CSCI 335 Software Design and Analysis III

Graph Algorithms III

(Network Flow,Minimum Spanning Trees, DFS, Biconnectivity, Strongly connected components)

Network Flow Problems

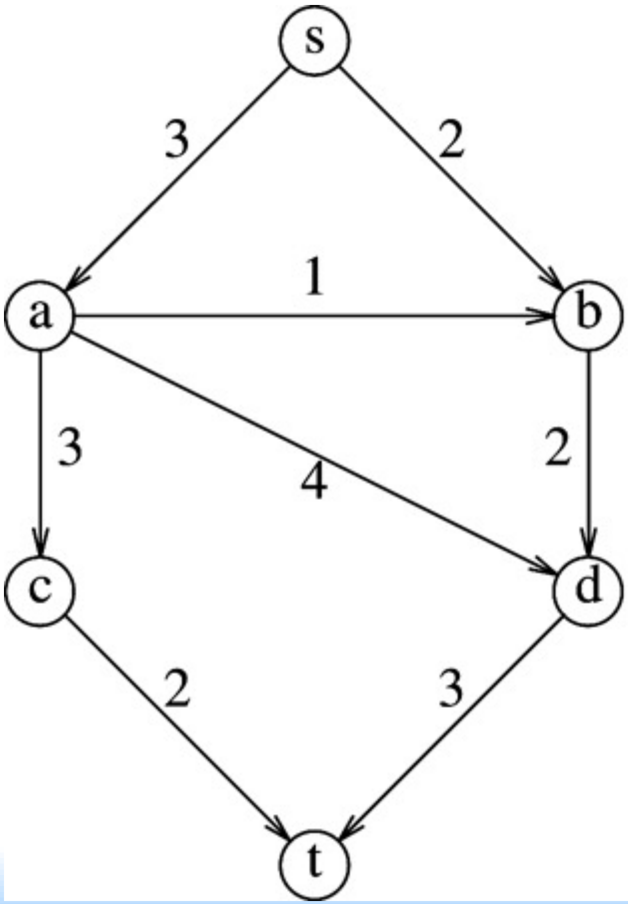
- Directed graph $G=(V,E)$, with edge capacities $c_{v,w}$
- **Capacity**: amount of water that could flow through edge
- One **source s** and one **sink t** vertex
- At any edge at <u>most</u> $c_{v,w}$ units of flow may pass
- At any vertex v (except **s** and **t**):

  flow coming in = flow coming out
- Determine: **maximum amount flow** that can pass through the graph from source **s** to sink **t**.

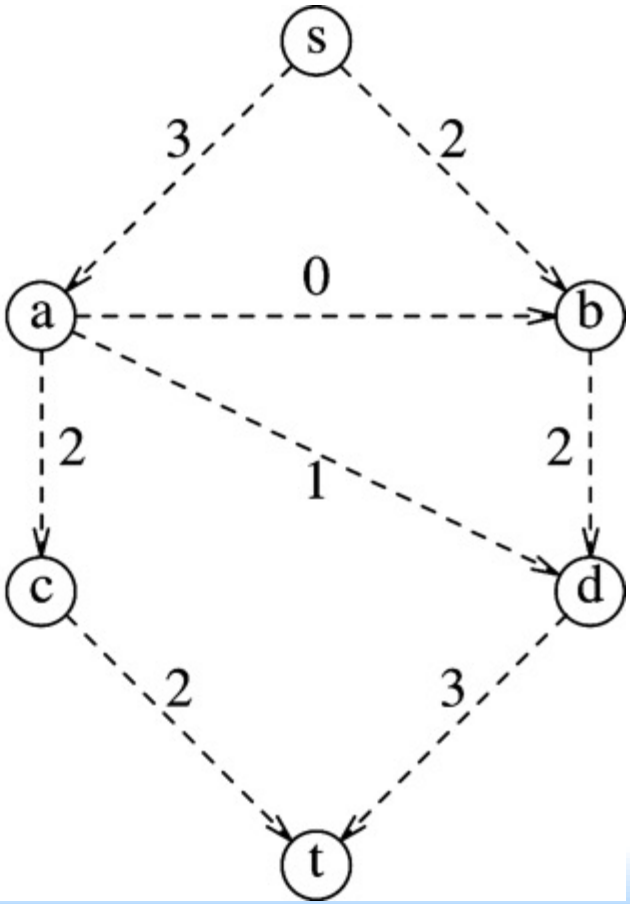
Applications

- Transportation networks
- Electricity networks
- Internet
- Ecology
- …

Example



Graph with capacities

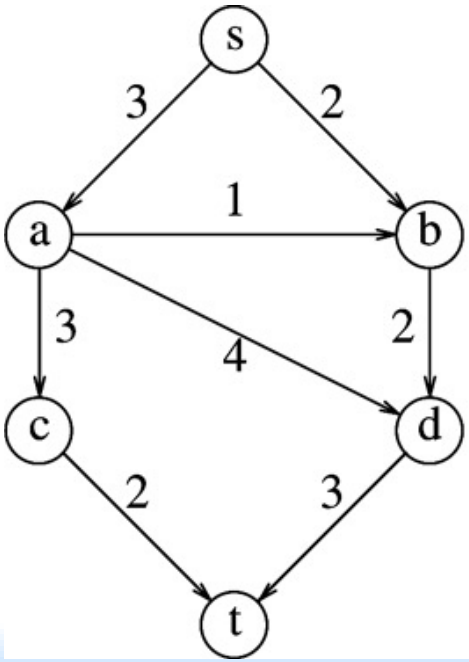


Maximum Flow

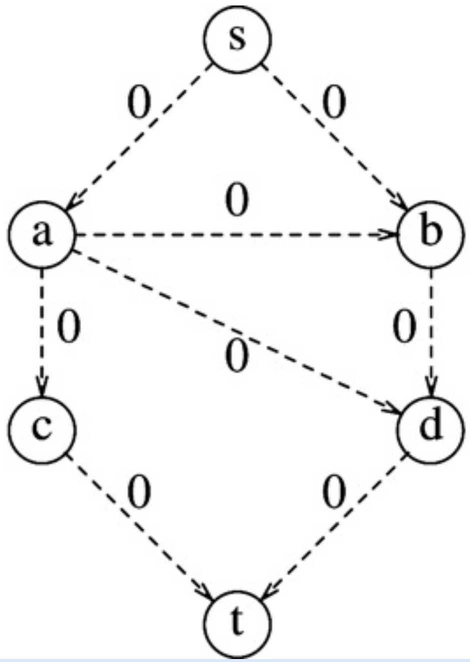
Solution

- At each algorithm stage keep:
- A flow graph G_f: current flow at each edge
  - Initial state: zero flow at each edge
- A residual graph G_r: amount of extra flow that can be pushed at each edge
  - Initial state: G_r is same as input G
- Augmenting path: a <u>path</u> from **s** to **t**
  - How much flow can we push through an augmenting path?
  - Flow that is as much as the minimum edge on the path.

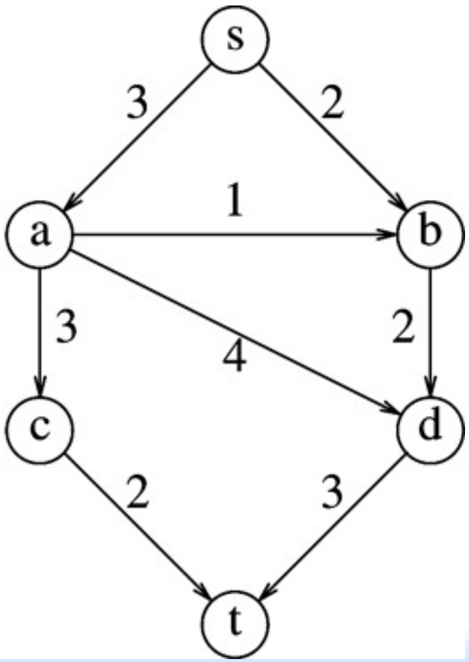
Initial state



G

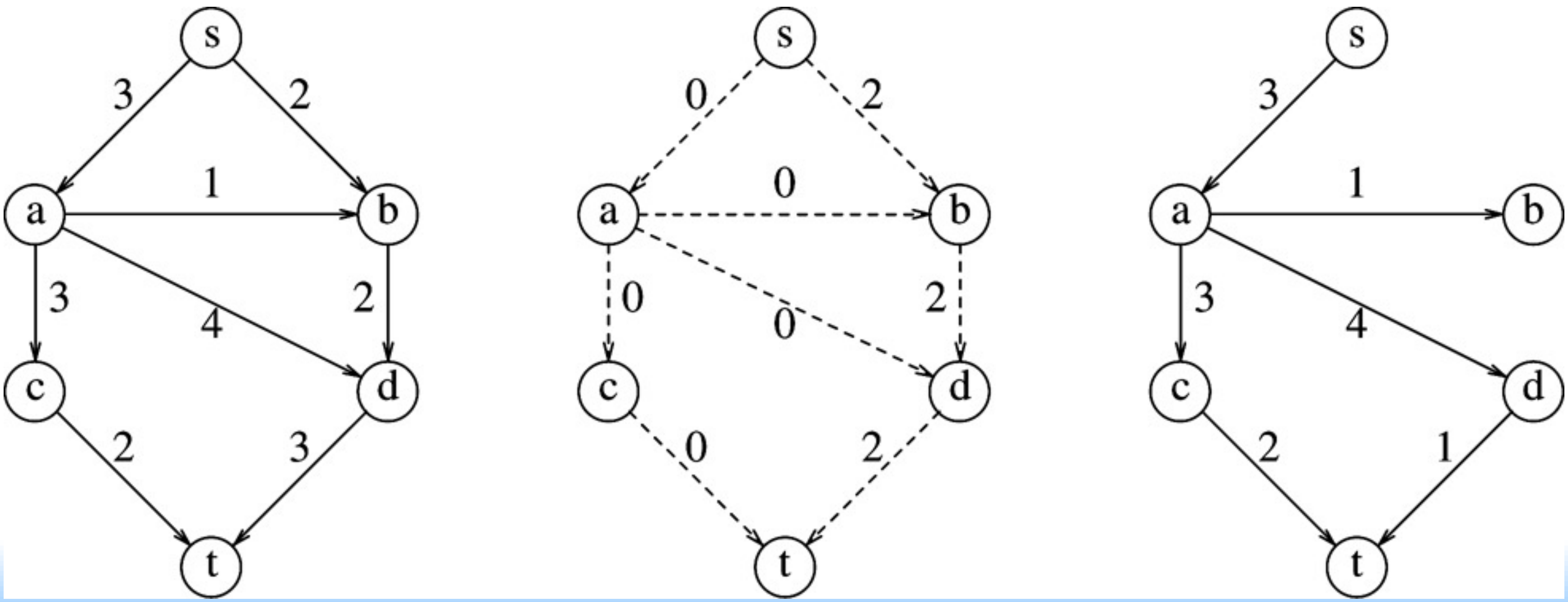


Flow graph Gf



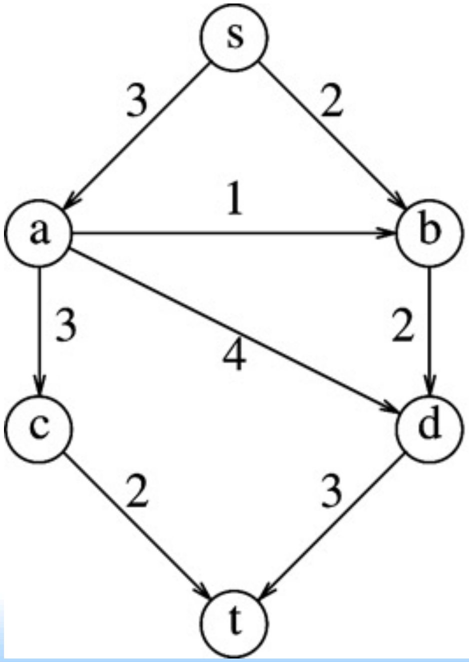
Residual Graph Gr

Augmenting path: s,b,d,t

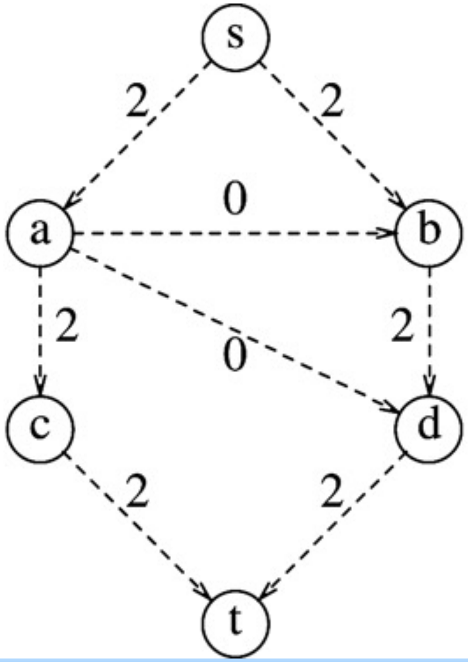


G **Flow graph Gf** **Residual Graph Gr**

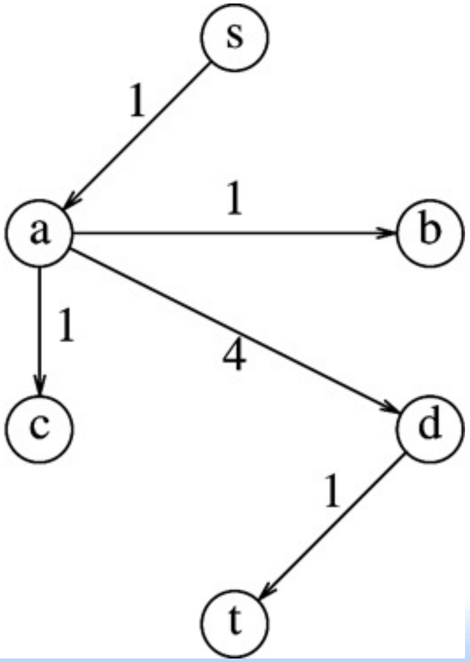
Augmenting path: s,a,c,t



G



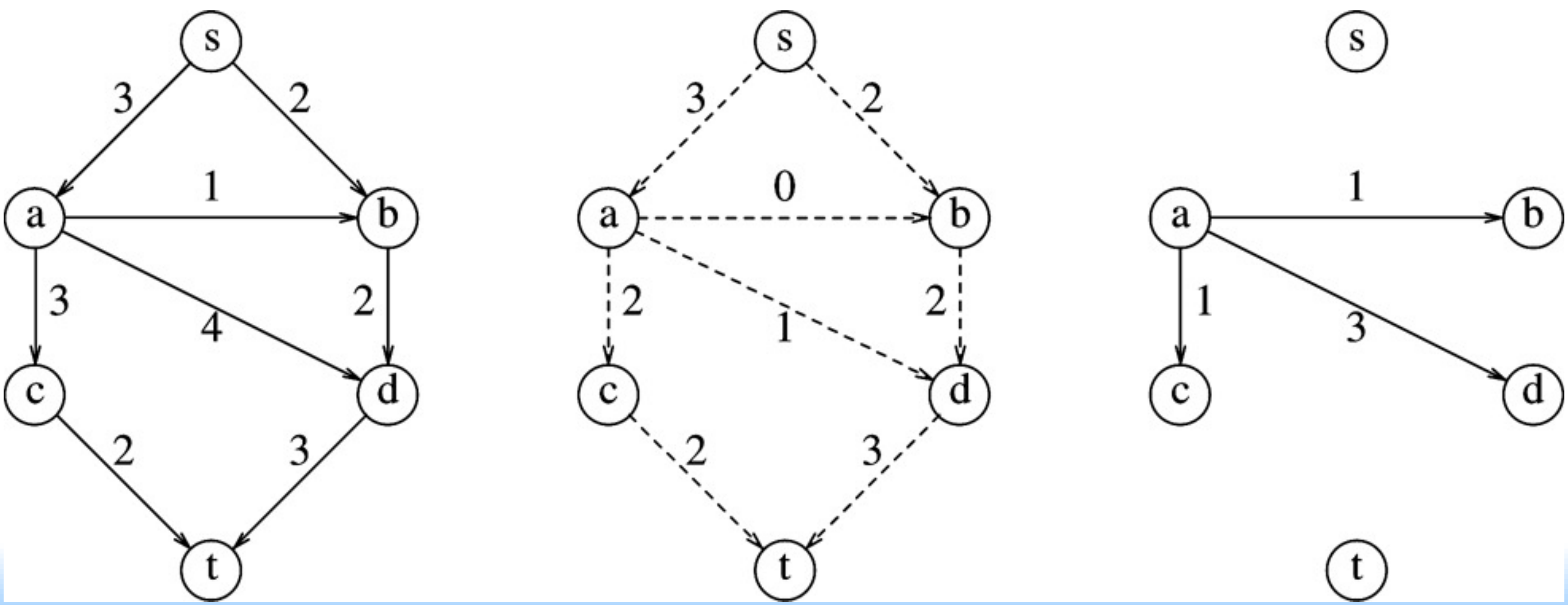
Flow graph Gf



Residual Graph Gr

Augmenting path: s,a,d,t

Algorithm termination



G **Flow graph Gf** **Residual Graph Gr**

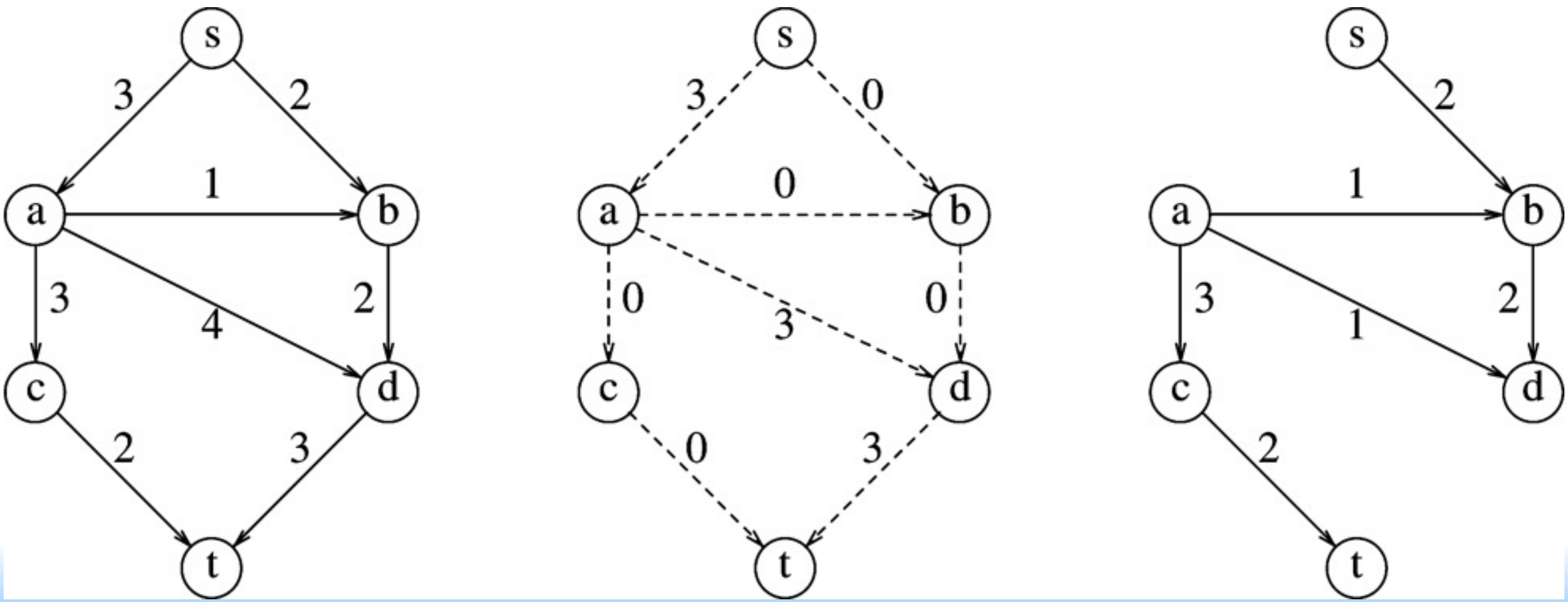
Concern

- Selection of "the wrong" augmenting path may lead to errors.
- Suppose the augmenting path of maximum flow is chosen at each step (GREEDY)

Augmenting path s,a,d,t

GREEDY ALGORITHM DOES NOT PROVIDE OPTIMAL RESULT

Algorithm terminates: wrong result !



