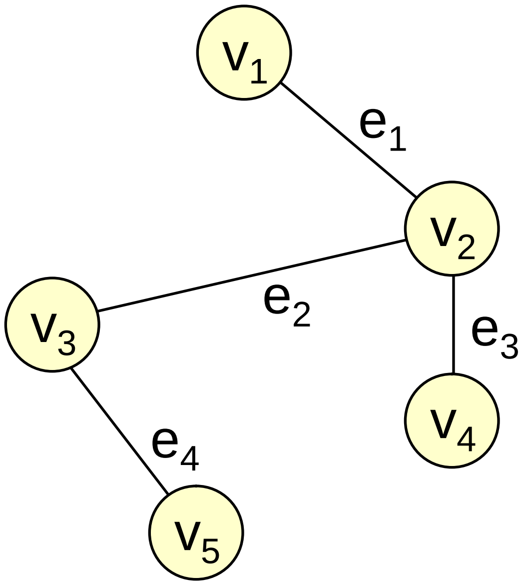
Unit III : Graphs

Basic Concepts, Storage representation, Adjacency matrix, adjacency list, adjacency multi list, inverse adjacency list. **Traversals-depth first and breadth first, Minimum spanning Tree, Greedy algorithms** for computing minimum spanning tree- Prims and Kruskal Algorithms, Dikjtra's Single source shortest path, All pairs shortest paths- Flyod-Warshall Algorithm Topological ordering.

Case Study : Data structure used in Webgraph and Google map

What is a Graph?

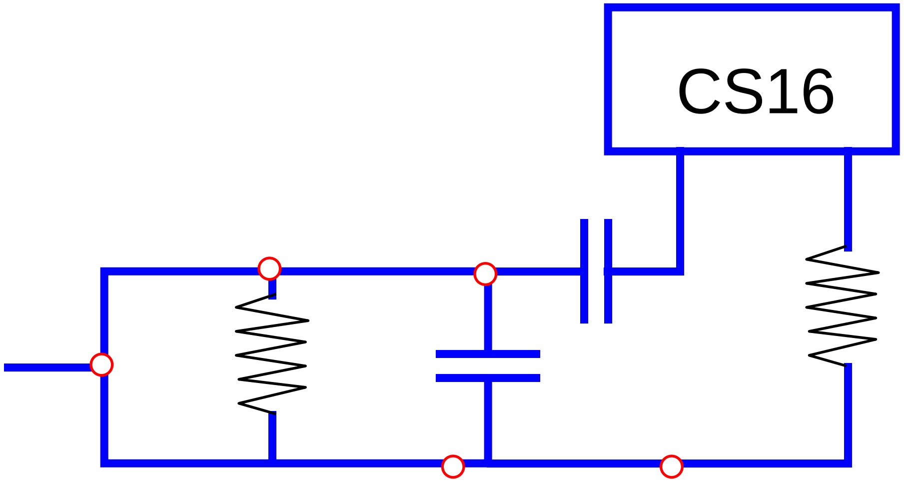
A Graph G consists of a set V of vertices or nodes and a set E of edges that connect the vertices. We write G=(V,E).



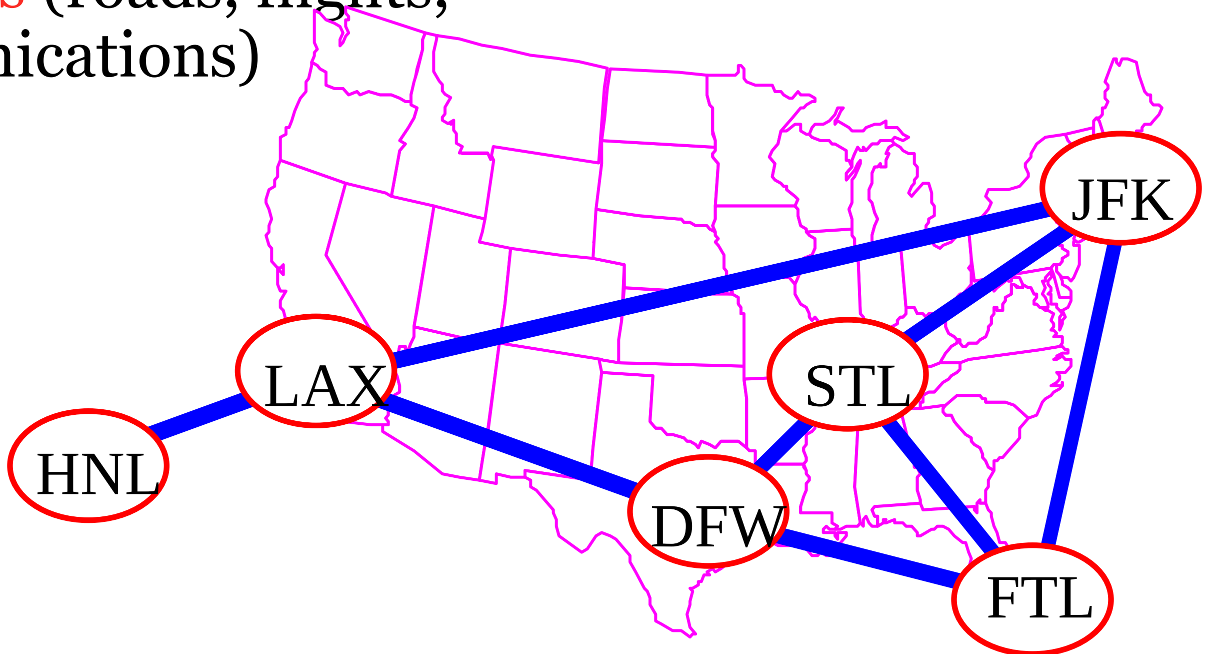
$G=(V,E)$
$V=\{v_1,v_2,v_3,v_4,v_5\}$
$E=\{e_1,e_2,e_3,e_4\}$
$e_1=(v_1,v_2)$
$e_2=(v_2,v_3)$
$e_3=(v_2,v_4)$
$e_4=(v_3,v_5)$

Applications

- electronic circuits



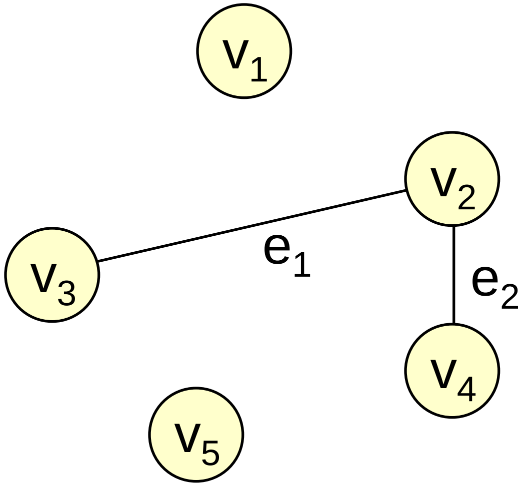
- networks (roads, flights, communications)



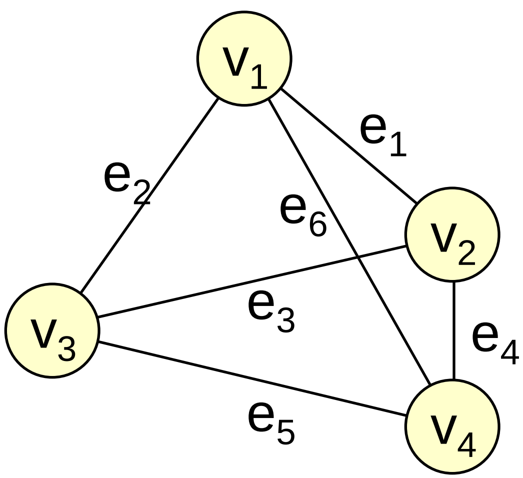
Applications

- Graphs are the basic mathematical formulation we use too tackle such problems.
  - Campus map
  - Travelling salesperson
  - Electronic circuits layout
    - Printed circuit board
    - Integrated circuit
  - Project scheduling
  - Oil flow
  - Transportation networks
    - Highway network
    - Flight network(Flight scheduling)
  - Computer networks
    - Local area network
    - Internet
    - Web
  - Databases
    - Entity relationship diagram

Graphs --- Examples



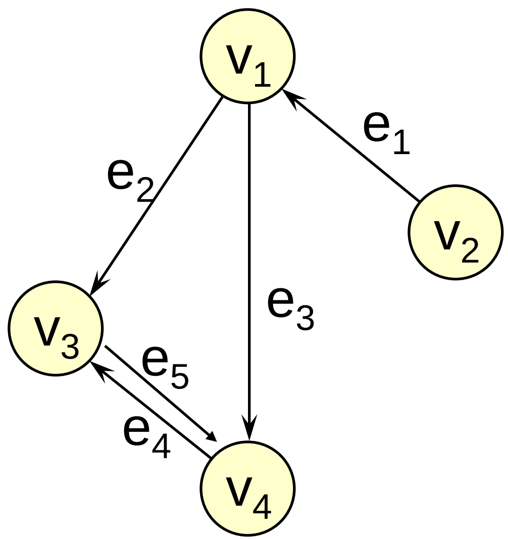
$G=(V,E)$
$V=\{v_1,v_2,v_3,v_4,v_5\}$
$E=\{e_1,e_2\}$
$e_1=(v_2,v_3)$
$e_2=(v_2,v_4)$



$G=(V,E)$
$V=\{v_1,v_2,v_3,v_4\}$
$E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$
$e_1=(v_1,v_2)$ $e_2=(v_1,v_3)$
$e_3=(v_2,v_3)$ $e_4=(v_2,v_4)$
$e_5=(v_3,v_4)$ $e_6=(v_1,v_4)$

Directed Graphs

In some cases we want the edges to have directions associated with them; we call such a graph a directed graph or a digraph.



$G=(V,E)$

$V=\{v_1,v_2,v_3,v_4\}$

$E=\{e_1,e_2,e_3,e_4\}$

$e_1=(v_2,v_1)$

$e_2=(v_1,v_3)$

$e_3=(v_1,v_4)$

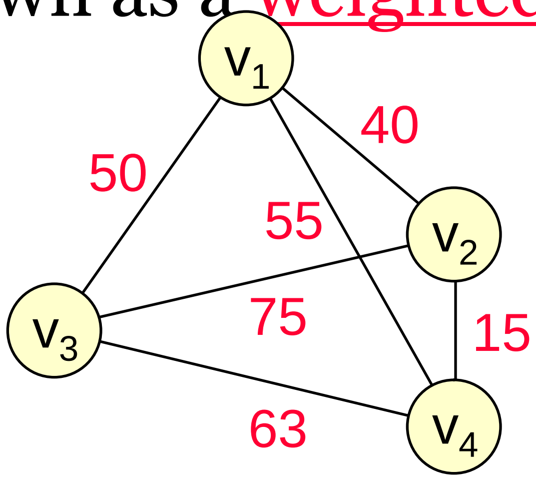
$e_4=(v_4,v_3)$

$e_5=(v_3,v_4)$

ordered pair
(predecessor, successor)

Weighted Graphs

In some cases, we want to associate a weight with each edge in the graph. Such a graph is known as a weighted graph.



G=(V,E)
$V=\{v_1,v_2,v_3,v_4\}$
$E=\{e_1,e_2,e_3,e_4,e_5\}$
……………………
……………………

Graphs with no weights are called unweighted graphs (or simply graphs). Directed graphs can also be weighted (directed weighted graphs).

More Graph Terminology

- A vertex v_j is said to be adjacent to a different vertex v_i if an edge connects v_i to v_j, i.e., if there exists and edge $e \in E$ such that $e=(v_i,v_j)$.
- A path is a sequence of vertices in which each vertex is adjacent to the next one. That is, a path $p = v_1, v_2, \ldots, v_n$
  $(n > 1)$ such that each vertex v_{i+1} is adjacent to v_i, $1 \leq i < n$.
- The length of a path is the number of edges in it.

More Graph Terminology (Cont'd)

- A cycle is a path of length greater than one that begins and ends at the same vertex. In other words, a cycle is a path $p = v_1, v_2, \ldots, v_n$, such that $v_1 = v_n$.
- A graph with no cycles is called an acyclic graph. A directed acyclic graph is called a DAG.
- A simple cycle is a cycle formed from three or more distinct vertices in which no vertex is visited more than once along the simple cycle's path (except starting and ending vertex). That is, if $p = v_1, v_2, \ldots, v_n$ $(n > 3)$ is a path, then p is a simple cycle if $v_1 = v_n$, and $v_i \neq v_j$ for different i and j in the range $1 \leq i,j < n$.

More Graph Terminology (Cont'd)

- Two different vertices are connected if there is a path between them.
- A subset of vertices S is said to be a connected component of G if there is a path from each vertex v_i to any other distinct vertex v_j of S. If S is the largest such subset, then it is called a maximal connected component.
- The degree of a vertex is the number of edges connected to it.

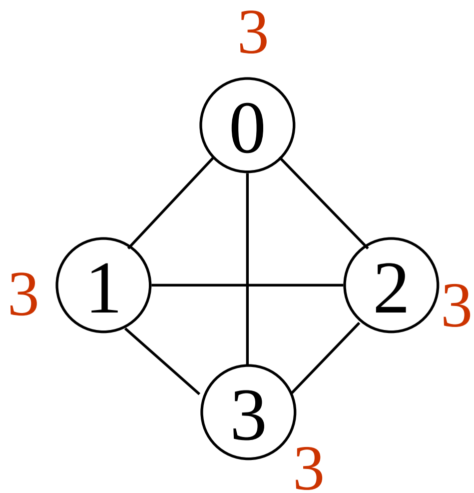
Terminology: Degree of a Vertex

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

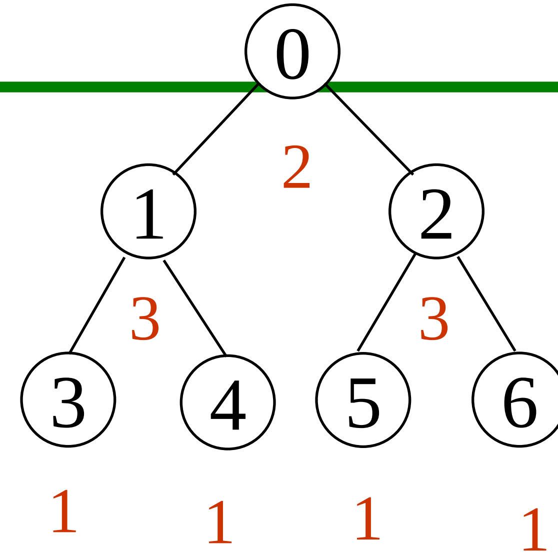
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples



