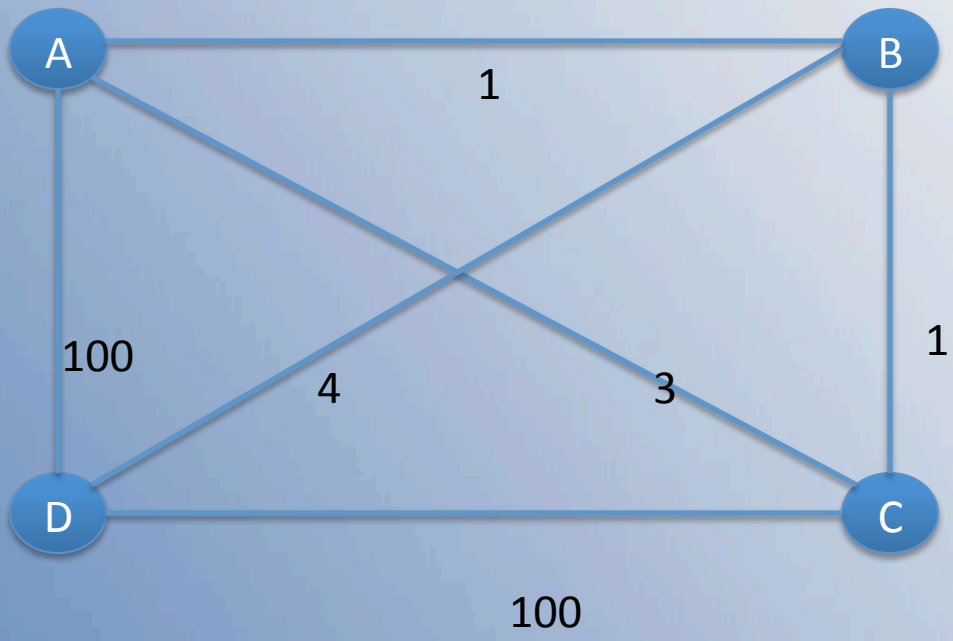
Graph Algorithms I

DFS, BFS, Spanning Tree, Topological Sort, Disjoint Sets

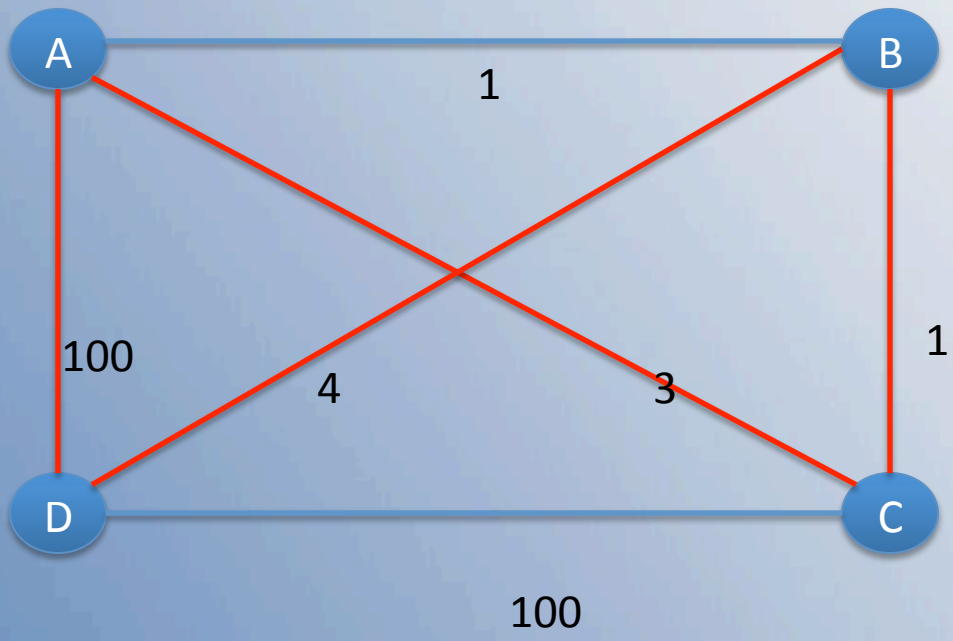
When DP does not apply

- Find shortest loop through this network:



When DP does not apply

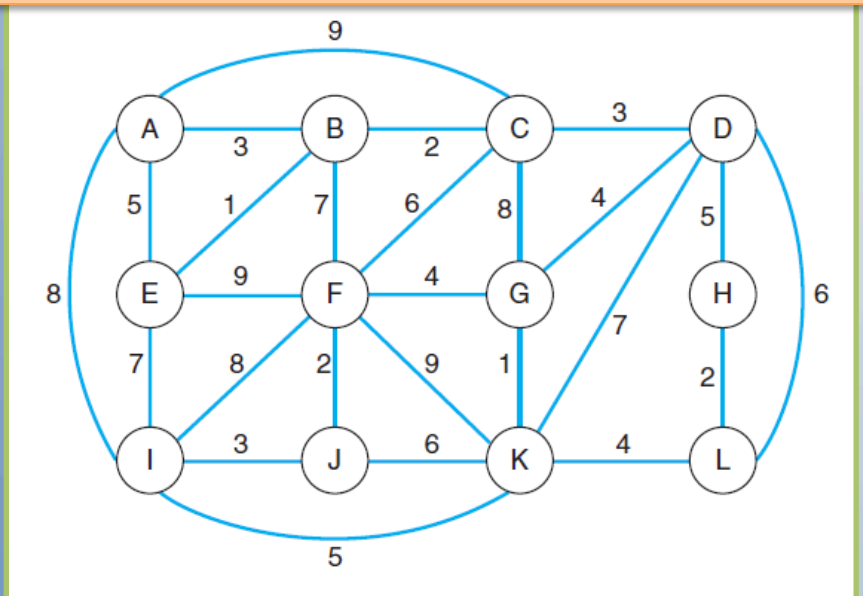
- Find shortest loop through this network:



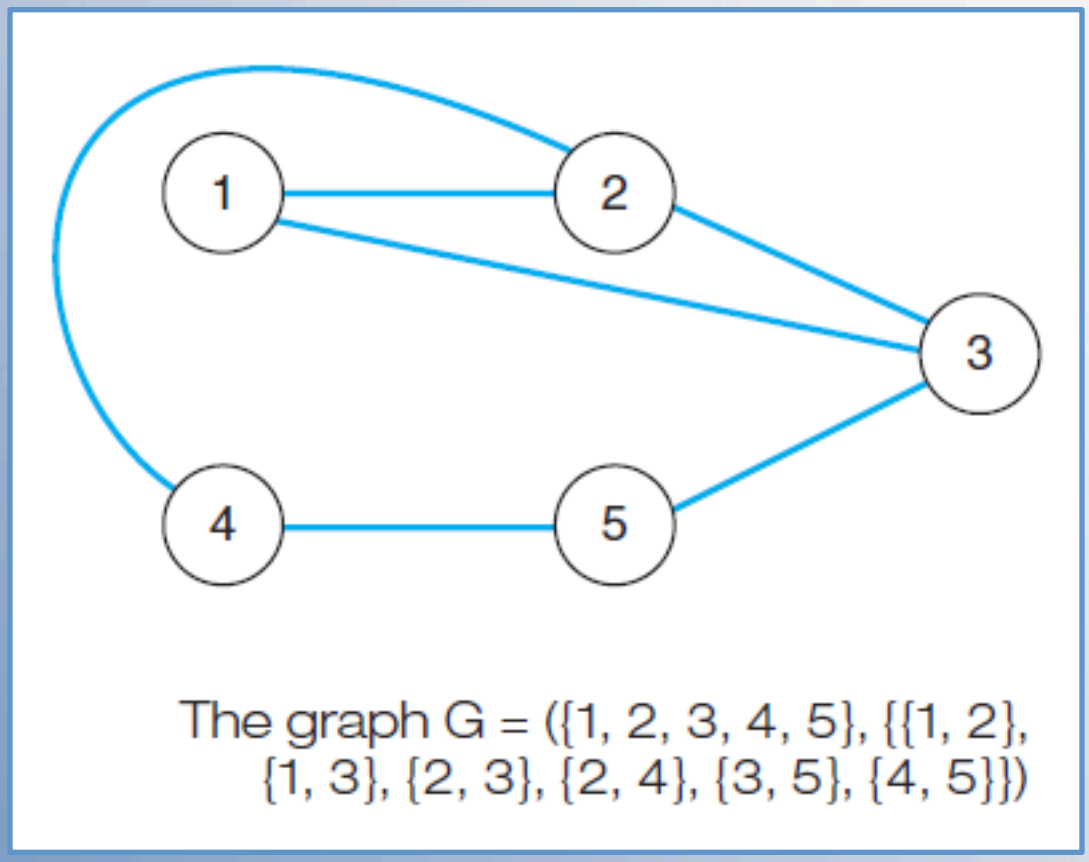
Does not use shortest path from A to C or from A to B

Graph

- Graphs are formal descriptions for a variety of situations: road maps, routes, shipping points, nodes in a network, employees and tasks, etc.
- A graph G(V,E) consists of vertices (nodes) and edges (arcs). Edges can have weights and direction. An edge connects two nodes
- We can “traverse” an edge from A to B, “visiting” each node.