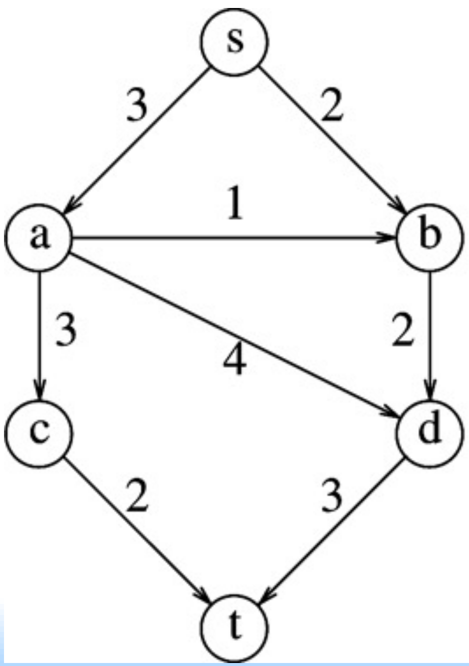
G **Flow graph Gf** **Residual Graph Gr**

Solution

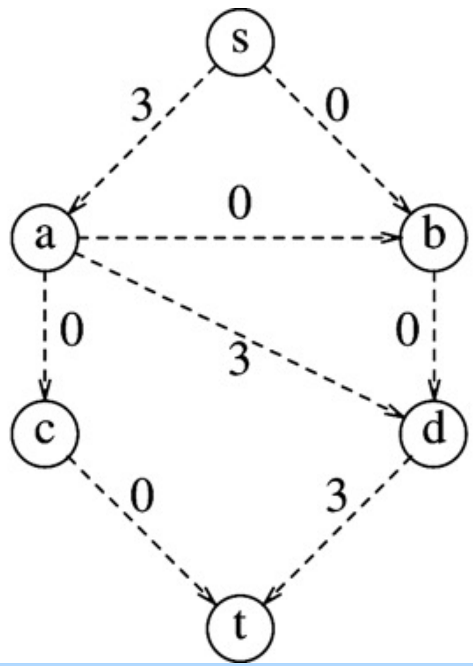
- Add the capability to **UNDO** action in case of wrong decision

Augmenting path s,a,d,t

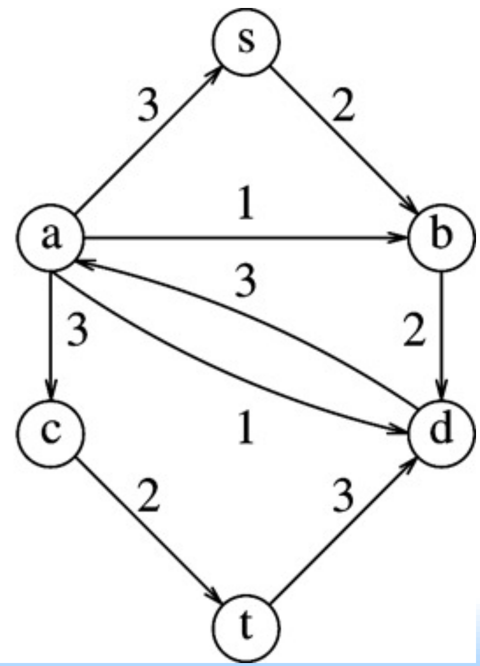
In residual graph action can be undone !



G



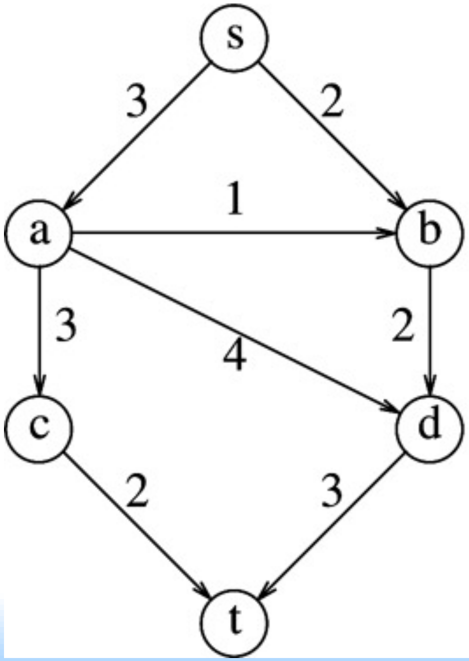
Flow graph Gf



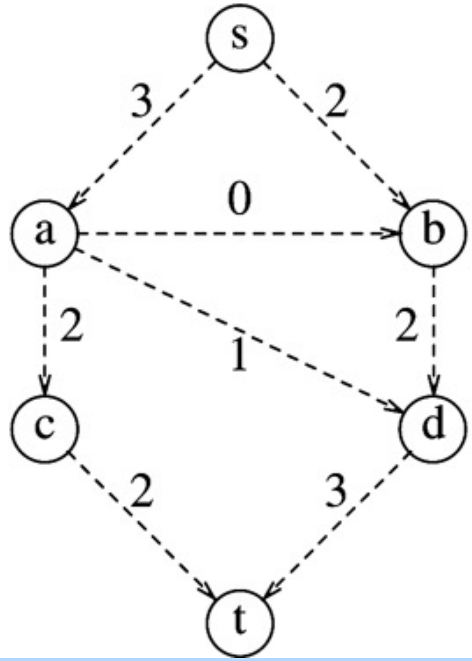
Residual Graph Gr

Augmenting path s,b,d,a,c,t

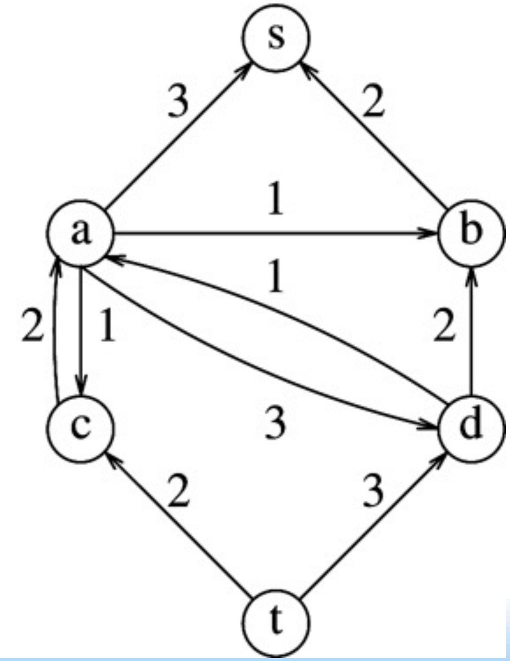
Algorithm terminates



G



Flow graph Gf

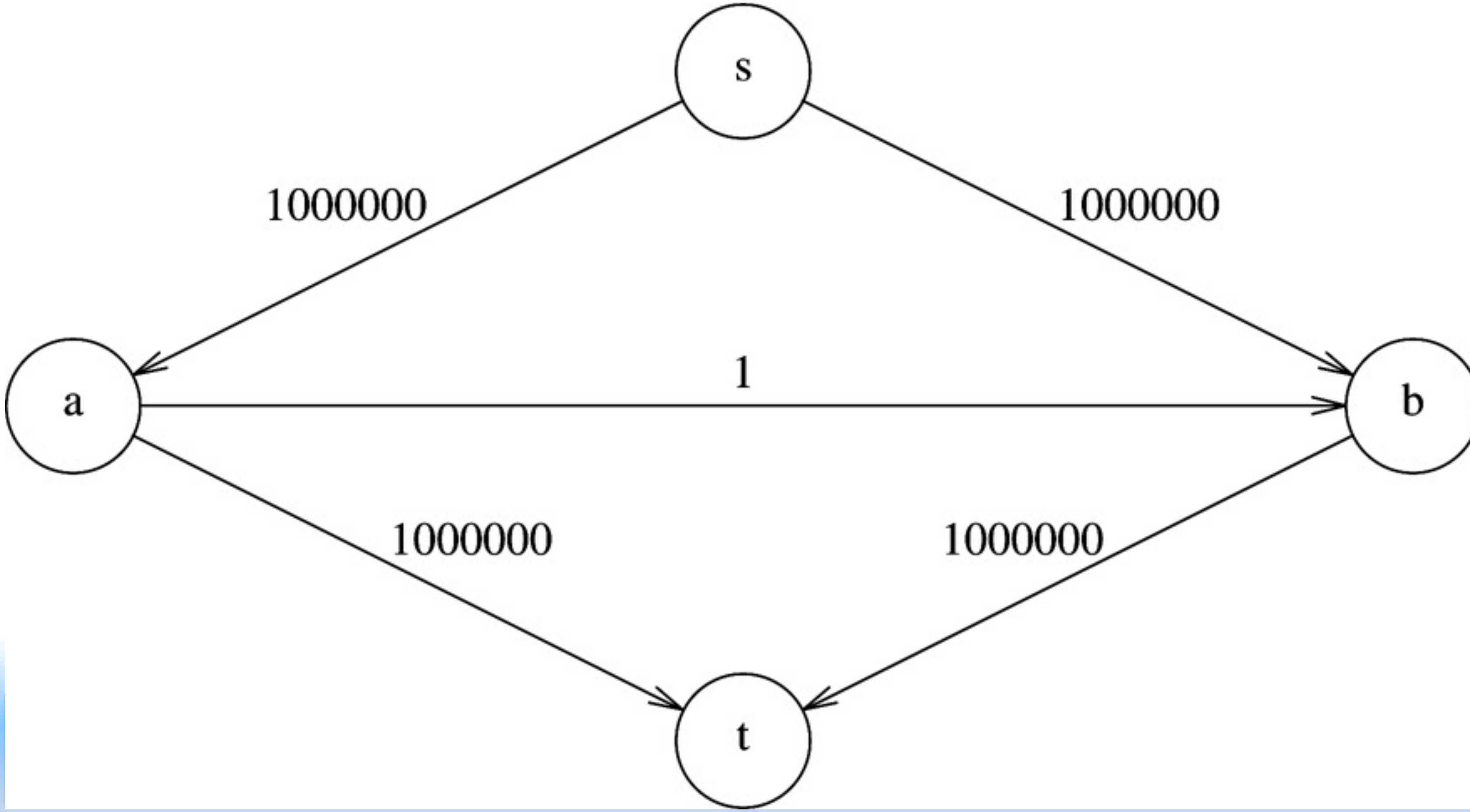


Residual Graph Gr

Analysis

- If capacities are rational numbers algorithm converges
- If capacities are integers (positive) and maximum flow is **f** then the algorithm would require at most **f** steps
- At each step, an augmenting path needs to be found: O(|E|) with unweighted shortest path
- Total cost (worst case): O (**f** * |E|)
  - Each augmenting path increases flow by 1
  - Not good, should be improved

Bad example



Improvements

- Always select path of maximum flow
  - How ?
- If cap_{max} is maximum edge capacity then $O(\ |E|\ \log cap_{max}\ )$ augmentations needed
- Augmentation time is now $O(\ |E|\ \log|V|\ )$
- Total cost is thus:

$$O(\ |E|^2\ \log|V|\ \log cap_{max}\ )$$

Improvements

- Always select path with smaller number of edges
  - How ?
- $O(|E|\ |V|)$ augmentation needed
- Augmentation time is now $O(\ |E|\ )$
- Total cost is thus:

$$O(\ |E|^2\ |V|\ )$$

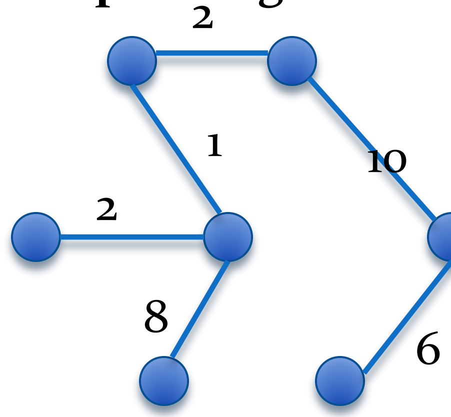
Spanning Trees

For a graph of N vertices exactly N-1 edges are needed for construction of tree.

Tree is defined as a graph such that:

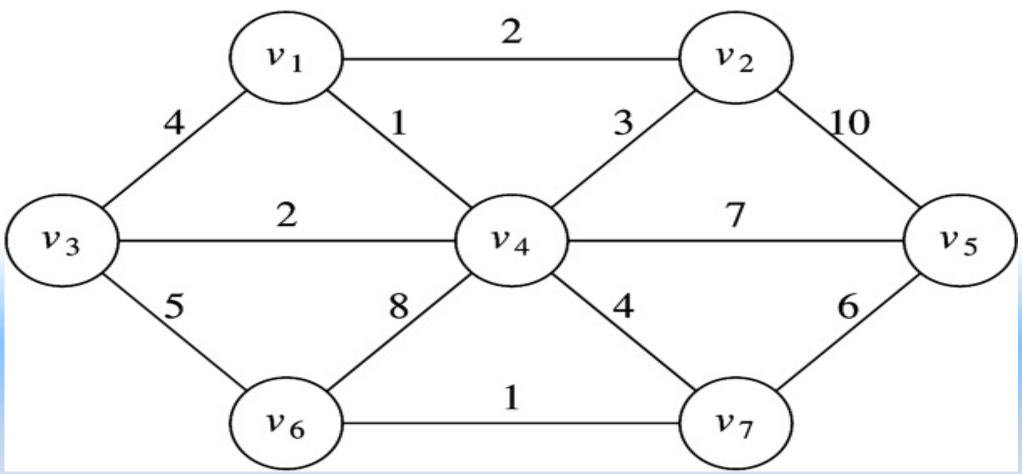
(a) there is a path between every pair of nodes
(b) There are no cycles

A spanning tree of graph G



Cost = 2 + 1 + 10 + 2 + 8 + 6 = **29**

Input Undirected Graph G



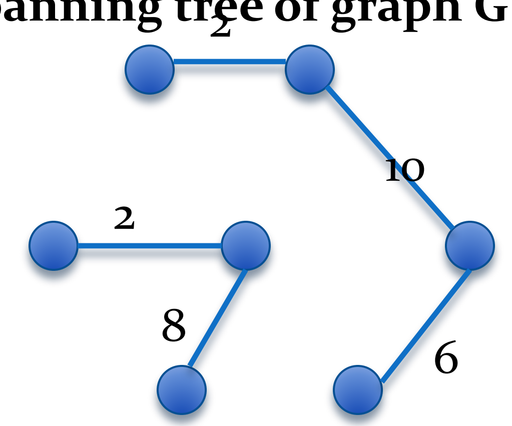
Spanning Trees

For a graph of N vertices exactly N-1 edges are needed for construction of tree.

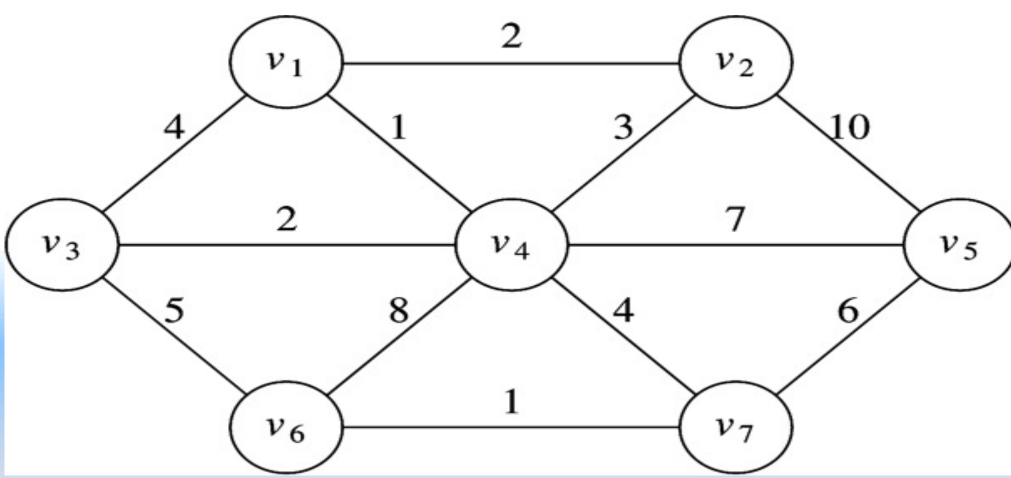
Tree is defined as a graph such that:

(a) there is a path between every pair of nodes
(b) There are no cycles

Not a spanning tree of graph G (why?)



Input Undirected Graph G



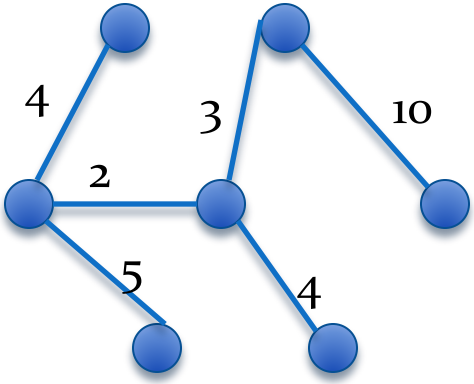
Spanning Trees

For a graph of N vertices exactly N-1 edges are needed for construction of tree.

Tree is defined as a graph such that:

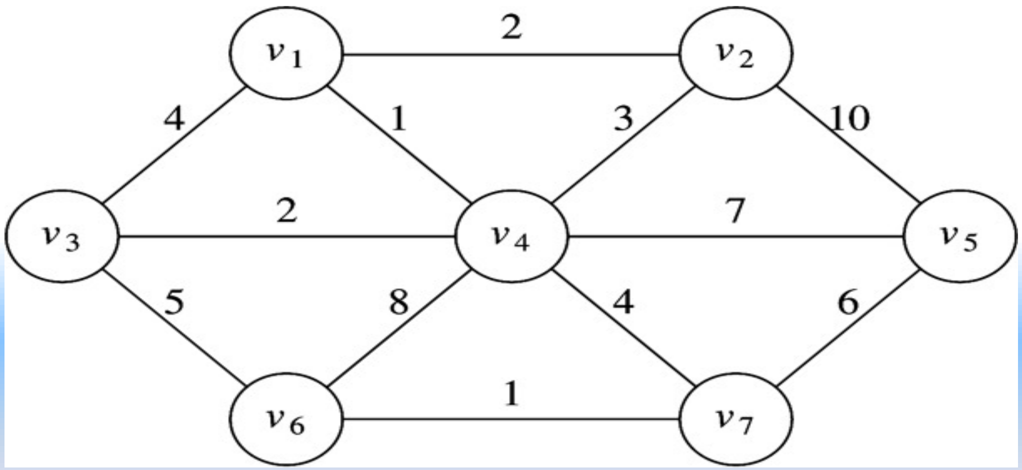
(a) there is a path between every pair of nodes
(b) There are no cycles

Another spanning tree of graph G



Cost = 4 + 2 + 5 + 3 + 4 + 10 = **28**

Input Undirected Graph G



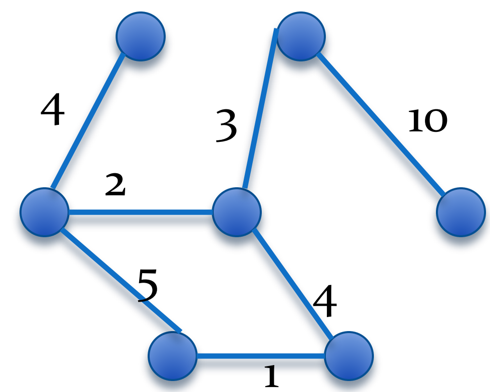
Spanning Trees

For a graph of N vertices exactly N-1 edges are needed for construction of tree.

