<a, h>	8	
<b, c>	8	
<d, e>	9	
<e, f>	10	
<b, h>	11	
<d, f>	14	



Kruskal's Algorithm

Edge	Weight	
<h, g>	1	✓
<c, i>	2	✓
<g, f>	2	✓
<a, b>	4	✓
<c, f>	4	✓
<g, i>	6	✗
<c, d>	7	✓
<h, i>	7	✗
<a, h>	8	✓
<b, c>	8	✗
<d, e>	9	✓
<e, f>	10	
<b, h>	11	
<d, f>	14	



Algorithm will stop here since there are already (n-1) edges found.

Prim's Algorithm

- Initialize the current tree *T* to have any one vertex.
- While *T* has fewer than *n* - 1 edges
  - Pick the edge around *T* whose weight is the smallest and does not form a cycle.
  - (When no such edge is found, graph is disconnected and no MST exists.)

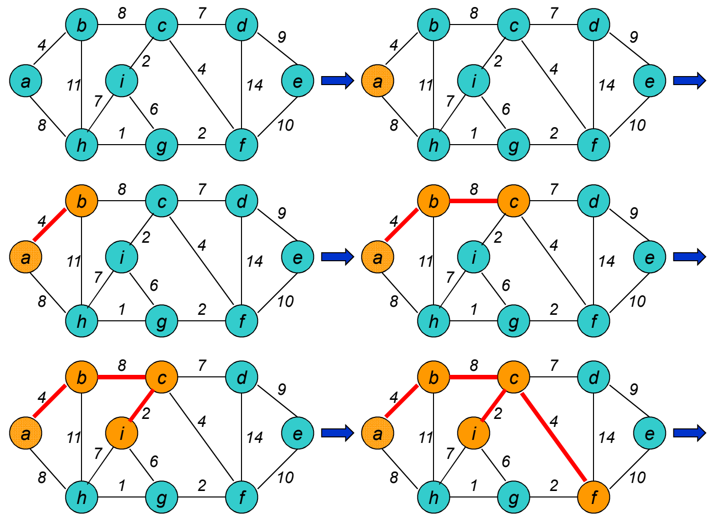
Prim's Algorithm

```
MST-Prim(G, w, r)
    for each u ∈ V[G]
       do key[u] = ∞;
       Π[r] = NULL;
 key[r] = 0;
 Q = V[G];
while (Q ≠  )
        u = Extract_Min(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                Π[v] = u;
               key[v] = w(u,v);
```

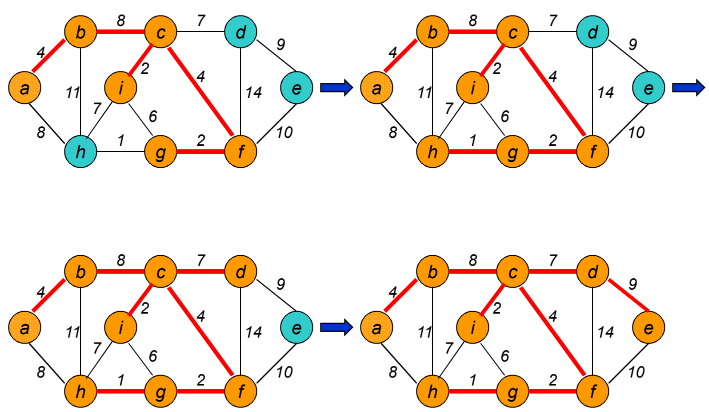
--G is a connected graph
--w is edge weights
--r is root

Running Time: ***O(E + V lg V)***
[using a Fibonacci heap for the priority queue]

Prim’s Algorithm



Prim’s Algorithm



Batch B over