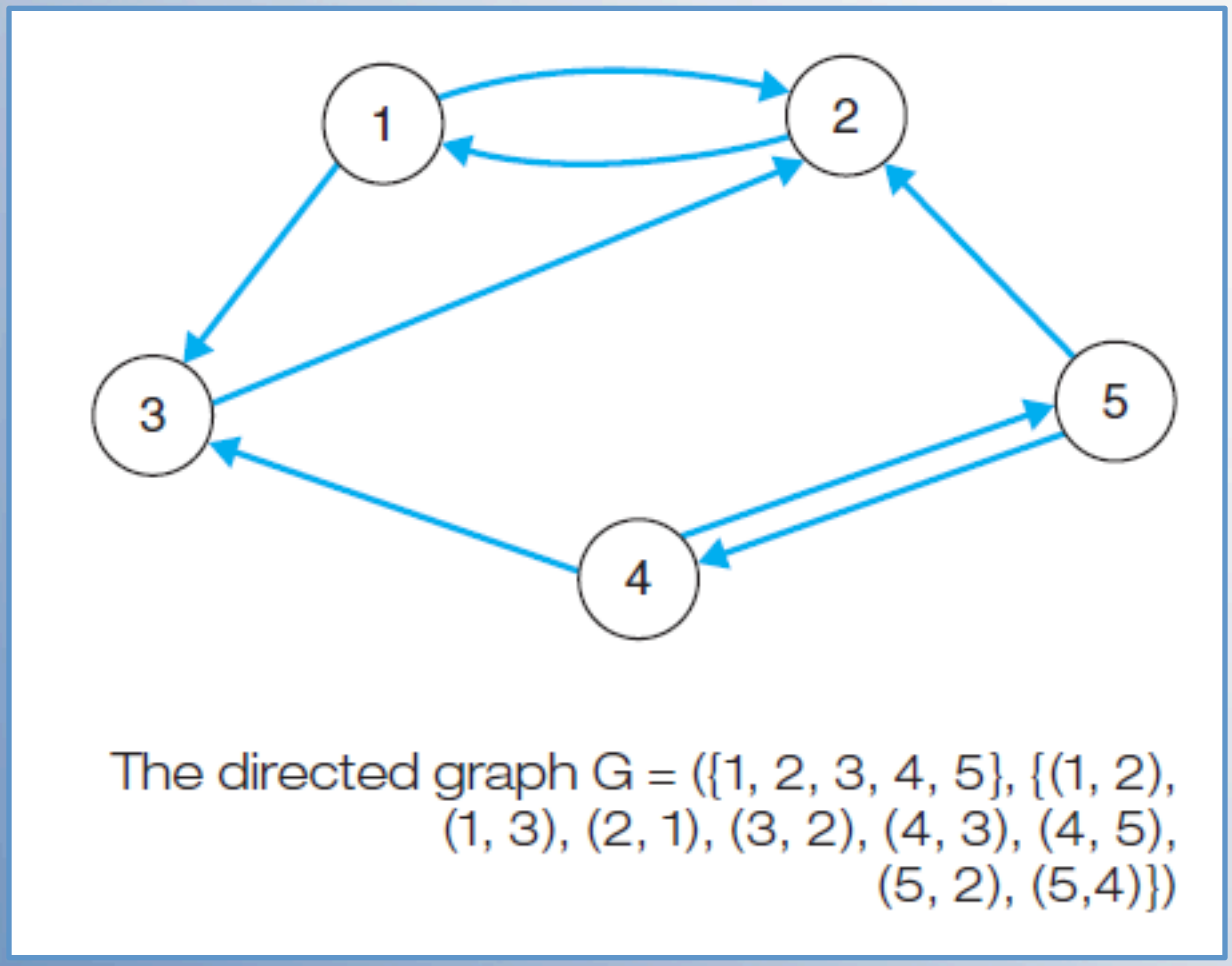


Undirected Graph



Edge linking u and v usually written (u,v)

Directed Graph



The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

Edge linking u and v usually written (u,v); order matters if directed

Terminology

- A **complete** graph has a direct edge between every pair of nodes.
  - Undirected: $n(n-1)/2 = (n^2 - n)/2$
  - Directed: $n^2 - n$
- A **subgraph** is a graph S(V,E) with a subset of the vertices and edges of graph G(V,E).

Terminology

- A **path** between any two nodes is a sequence of *adjacent* (connected) edges.
- The **cost** of a path is the sum of the edge weights along that path.
- A **connected** (undirected) graph is such that every pair of nodes is connected via some path.
- A **strongly connected** graph is a directed graph that is connected by paths that follow direction.

Terminology (Directed Graphs Only)

- A **source** is a vertex with all edges going out.
- A **sink** is a vertex with all edges coming in.

Trees

- A graph is **acyclic** if there are no paths in the graph that start and end on the same node. (e.g. There are no *cycles*.)
- A *tree* is a connected acyclic graph.
- A tree on n nodes must have n-1 edges
- Fewer edges guarantees disconnected
- More guarantees cycle

Terminology

- A **spanning tree** is a subgraph which is a tree and contains every node.
- A **minimum spanning tree** is a spanning tree with the lowest sum of edge costs.
- A graph is **acyclic** if there are no paths in the graph that start and end on the same node. (e.g. There are no *cycles*.)

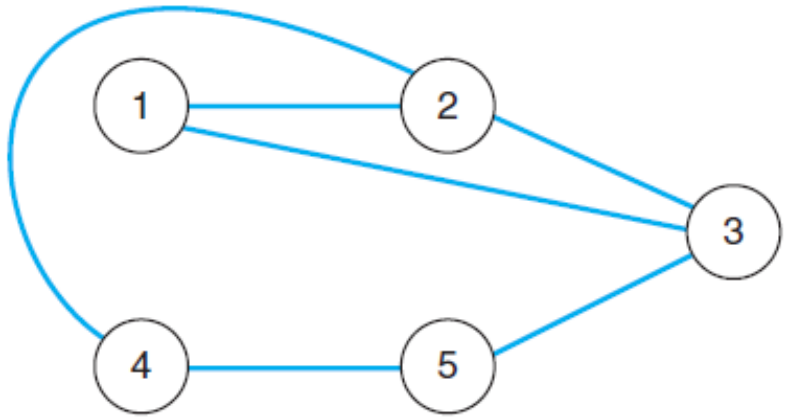
Representation

- A graph can be represented in two forms:
  - **Adjacency List** (Sparse)
  - **Adjacency Matrix** (Dense)

Adjacency Matrix

N x N matrix.

$$\text{AdjMat}[i, j] = \begin{cases} 1 & \text{if } v_i v_j \in E \\ 0 & \text{if } v_i v_j \notin E \end{cases} \qquad \text{for all } i \text{ and } j \text{ in the range 1 to } N$$



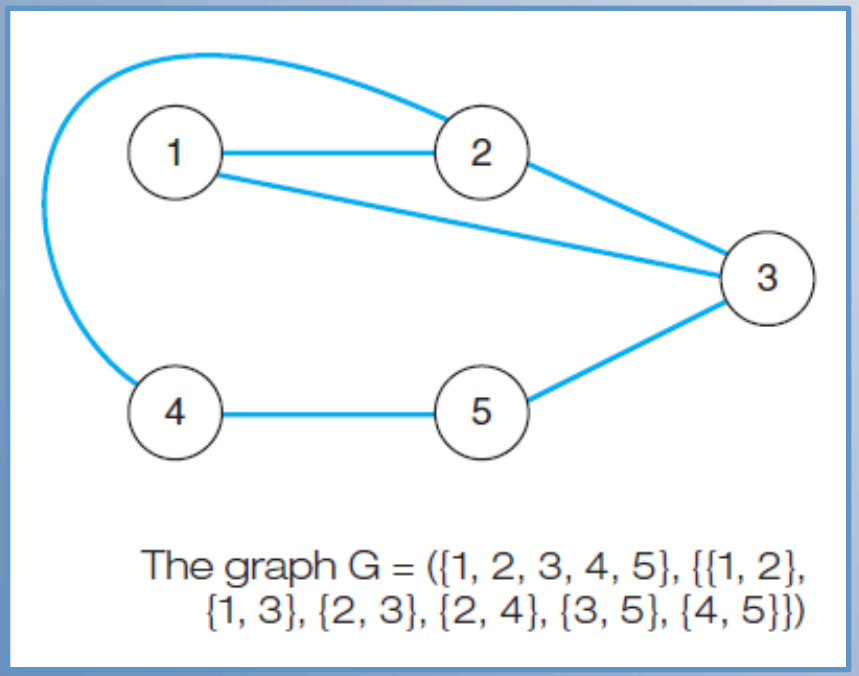
The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

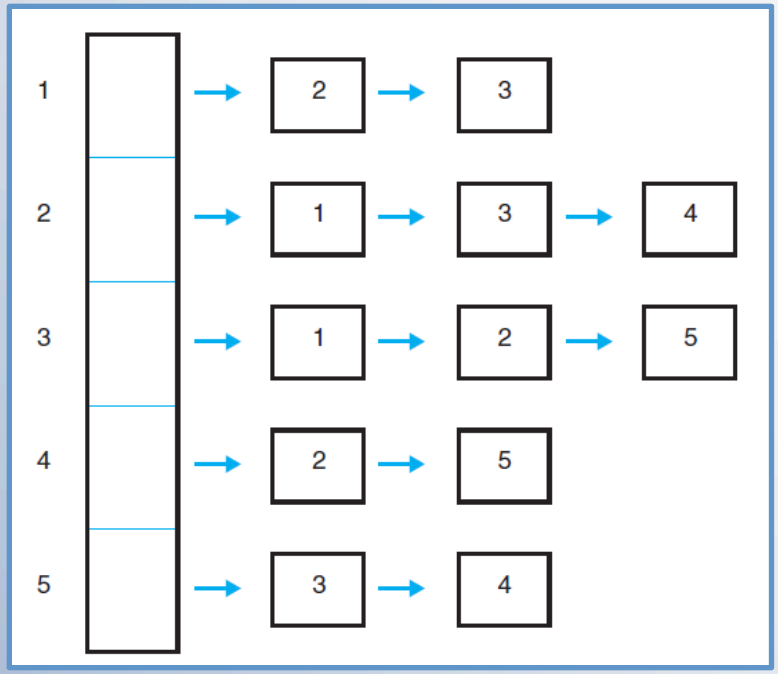
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	1
5	0	0	1	1	0

Adjacency List

Array of size N, with linked lists.

One array element per node, with a linked list or array of outgoing edges.





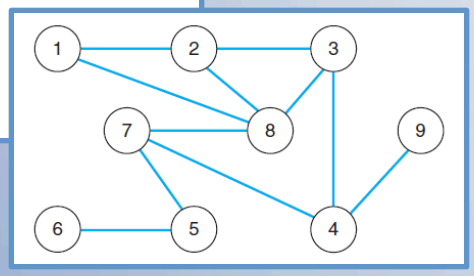
Graph Traversal

Goal: Visit each node once in a graph.

- Depth-First (uses stack)
- Breadth-First (uses queue)

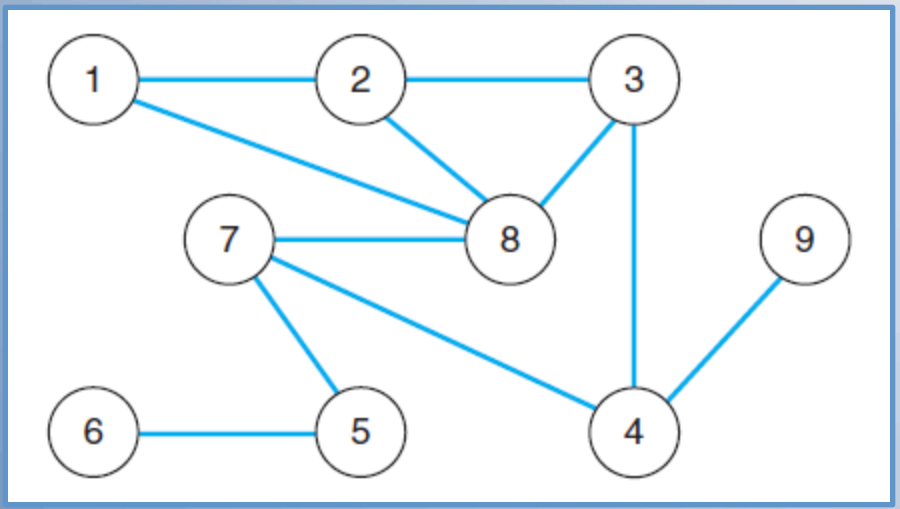
Breadth-First

```
BreadthFirstTraversal(G, v)
G  is the graph
v  is the current node

Visit( v )
Mark( v )
Enqueue( v )
while the queue is not empty do
   Dequeue( x )
   for every edge xw in G do
      if w is not marked then
         Visit( w )
         Mark( w )
         Enqueue( w )
      end if
   end for
end while
```