G1

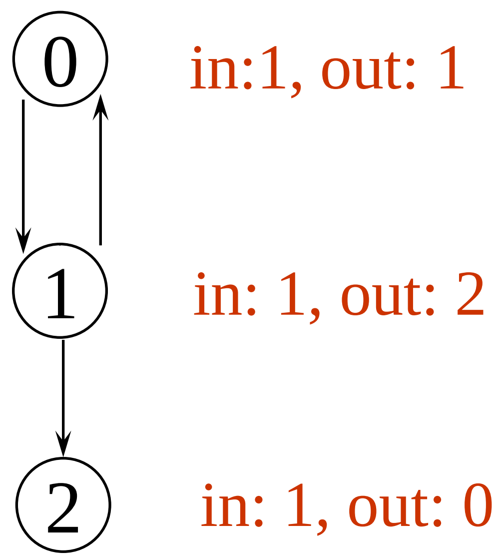


G2

directed graph

in-degree

out-degree



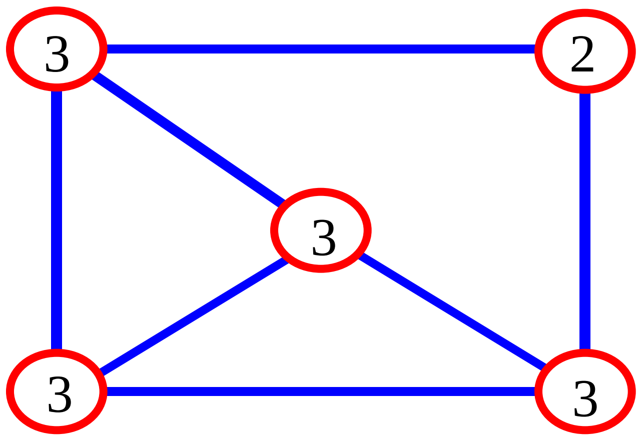
G3

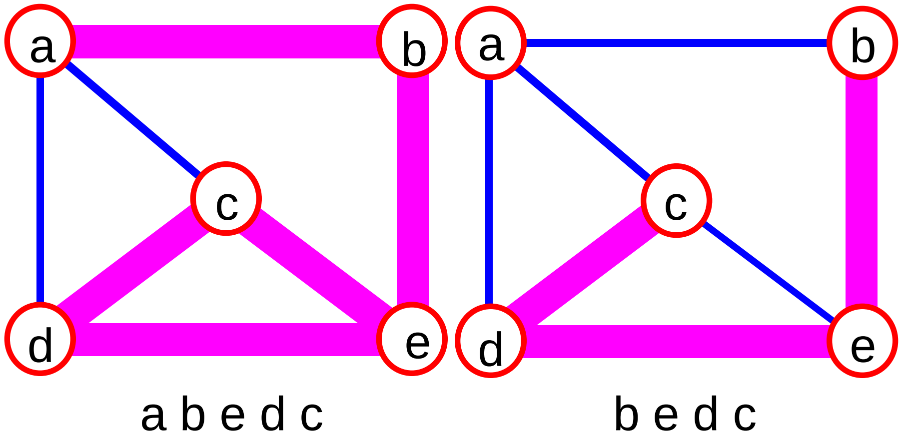
Terminology: Adjacent and Incident

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

Terminology: Path

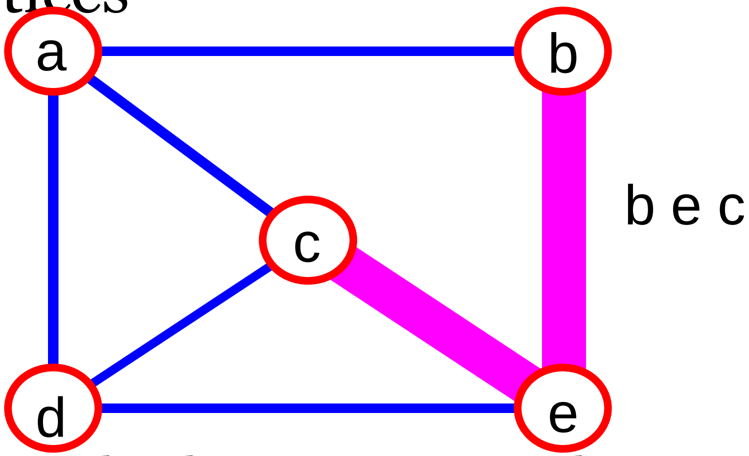
- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



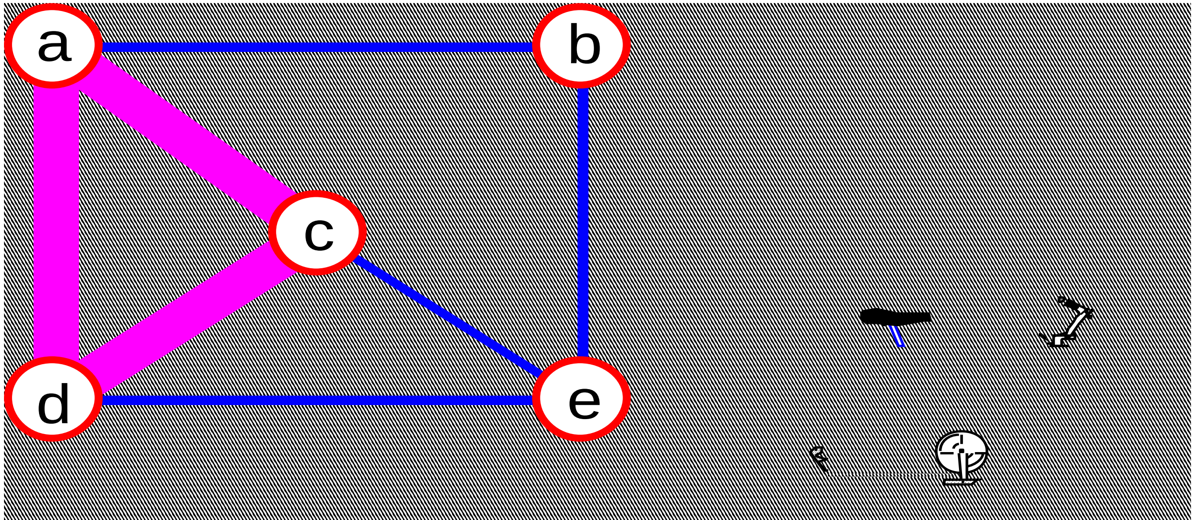


More Terminology

- simple path: no repeated vertices

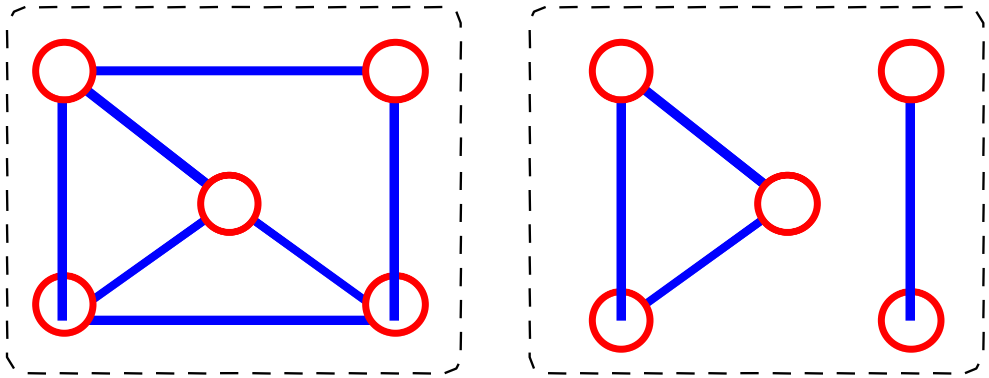


- cycle: simple path, except that the last vertex is the same as the first vertex



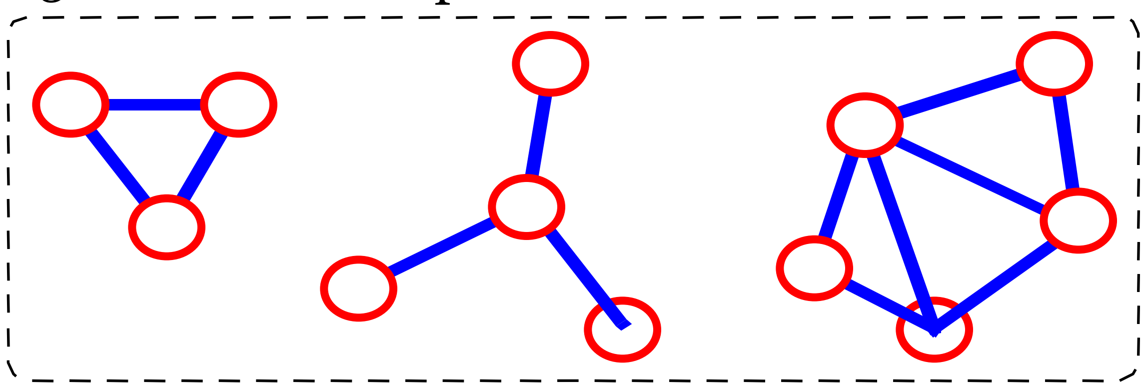
Even More Terminology

•connected graph: any two vertices are connected by some path

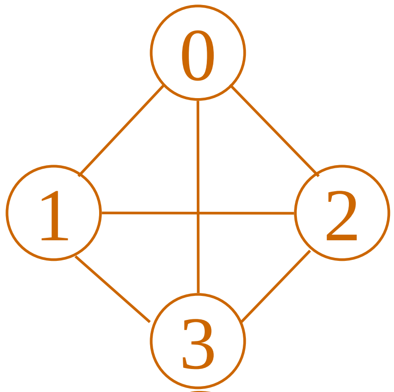


connected   not connected

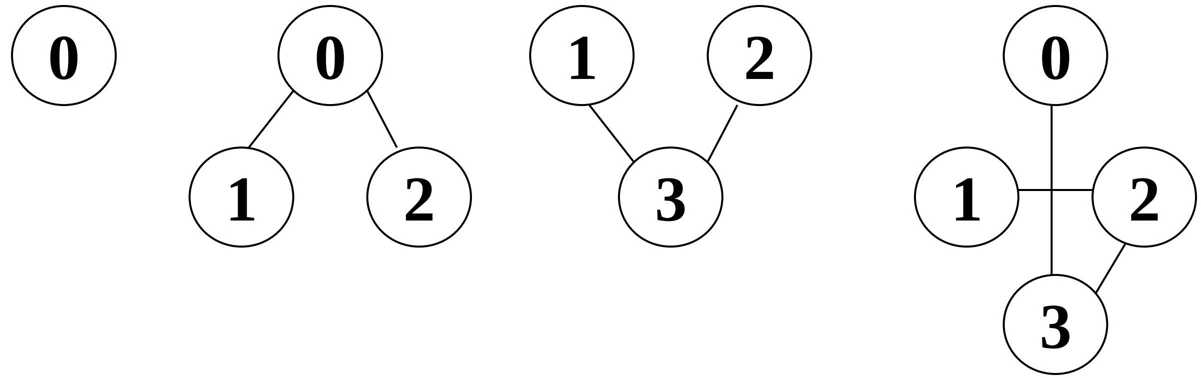
- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Subgraphs Examples

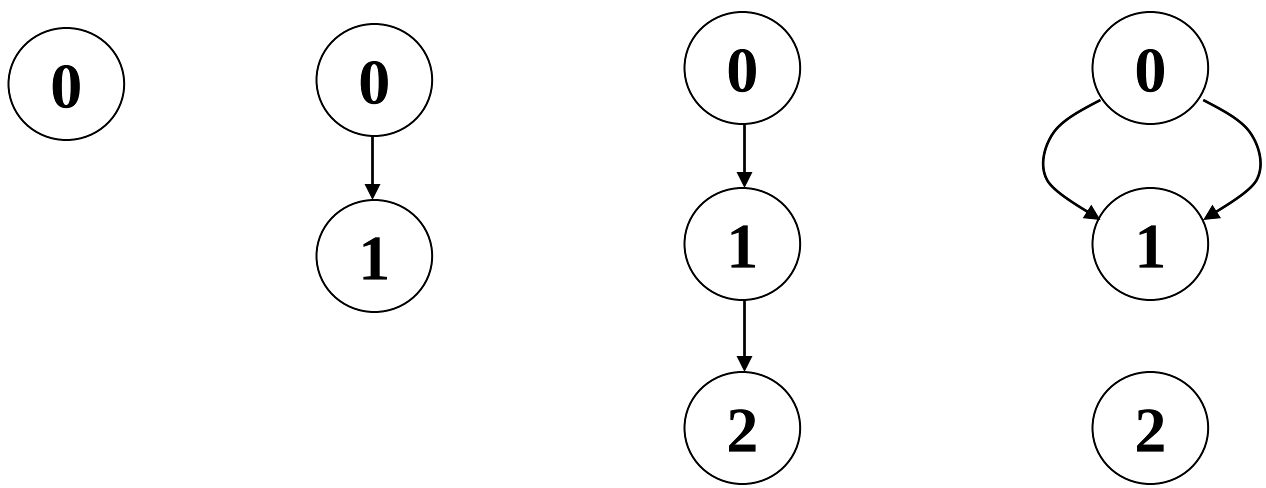


G_1



(i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



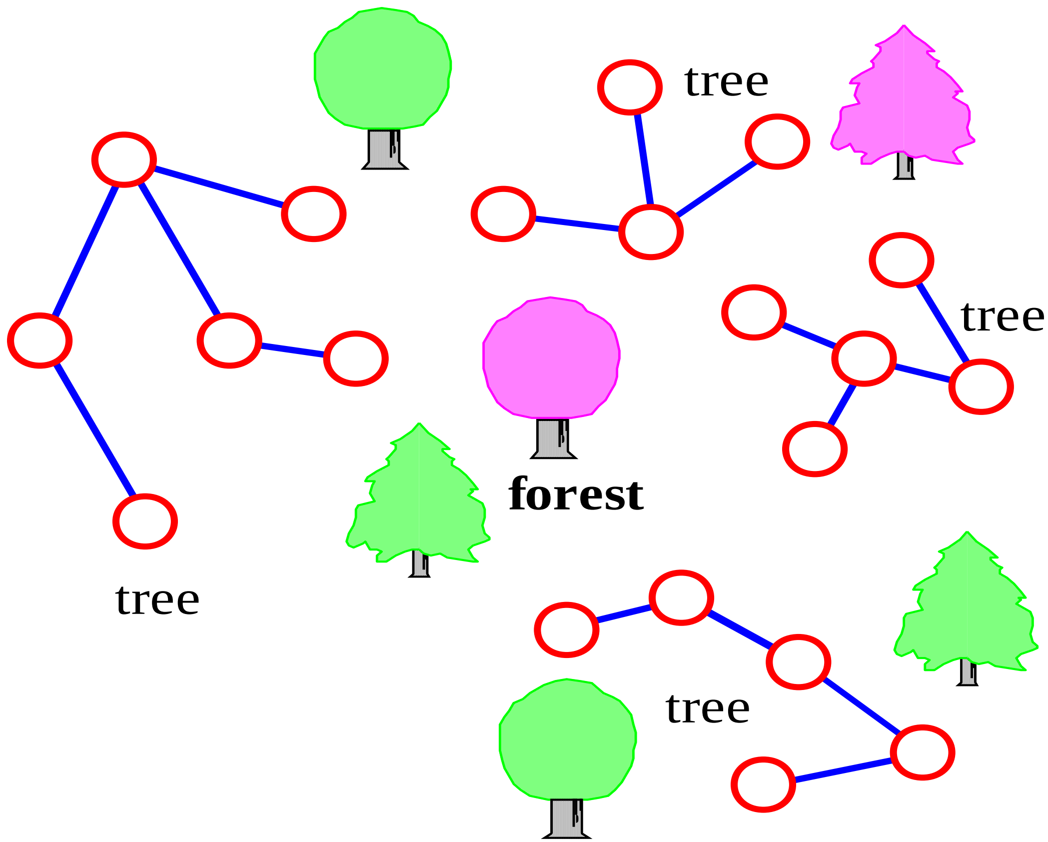
(i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