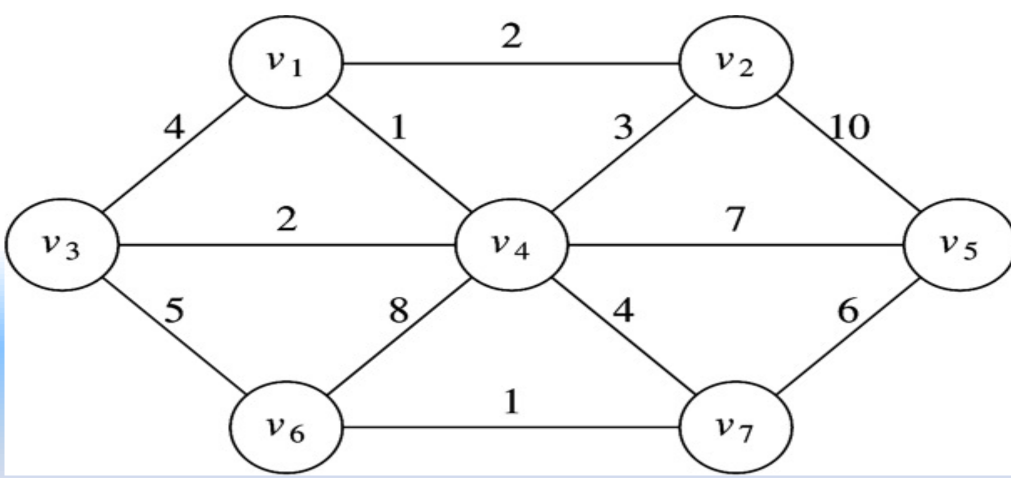
Tree is defined as a graph such that:

(a) there is a path between every pair of nodes
(b) There are no cycles

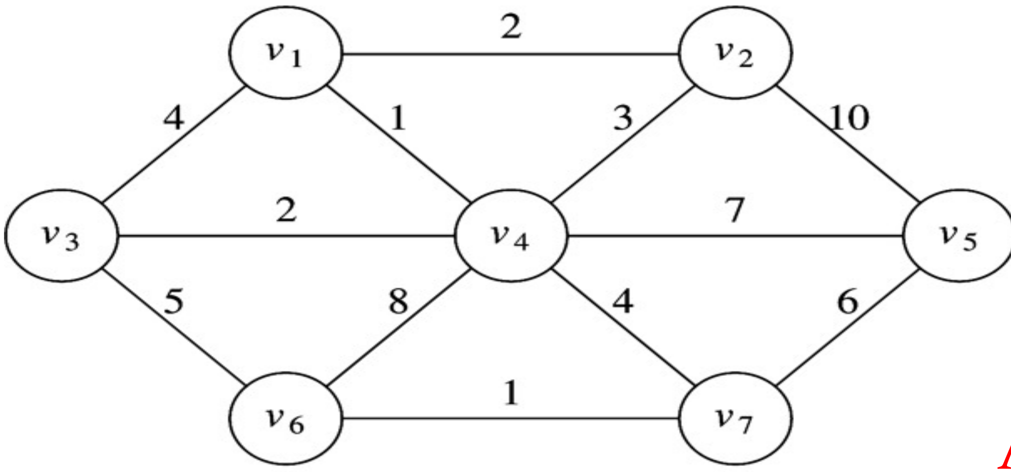
Not a spanning tree of graph G (why?)



Input Undirected Graph G

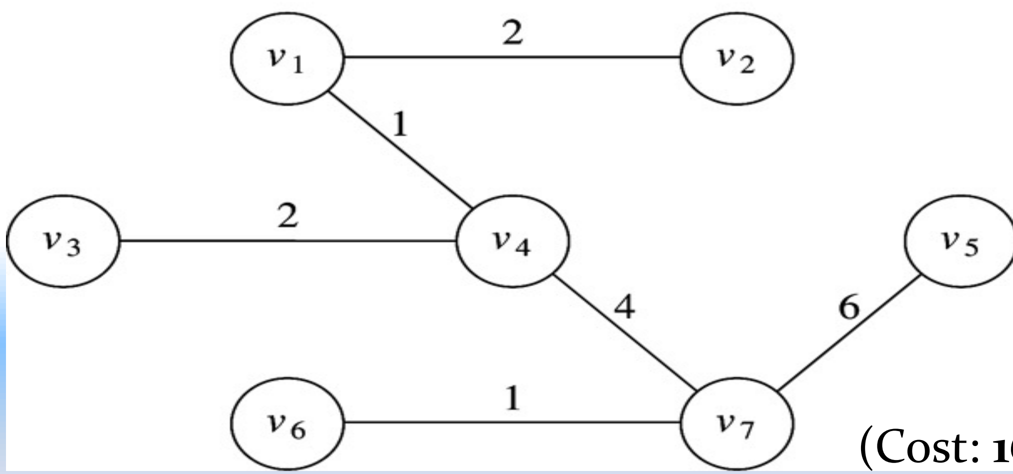


Minimum Spanning Trees



Input Undirected Graph

APPLICATIONS?



Corresponding Minimum Spanning Tree

(Cost: **16**): Spanning tree with minimum cost

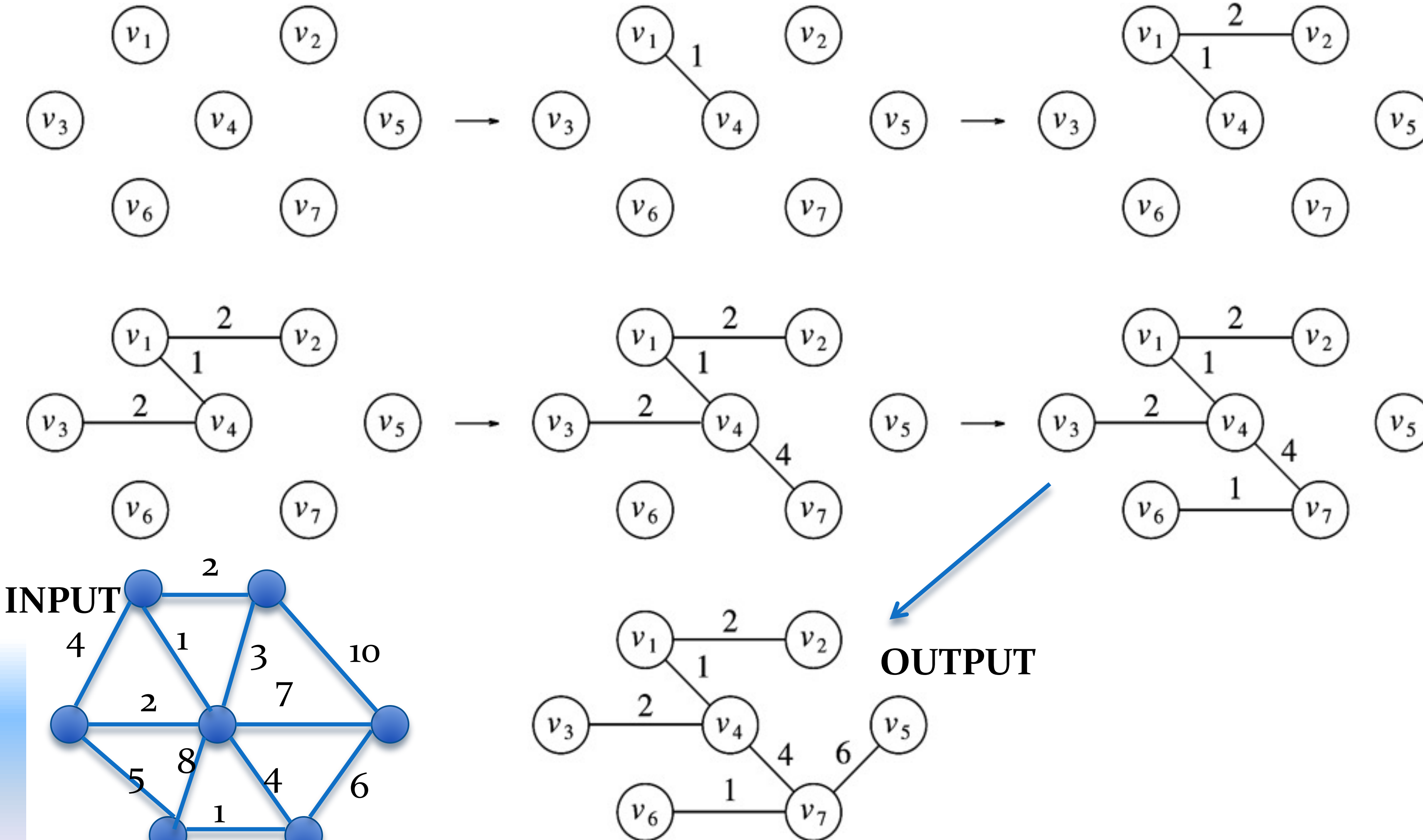
MST

- Two **<u>Greedy</u>** Algorithms
- **<u>Prim's algorithm</u>** [undirected graphs]
- Very similar to dijkstra:
  - Sets of known (T) and uknown (F) vertices
  - d_v: weight of shortest edge connecting v with known
  - p_v: last vertex to cause a change in d_v
  - Update rule: after v is selected update d_w for all uknown vertices adjacent to v as:

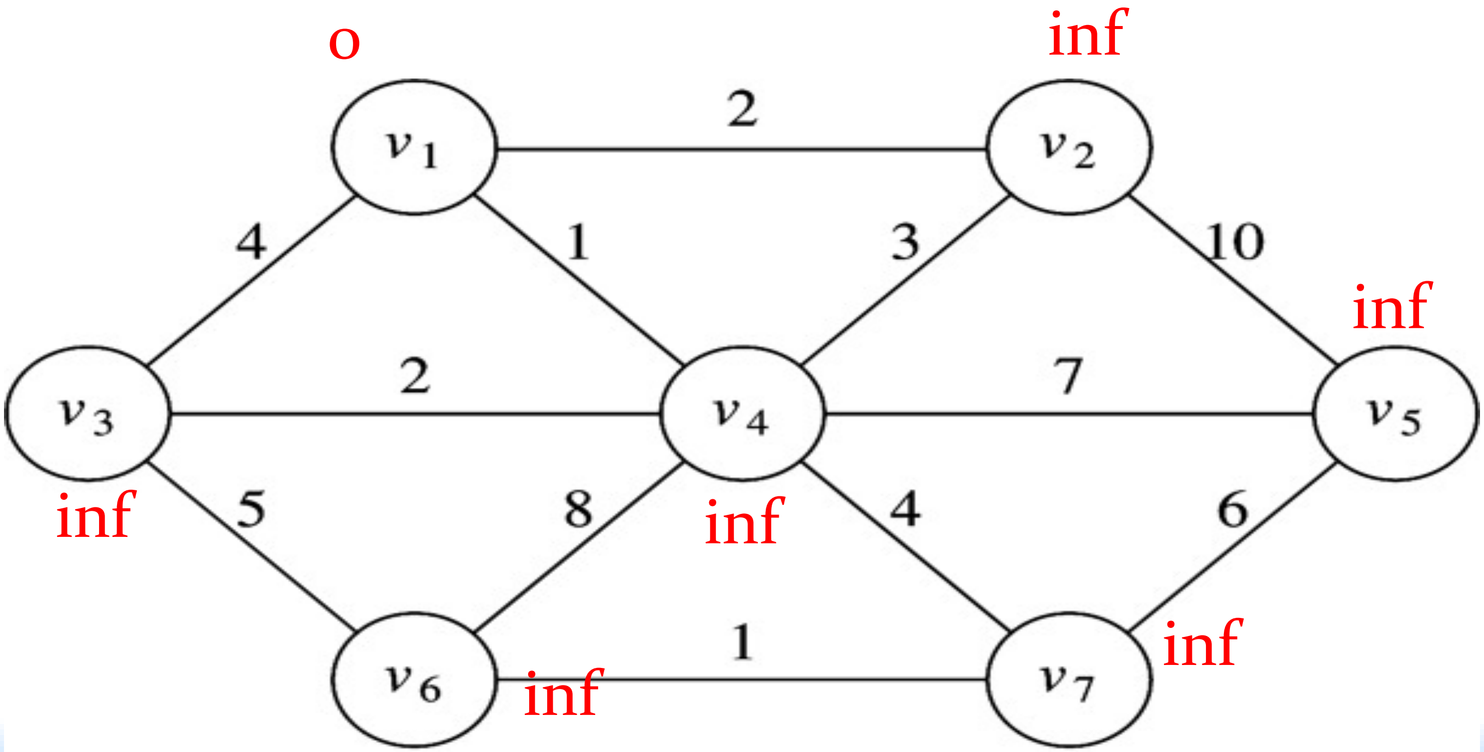
    $d_w = \min(d_w, c_{w,v})$
- Cost: Same as Dijkstra -> **<u>O(|E| log|V|)</u>** for sparse graphs