Depth-First

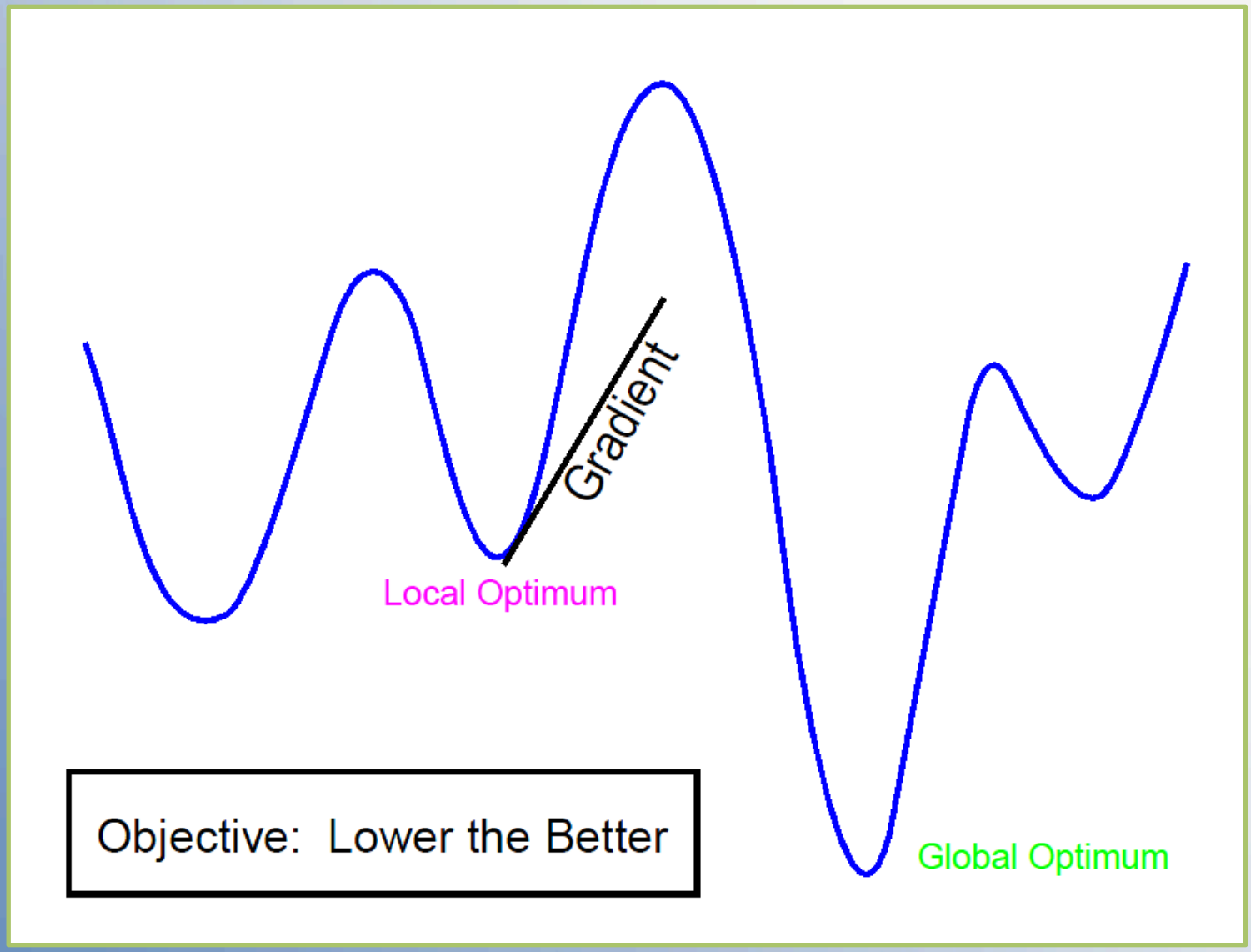
```
DepthFirstTraversal(G, v)
G  is the graph
v  is the current node

Visit( v )
Mark( v )
for every edge vw in G do
   if w is not marked then
      DepthFirstTraversal(G, w)
   end if
end for
```



Minimum Spanning Tree

- **Dijkstra-Prim** Algorithm
  - Independently published in 1950's
  - Edsger Dijkstra
  - RC Prim

- **Greedy Algorithm**
  - Design technique that makes locally optimal choices in hopes of reaching a global optimum.



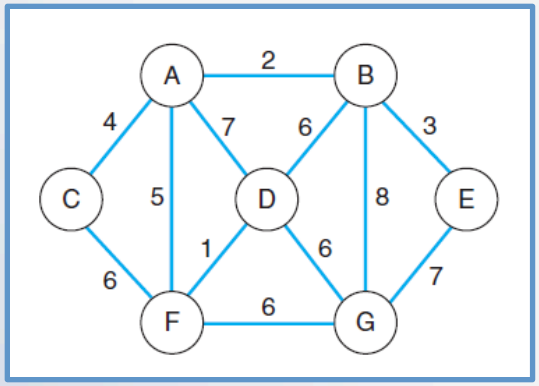
Greedy algorithms are simple to implement, but only certain greedy algorithms produce a true global optimum. Prim and Kruskal are two such algorithms.

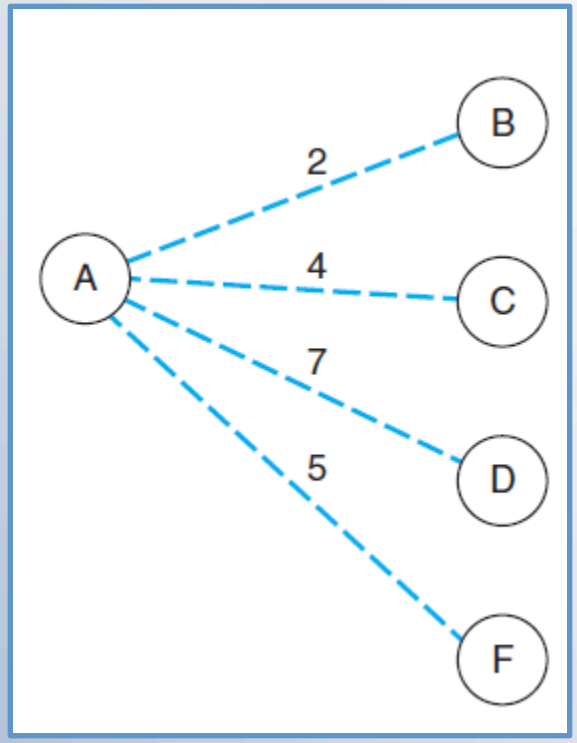
Why do they work?

- We will look at how the algorithms work first, then convince ourselves that they are indeed correct!

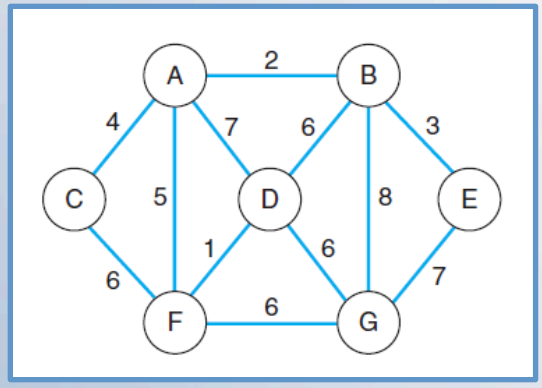
Dijkstra-Prim : MST

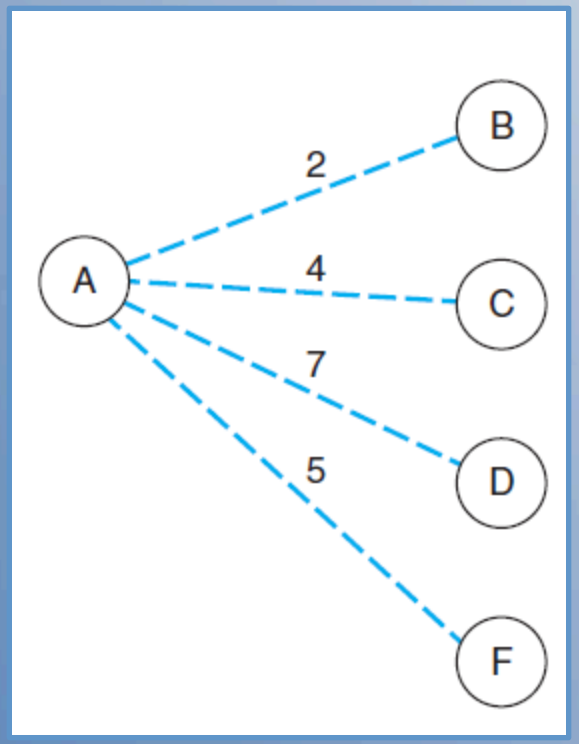
- Start at some node (A)
- Keep adding the lightest edge that joins a new node to current tree
- Break ties arbitrarily (there can be more than one MST)