G_3

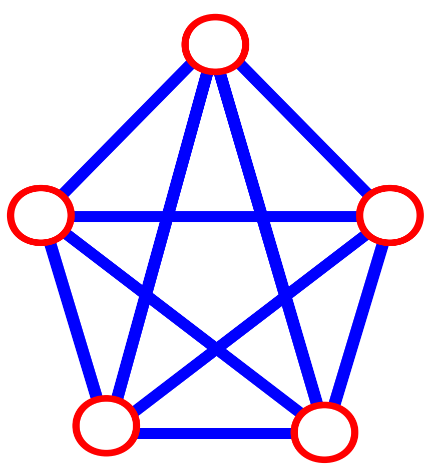
More…

- tree - connected graph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



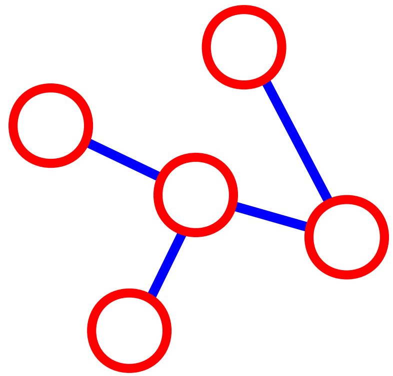
n = 5
m = (5 ∗ 4)/2 = 10

More Connectivity

n = #vertices

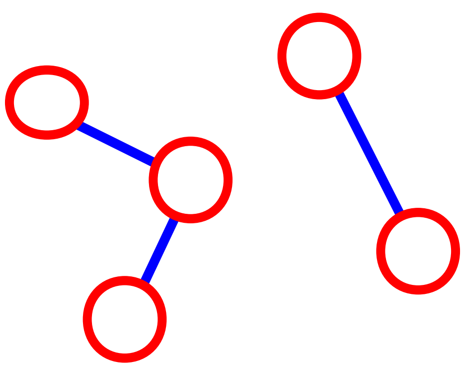
m = #edges

- For a tree **m** = **n** - 1



n = 5
m = 4

If **m** < **n** - 1, G is not connected



n = 5
m = 3

ADT for Graph

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, v, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, v)::= return a graph with v inserted. v has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, v)::= return a graph in which v and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,v)::= return a list of all vertices that are adjacent to v

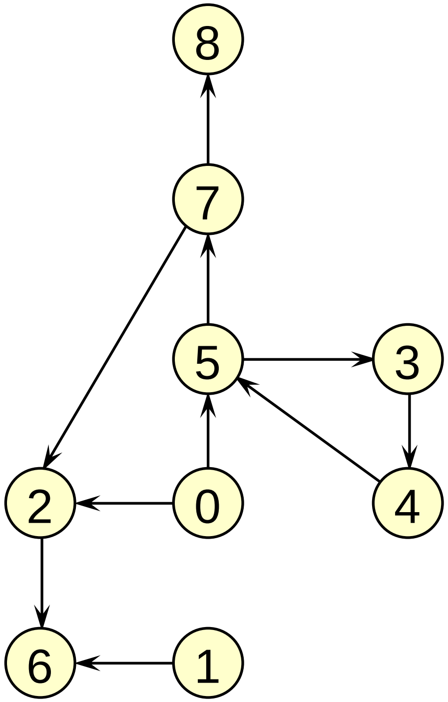
Graph Representations

- Adjacency Matrix
- Adjacency Lists
- Adjacency multi list
- Inverse adjacency list

Adjacency Matrix

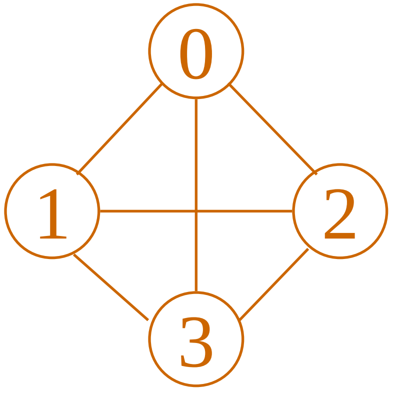
- The adjacency matrix for a graph G=(V,E) with n (or |V|) vertices numbered 0, 1, …, n-1 is an n x n array M such that M[i][j] is 1 if and only if there is an edge from vertex i to vertex j.
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph
need not be symmetric

Adjacency Matrix --- Example 1



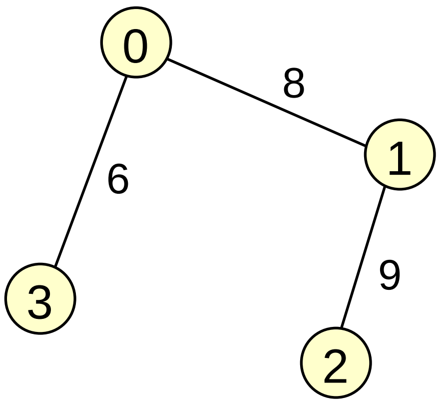
	0	1	2	3	4	5	6	7	8
0	0	0	**1**	0	0	**1**	0	0	0
1	0	0	0	0	0	0	**1**	0	0
2	0	0	0	0	0	0	**1**	0	0
3	0	0	0	0	**1**	0	0	0	0
4	0	0	0	0	0	**1**	0	0	0
5	0	0	0	**1**	0	0	0	**1**	0
6	0	0	0	0	0	0	0	0	0
7	0	0	**1**	0	0	0	0	0	**1**
8	0	0	0	0	0	0	0	0	0

Adjacency Matrix --- Example 2



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency Matrix --- Example 3



	0	1	2	3
0	∞	**8**	∞	**6**
1	**8**	∞	**9**	∞
2	∞	**9**	∞	∞
3	**6**	∞	∞	∞

The matrix is symmetric for undirected graphs.

Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree

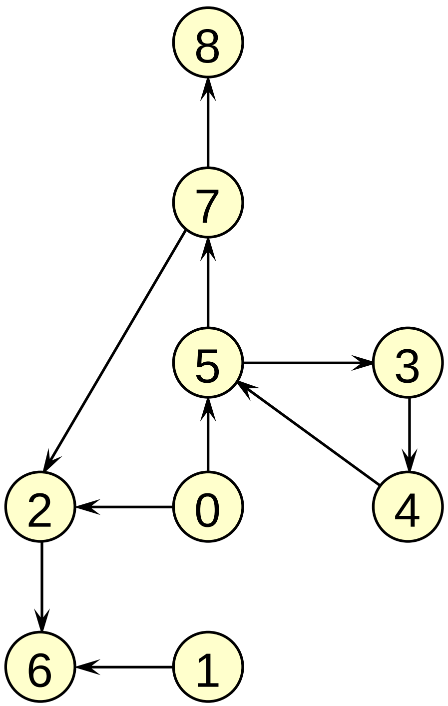
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

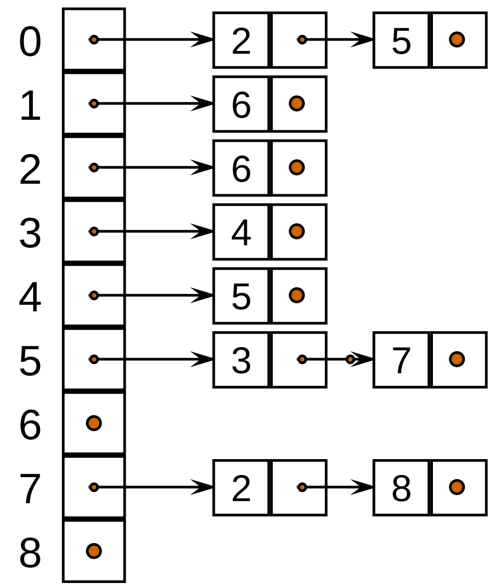
Adjacency Lists

- The adjacency list for a graph G=(V,E) with n vertices numbered 0, 1, …, n-1 consists of n linked lists. The i^{th} linked list has a node for vertex j if and only if the graph contains and edge from vertex i to vertex j.

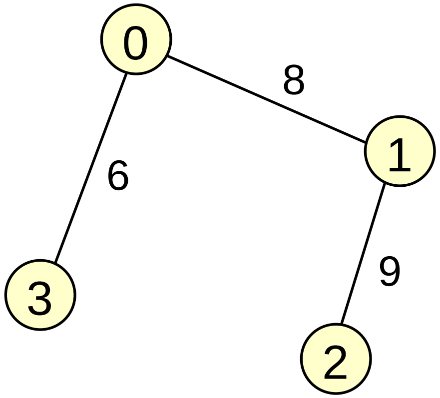
- Each row in adjacency matrix is represented as an adjacency list.

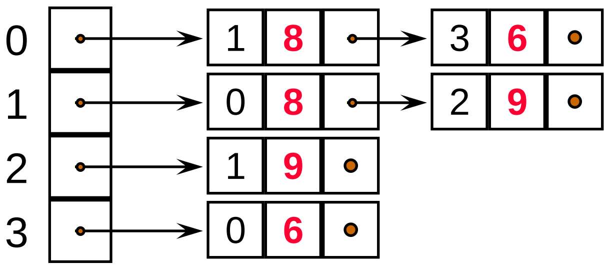
Adjacency List --- Example 1





Adjacency List --- Example 2

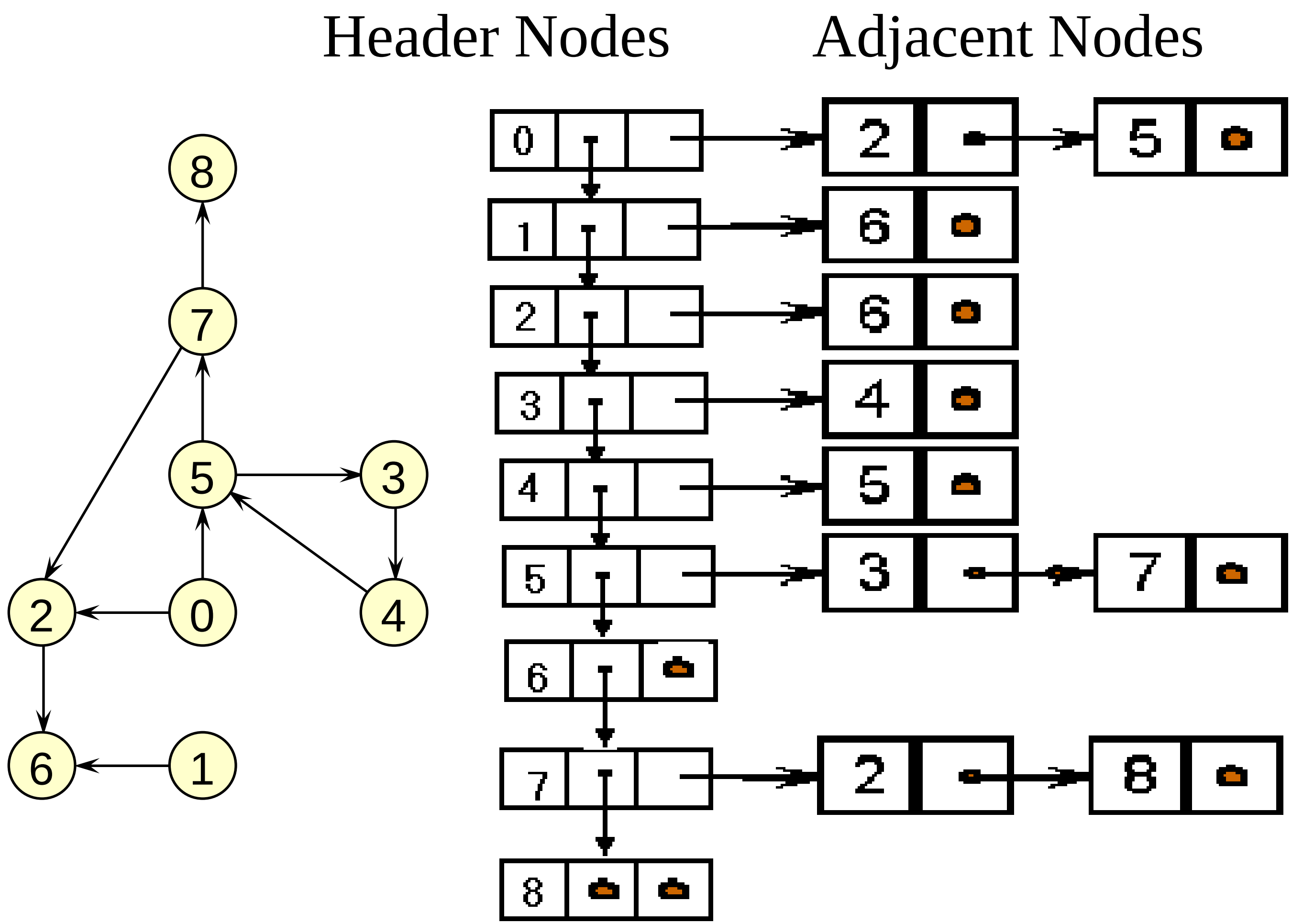




Adjacency Lists (data structure)

```
#define MAX_VERTICES 50
struct Adj_node {
    int vertex;
    struct Adj_node *rlink;
};
struct Adj_node *G[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Adjacency List – Good representation



Adjacency Lists (data structure)

```
struct Adj_node {
    char vertex;
    struct Adj_node *rlink;
};