Visualization: https://www.cs.usfca.edu/~galles/visualization/Prim.html

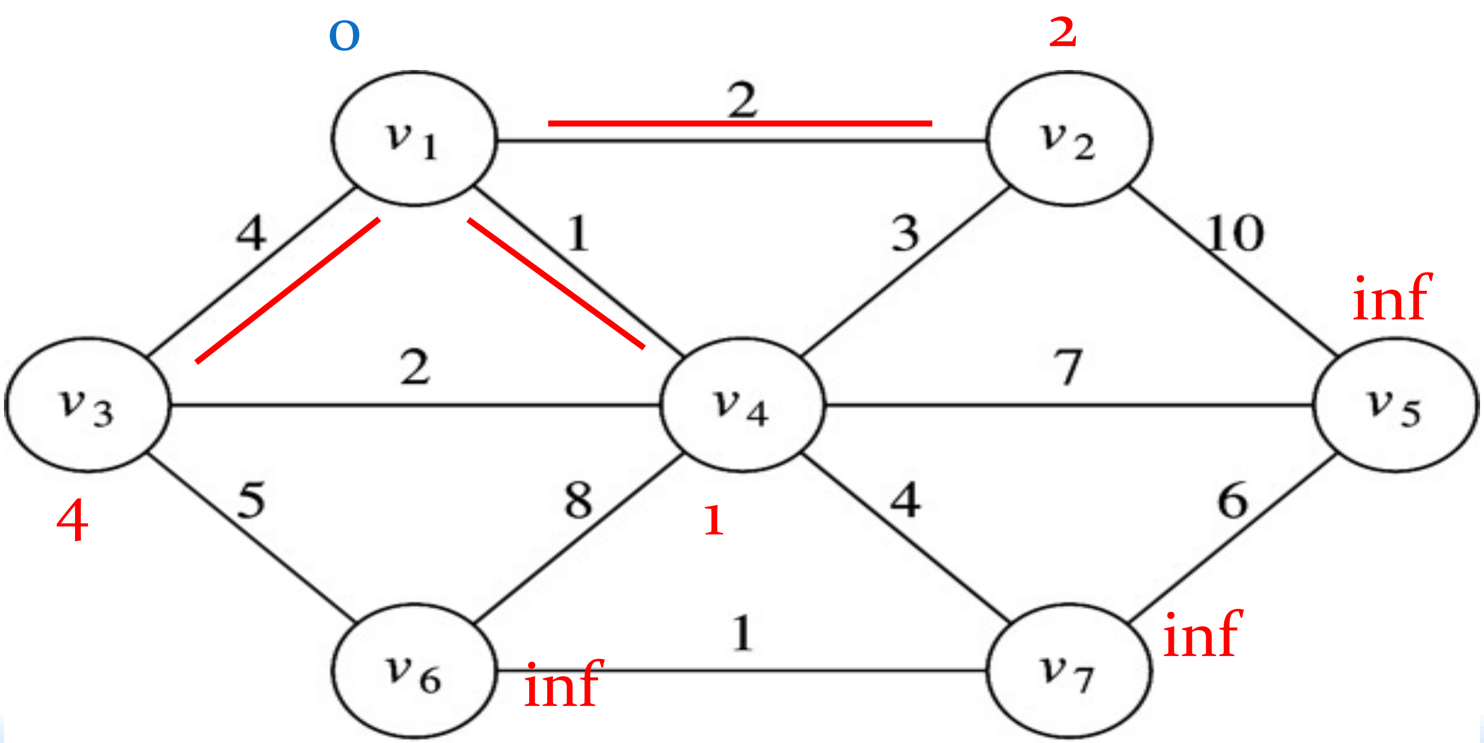
Prim’s algorithm



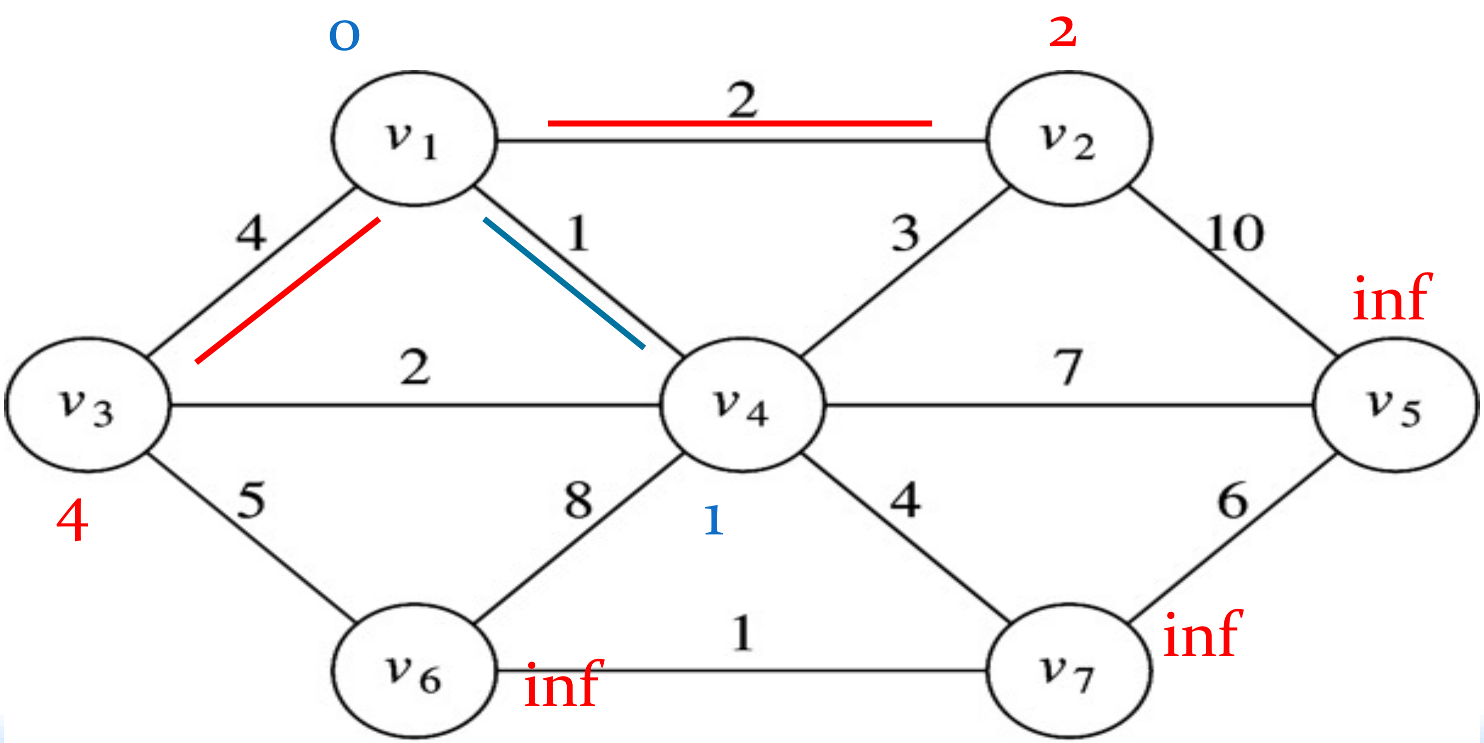
Minimum Spanning Trees



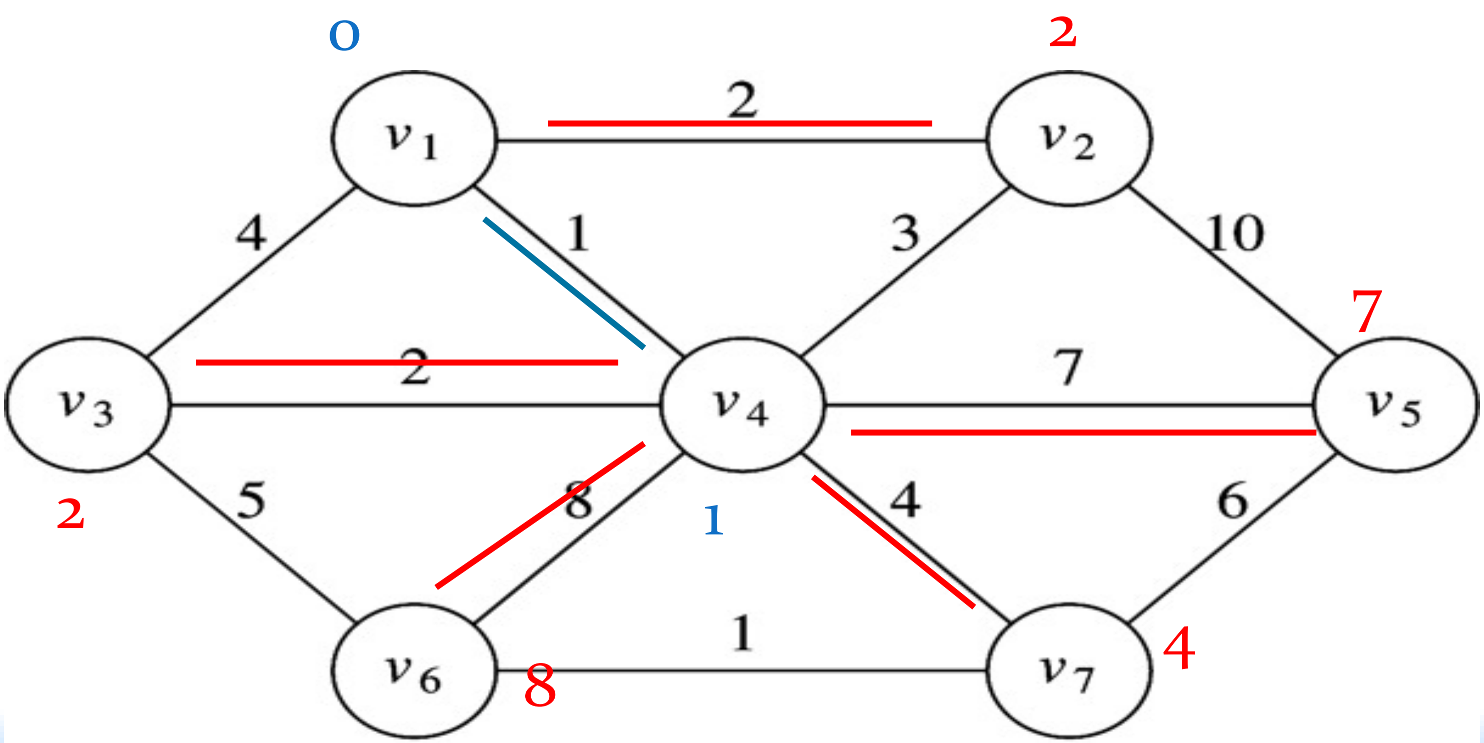
Minimum Spanning Trees



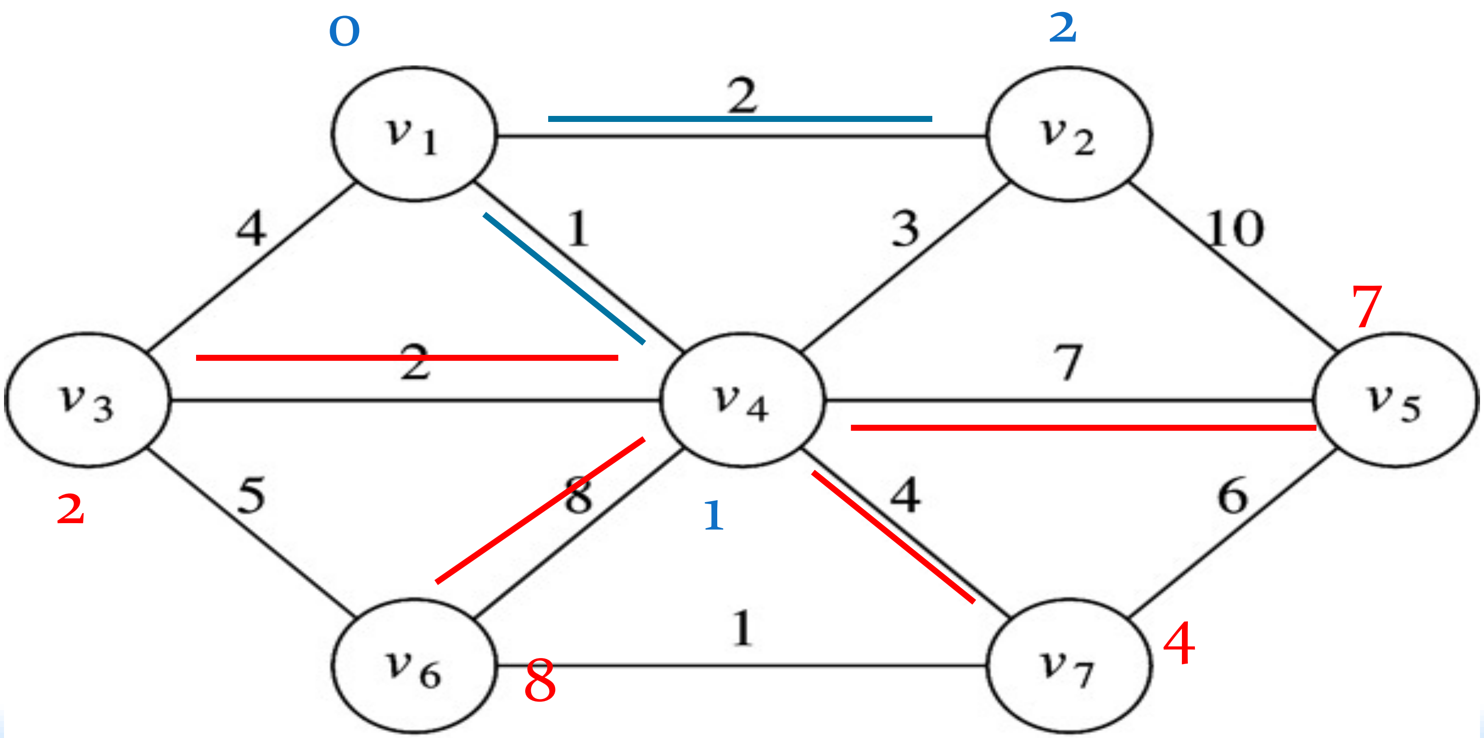
Minimum Spanning Trees



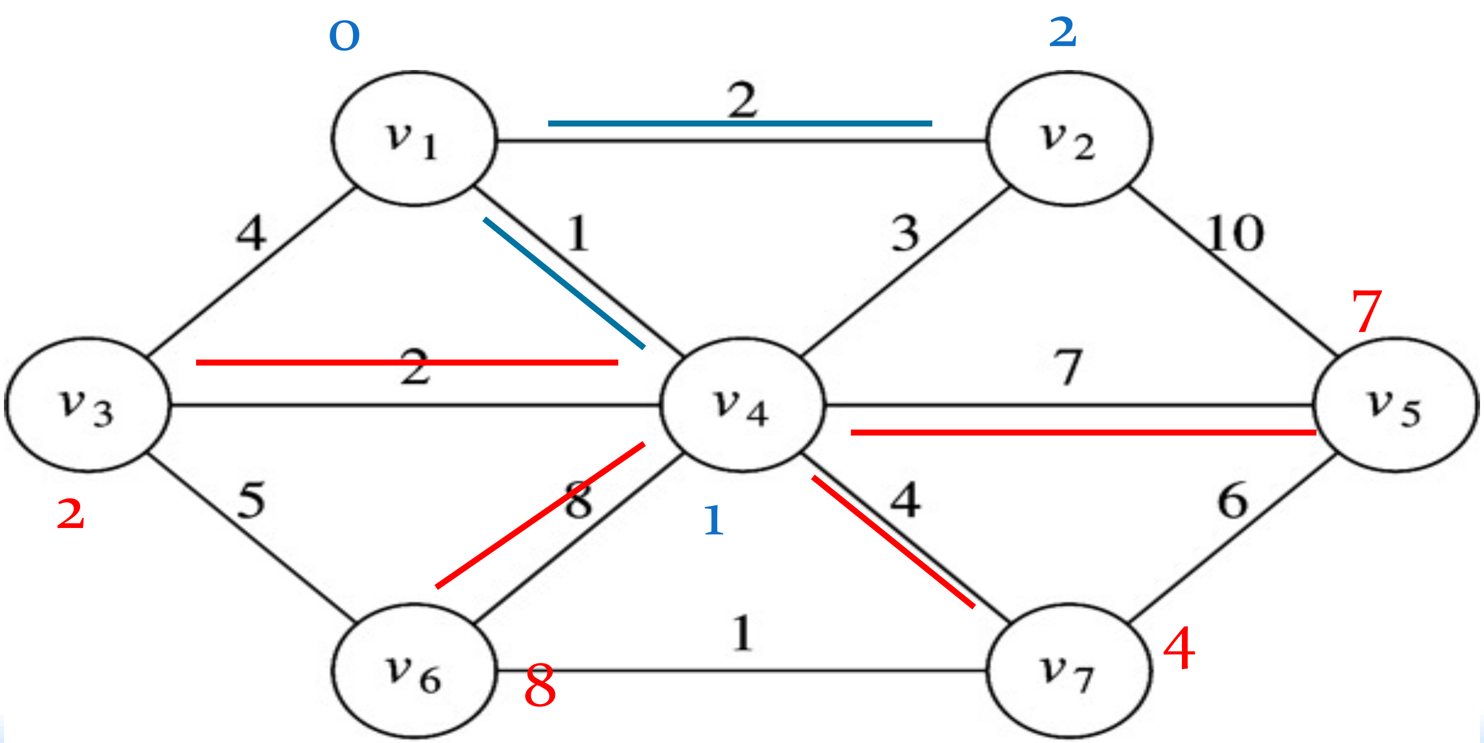
Minimum Spanning Trees



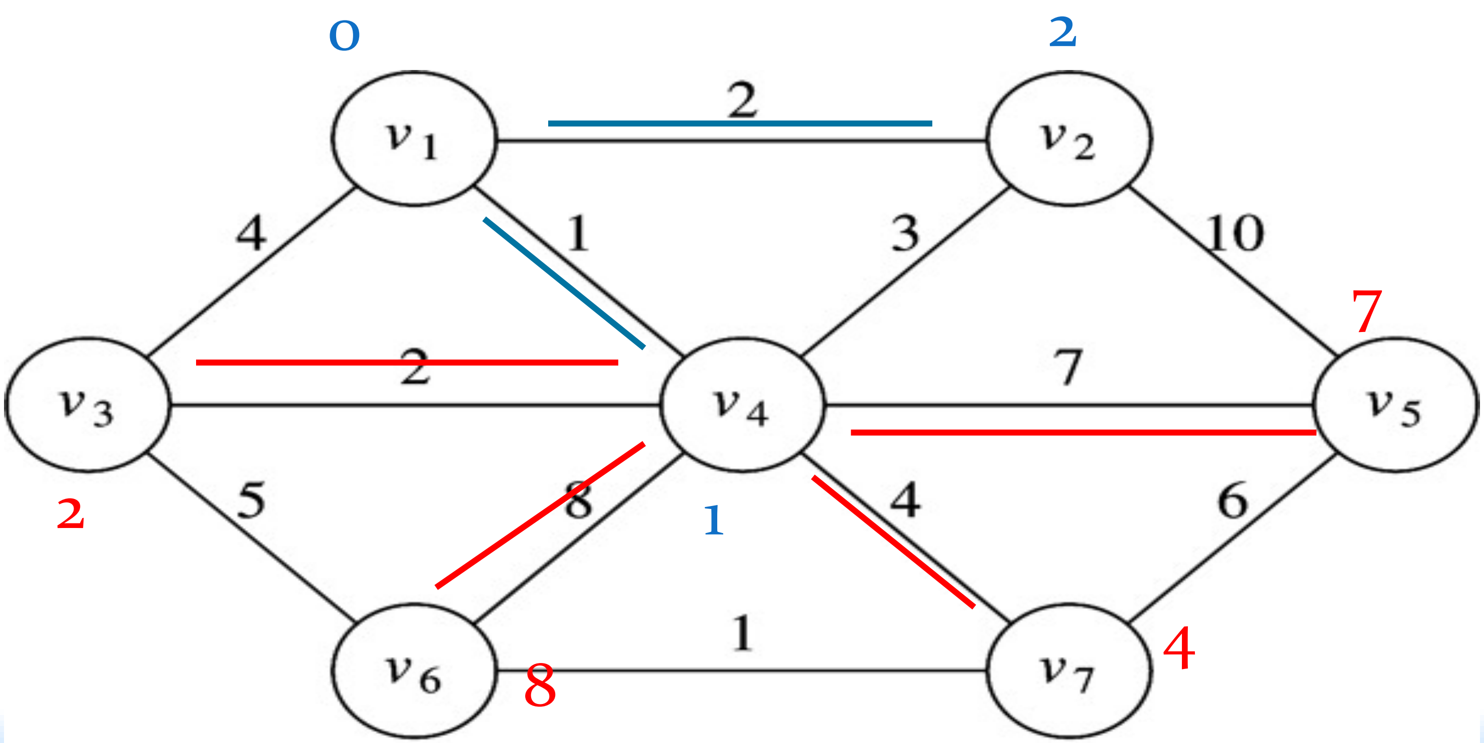
Minimum Spanning Trees



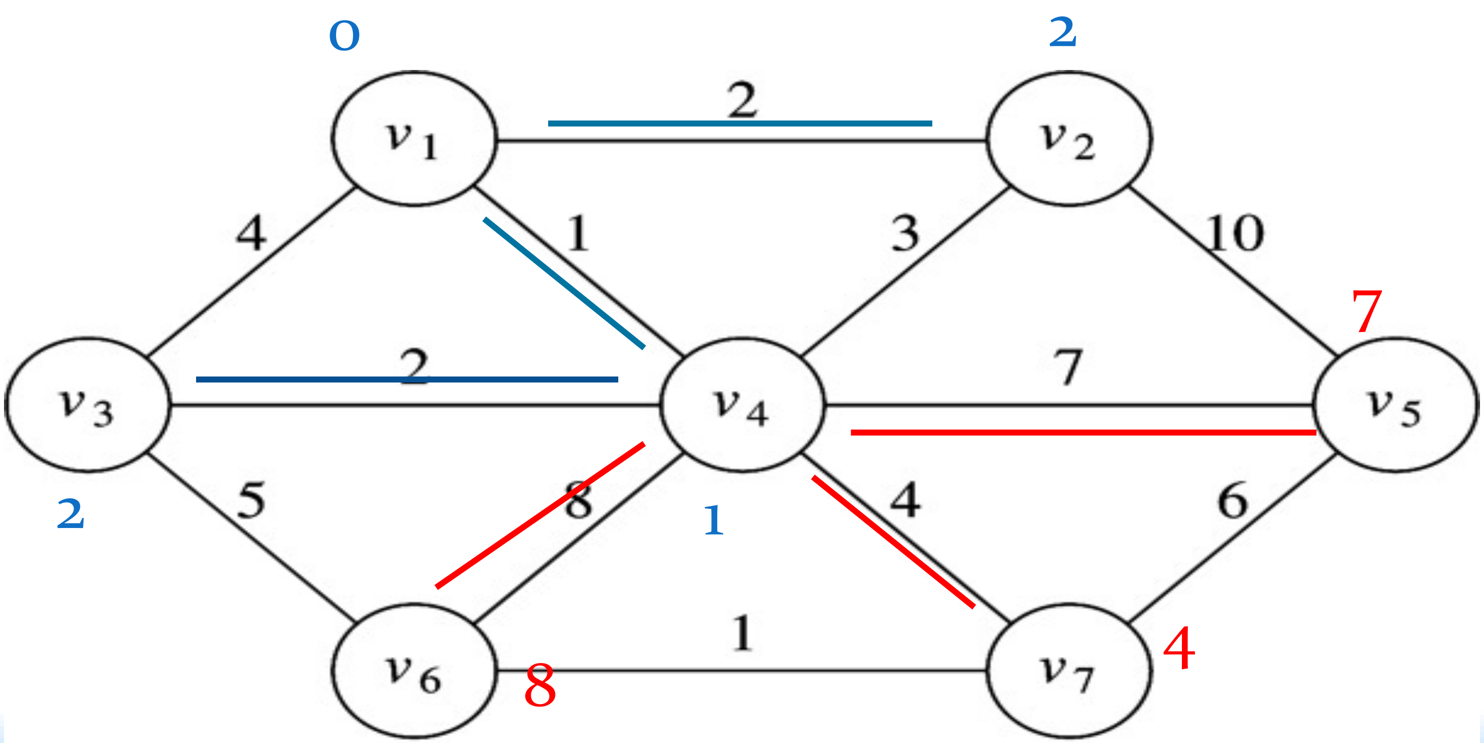
Minimum Spanning Trees



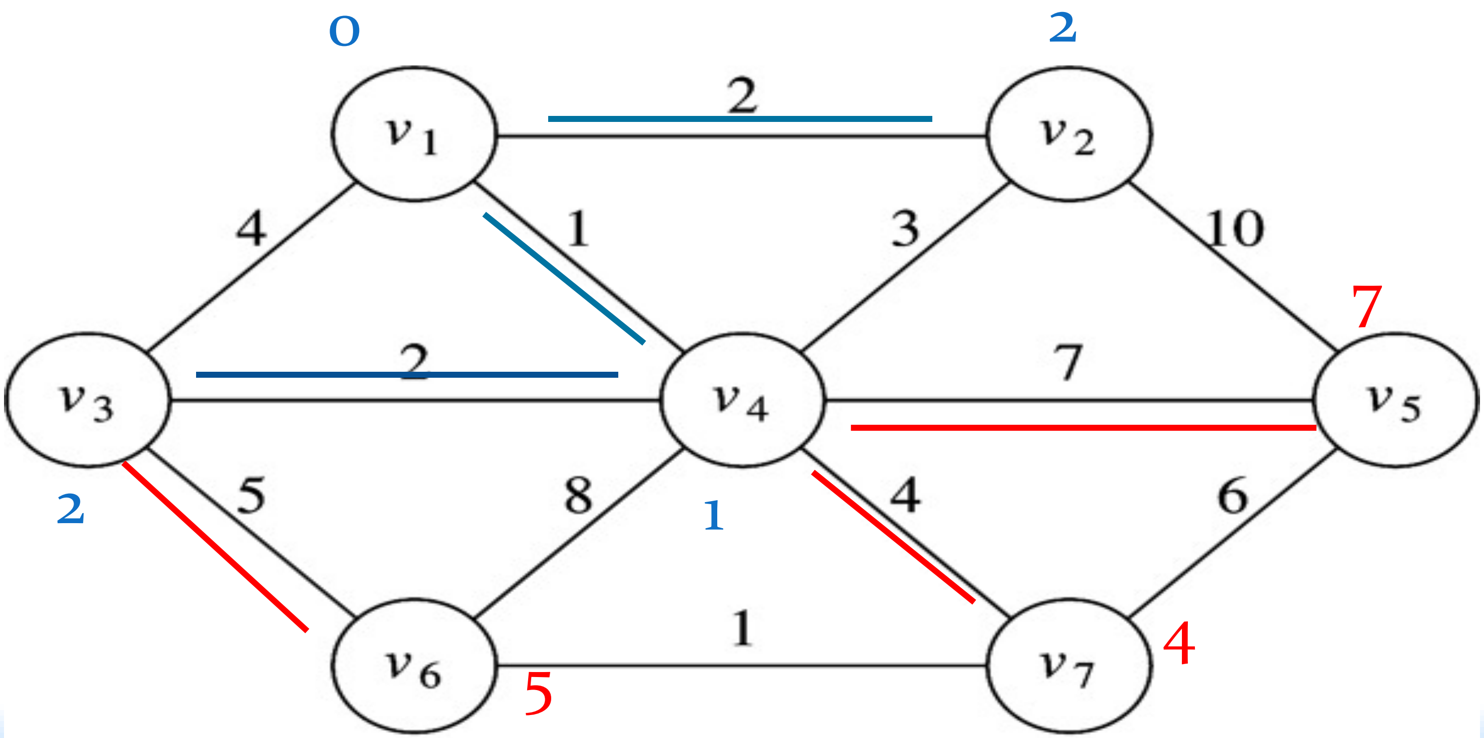
Minimum Spanning Trees



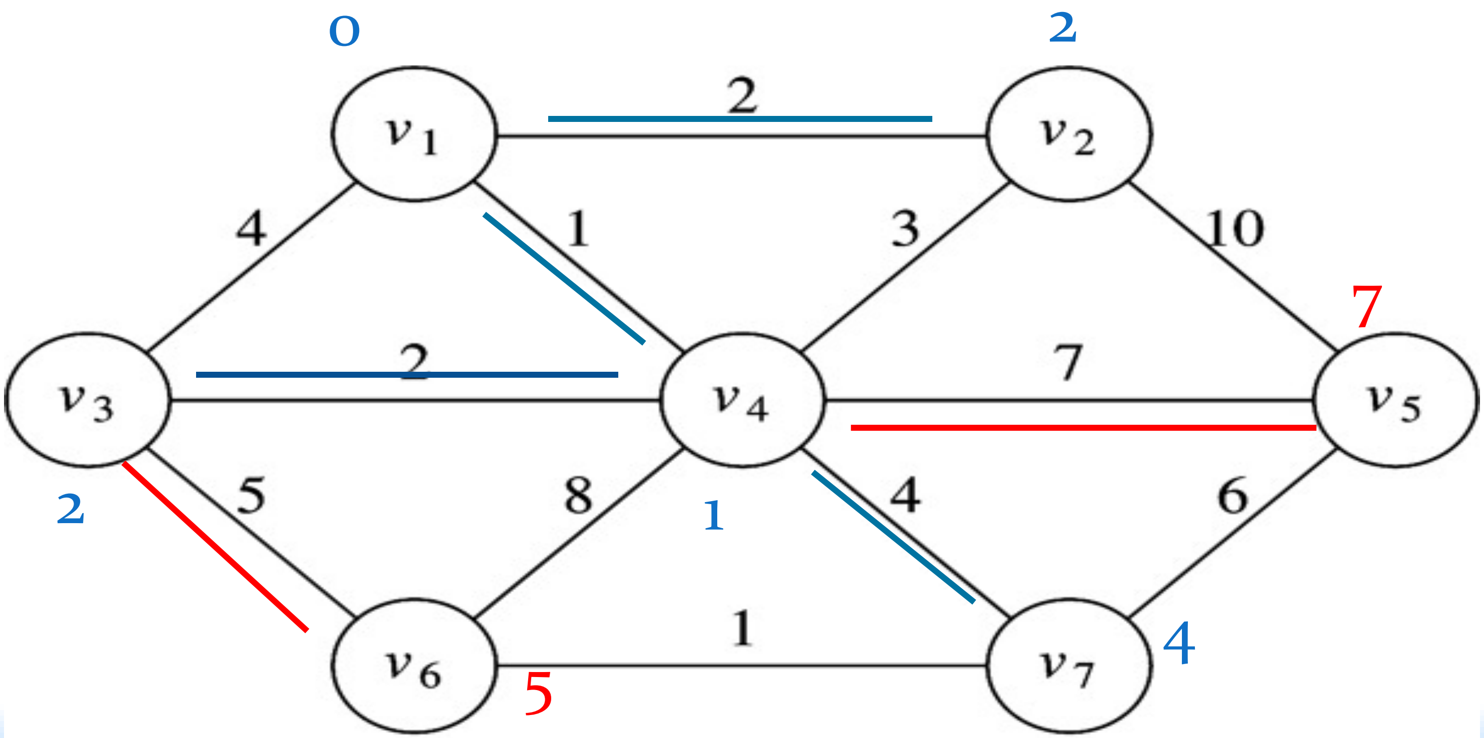
Minimum Spanning Trees



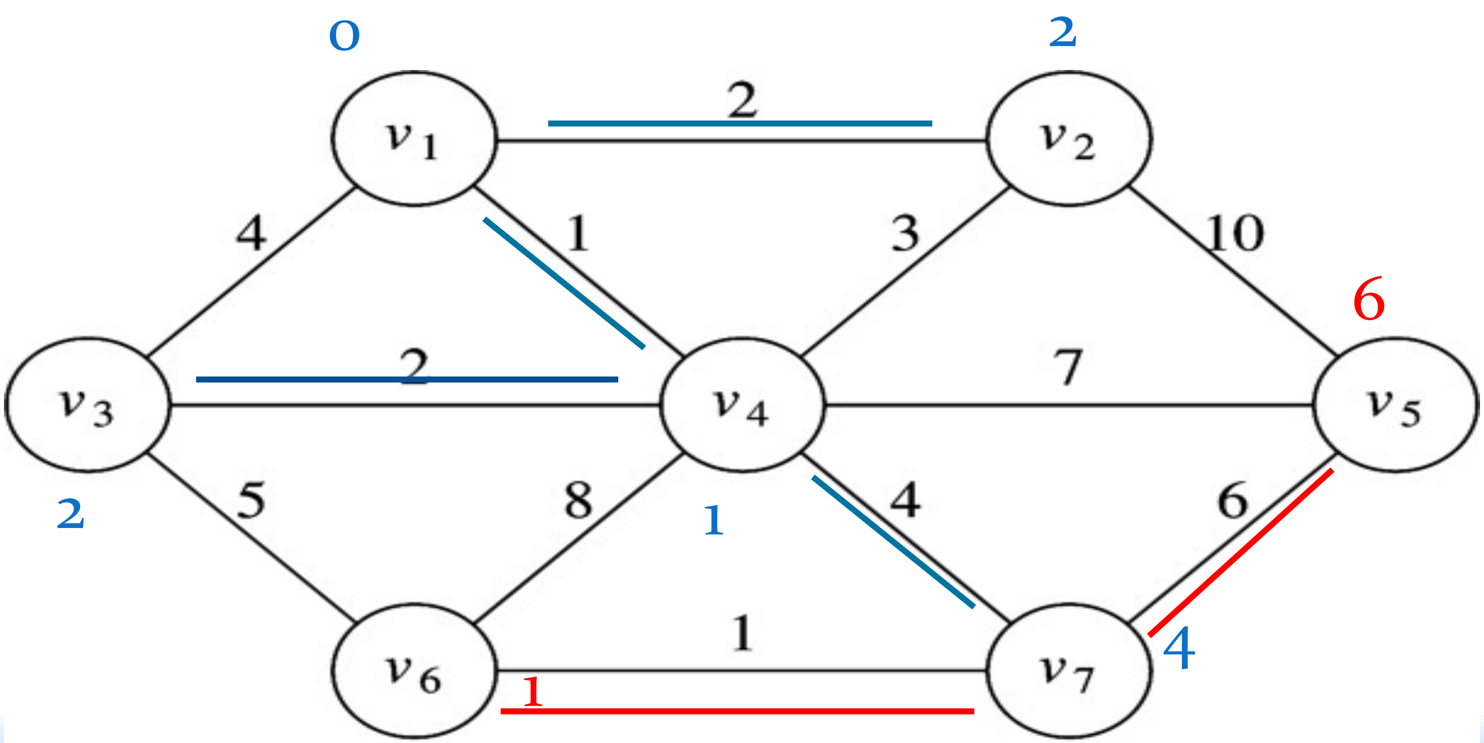
Minimum Spanning Trees



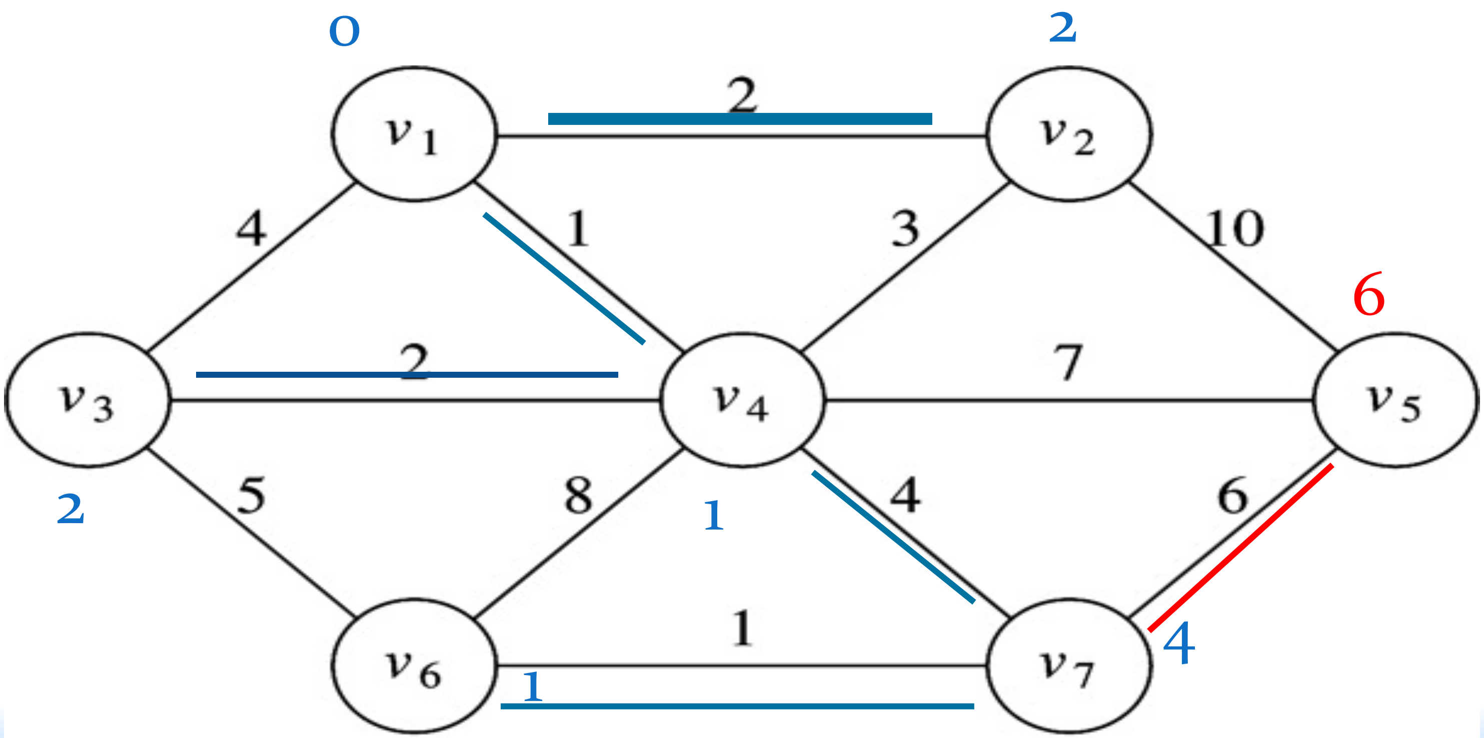
Minimum Spanning Trees



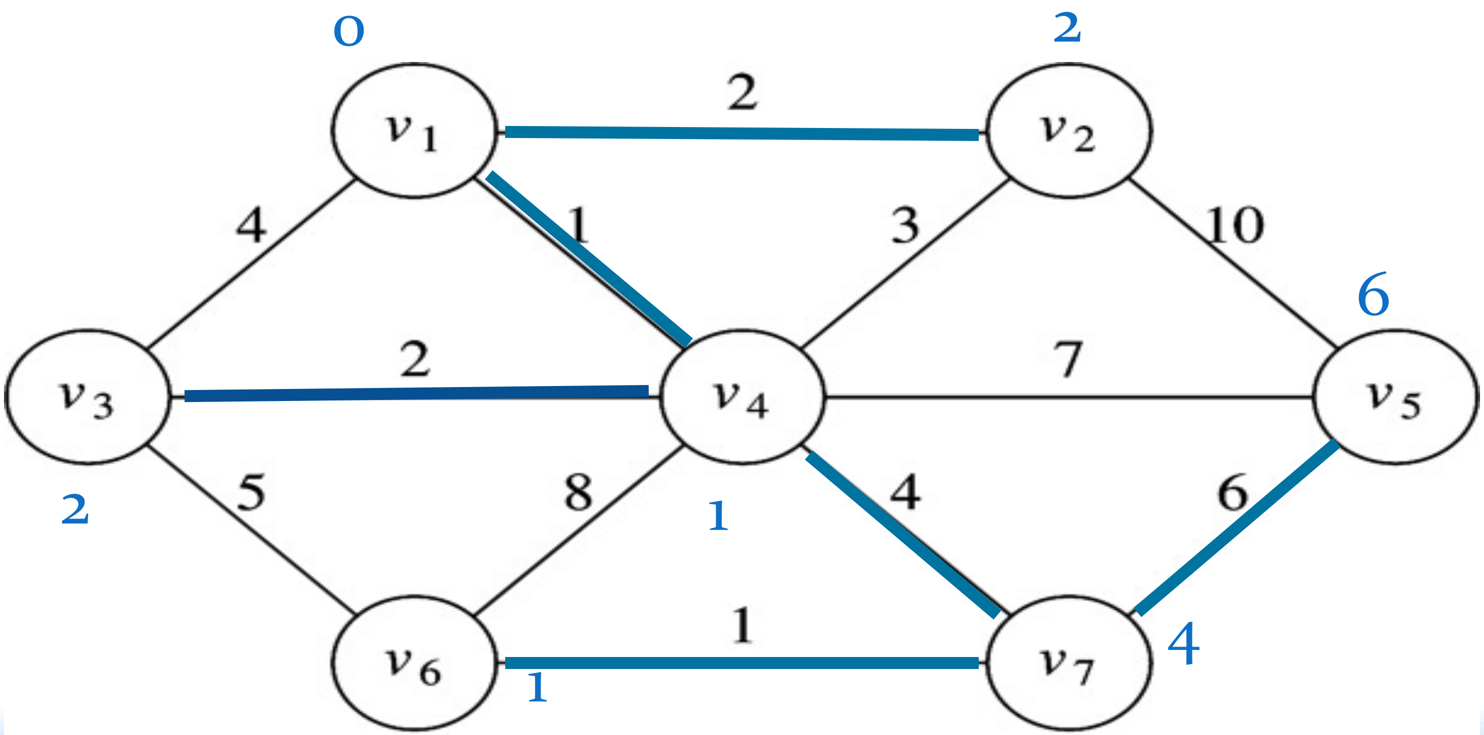
Minimum Spanning Trees



Minimum Spanning Trees



Minimum Spanning Trees



Prim's

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v1 selected