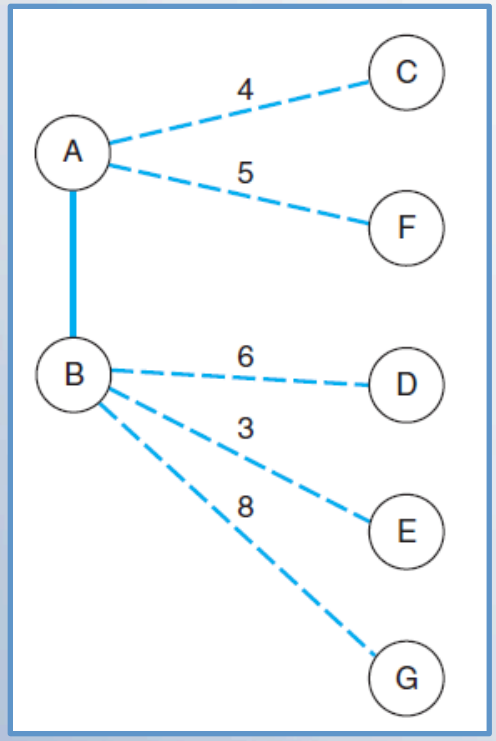




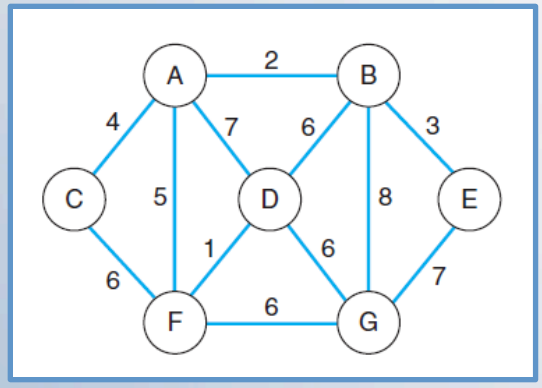
Dijkstra-Prim : MST

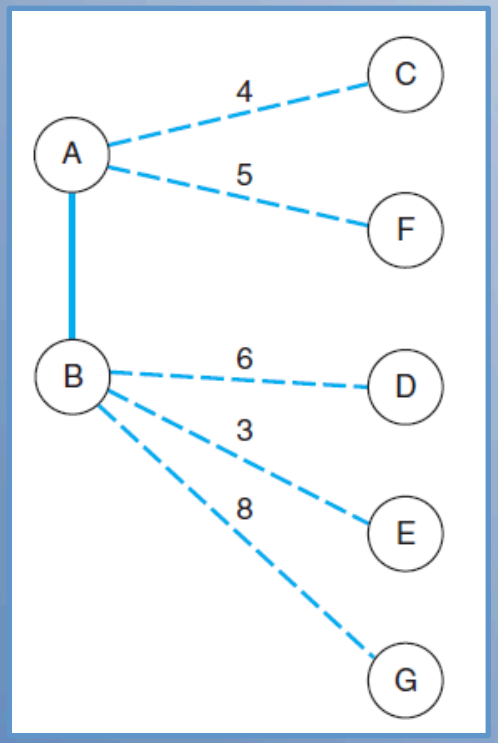


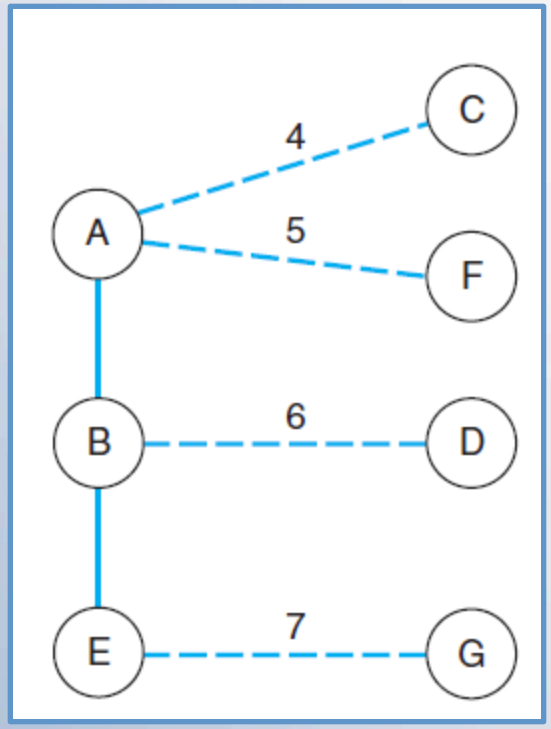




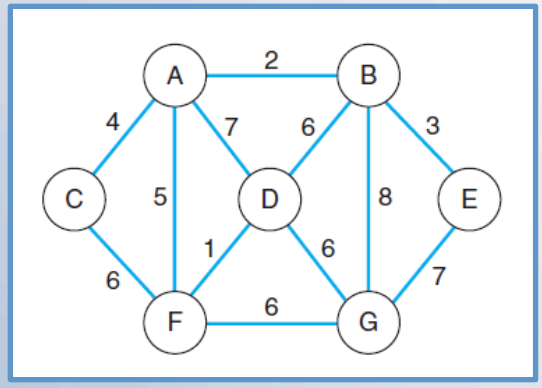
Dijkstra-Prim : MST

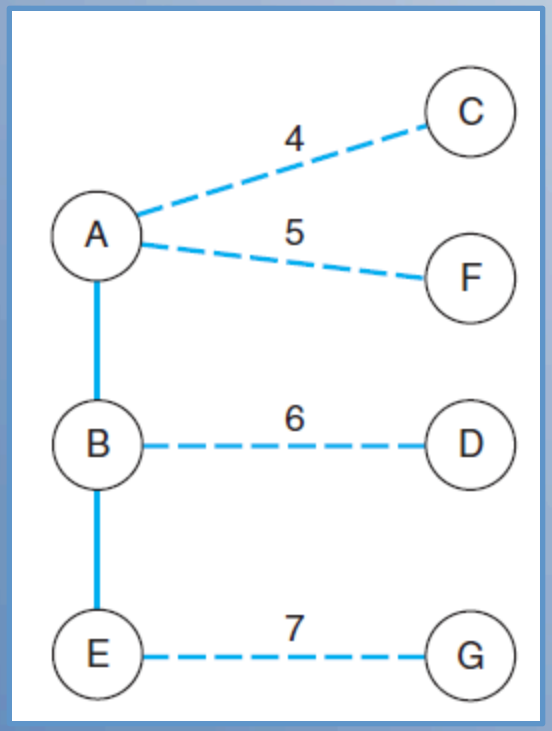


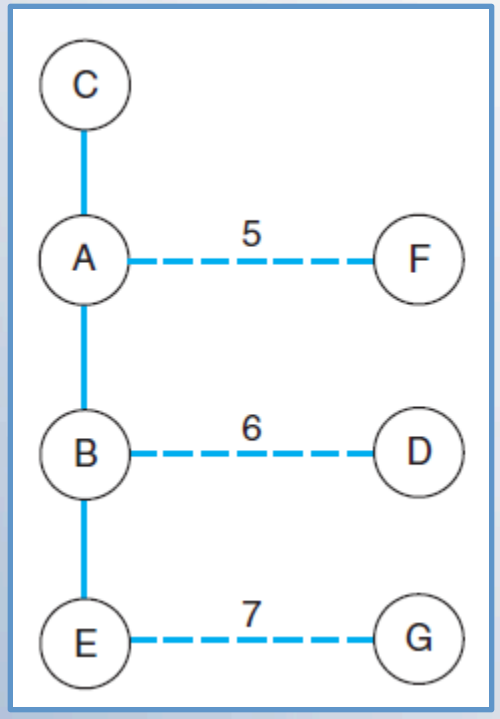




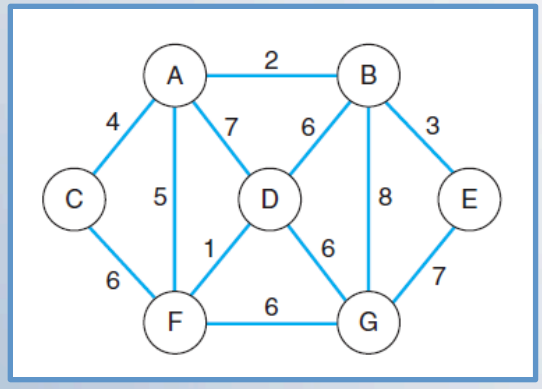
Dijkstra-Prim : MST

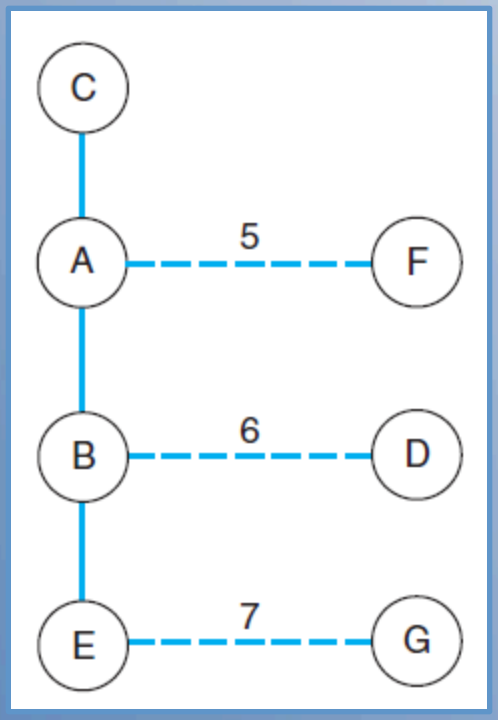


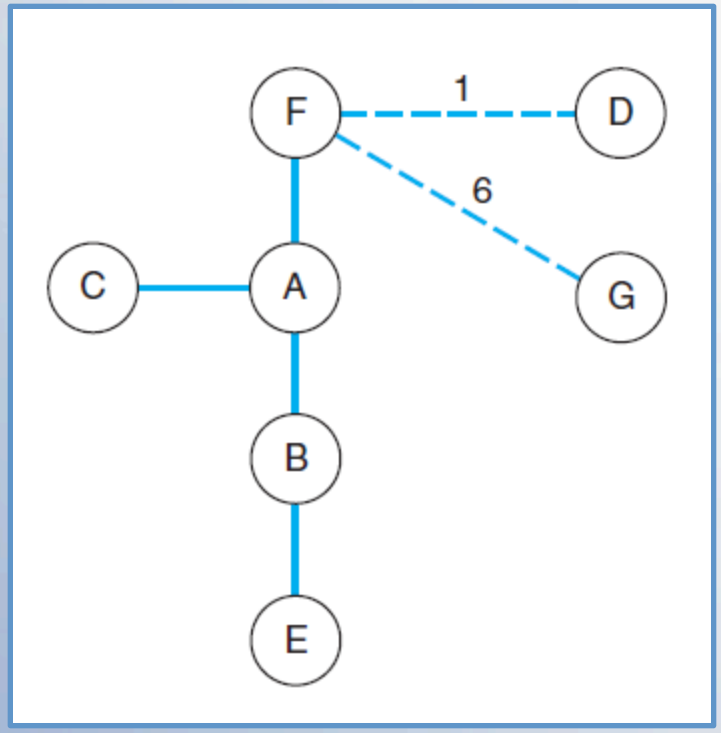




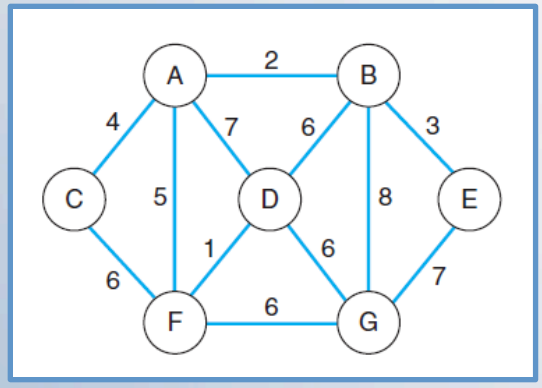
Dijkstra-Prim : MST