struct Head_node {
    char vertex;
    struct Head_node *dlink;
    struct Adj_node *rlink;
};
struct Head_node *head;
```

Which is Better?

- Operation 1: Is there an edge from vertex i to vertex j?
- Operation 2: Find all vertices adjacent to vertex i.
- **Time** (d is degree of the vertex):

	Matrix	List
Operation 1	M[i][j] O(1)	Search List O(d)
Operation 2	Traverse row O(n)	Traverse List O(d)

- Determine which operation is most frequent.

Which is Better?

Space:

Matrix: n^2 x size of integer; i.e., $O(n^2)$.

List: n x size of pointer
+ O(|E|) x (size of integer + size of pointer)

O(n+|E|) = O(|V| + |E|)

How big is this?

- Consider space given graph properties.

Which is Better?

- An Adjacency matrix gives us the ability to quickly access edge information, but if the graph is far from being a complete graph, there will be many more empty elements in the array than there are full elements.
- An Adjacency list uses space that is proportional to the number of edges in the graph, but the time to access edge information may be greater.

Which is Better & which to Use?

- There is no clear benefit to either of these methods.
- The choice between these two will be closely linked to knowledge of the graphs that will be input to the algorithm.
- In situations where the graph has many nodes, but they are each connected to only a few other nodes, an adjacency list would be best because it uses less space, and there will not be long edge lists to traverse.
- In situations were the graph has few nodes, an adjacency matrix would be best because it would not be very large, so even a sparse graph would not waste many entries.
- In situations where the graph has many edges and begins to approach a complete graph, an adjacency matrix would be best because there would be few entries.

Adjacency multilists

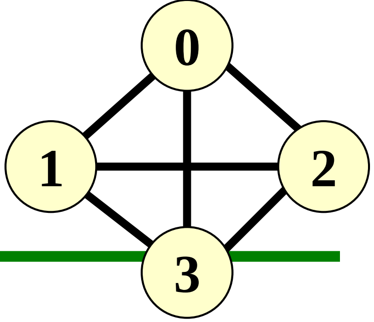
- In the adjacency-list representation of an undirected graph, each edge (u, v) is represented by two entries.
- Multilists: To be able to determine the second entry for a particular edge and mark that edge as having been examined, we use a structure called multilists.
  - Each edge is represented by one node.
  - Each edge node will be in two lists.

Adjacency multilists

- Adjacency Multilists
  - Lists in which nodes may be shared among several lists. (an edge is shared by two different paths)
  - There is exactly one node for each edge.
  - This node is on the adjacency list for each of the two vertices it is incident to

marked	vertex1	vertex2	path1	path2

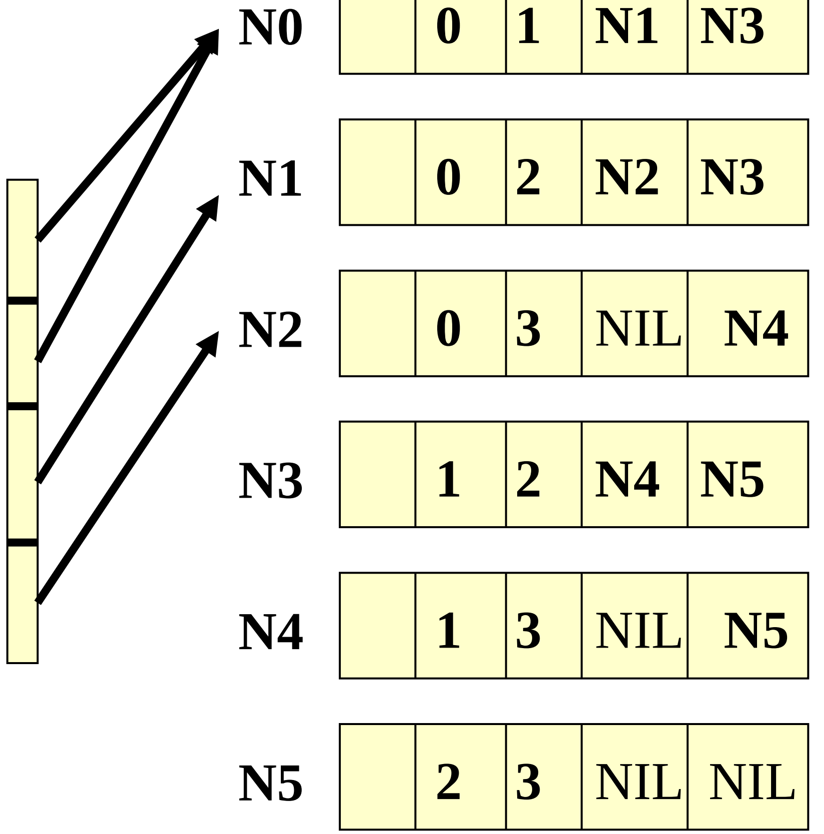
Adjacency multilists



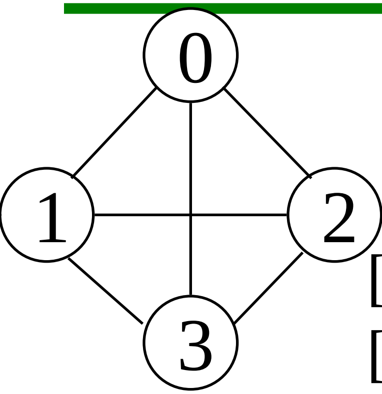
G1

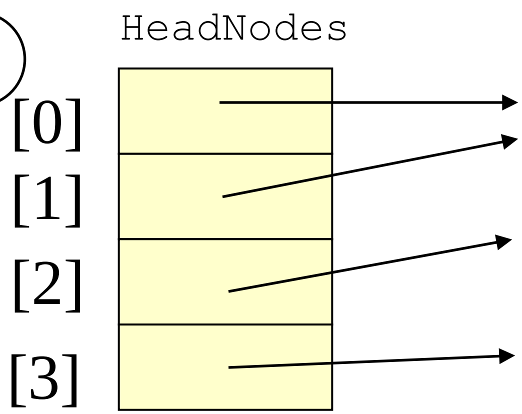
m	vertex1	vertex2	list1	list2

```
typedef struct edge *edge_ptr;
Typedef struct edge {
        int marked;
        int vertex1;
        int vertex2;
        edge_ptr path1;
        edge_ptr path2;
} edge;
edge_ptr graph[MAX_VERTICES];
```



Example for Adjacency Multlists





N0		0	1	N1	N3	edge (0, 1)
N1		0	2	N2	N3	edge (0, 2
N2		0	3	0	N4	edge (0, 3)
N3		1	2	N4	N5	edge (1, 2)
N4		1	3	0	N5	edge (1, 3)
N5		2	3	0	0	edge (2, 3)

The lists are

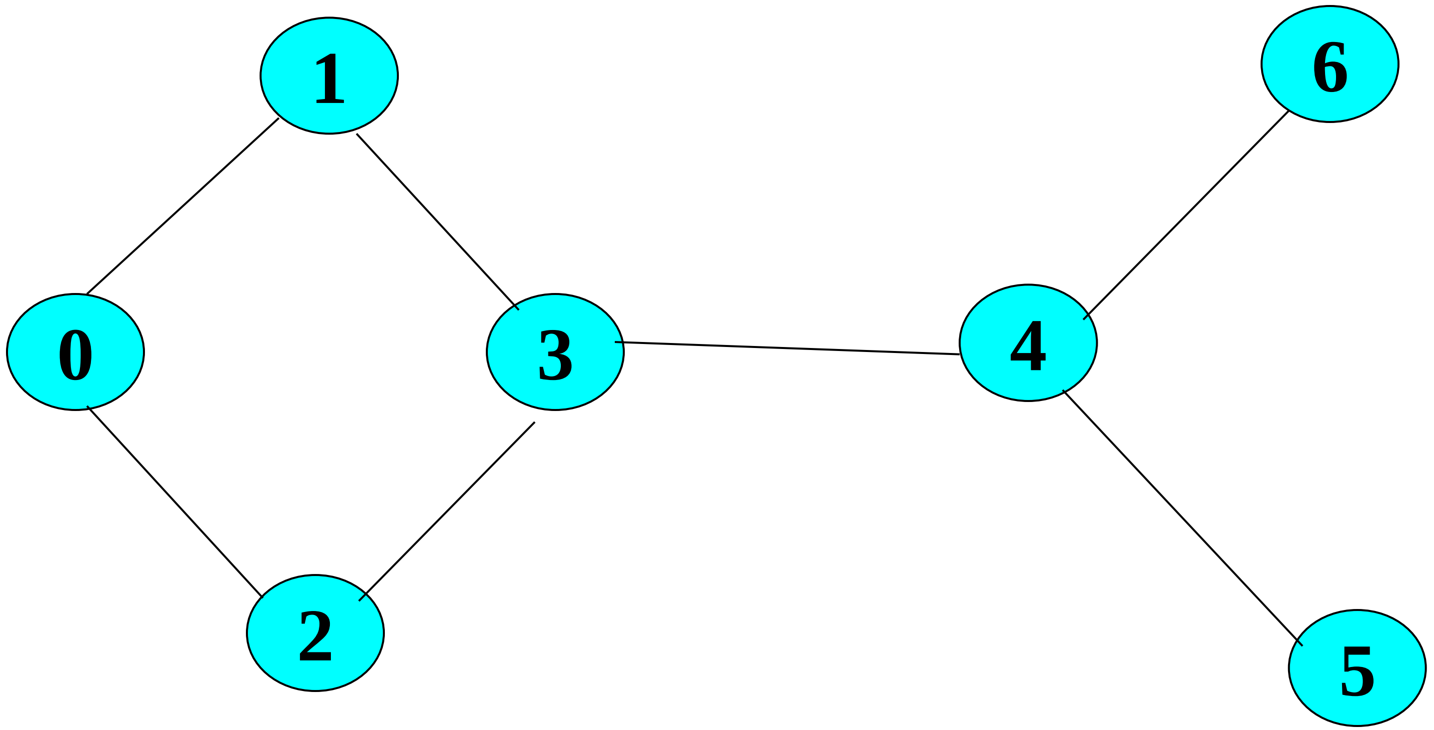
Vertex 0: N0 -> N1 -> N2

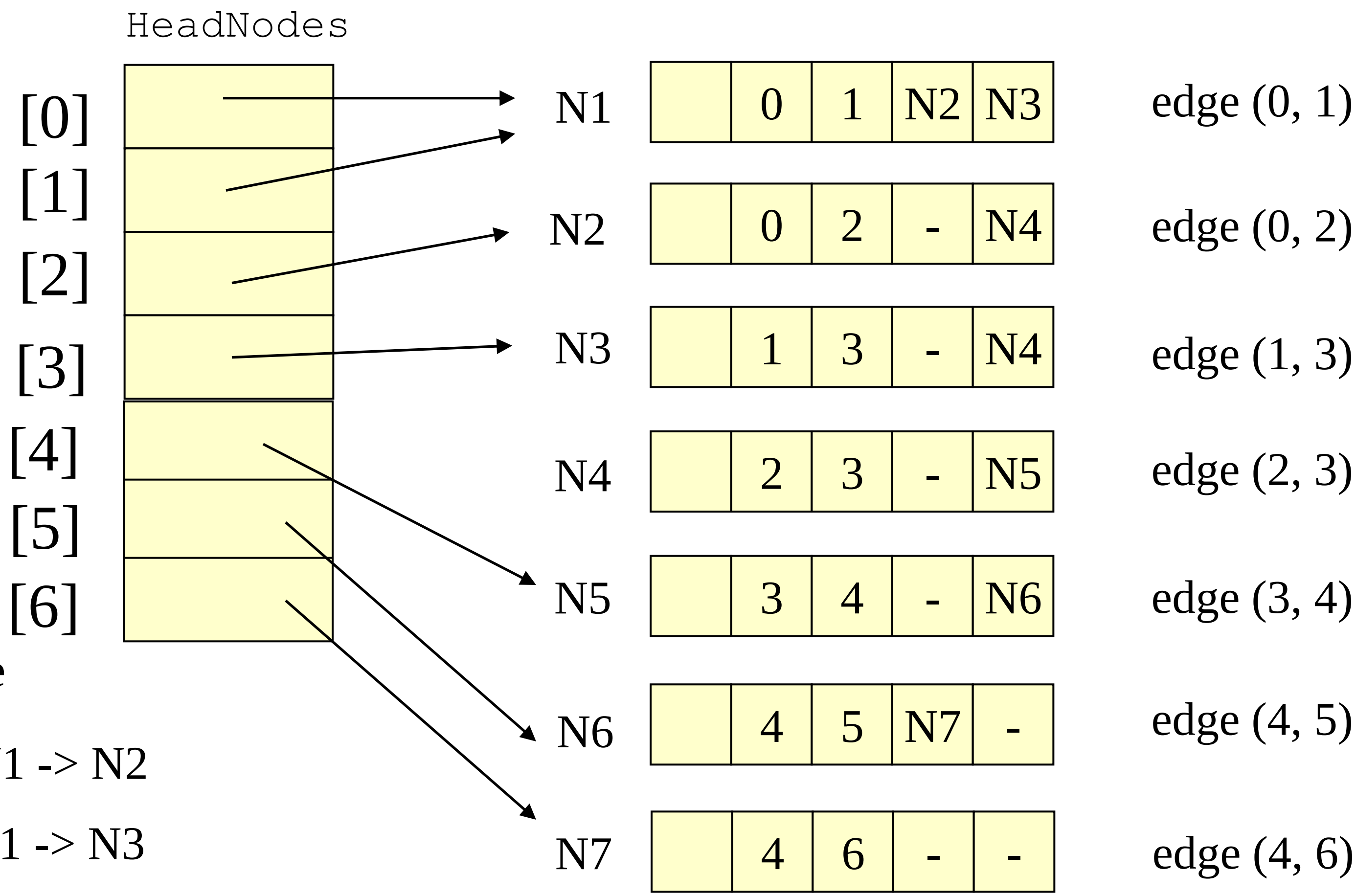
Vertex 1: N0 -> N3 -> N4

Vertex 2: N1 -> N3 -> N5

Vertex 3: N2 -> N4 -> N5

(Practice Example)





The lists are

Vertex 0: N1 -> N2

Vertex 1: N1 -> N3

Vertex 2: N2 -> N4

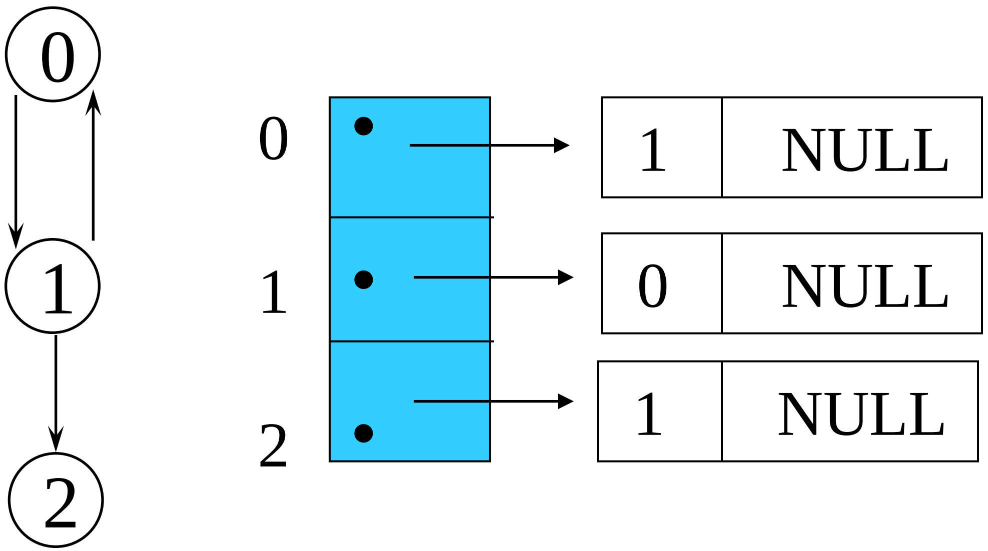
Vertex 3: N3 -> N4 -> N5

Vertex 4: N5 -> N6 -> N7

Vertex 5: N6

Vertex 6: N7

Inverse adjacency list



Determine in-degree of a vertex in a fast way.

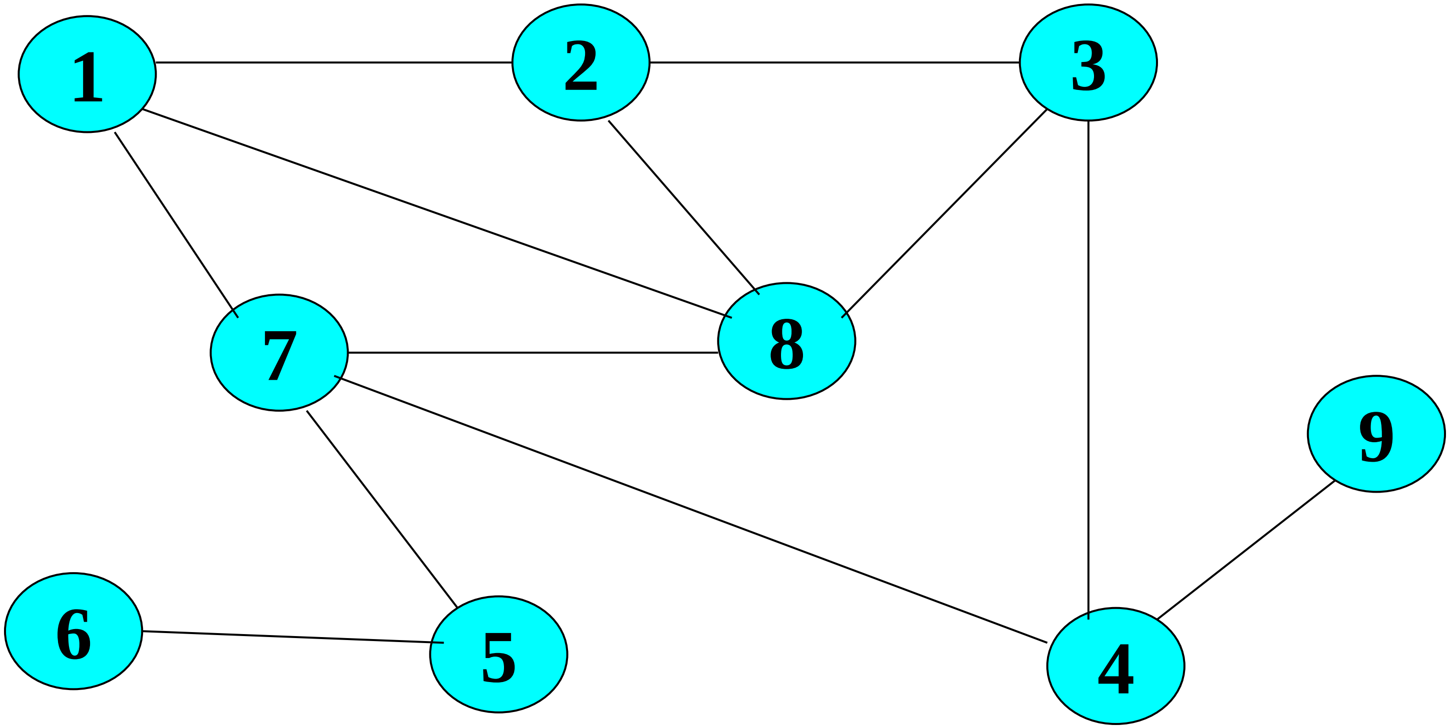
Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph
- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time

Depth-First Traversal (DFS)

- In depth-first traversal, we visit the starting node and then proceed to follow links through the graph until we reach a dead end.
- In an undirected graph, a node is a dead end if all of the nodes adjacent to it have already been visited.
- In a directed graph, if a node has no outgoing edges, we also have a dead end.
- When we reach a dead end, we back up along our path until we find an unvisited adjacent node and then continue in that new direction.
- The process will have completed when we back up to the starting node and all the nodes adjacent to it have been visited.

Depth-First Search (Example)



DFS : 1 2 3 4 7 5 6 8 9

Depth-First Search (Recursive algo)

Algorithm DepthFirstTraversal (G , v)

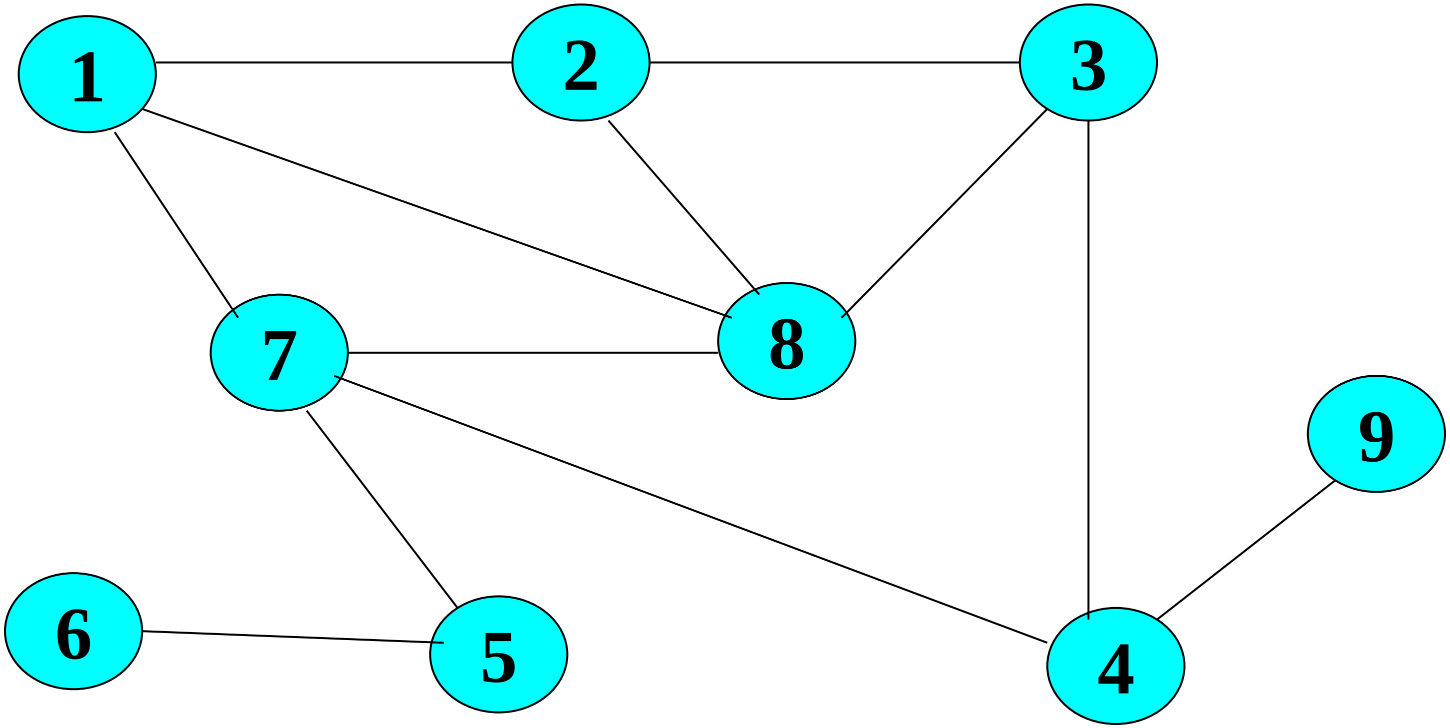
// G is the graph and v is the starting vertex

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

Breadth-First Search

- In a breadth-first traversal, we visit the starting node and then on the first pass visit all of the nodes directly connected to it.
- In the second pass, we visit nodes that are two edges “away” from the starting node.
- With each new pass we visit nodes that are one more edge away.
- Because there might be cycles in the graph, it is possible for a node to be on two paths of different lengths from the starting node.
- Because we will visit that node for the first time along the shortest path from the starting node, we will not need to consider it again.
- We will, therefore, either need to keep a list of the nodes we have visited or we will need to use a variable in the node to mark it as visited to prevent multiple visits.

Breadth-First Search (Example)



BFS : 1 2 7 8 3 4 5 9 6

Breadth-First Search (algo)

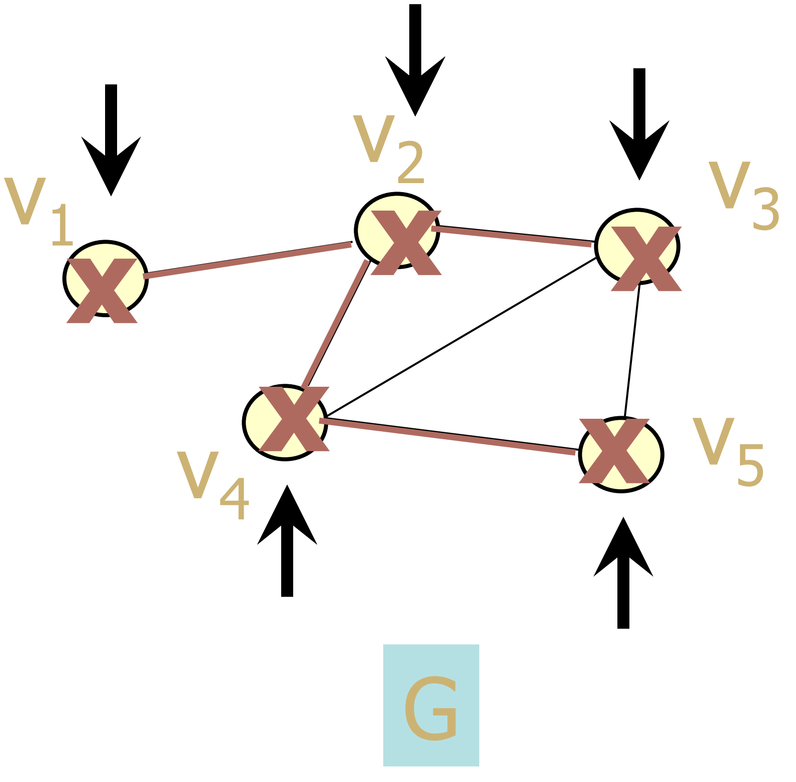
```
Algorithm BreadthFirstTraversal (G , sv )
  // G is the graph and sv is the starting vertex
 {
      Visit  ( sv )
      Mark ( sv )
      Enqueue ( sv )
      while the queue is not empty do
           Dequeue ( v )
           for every edge vw in G do
                if w is not marked then
                    Visit  ( w )
                    Mark ( w )
                    Enqueue ( w )
                end if
            end for
      end while
}
```

Non-recursive version of DFS algorithm