v	*known*	d_v	p_v
$\cancel{v_1}$	~~T~~	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v4 selected

v	*known*	d_v	p_v
$\cancel{v_1}$	~~T~~	0	0
v_2	F	2	v_1
v_3	F	2	v_4
$\cancel{v_4}$	~~T~~	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

v2 selected

Prim's

v	known	d_v	p_v
~~v_1~~	~~T~~	0	0
~~v_2~~	~~T~~	2	v_1
~~v_3~~	~~T~~	2	v_4
~~v_4~~	~~T~~	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

v	known	d_v	p_v
v_1	~~T~~	0	0
v_2	~~T~~	2	v_1
v_3	~~T~~	2	v_4
v_4	~~T~~	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	~~T~~	4	v_4

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

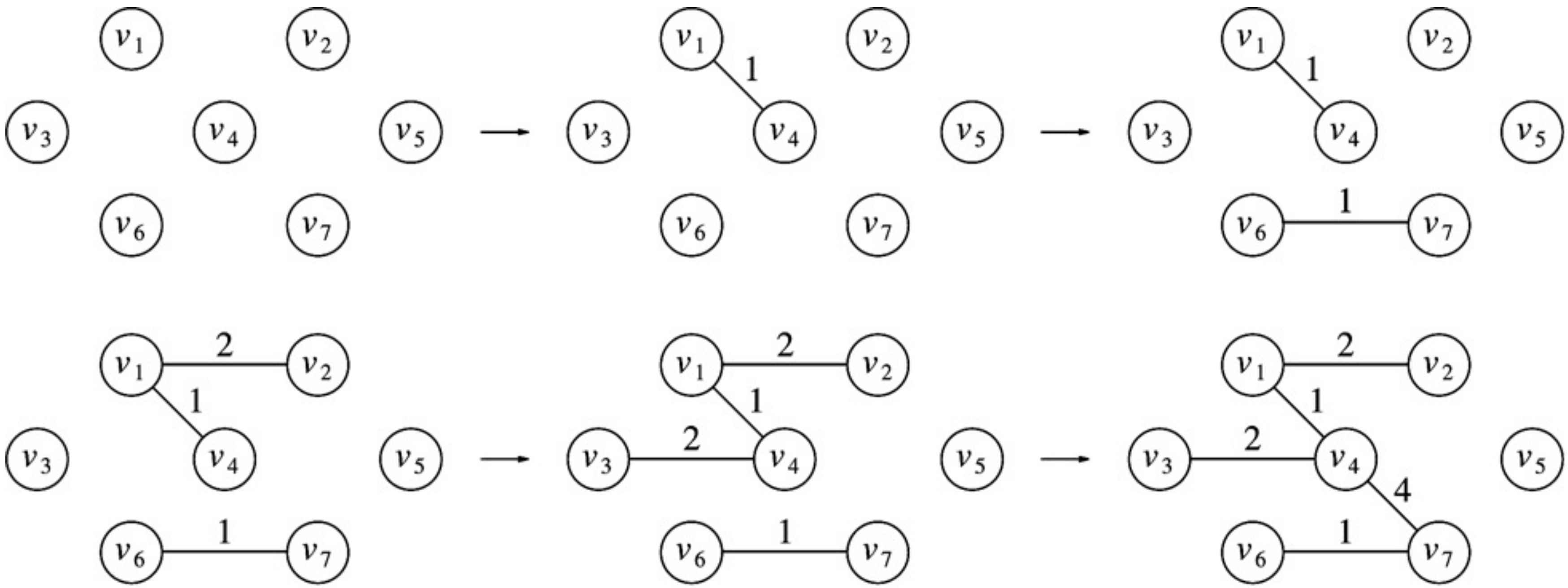
v7 selected **v6 selected** **The End**

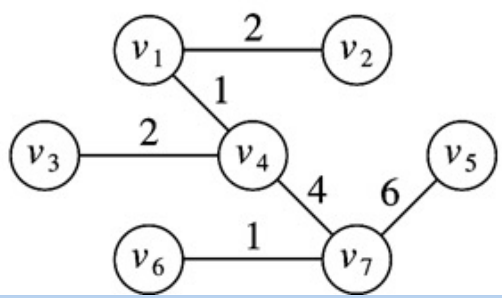
Kruskal

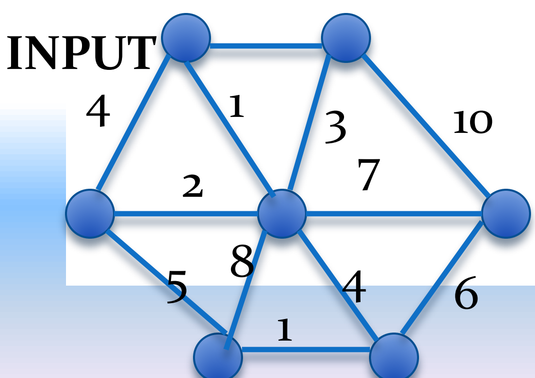
- Maintain a forest (collection of trees)
  - Initially: each node is a tree
- By adding one edge two trees are merged
- Always pick the minimum edge
  - Note that you should not connect via an edge two nodes of the same tree !