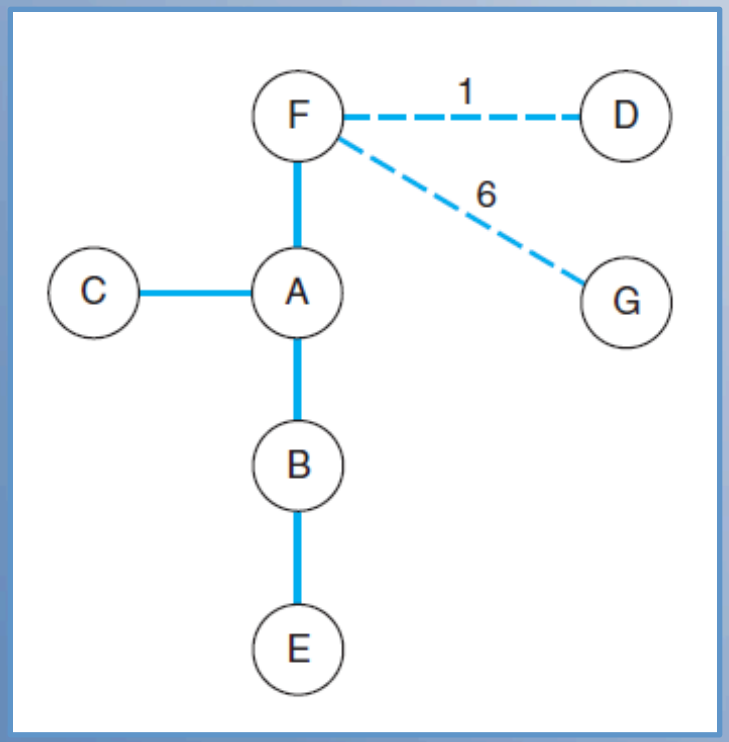


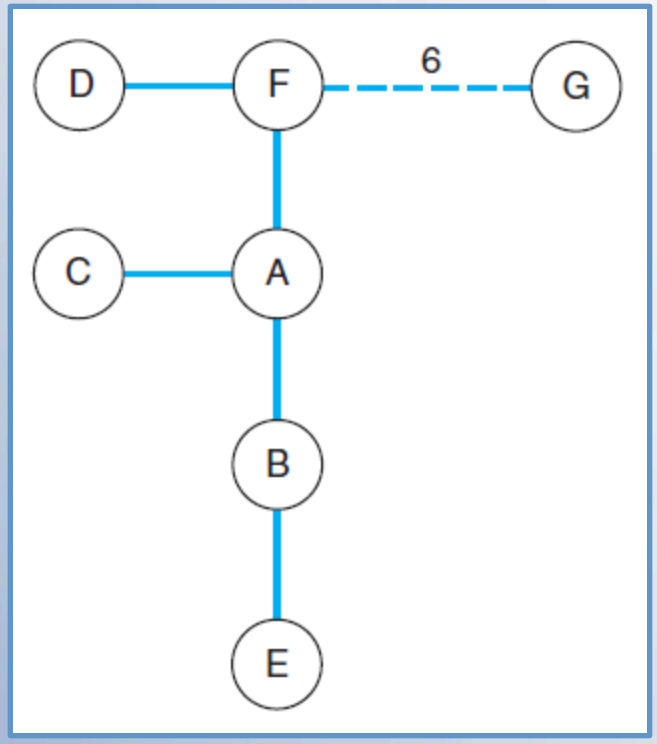




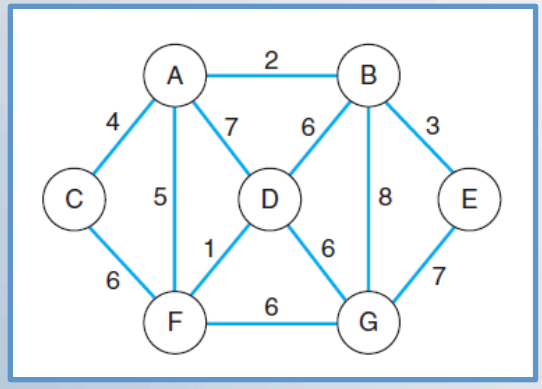
Dijkstra-Prim : MST

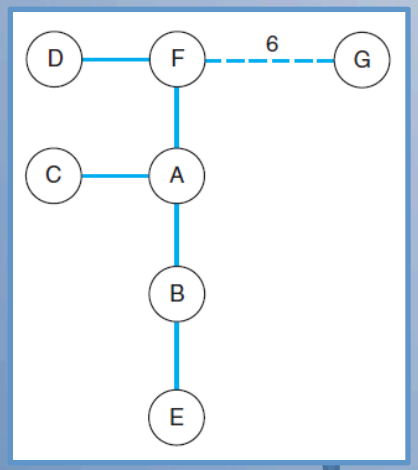


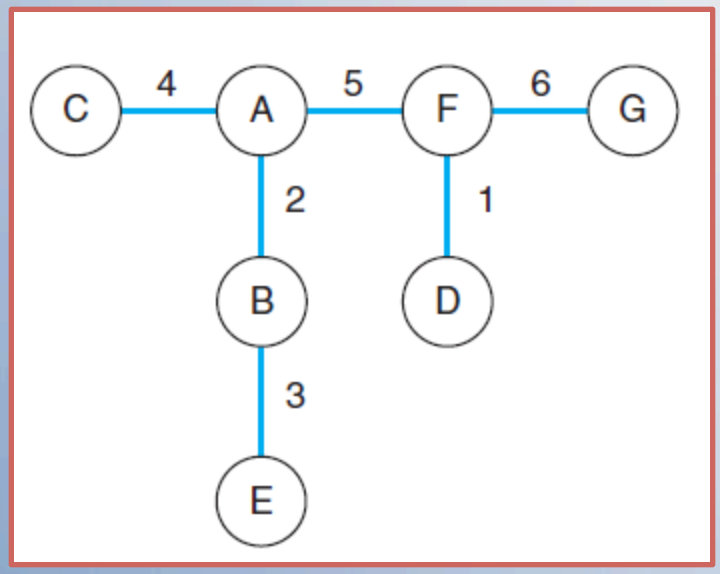




Dijkstra-Prim : Completed MST

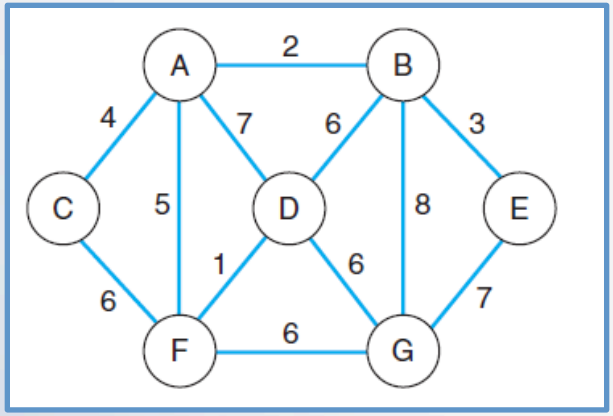


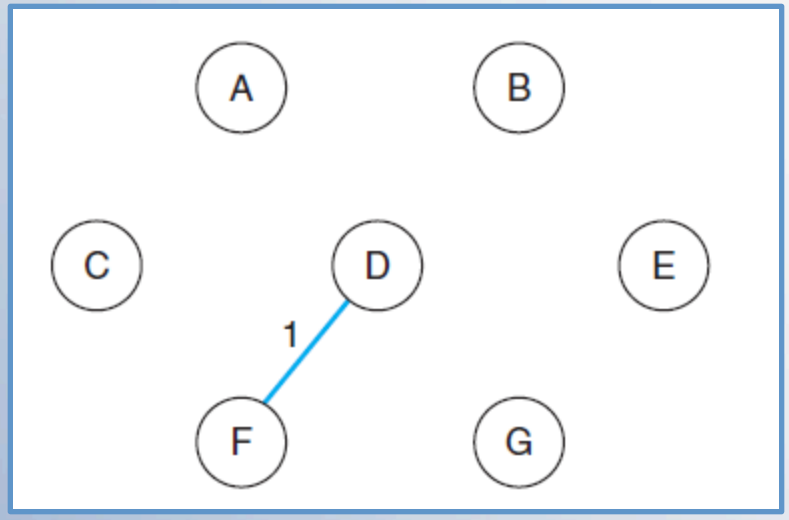




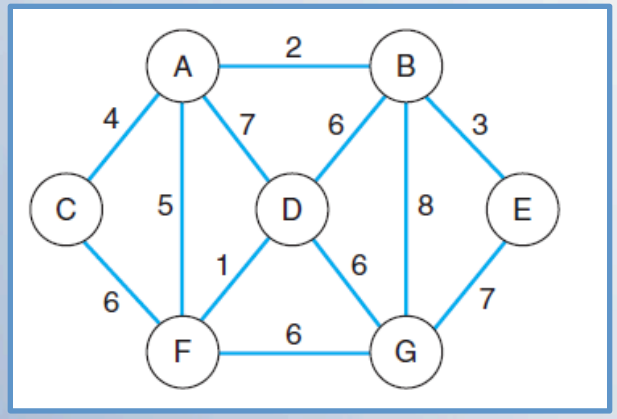
Kruskal's Algorithm: Min. Spanning Tree

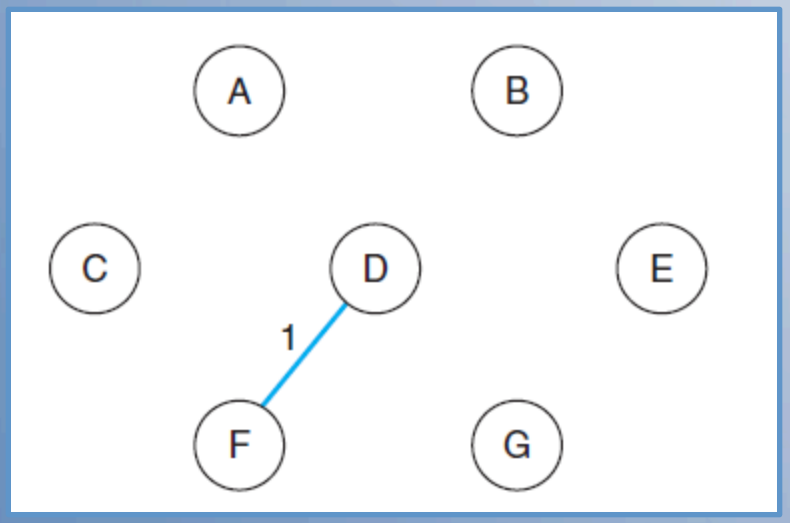
- Keep adding the lightest edge that does not create a cycle
- Break ties arbitrarily