```
Algorithm DepthFirstTraversal_nonrecursive (G , sv )
  // G is the graph and sv is the starting vertex
 {
     Push(sv);
     Visit(sv);
      Mark(sv);
     While (Stack is not Empty)
    {
          let v be the node on the top of the stack
          if (no unvisited nodes are adjacent to v)
                pop();   // backtrack
         else
         {
            select an unvisited node w adjacent to v;
            push(w);
            Mark(w);
           Visit(w);
         }
    }
 }
```

Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4, v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Complexity of Graph Traversals

- Each vertex must be visited exactly once.
- At a vertex, we must determine all other vertices connected to the vertex.
- Adjacency matrix: $O(|V|^2)$.
- Adjacency list: $O(|V| + |E|)$.
- Each edge is examined once (directed) or twice (undirected).
- Typically, lists are better than matrices. The complexity of the traversal is linear in the number of edges.

Elementary Graph Operations

- Graph traversals provide the basis for many elementary graph operations:
  - Spanning trees on graphs
  - Graph cycles
  - Connected components of a graph

Applications: Finding a Path

- Find path from source vertex s to destination vertex d
- Use graph search starting at s and terminating as soon as we reach d
  - Need to remember edges traversed
- Use depth – first search ?
- Use breath – first search?

Shortest Path Algorithm

- Useful to find the shortest path among various given path
- Example : Railway network connecting several cities.
- Vertices represent the cities
- Edges represent the railway route
- Weight of the edge is the distance between two cities.

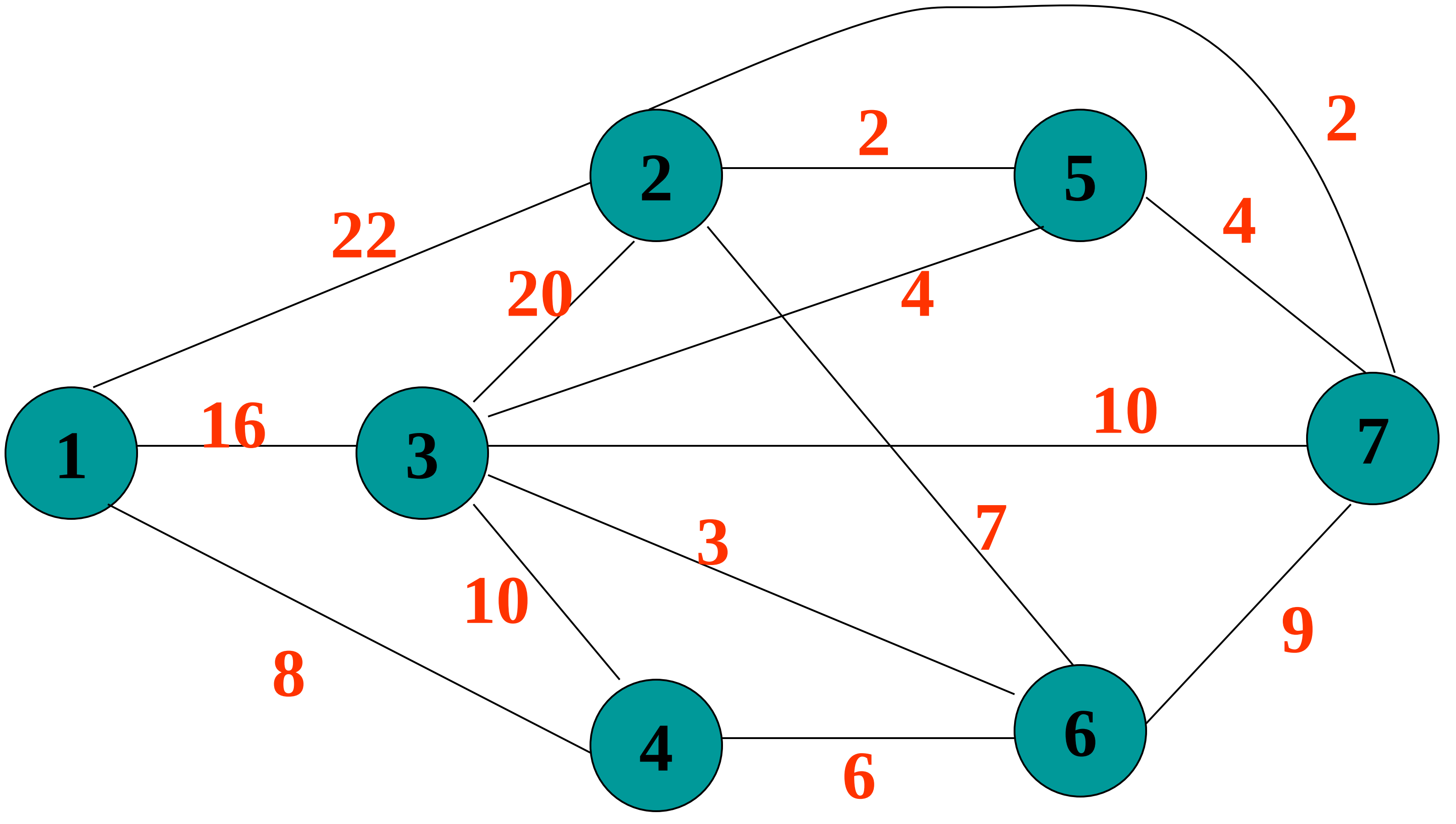
Dijkstra’s shortest path algorithm

- Let G = (V,E) be a simple weighted graph represented by adjacency matrix.
- Let ‘ s ’ be the source vertex
- Let Dist(i) denote the length of the shortest path from source vertex ‘ s ’ to the vertex ‘ i ’.
- Let G[i][j] denote the weight of edge e_{ij}
- Let Visit(i) denote whether the vertex ‘ i ’ is visited or not visited.
- Let From(i) denote the predecessor vertex from which the shortest path to reach to vertex ‘ i ’ will be given.

Dijikstra's Shortest path algorithm

- Step 1
  - For all i Initialize Visit(i) → 0 , Dist(i) → ∞ , From(i) → ∞
  - Set Dist(s) → 0 From(s) → s
- Step 2
  - Select a Vertex ' v ' which is not yet visited and has the smallest value in the Dist array
  - Mark the Vertex ' v ' as visited i.e. Visit(v) → 1
  - If v == destination vertex ' d ' then stop
- Step 3
  - Revise the Dist array for those vertices which are not yet visited by
    - Dist(x) = min(old Dist(x) , Dist(v) + G[v][x])
  - For all vertices x for which the distance are revised
    - From(x) = v
- Step 4
  - Repeat Step 2 and 3 till all vertices are visited or destination vertex is reached.

Example (Dijikstra's Algorithm)



Step 1

Step 1 : Initialize all the arrays and set Dist(s) = 0 From(s) = s

	1	2	3	4	5	6	7
Visit	0	0	0	0	0	0	0
From	1	∞	∞	∞	∞	∞	∞
Dist	0	∞	∞	∞	∞	∞	∞

Step 2

Step 2 : Select vertex v = 1 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	0	0	0	0	0	0
From	1	1	1	1	∞	∞	∞
Dist	0	22	16	8	∞	∞	∞

Step 3

Step 3 : Select vertex v = 4 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	0	0	1	0	0	0
From	1	1	1	1	∞	4	∞
Dist	0	22	16	8	∞	14	∞

Step 4

Step 4 : Select vertex v = 6 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	0	0	1	0	1	0
From	1	6	1	1	∞	4	6
Dist	0	21	16	8	∞	14	23

Step 5

Step 5 : Select vertex v = 3 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	0	1	1	0	1	0
From	1	6	1	1	3	4	6
Dist	0	21	16	8	20	14	23

Step 6

Step 6 : Select vertex v = 5 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	0	1	1	1	1	0
From	1	6	1	1	3	4	6
Dist	0	21	16	8	20	14	23

Step 7

Step 7 : Select vertex v = 2 with minimum label , mark it and revise the labels

	1	2	3	4	5	6	7
Visit	1	1	1	1	1	1	0
From	1	6	1	1	3	4	6
Dist	0	21	16	8	20	14	23

Step 8

Step 8 : Select vertex v = 7 with minimum label , mark it and stop

	1	2	3	4	5	6	7
Visit	1	1	1	1	1	1	1
From	1	6	1	1	3	4	6
Dist	0	21	16	8	20	14	23

Shortest path and length

Sr. No	Source - destination	Path length	Path
1	1 – 1	0	1 → 1
2	1 – 2	21	1 → 4 → 6 → 2
3	1 – 3	16	1 → 3
4	1 – 4	8	1 → 4
5	1 – 5	20	1 → 3 → 5
6	1 – 6	14	1 → 4 → 6
7	1 – 7	23	1 → 4 → 6 → 7

DijkstrasAlgorithm (G,n,s,d)