Kruskal visualization:
https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/mst.html

Kruskal’s algorithm







Edge's sorted via length

Edge	Weight	Action	
(v_1, v_4)	1	Accepted	Merge Trees
(v_6, v_7)	1	Accepted	-//-
(v_1, v_2)	2	Accepted	-//-
(v_3, v_4)	2	Accepted	-//-
(v_2, v_4)	3	Rejected	
(v_1, v_3)	4	Rejected	
(v_4, v_7)	4	Accepted	-//-
(v_3, v_6)	5	Rejected	
(v_5, v_7)	6	Accepted	-//-

Edge's sorted via length

Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

Sets: {v1}, {v2}, {v3}, … {v7}

{v1, v4}, {v2}, {v3},{v5},{v6}, {v7}

{v1, v4}, {v2}, {v3},{v5},{v6, v7}

{v1, v4, v2}, {v3},{v5},{v6, v7}

{v1, v4, v2, v3},{v5},{v6, v7}

WHY?

{v1, v4, v2, v3, v6, v7} {v5}

WHY?

{v1, v4, v2, v3, v6, v7, v5}

Implementation

- Via union-find. Each **set** of vertices is a tree.
- Initially each vertex is in own **set**.
- Keep edges into a priority queue (why?)
- Cost is **$O(\,|E|\,\log|E|\,)$**
  - **But in practice cost is much lower**

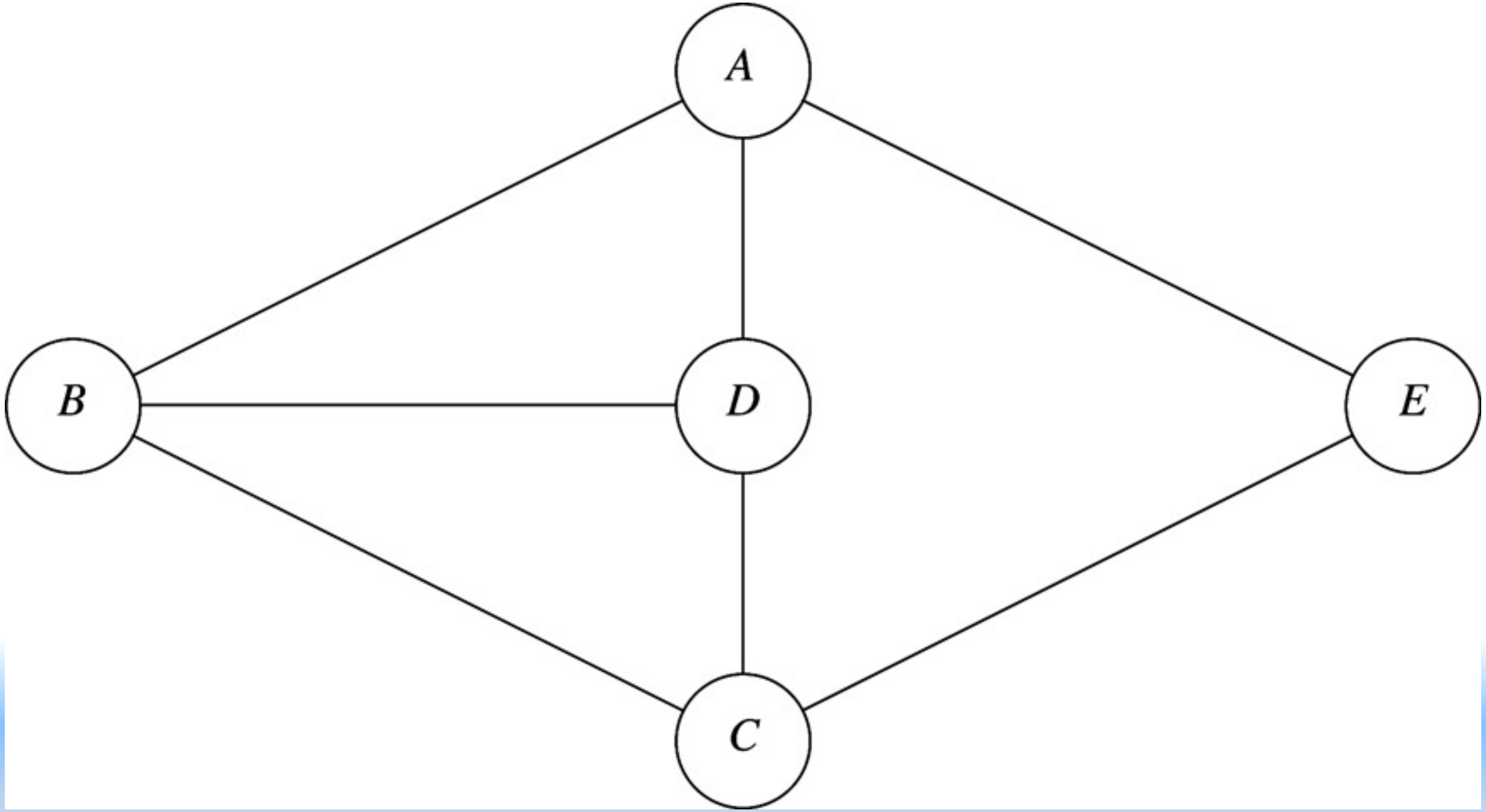
```
void Graph::kruskal() {
    int accepted_edges = 0;
    DisjointSet ds(num_vertices_);

    PriorityQueue<Edge> pq;
    pq.BuildQueue(GetAllEdges());
    Edge e;
    Vertex u, v;
    while (accepted_edges < num_vertices_ - 1) {
      pq.deleteMin(e); // Edge e = (u, v)
      SetType uset = ds.find(u);
      SetType vset = ds.find(v);
      if (uset != vset) {
        accepted_edges++;
        ds.unionSets(uset, vset);
      }
    }
}
```

Depth First Search

```
void Graph::dfs(Vertex v) {
    v.visited = true;
    for each Vertex w adjacent to v
        if (!w.visited)
            dfs(w);
}
```

A recursive implementation

A graph



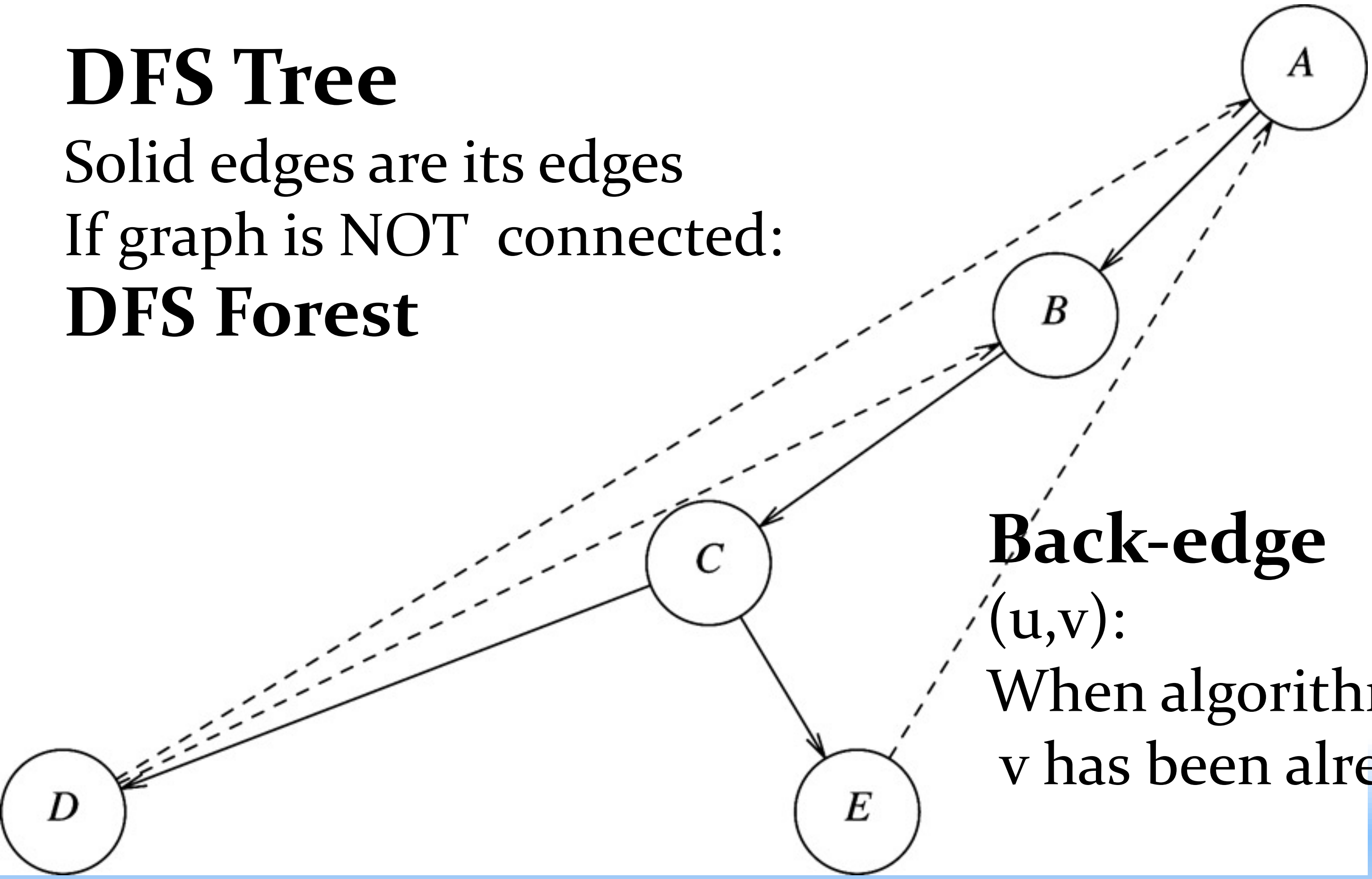
DFS of graph

DFS Tree

Solid edges are its edges

If graph is NOT connected:

DFS Forest



Back-edge

(u,v):

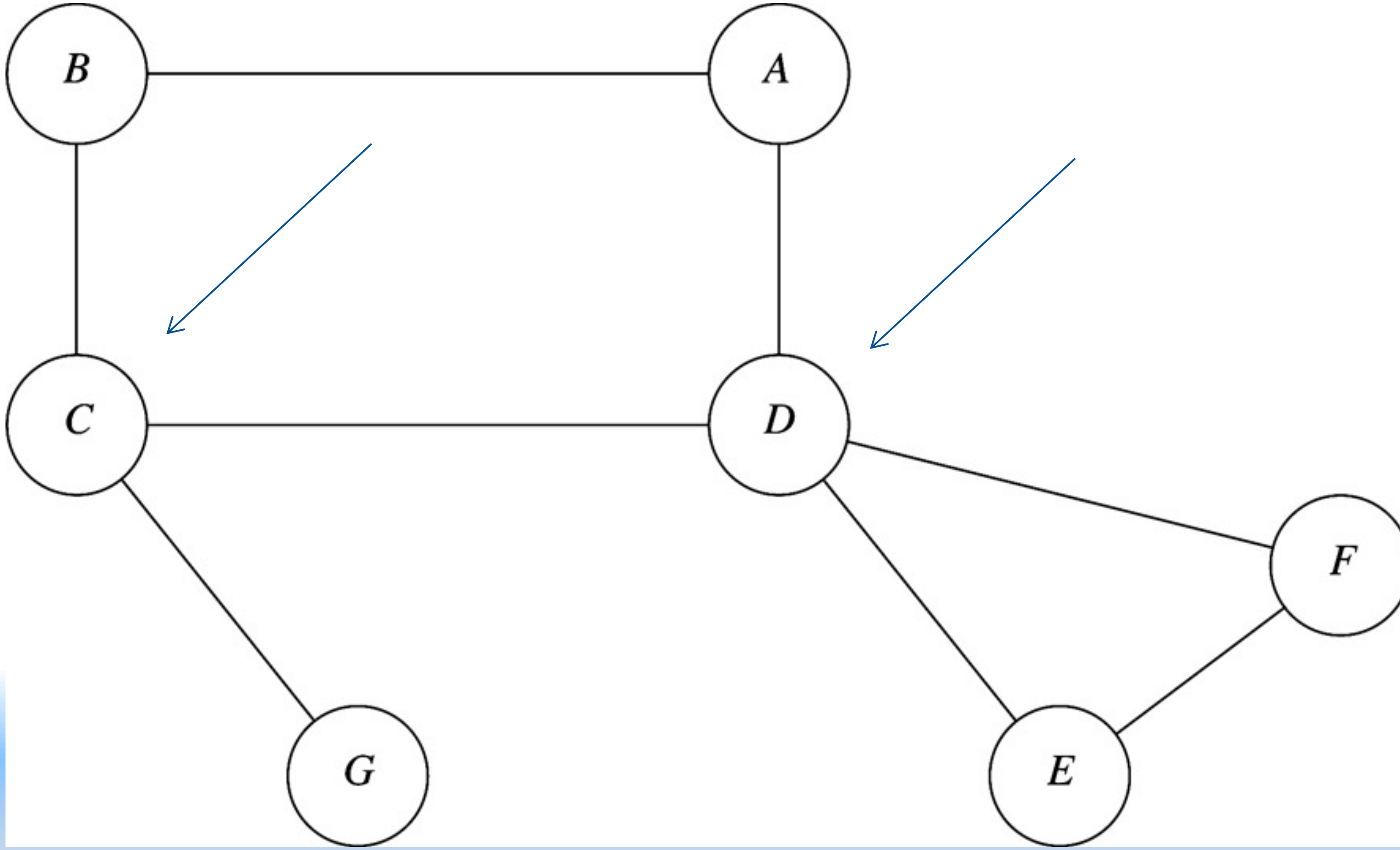
When algorithm selects u

v has been already visited

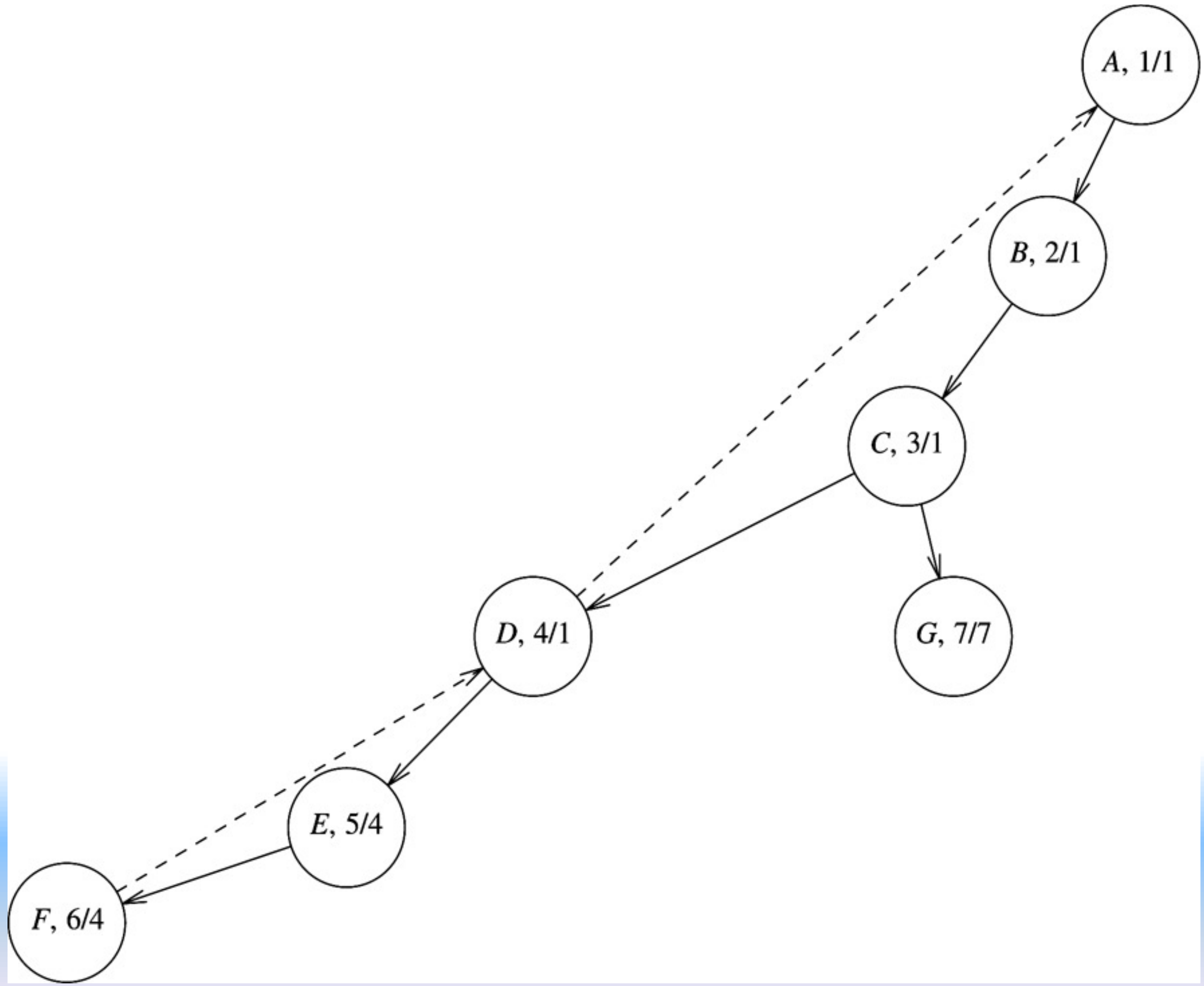
Biconnectivity

- A connected undirected graph is **biconnected** iff there is no vertex whose removal disconnects the rest of the graph.
- **Articulation points:** in a not biconnected graph the vertices whose removals break the connectivity.
- Can be found via **DFS** in linear time on a connected graph.