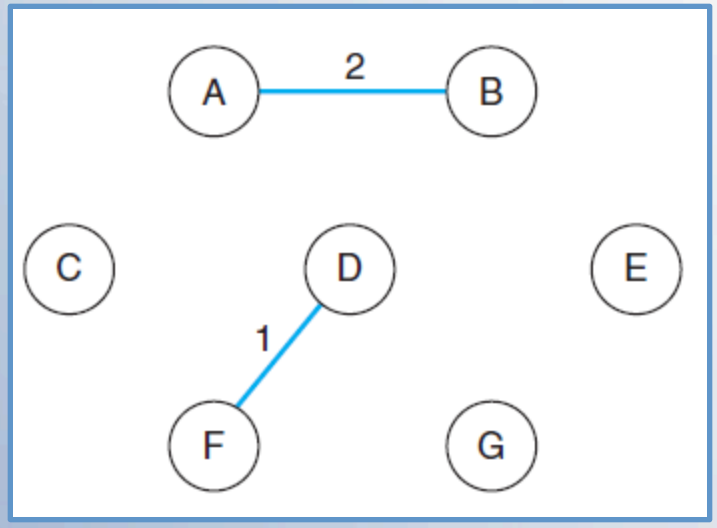




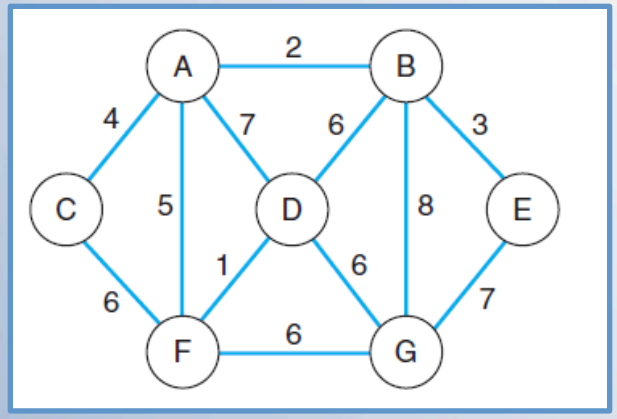
Kruskal's Algorithm

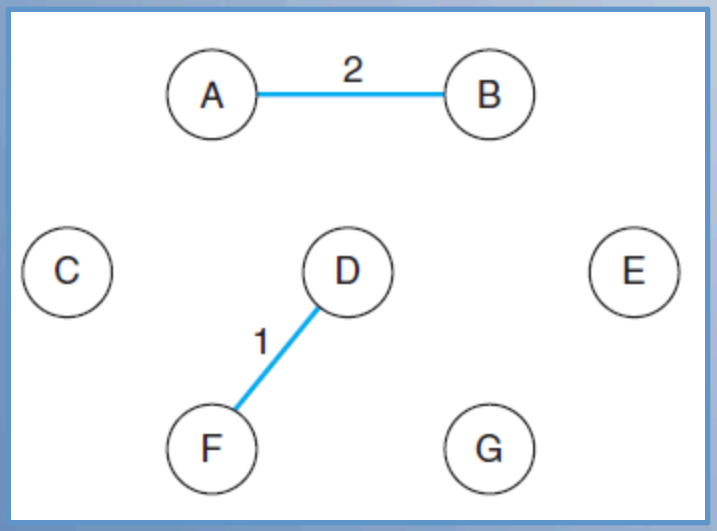


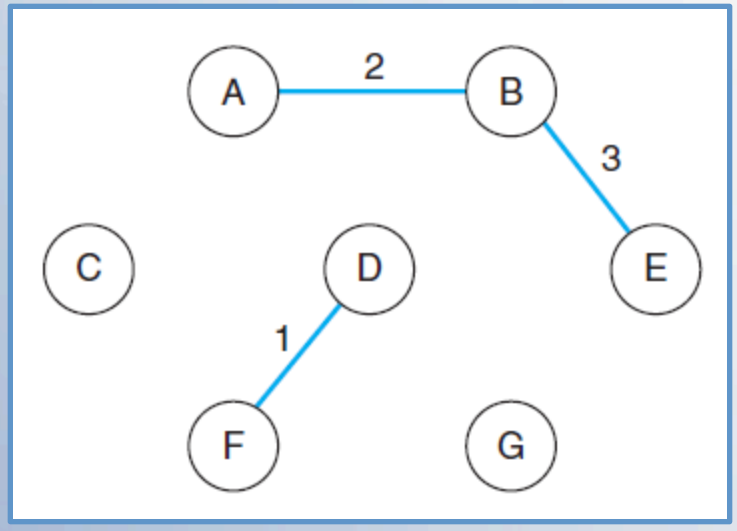




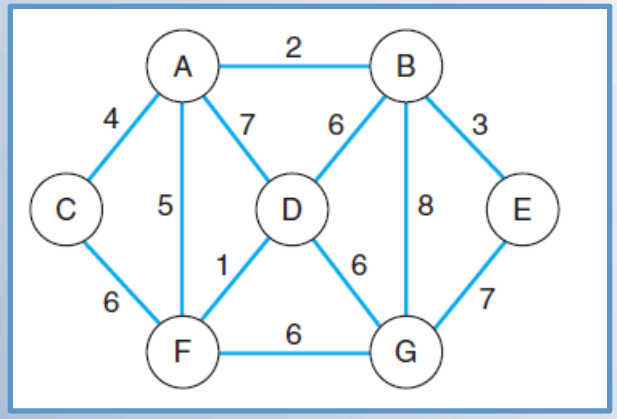
Kruskal's Algorithm

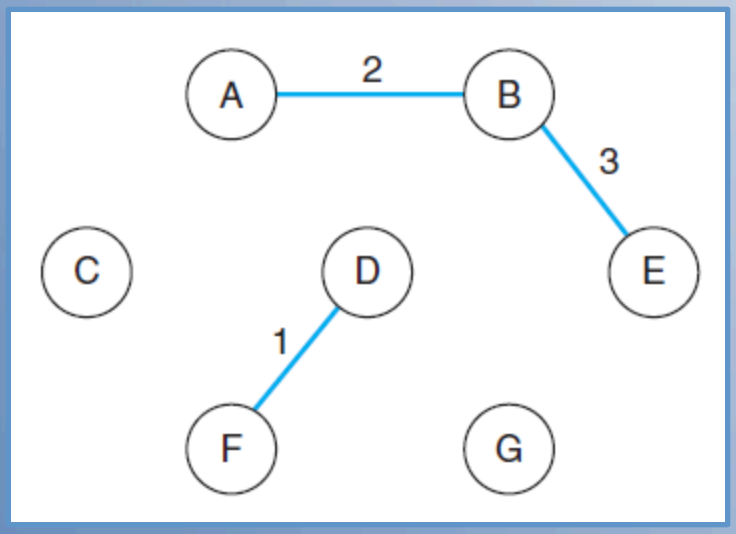


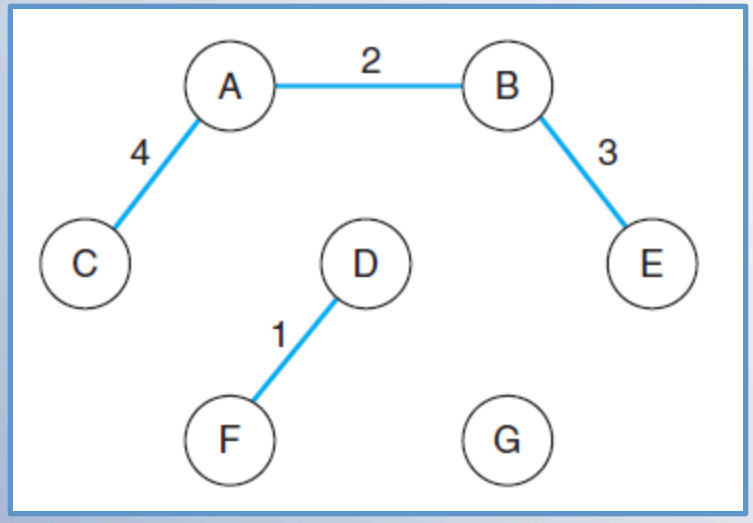




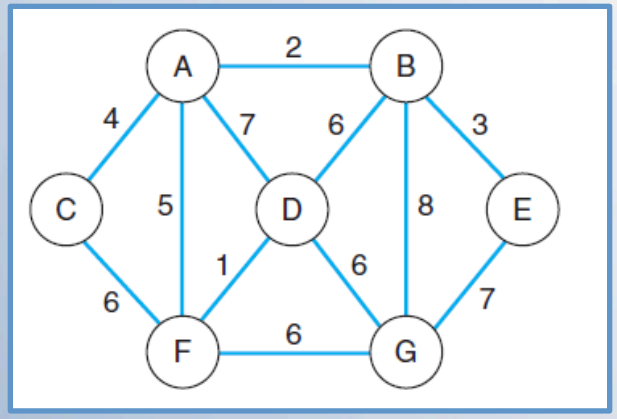
Kruskal's Algorithm

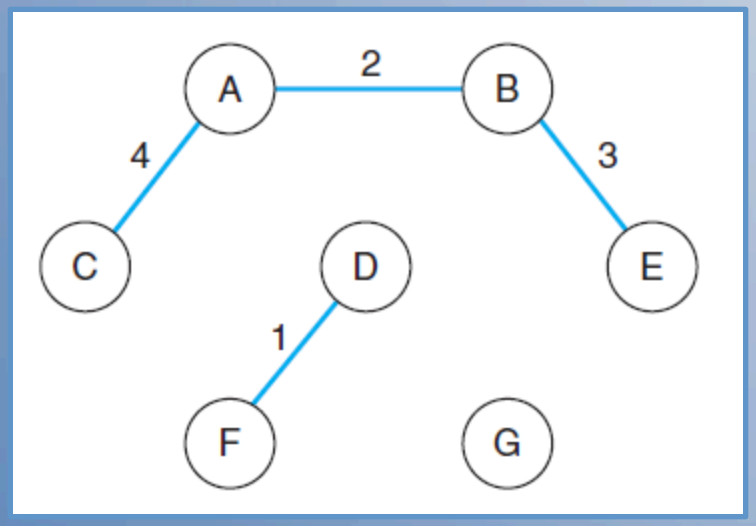


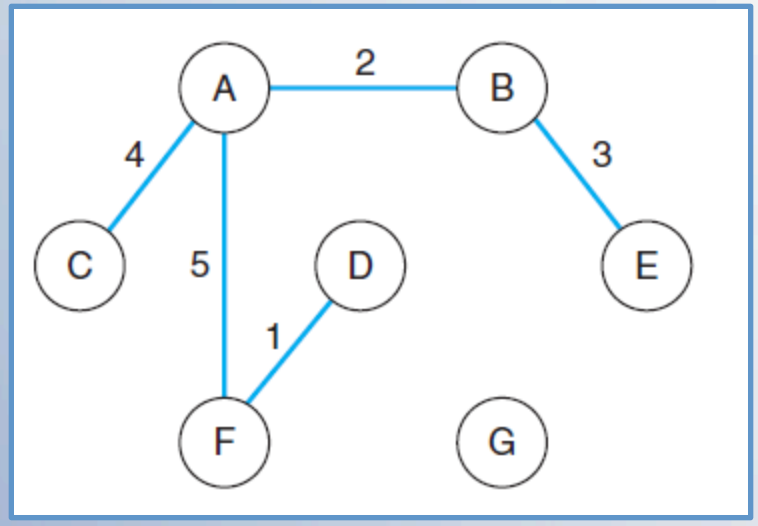




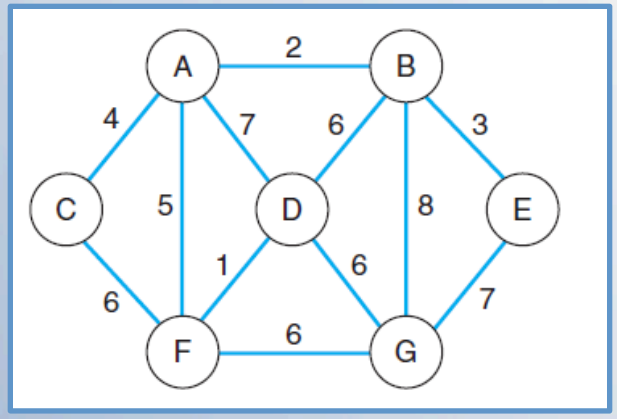
Kruskal's Algorithm

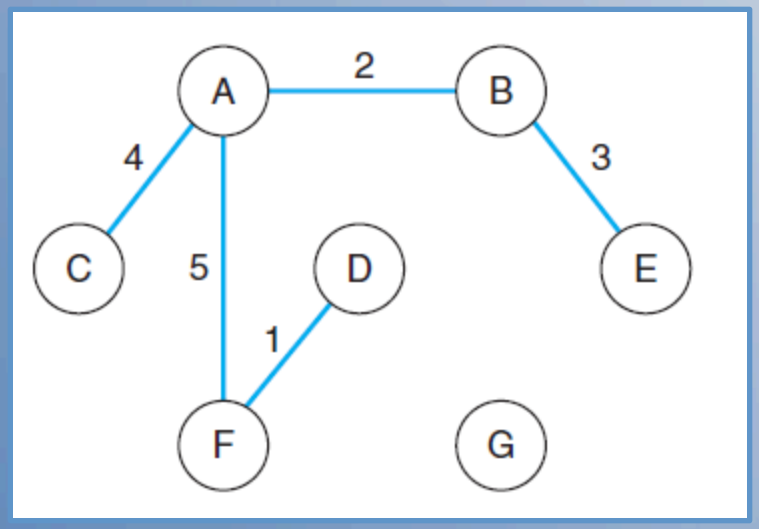


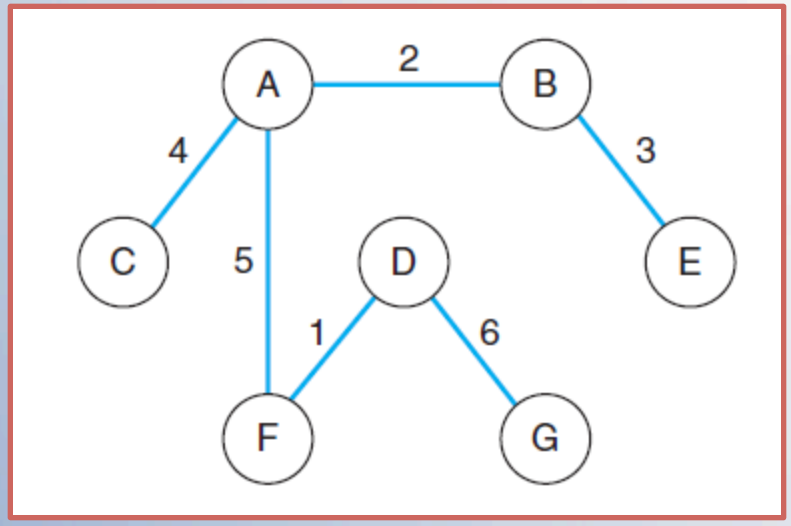




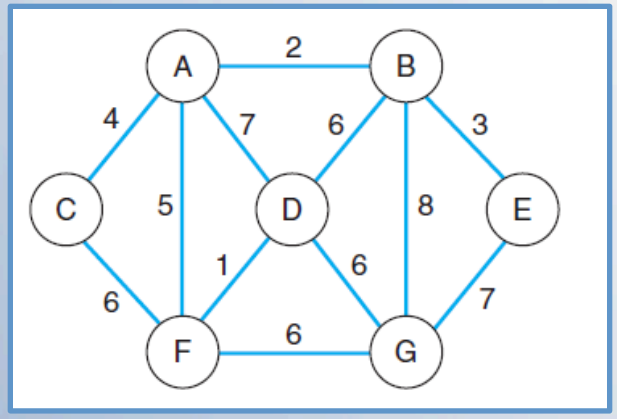
Kruskal's Algorithm

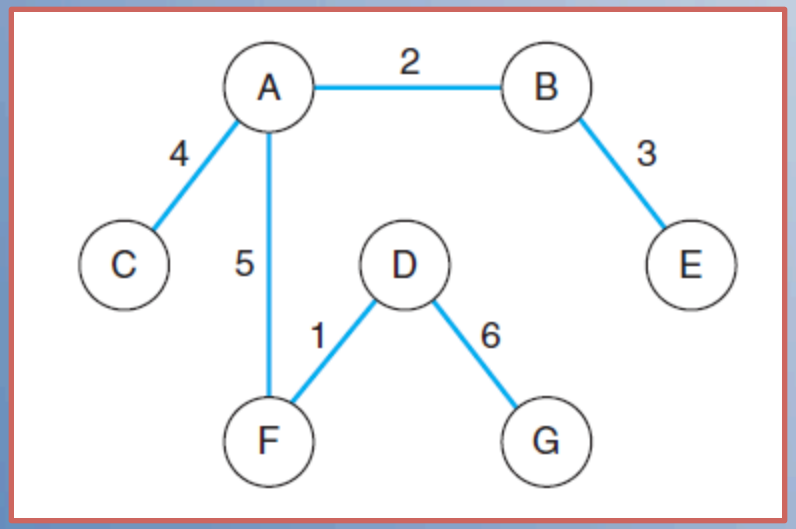


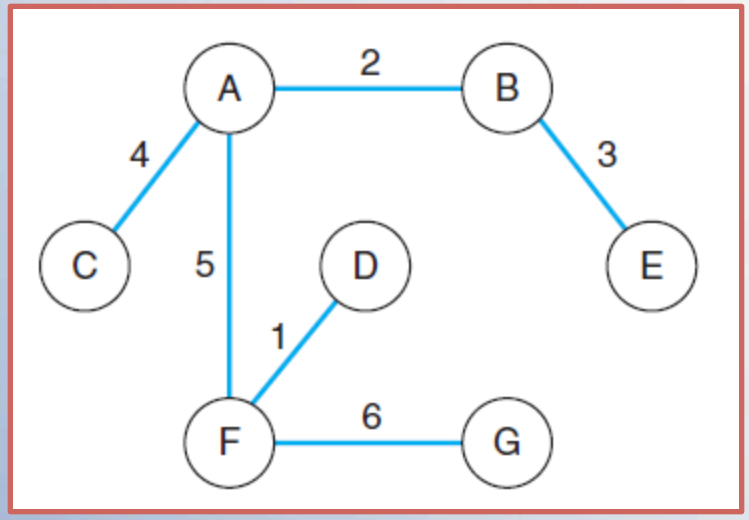




Kruskal's Algorithm







- Two alternative minimum spanning trees.

Kruskal

```
sort the edges in nondecreasing order by weight
initialize partition structure
edgeCount = 1
includedCount = 0
while edgeCount ≤ E and includedCount ≤ N-1 do
   parent1 = FindRoot( edge[edgeCount].start )
   parent2 = FindRoot( edge[edgeCount].end )
   if parent1 ≠ parent2 then
      add edge[edgeCount] to spanning tree
      includedCount = includedCount + 1
      Union( parent1, parent2 )
   end if
   edgeCount = edgeCount + 1
end while
```

Review

- Appendix “B” in CLRS
  - Section on “Graphs”
  - Basic terminology

So..

- Give an example of a MST with nodes u,v such that the MST does not contain the shortest path between u and v