```
  for(i=1;i<=n;i++)
      From[i] = Infinity;   ,    Dist[i] = Infinity;  ,    Visit[i] =
   0;
 Dist[s] = 0;          From[s] = s;
do
{
    V = find_vertex_with_minimum_label(Dist,Visit,n);
    Visit[V] = 1;
    if(V == d)            break;
   for(i=1;i<=n;i++)
     if(Visit[i] != 1 && (Dist[V] + G[V][i] < Dist[i]) )
     {
        Dist[i] = Dist[V]+ G[V][i];
        From[i] = V;
     }
}while(1);
Shortest path length  from s to d will Dist[V];
```

How to get the path from From array

```
k = 0
Path[k++] = d;
j = From[d];
while(j != s)
{
    Path[k++]  = j;
    j = From[j];
}
Path[k++] = s;
printf("\n Path from source to destination : ");
 for(i=k -1;i>=0;i--)
      printf("  %d",Path[i]);
```

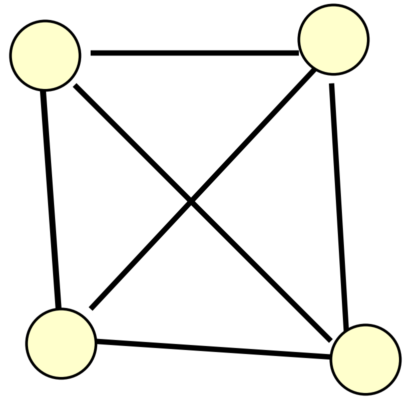
Find_vertex_with_minimum_label (Dist,Visit,n)

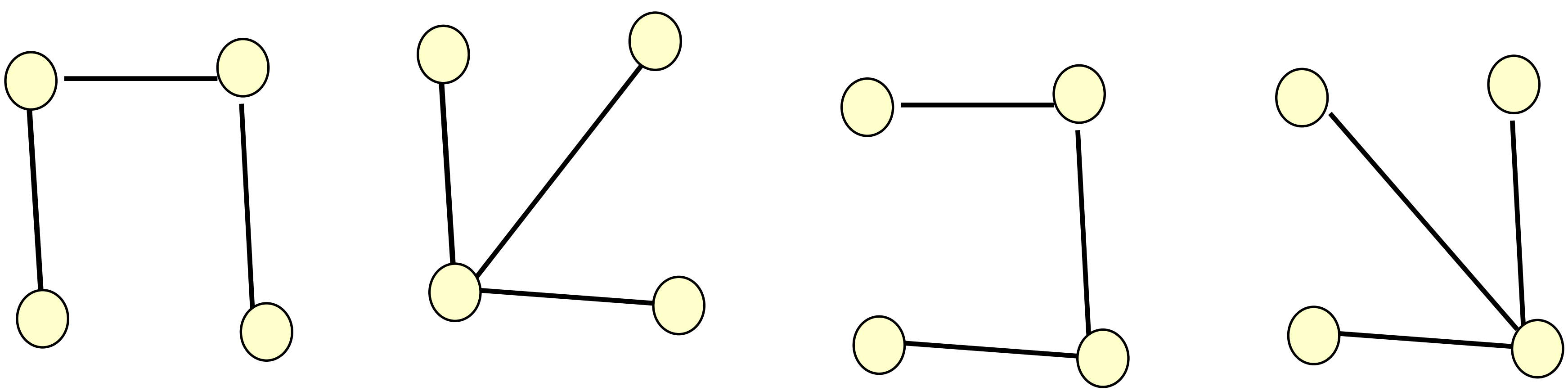
```
min=Infinity;
 for(i=1;i<=n;i++)
  if(min > Dist[i] && Visit[i] == 0)
  {
    index = i;
    min = Dist[i];
  }
 return index;
```

Spanning Tree

- Let G = (V,E) be an undirected connected graph. A subgraph T = (V,E’) of G is a spanning tree of G iff T is a tree.
- The spanning tree of a graph is actually a subset of a graph which is obtained by eliminating some edges of the graph.
- Used to obtain an independent set of circuit equations for an electric network.

Spanning Tree

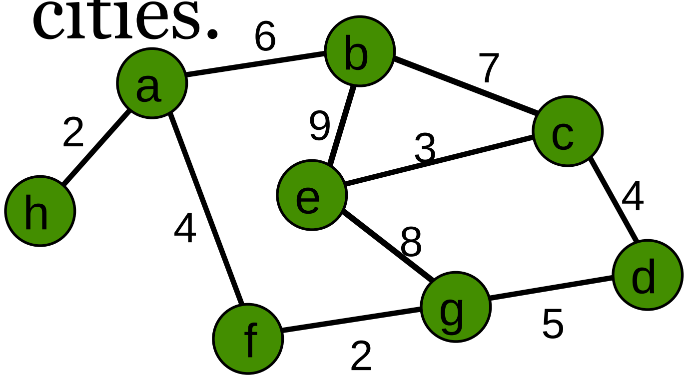


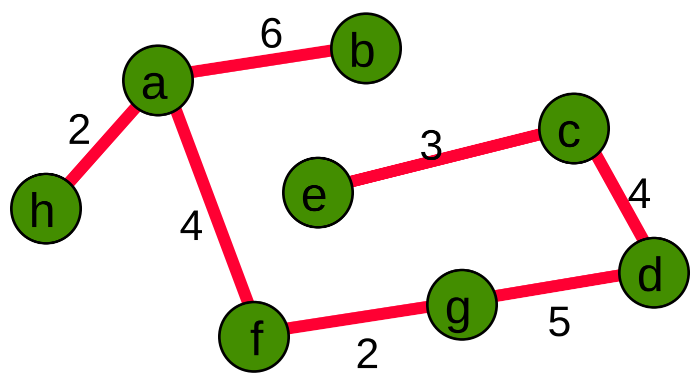


(a) (b) (c) (d)

Minimum Spanning Trees