Not biconnected graph



DFS of graph (back-edges shown)



DFS of graph (back-edges shown)

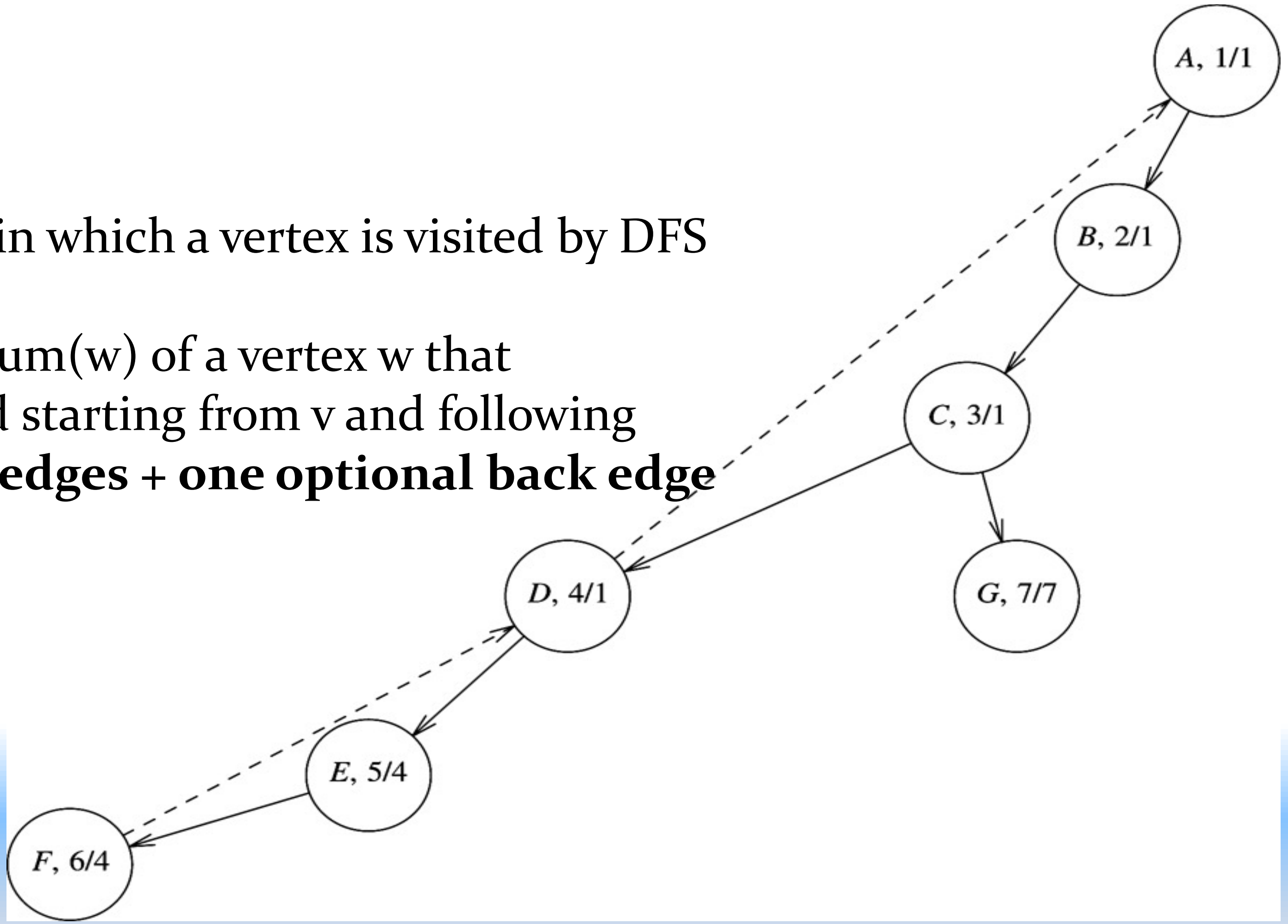
V, n/m

n=**Num(v)**

is the order in which a vertex is visited by DFS

m=**Low(v)**=

the smallest Num(w) of a vertex w that can be reached starting from v and following **zero or more edges + one optional back edge**



Low(v)

Low(v) is the minimum of:

1. Num(v) → No edges
2. The lowest Num(w) among all back edges **(v,w)**
   → Only back edge
3. The lowest Low(w) among all tree edges **(v,w)**
   → Follow some edges + optional back edge

How to compute Low(v) if you know Num(v) for all vertices? [recursive solution in linear time]

Articulation points

1. **Root** is articulation point iff it has more than one children. [why ?]
2. Any other vertex **v** is articulation point iff it has a child **w** such that **Low(w) >= Num(v)**

   [why?]

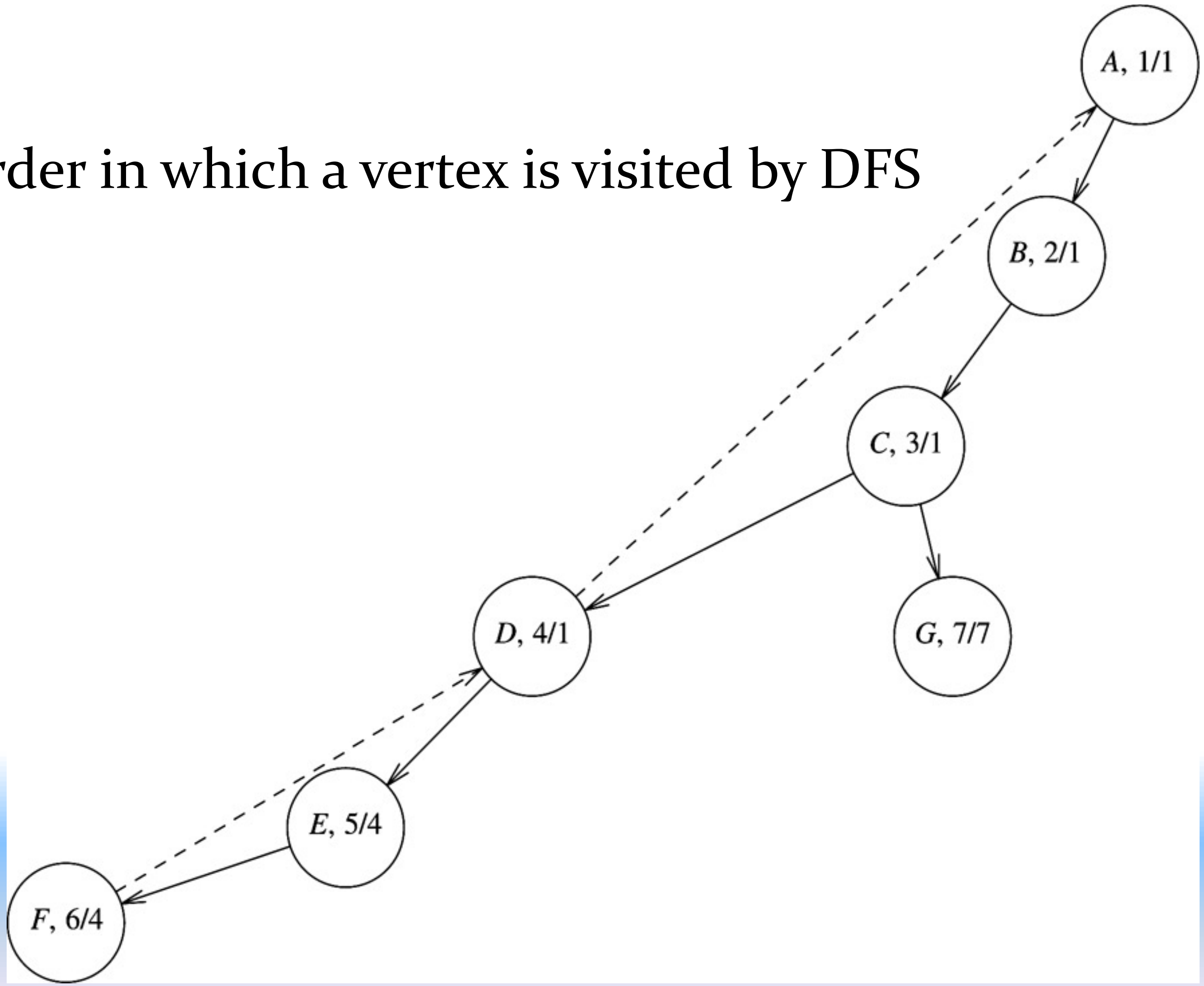
DFS of graph (back-edges shown)

V, n/m

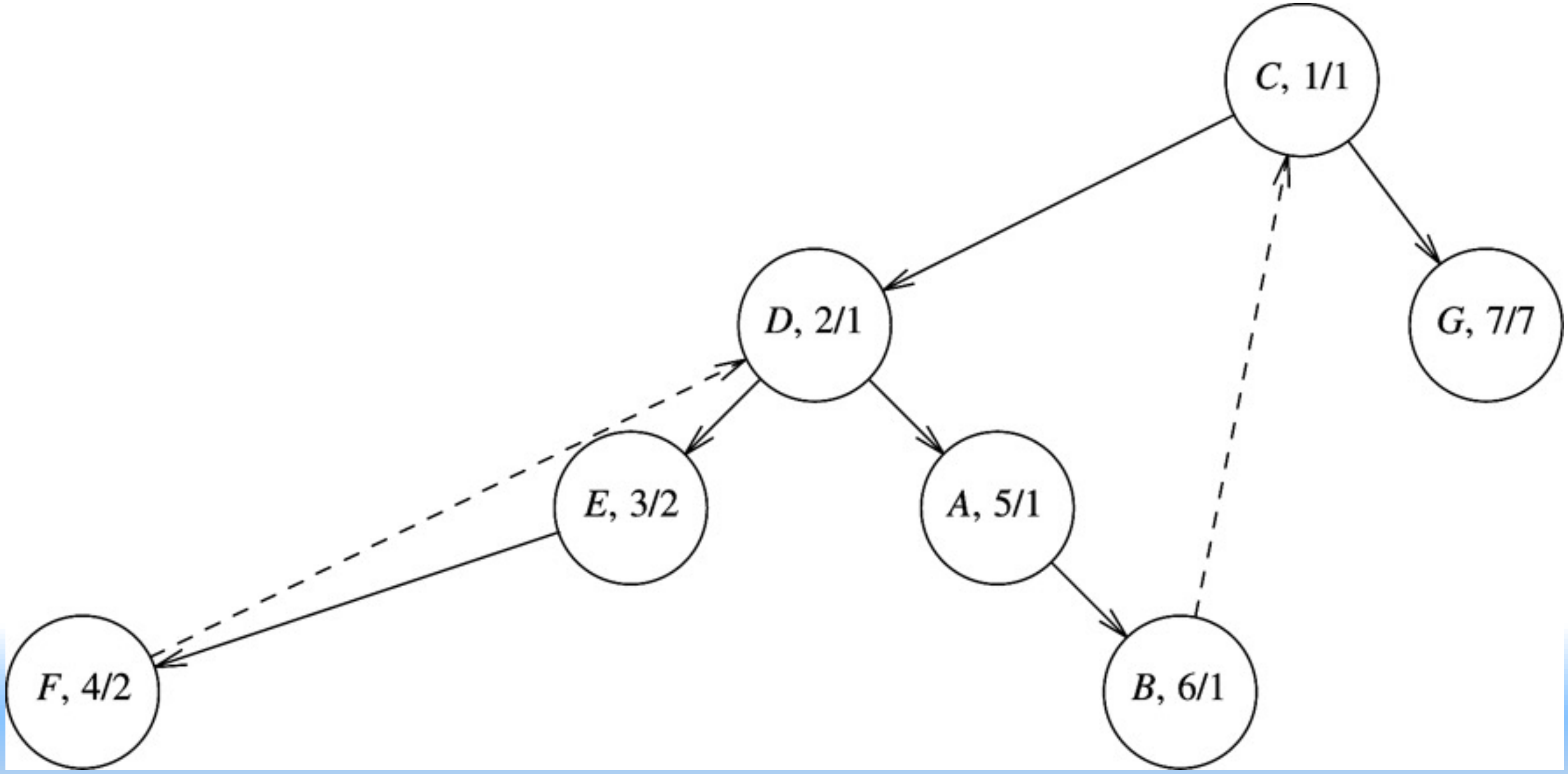
n=**Num(v)**

is the order in which a vertex is visited by DFS

m=**Low(v)**



Another DFS (starting from C)



Implementation

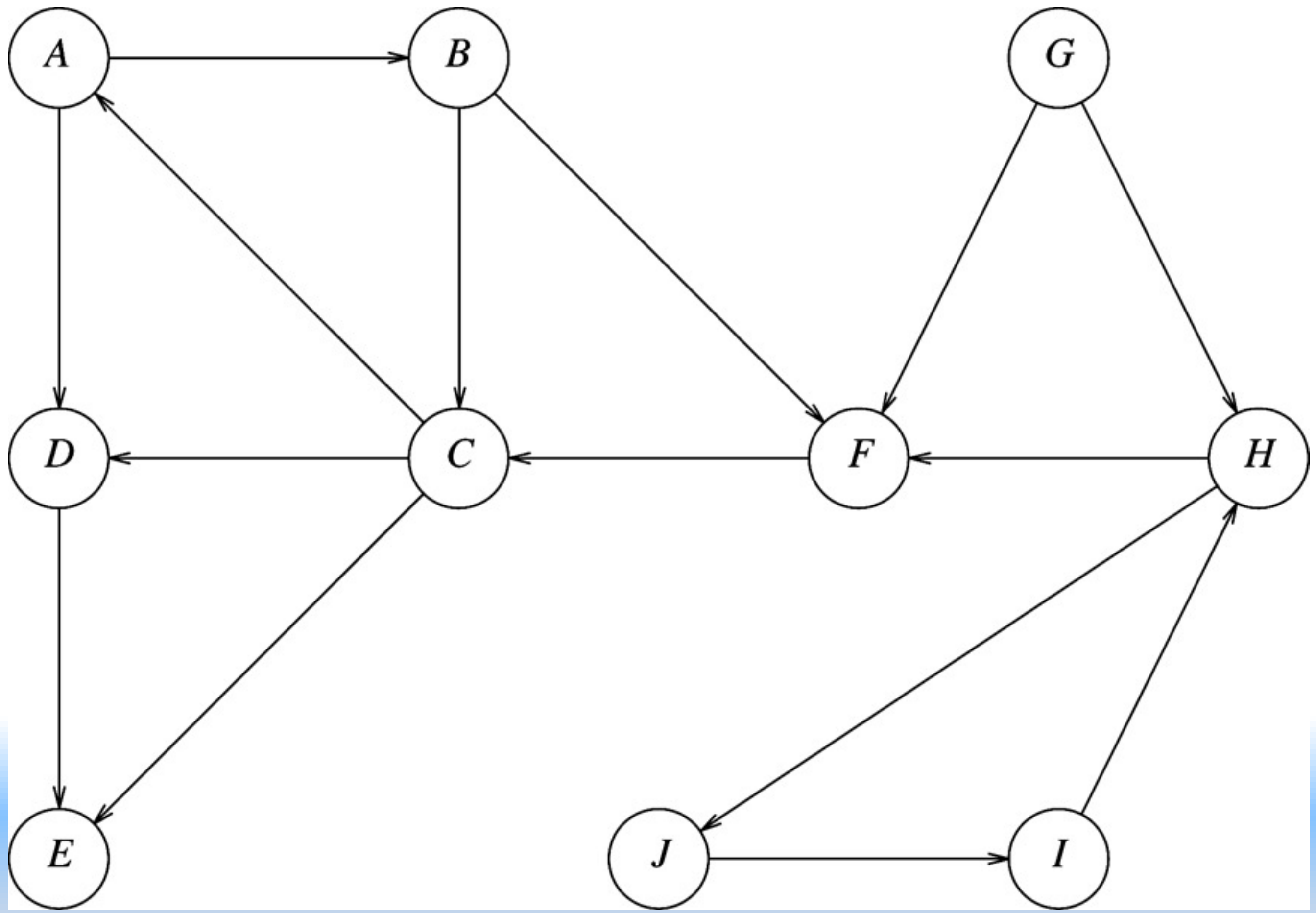
```
// Assign num and compute parents
void Graph::AssignNumber(Vertex v) {
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
        if (!w.visited) {
            w.parent = v;
            AssignNumber(w);
        }
}
```

```
// Assign low; also check for articulation points
void Graph::AssignLow( Vertex v ) {
    v.low = v.num; // Rule 1
    for each Vertex w adjacent to v {
        if (w.num > v.num) { // Forward edge
            AssignLow(w);
            if (w.low >= v.num)
                cout << v << "is an articulation point" << endl;
            v.low = min(v.low, w.low); // Rule 3
        }
        else if (v.parent != w) { // Back edge
            v.low = min(v.low, w.num) // Rule 2
        }
    } // End for each Vertex w
} // End AssignLow()
```