- Hint: a very small graph suffices

Consider This Algorithm

- Assume we start with a complete graph
- Randomly divide set of nodes into two equal parts
- Recursively find MST in each half
- Add lightest edge that joins the two halves
- *Show that this algorithm could fail to find MST*

Topological Sort

Topological sort of a directed acyclic graph G = (V, E): a linear order of vertices such that if there exists an edge (u, v), then u appears before v in the ordering.

- Directed acyclic graphs (**DAG**s)
  - Used to represent precedence of events or processes that have a **partial order**

a before b
b before c
} a before c

b before c
a before c
} What about a and b?

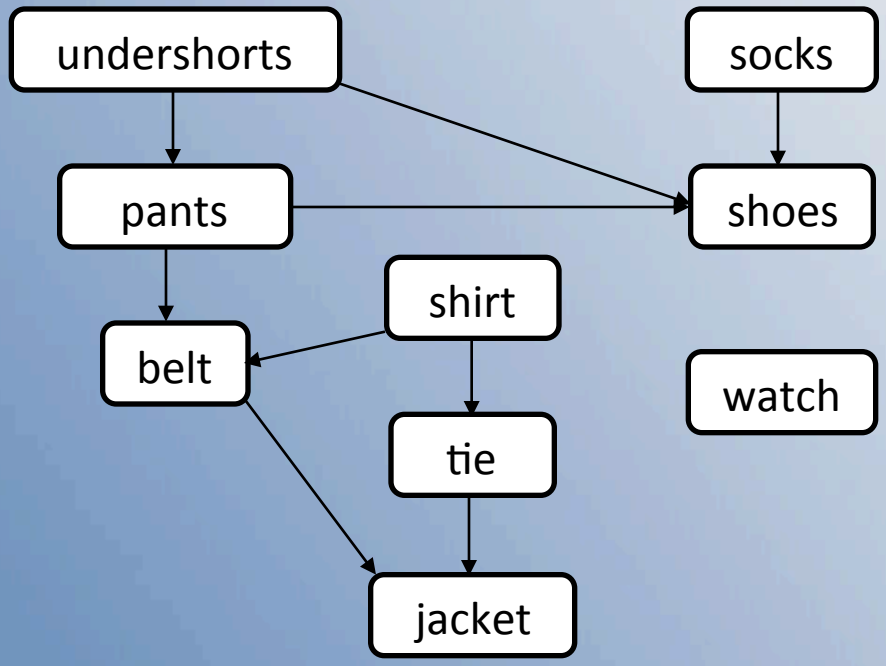
Topological sort helps us establish a **total order**

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)

DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

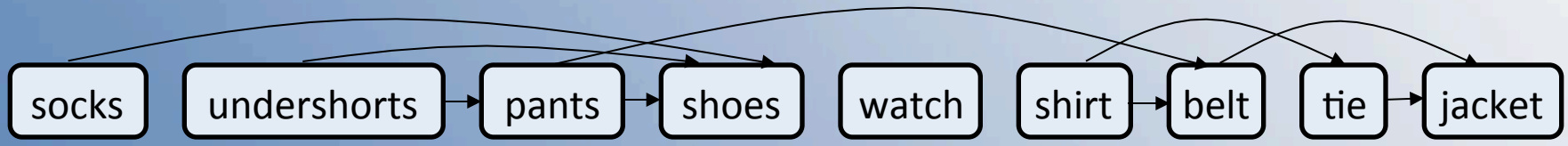
DFS

- To find everything reachable from u:
- Recursively find everything reachable from v where (u,v) is a directed edge
- Initially all nodes ‘white’
- Mark node ‘grey’ when we start to search from it {i.e. while we are inside call to dfs-visit(u)}
- Mark node ‘black’ when we are done (i.e. recursive call to search from u returns)
- Use global var to keep track of when we start, finish exploring from each node

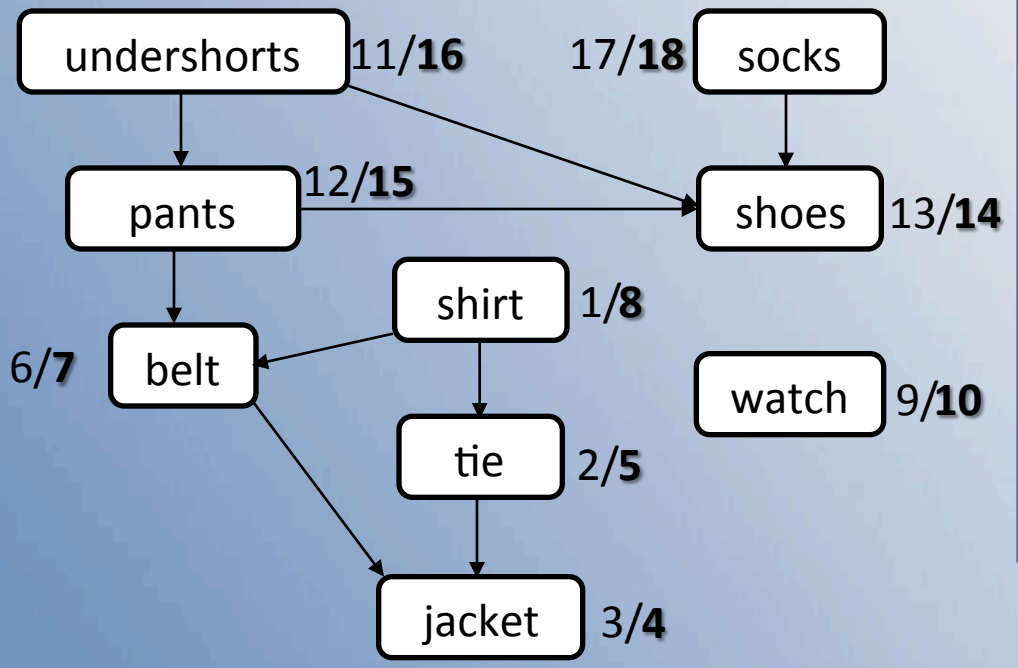
Example: Topological Sort



Topological sort:
a linear ordering of vertices such that all directed edges go from left to right.