- A minimum spanning tree of a *connected weighted graph* is a collection of edges connecting all vertices such that the sum of the weights of the edges is the smallest possible.
- MST's are useful in building a network or roads or railway lines connecting a number of cities.





Greedy Algorithms

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

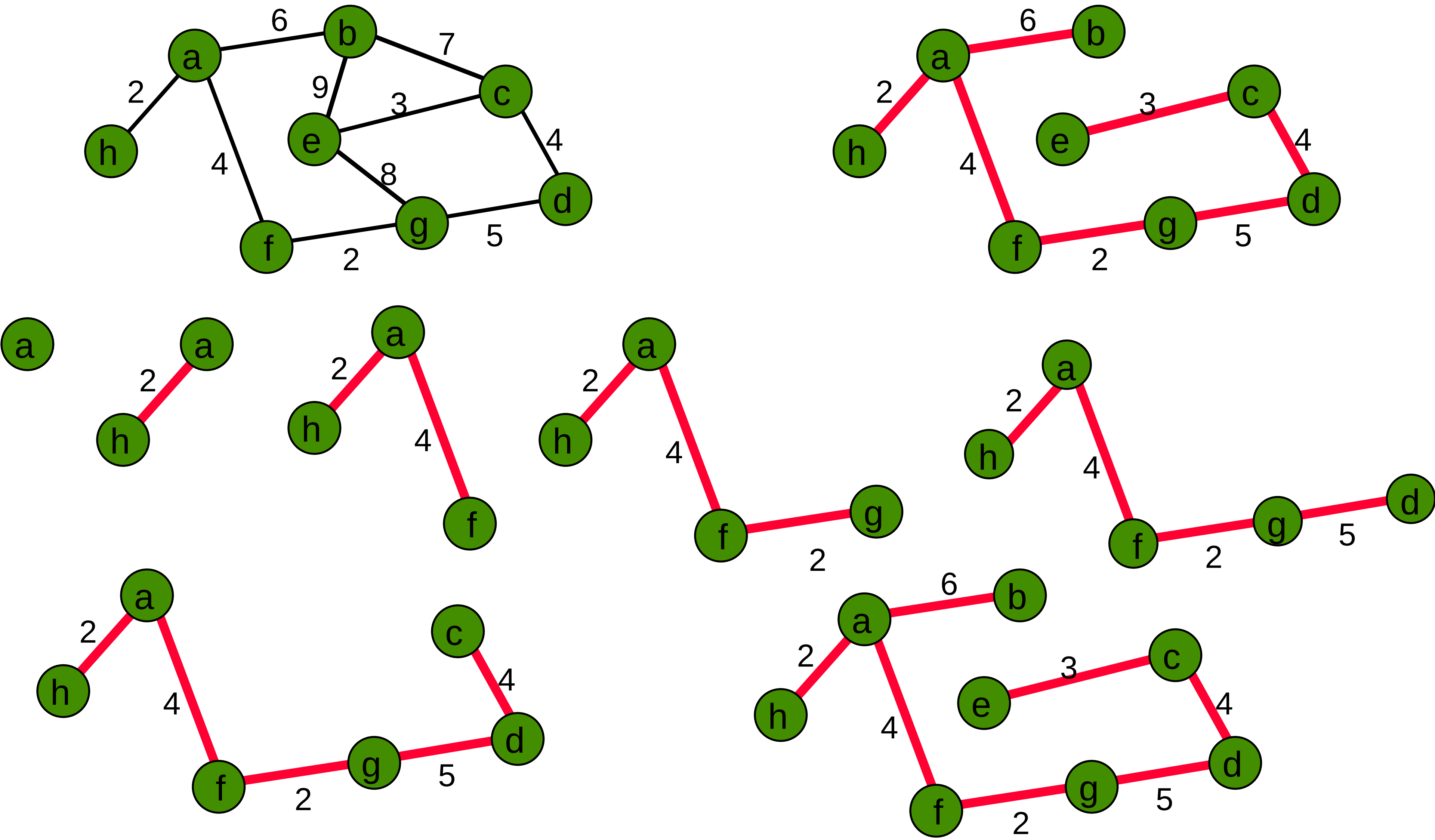
Prim's Algorithm to find MST.

- Let G = (V, E) be a connected weighted graph.
- Let T be the MST
- Step 1
  - Take a vertex V_0 in the graph G
  - Set T = $\{V_0\}$
- Step 2
  - Find the edge $E_1 = (V_0, V_1)$ from G such that its one end vertex V_0 is in T and its weight is minimum.
  - Include the vertex V_1 and the edge E_1 to T
    - i.e $T = \{ \{V_0, V_1\}, \{E_1\}\}$

Prim's Algorithm to find MST.

- Step 3
  - Choose the next edge $E_i = (V_i, V_j)$ in such a way that its one end vertex V_i is in T and the other end vertex V_j is not in T i.e (E_i should not form the circuit with the edges in T) and the weight of the edge E_i is as small as possible.
  - Include the edge E_i and vertex V_j to T
- Step 4
  - Repeat the step 3 until T contains all the vertices of G.
  - The set T will give the minimum spanning tree of the graph G.

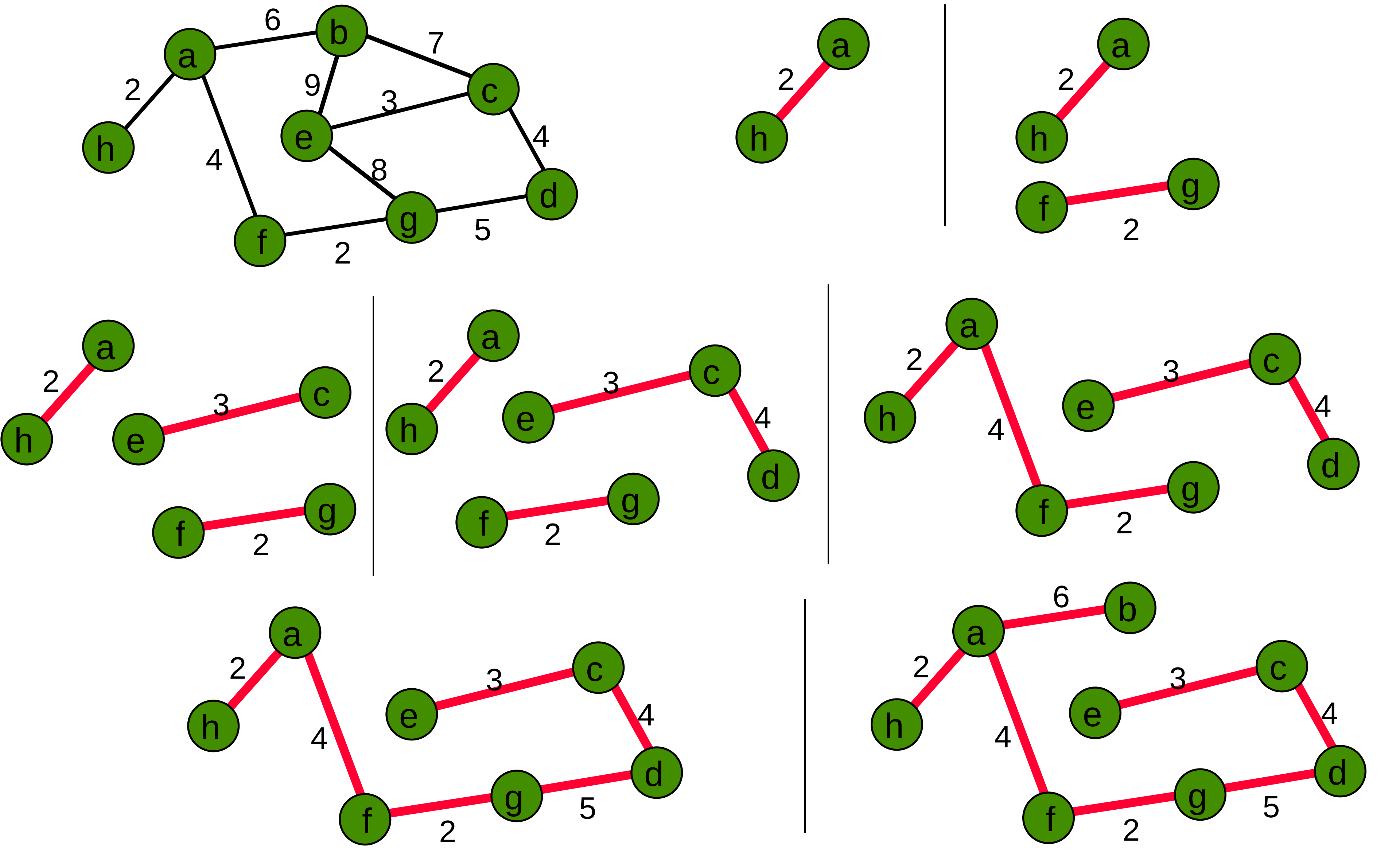
Example



Kruskal’s Algorithm to find MST.

- Let G = (V, E) be a connected weighted graph.
- Let T be the MST
- Step 1 : Pick up an edge e_i of G such that its weight is minimum.
- Step 2 : If edge e_1, e_2 , … e_k have been chosen then pick an edge e_{k+1} such that