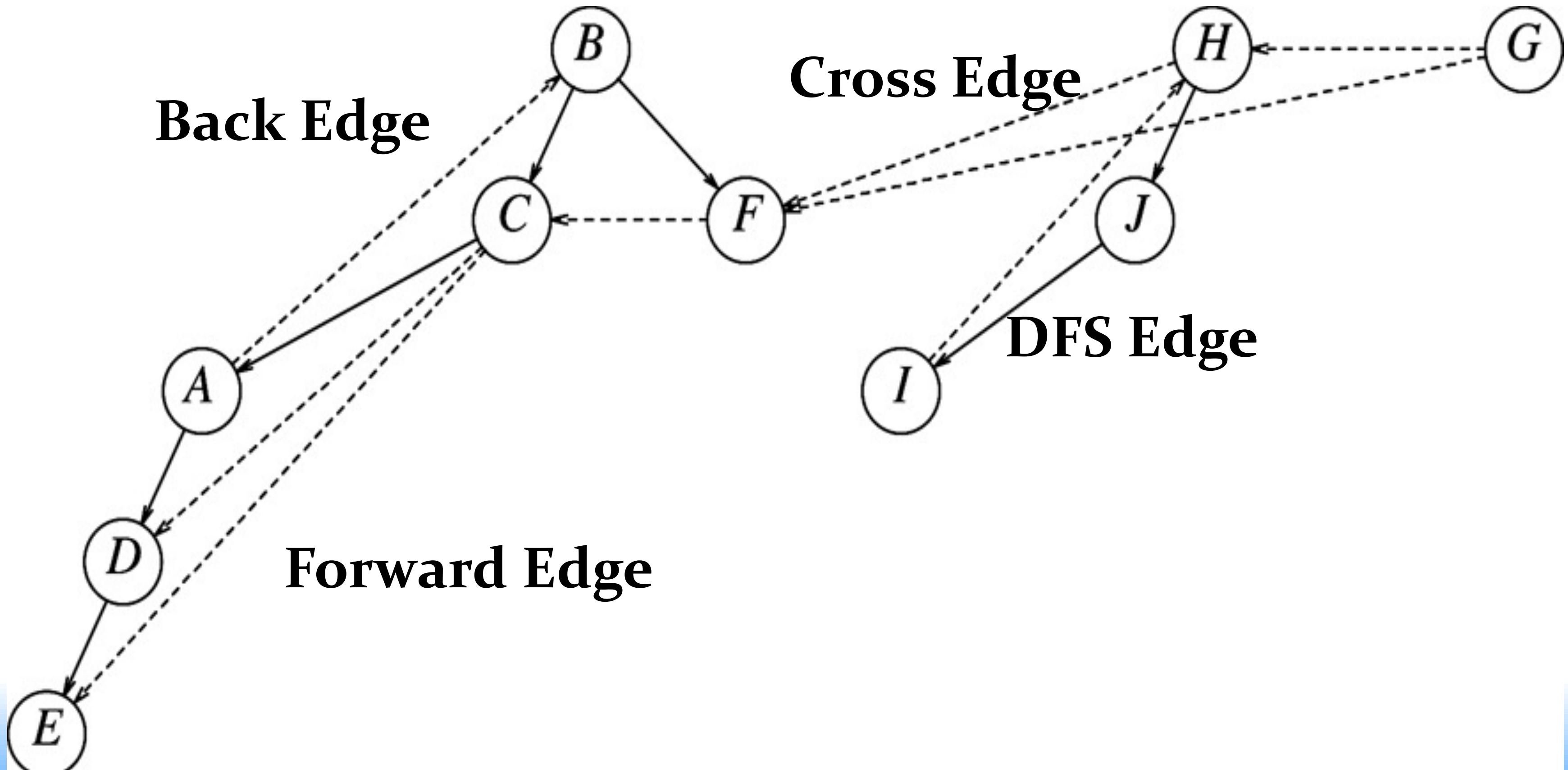
Directed Graphs

- DFS can traverse whole graph only if it is STRONGLY connected
- DFS from a starting vertex -> if not all vertices covered start DFS from unvisited vertex …

A directed graph



DFS



Start from B, then from H, and finally from G

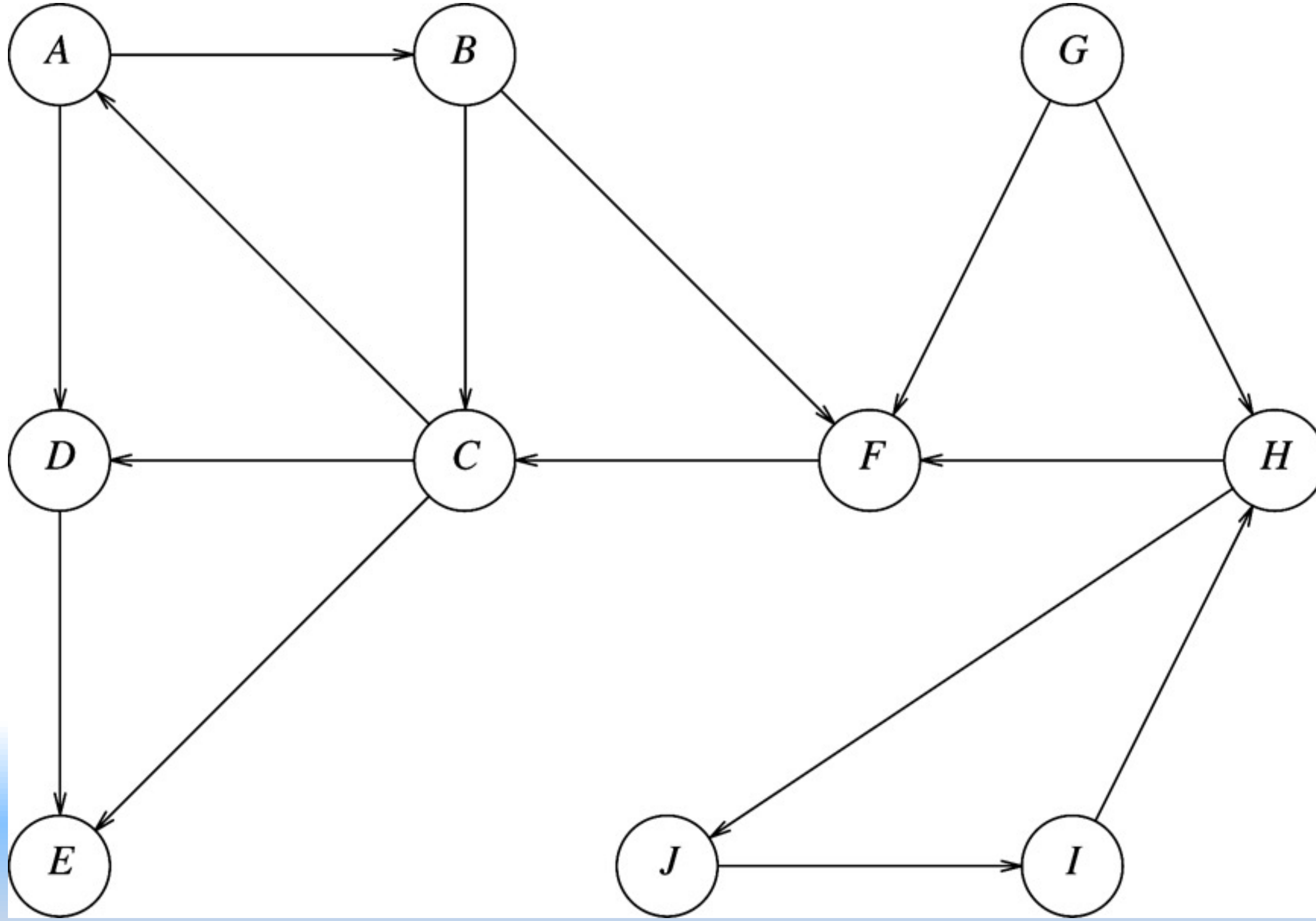
DFS directed graphs

- Determine whether a graph is acyclic.

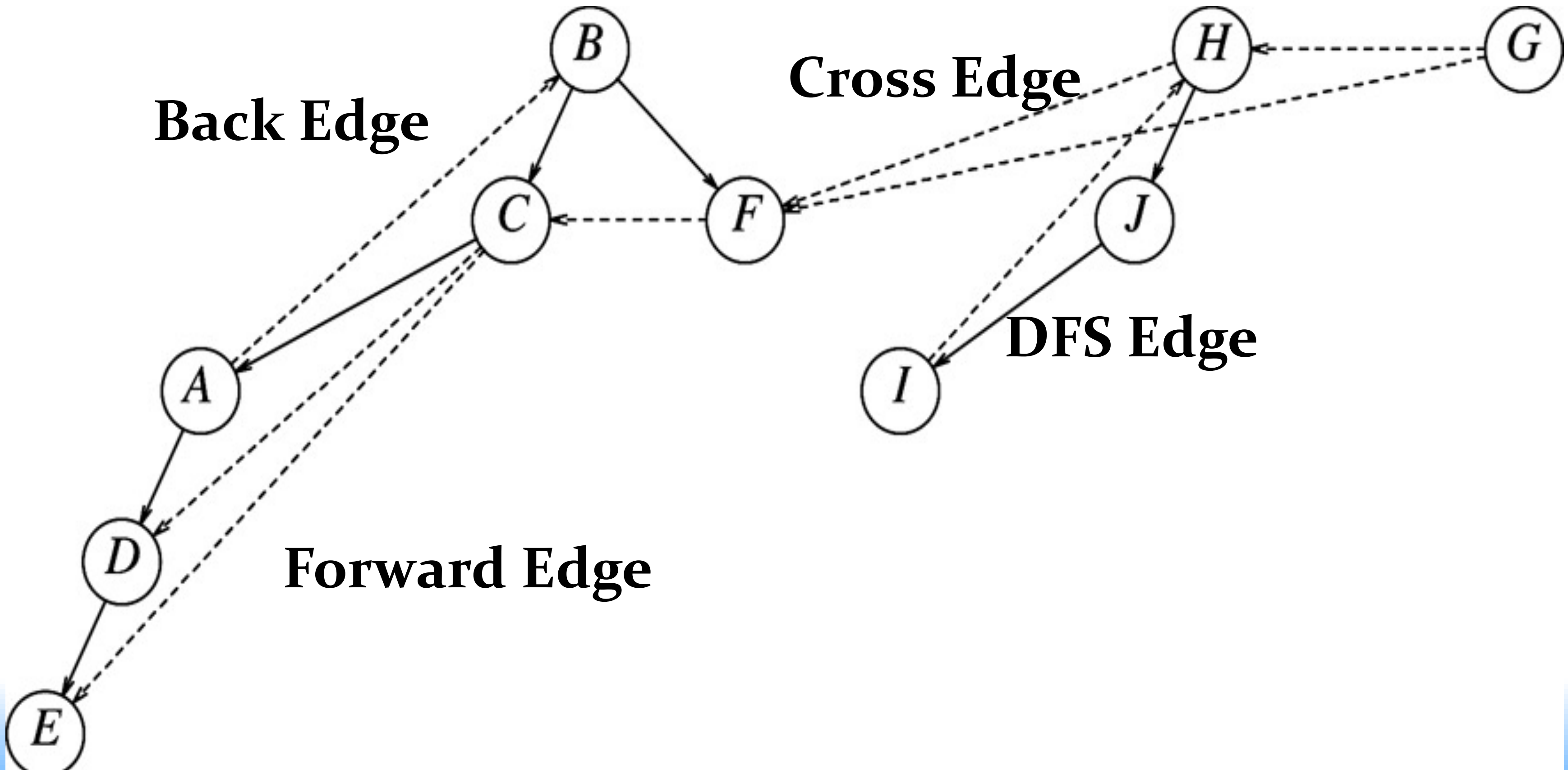
Finding strong components

- DFS on G-> DFS forest
- Postorder traversal assigns numbers to vertices => graph Gr
- Do DFS on Gr starting from the vertex with the highest number in postorder traversal.

A directed graph

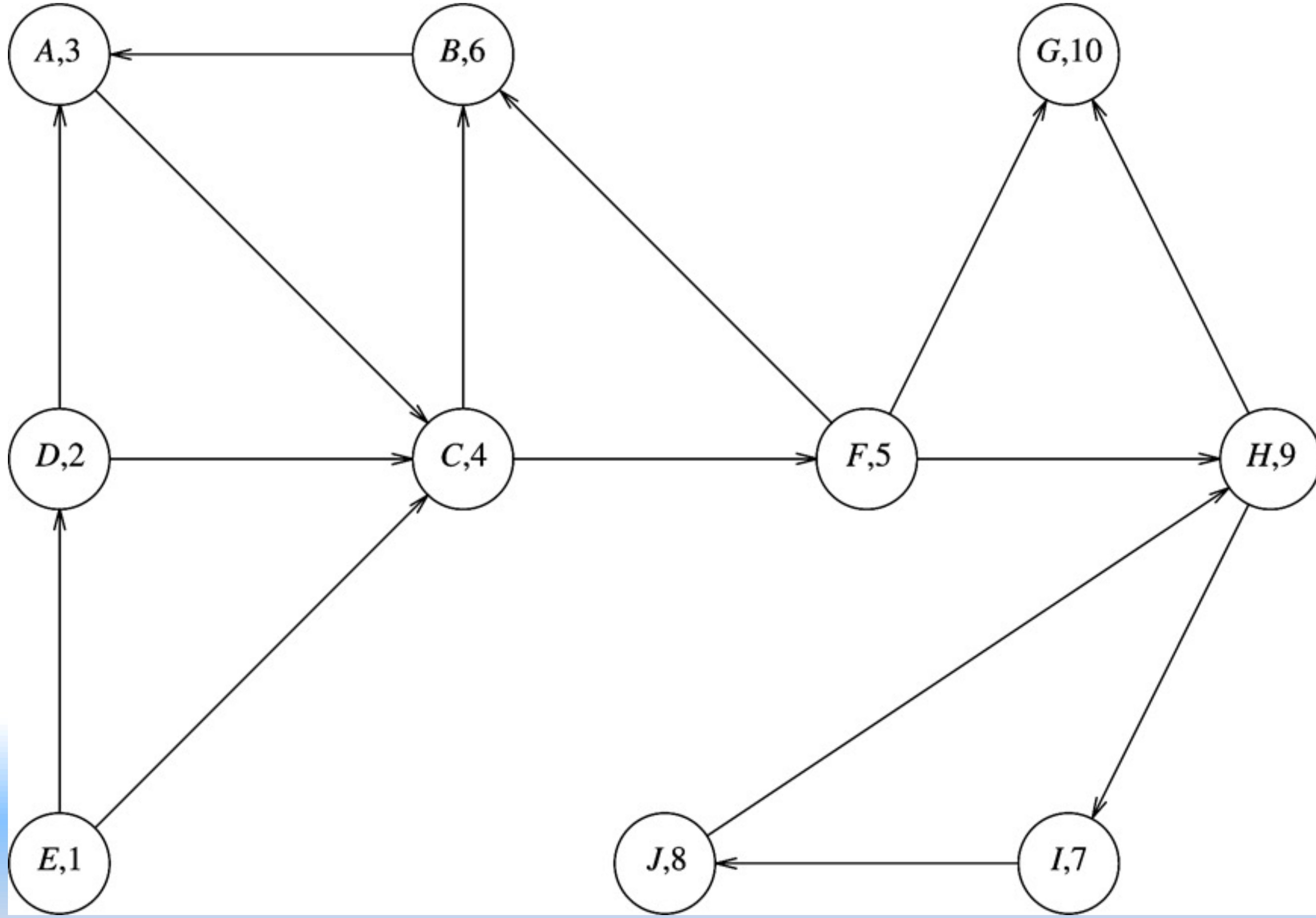


DFS

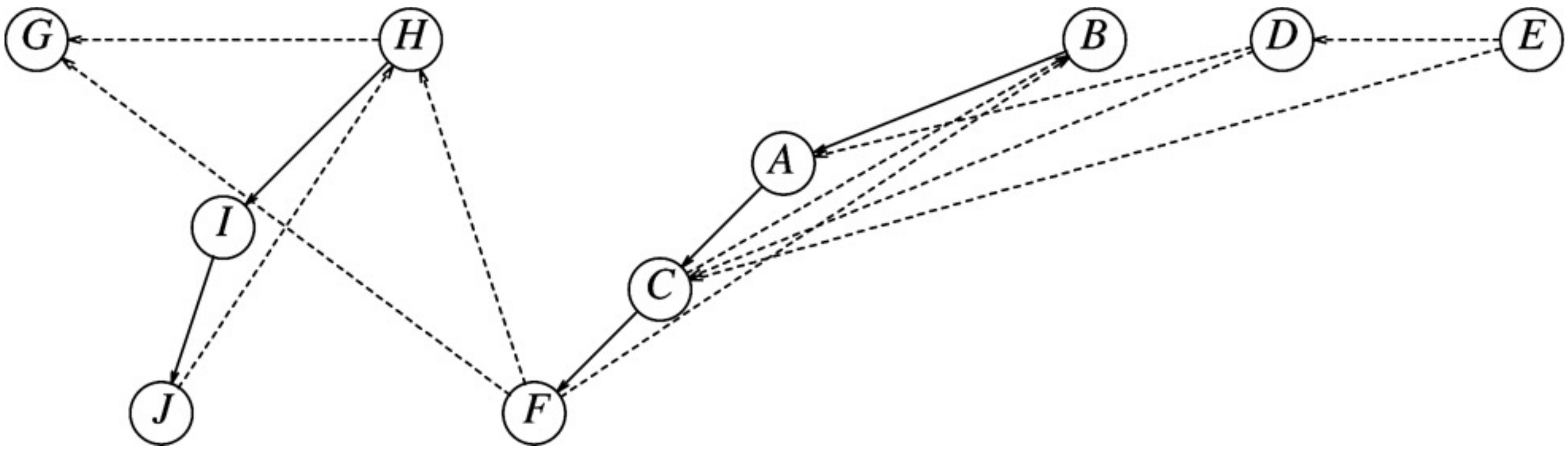


Start from B, then from H, and finally from G

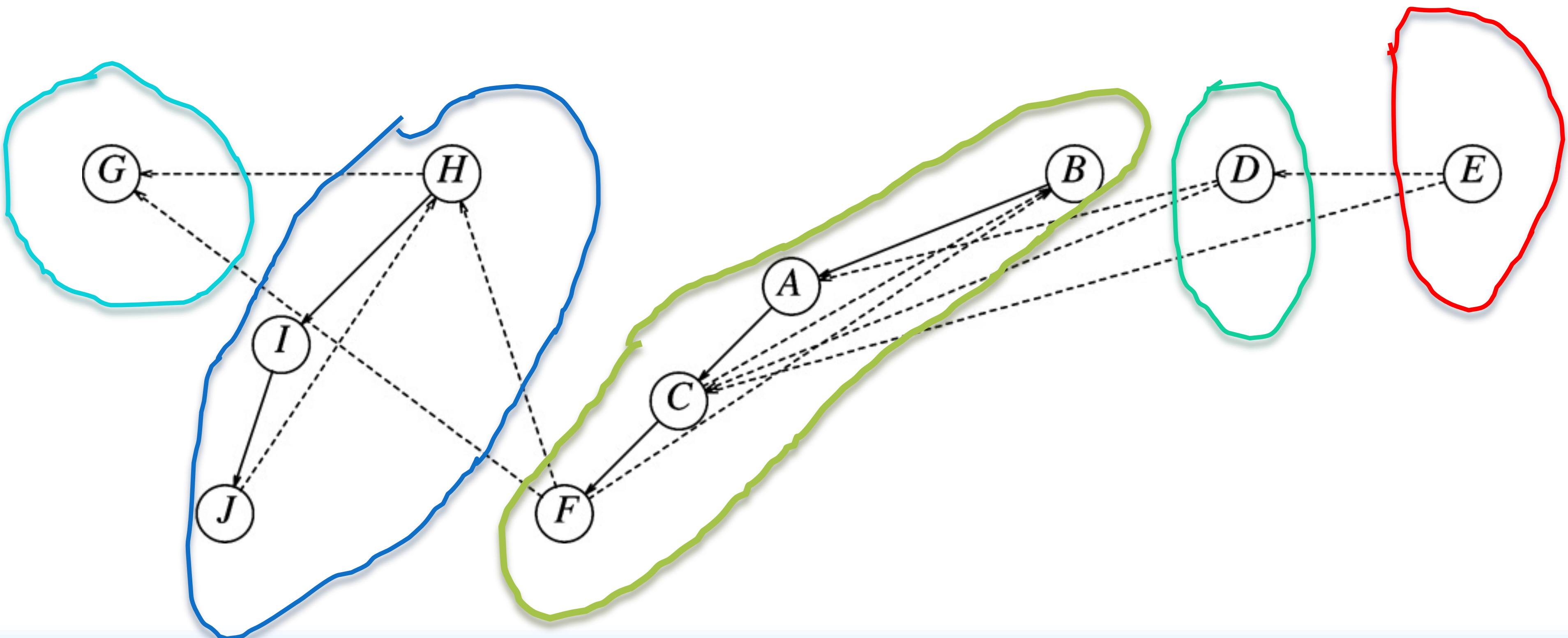
Numbers based on postorder traveral of DFS forest



DFS on Gr starting from higher numbered vertex



DFS on Gr starting from higher numbered vertex



STRONLY connected components

Why does the algorithm work?