Topological Sort



TOPOLOGICAL-SORT(V, E)

1. Call DFS(V, E) to compute **finish times** v.f for each vertex v
2. When each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

socks | undershorts | pants | shoes | watch | shirt | belt | tie | jacket

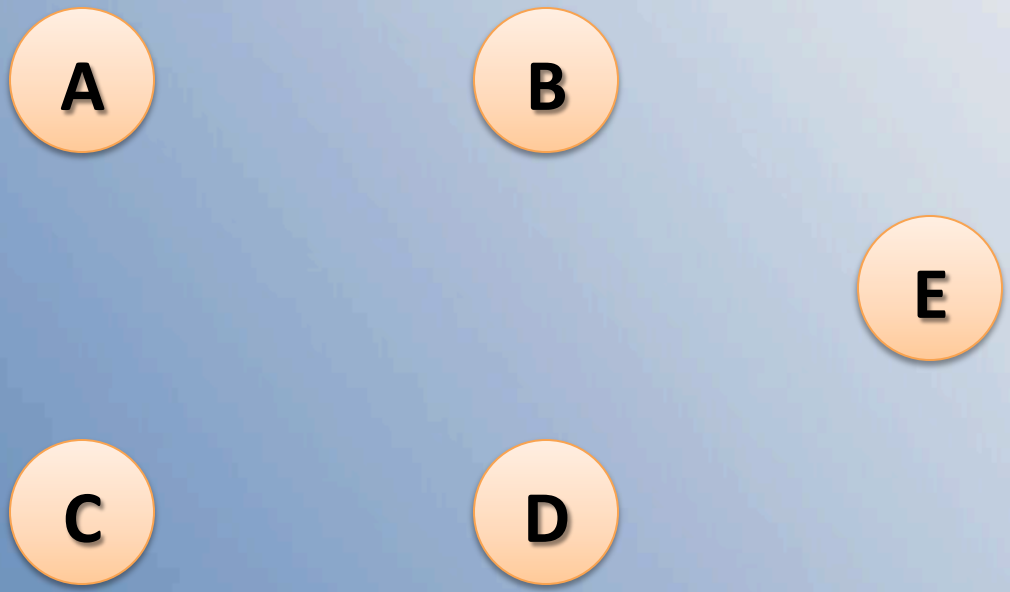
Running time: $\Theta(V + E)$

Topo Sort

- Claim: if u finishes after v, then can put u before v in order
- Finishing time of u is denoted u.f
- Sufficient: if (u,v) is an edge, then v.f < u.f
- Consider what happens when we explore the edge (u,v)
- v is white: will search from v and won't mark u as finished until v is finished
- v is grey: IMPOSSBLE since this would mean there is a loop in G
- v is black: v is already finished

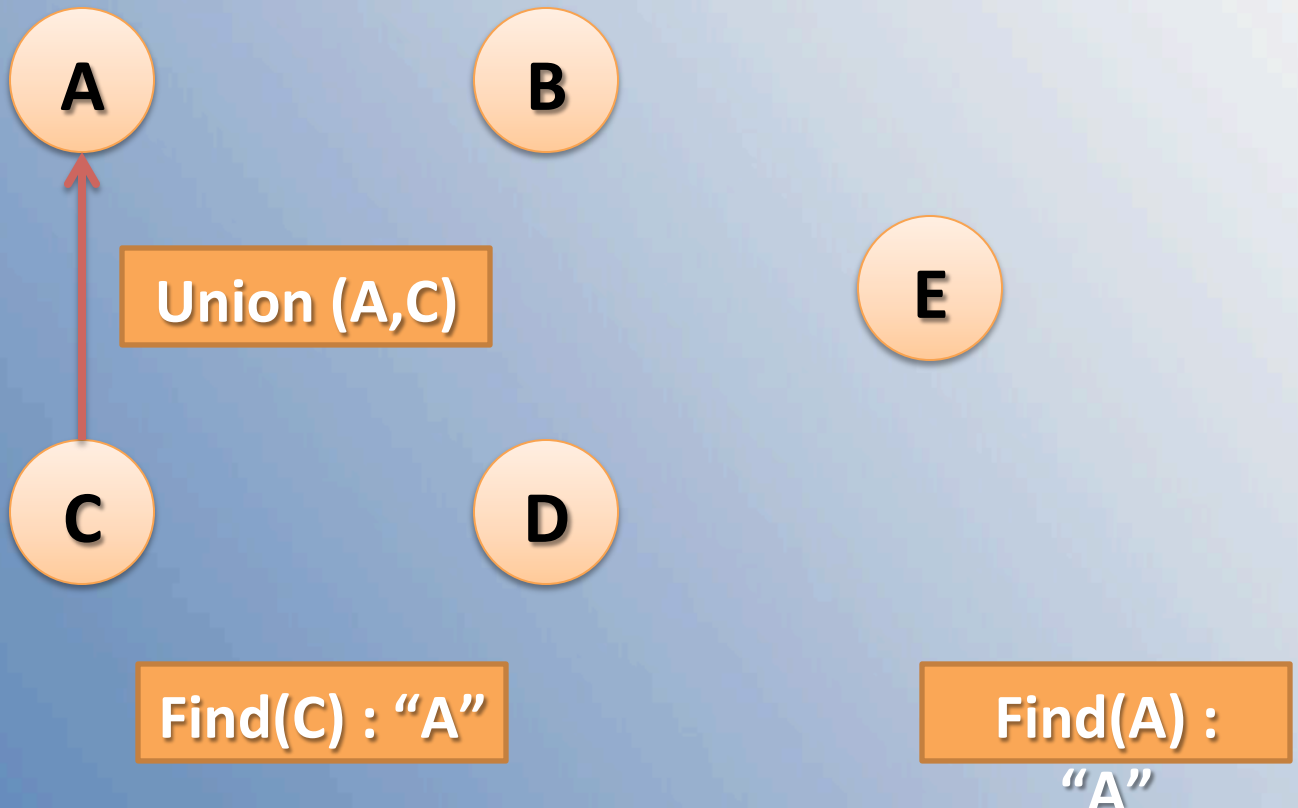
Disjoint Sets

- Data structure which supports merging **sets**
  - Union() : Joins two sets
  - Find() : Returns the "representative" of the set



Disjoint Sets

- Data structure which supports **sets**
  - Union() : Joins two sets
  - Find() : Returns the "representative" of the set



Disjoint Sets and MST

- This is exactly what we need for Kruskal's algorithm
- Edge (u,v) makes a loop if and only if u,v in same tree
- Adding edge (u,v) merges two smaller trees into one larger tree

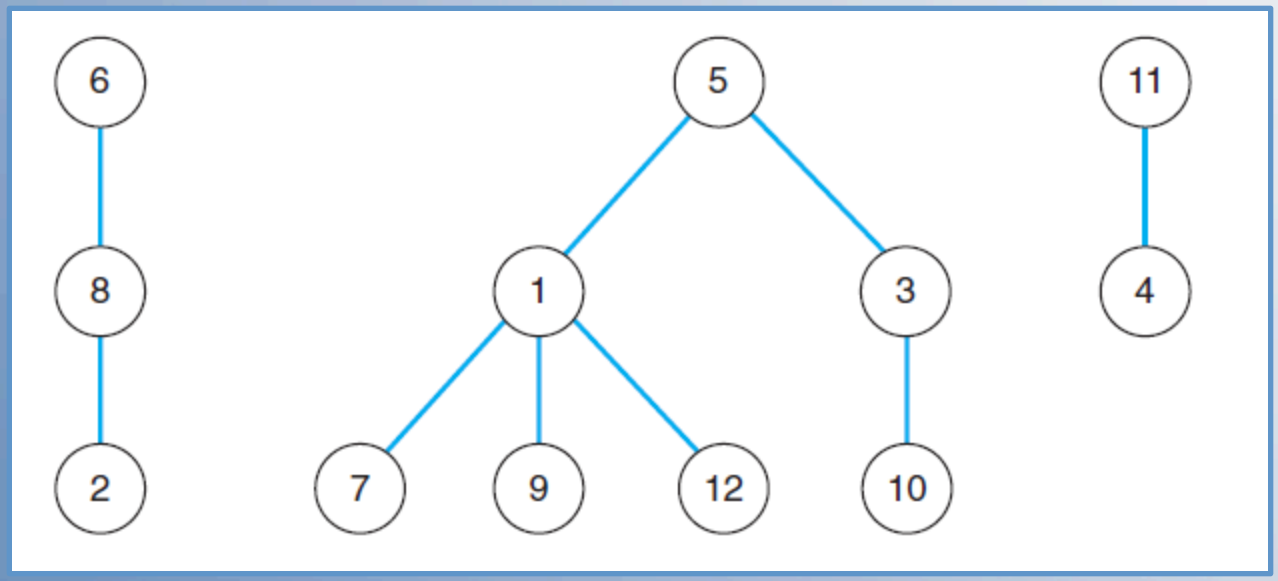
Initializing Disjoint Sets

```
InitializePartition( N )
N  the number of elements in the set

For i = 1 to N do
   Parent[i] = -1
end do
```

Disjoint Sets: Tree Representation

- Root records size of set (negative)
- Otherwise point up toward root

Index	1	2	3	4	5	6	7	8	9	10	11	12
Parent	5	8	5	11	–7	–3	1	6	1	3	–2	1



Union() and Find()

```
Union( i, j )
i, j  the partitions to join together

totalElements = Parent[i] + Parent[j]
if Parent[i] ≥ Parent[j] then
   Parent[i] = j
   Parent[j] = totalElements
else
   Parent[j] = i
   Parent[i] = totalElements
end if
```

```
FindRoot( s )
s  the element whose partition root we want

result = s
while Parent[result] > 0 do
   result = Parent[result]
end while
return result
```

Can improve performance with *path compression* (see text).