  - $e_{k+1} \neq e_i$ for any i = 1,2 , … , k
  - The edge e_1, e_2 , … e_k , e_{k+1} do not form a ckt.
  - The weight of is as small as e_{k+1} possible.
- Step 3 : Stop when all vertices are included & Step 2 cannot be implemented.

Example



a b c d e f g h
6 7 2 9 3 4 4 8 2 5
a h 2
a h 2 f g 2
a h 2 e c 3 f g 2
a h 2 e c 3 c d 4 f g 2
a h 2 a f 4 e c 3 c d 4 f g 2
a h 2 a f 4 e c 3 c d 4 f g 2 g d 5
a b 6 a h 2 a f 4 e c 3 c d 4 f g 2 g d 5

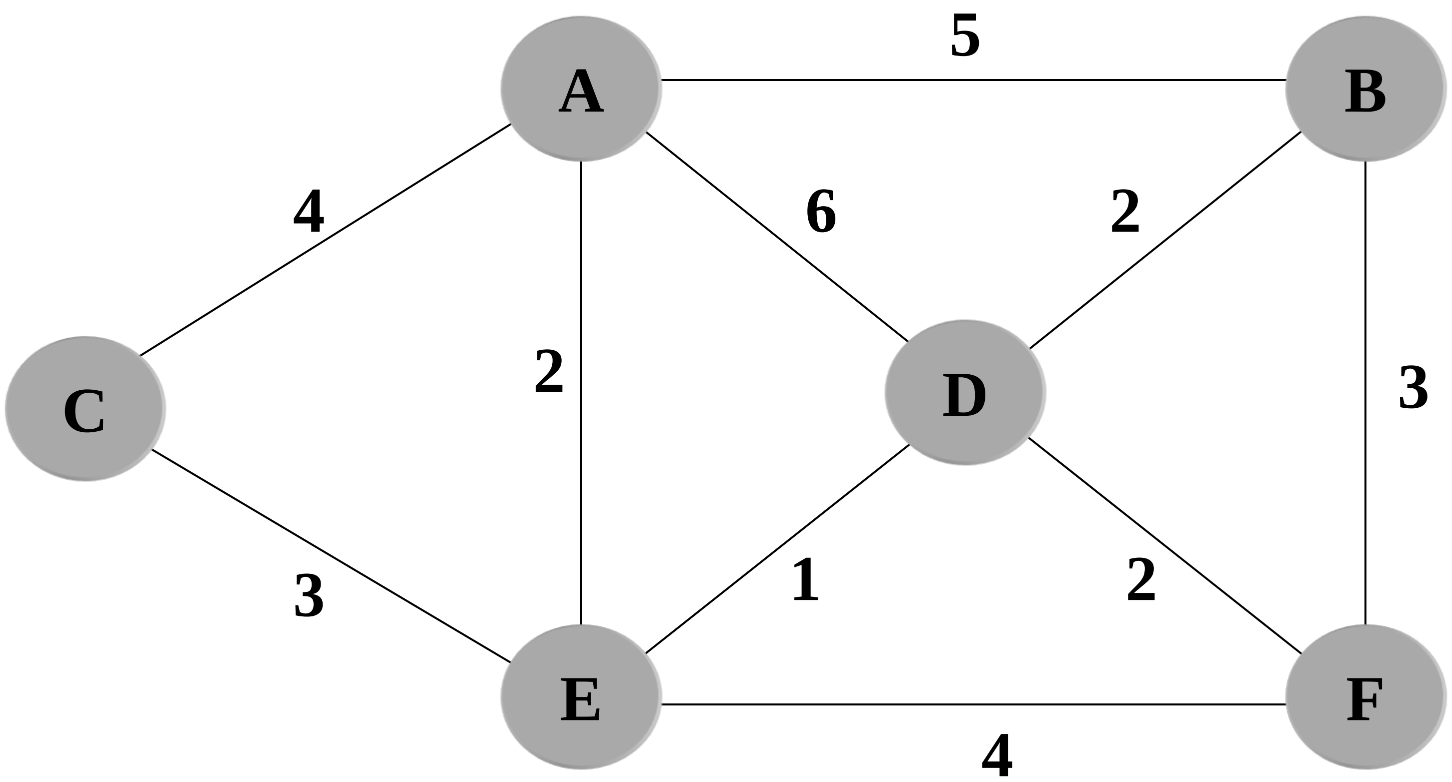
Prim‘s Algorithm

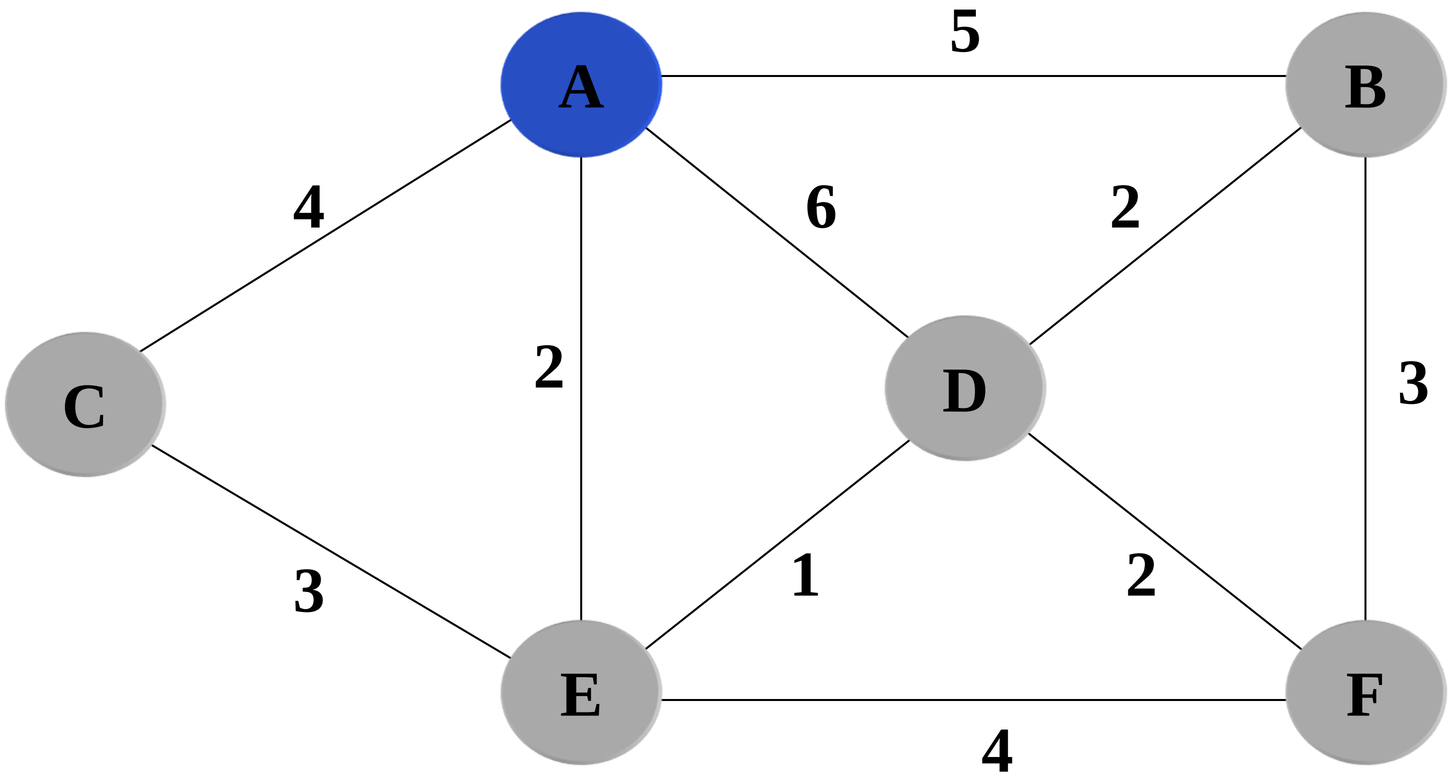
1. All vertices are marked as not visited

2. Any vertex ***v*** you like is chosen as starting vertex and is marked as visited (define a cluster ***C***)

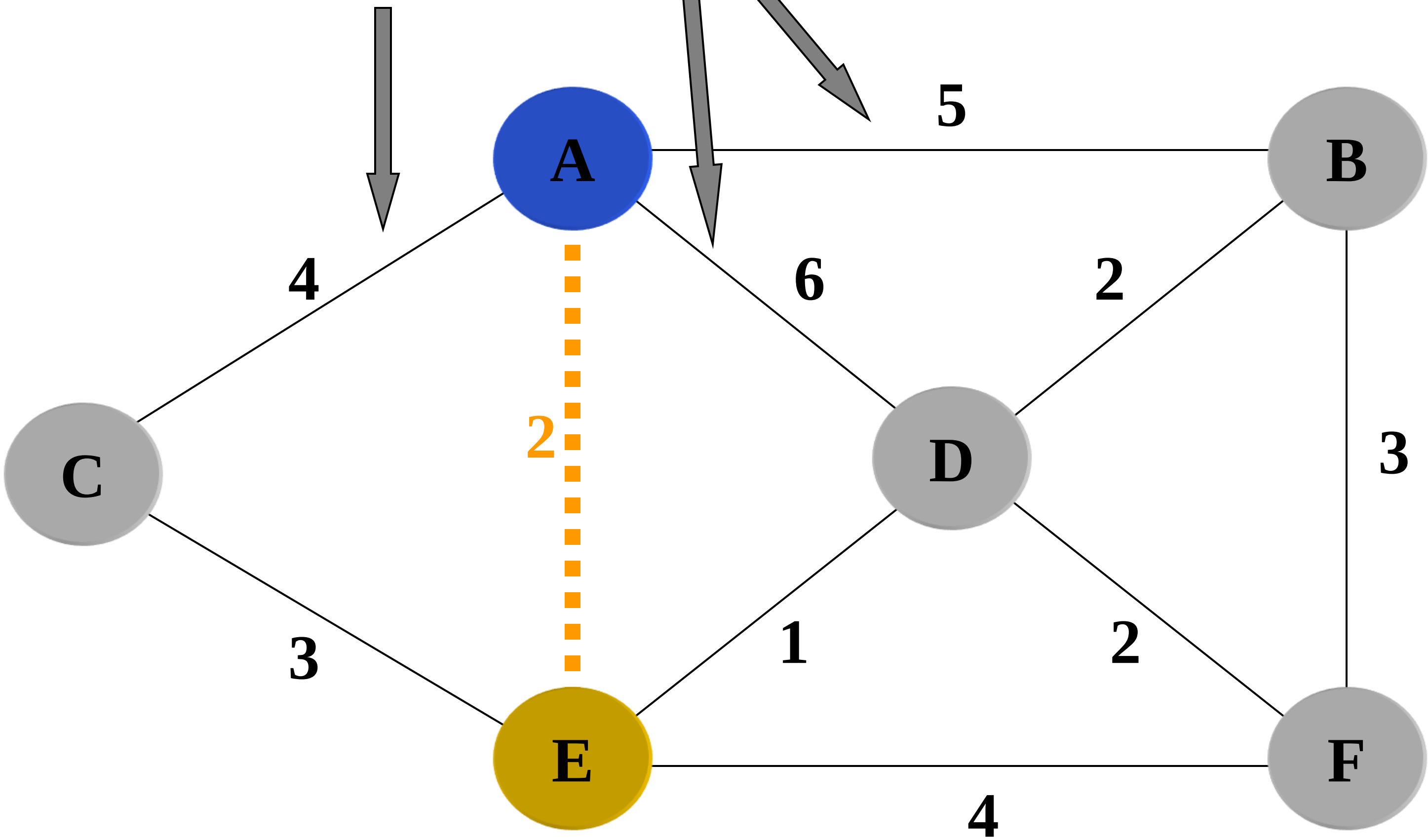
3. The smallest- weighted edge ***e = (v,u)***, which connects one vertex *v* inside the cluster ***C*** with another vertex ***u*** outside of ***C***, is chosen and is added to the MST.

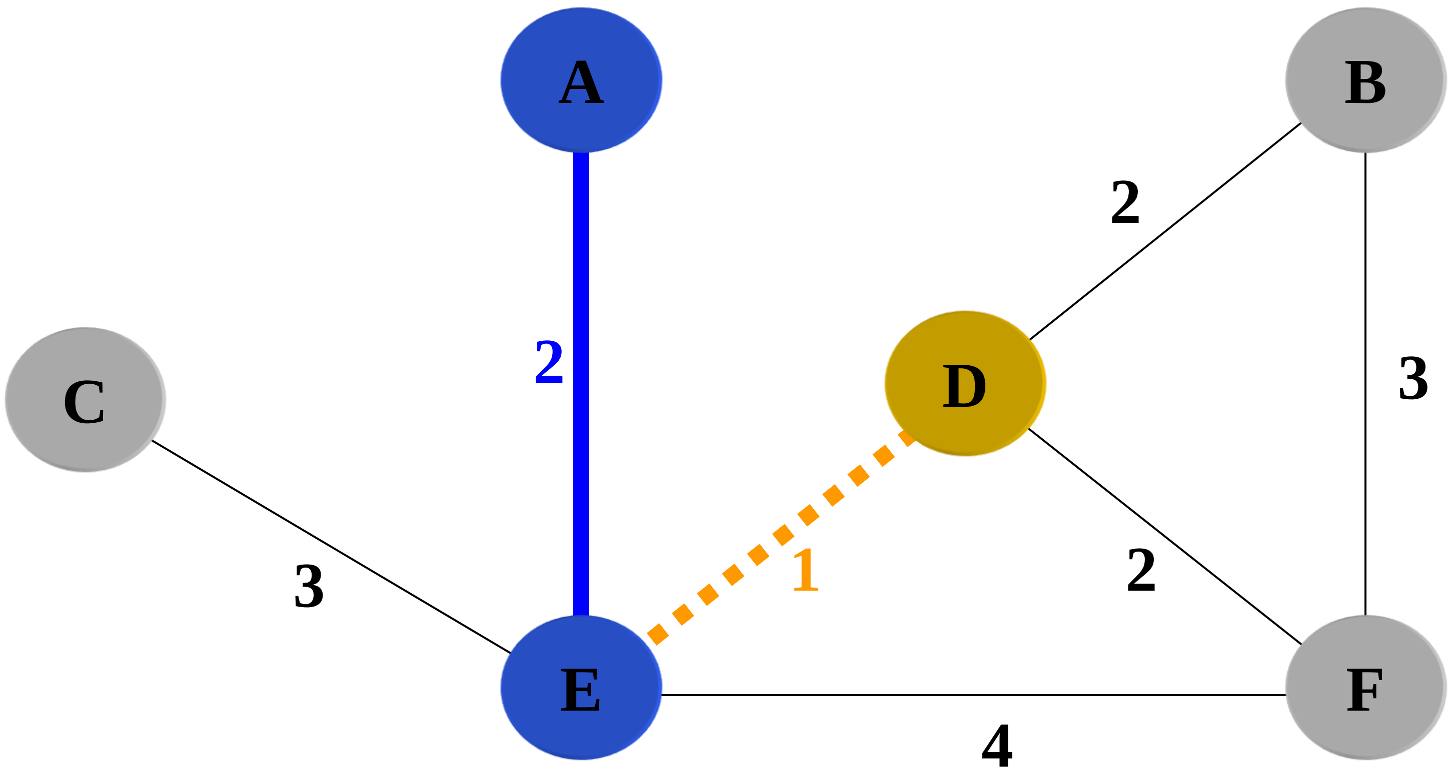
4. The process is repeated until a spanning tree is formed

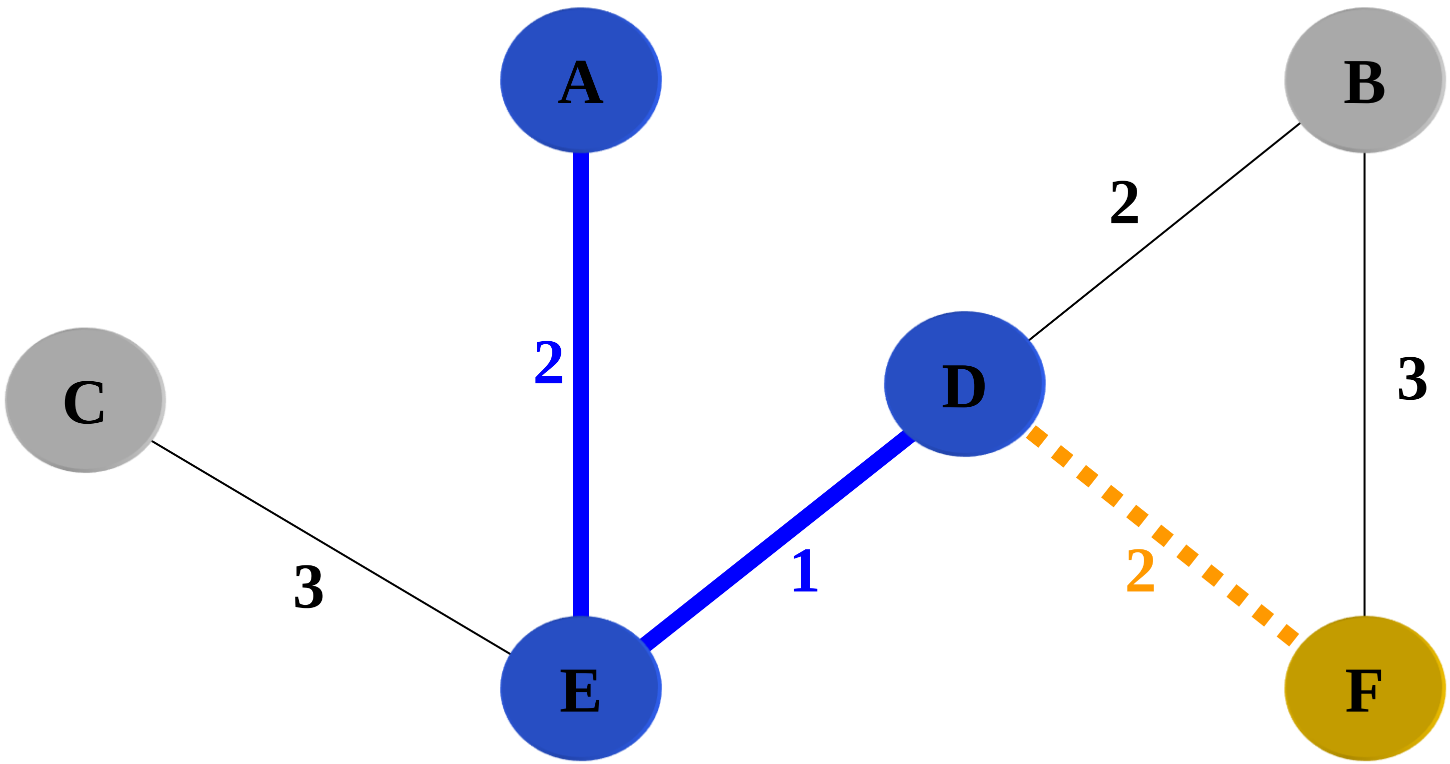

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4


A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

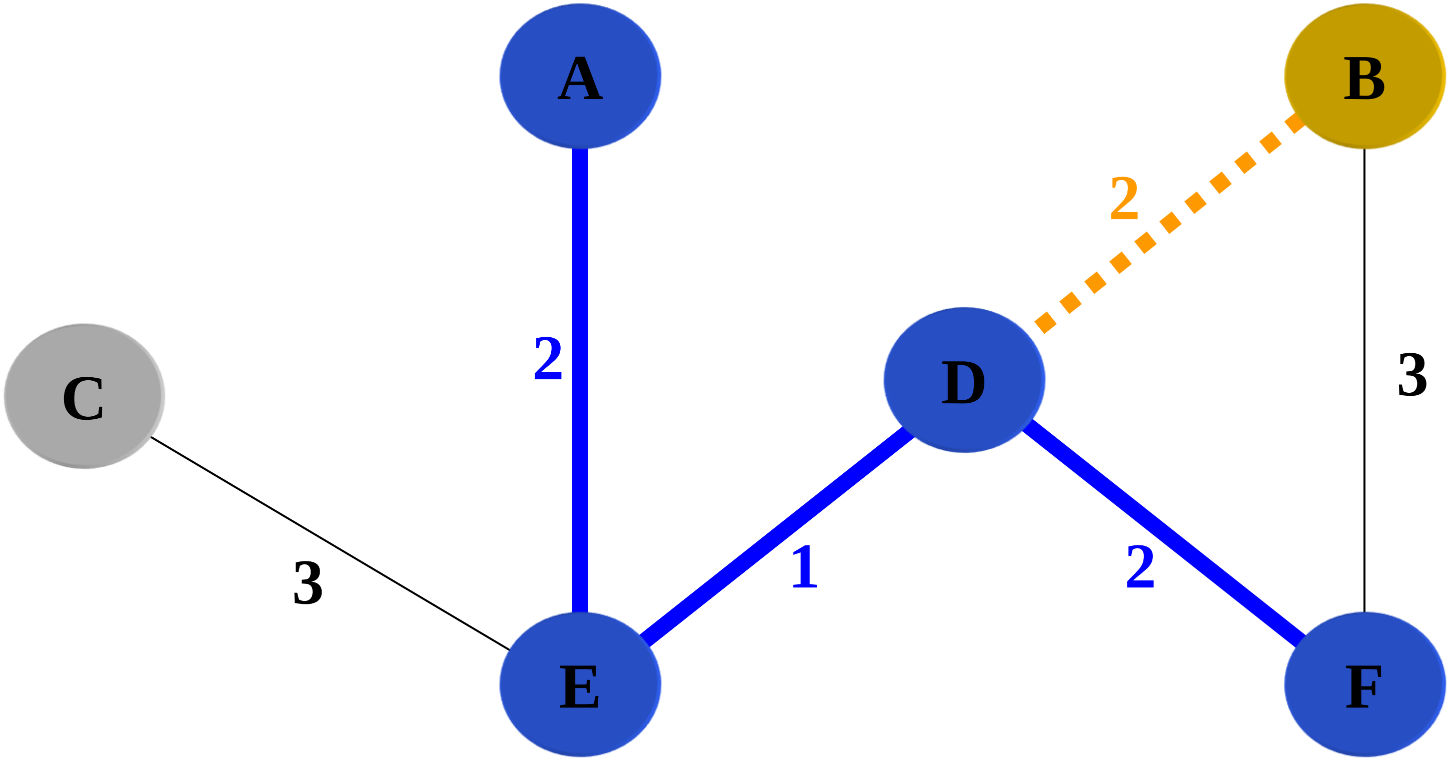
We could delete these edges because of Dijkstra‘s label ***D[u]*** for each vertex outside of the cluster

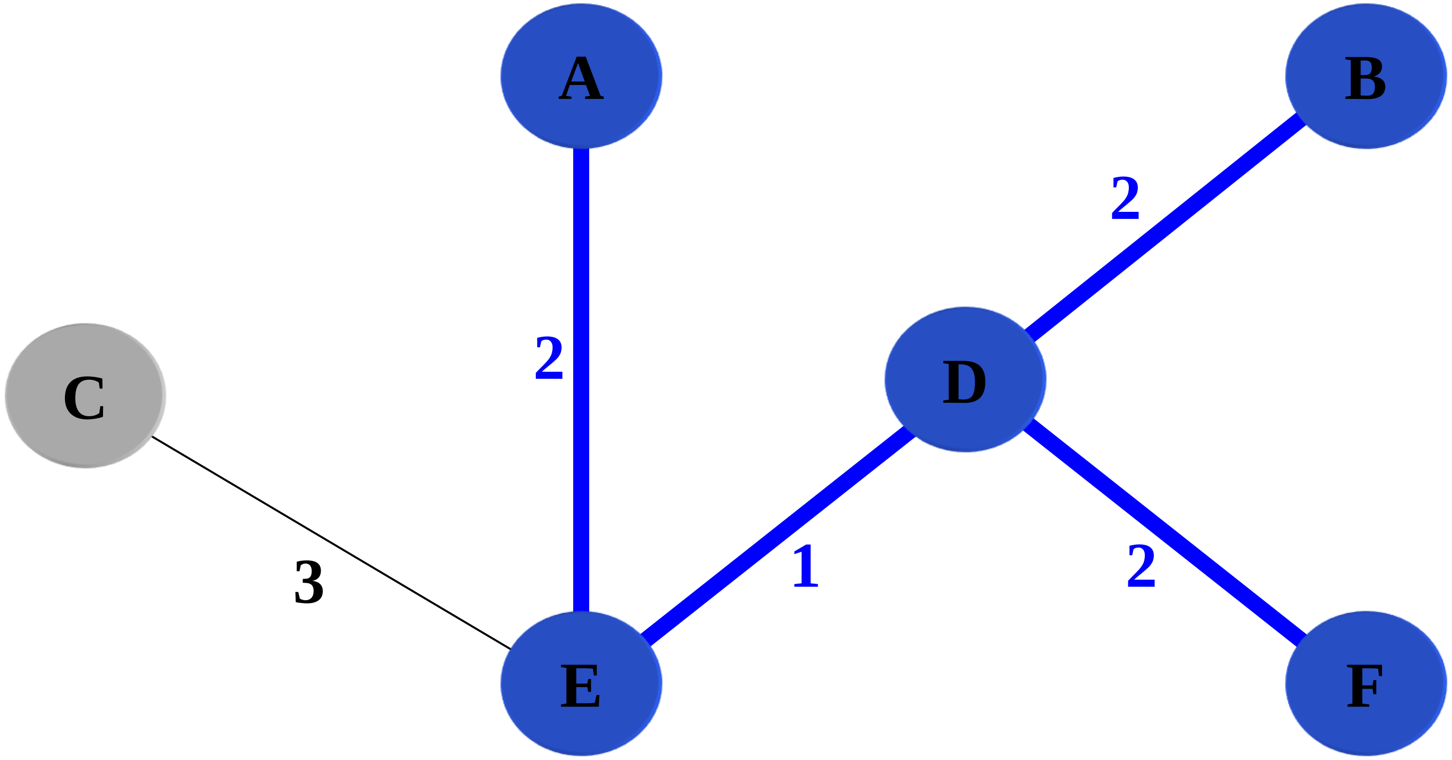


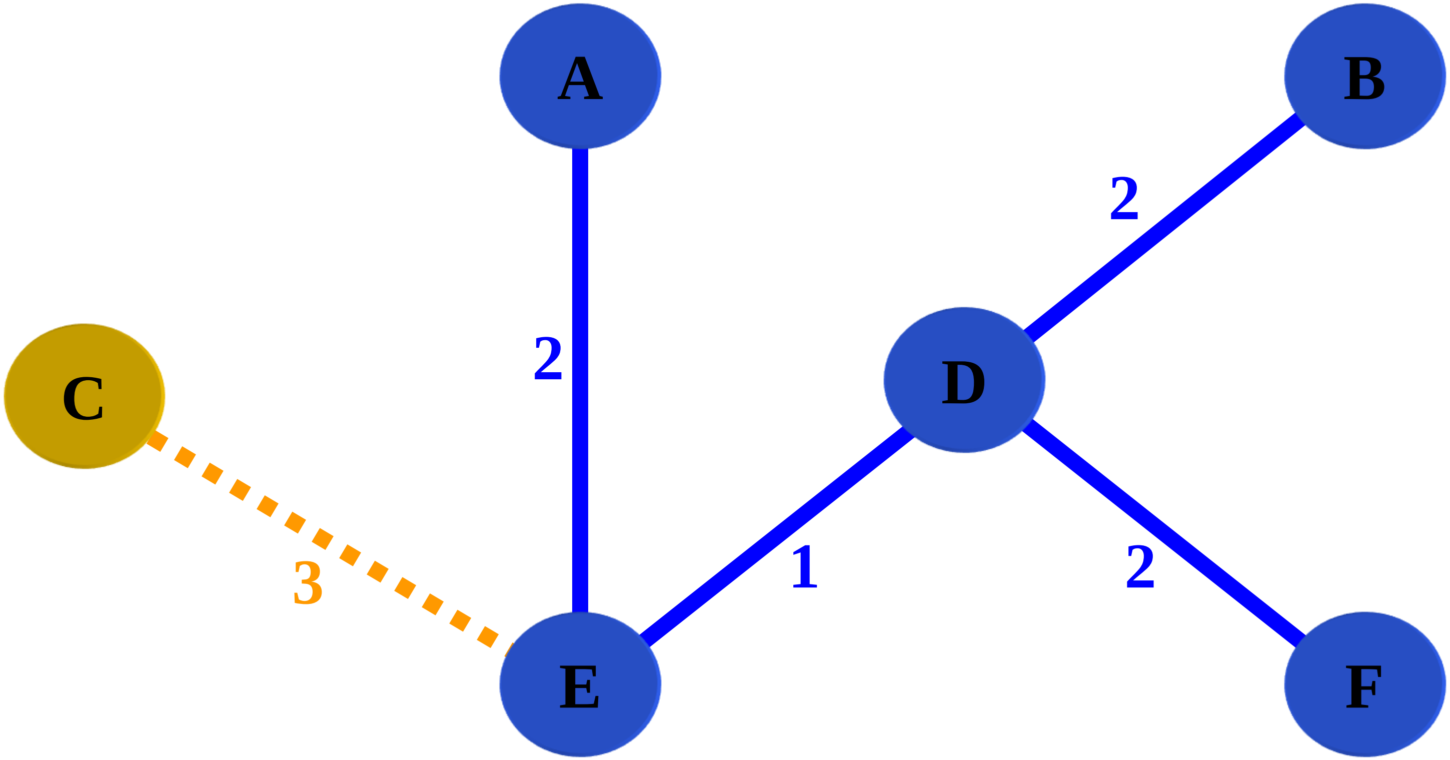

A
B
C
D
E
F
2
2
3
3
1
2
4



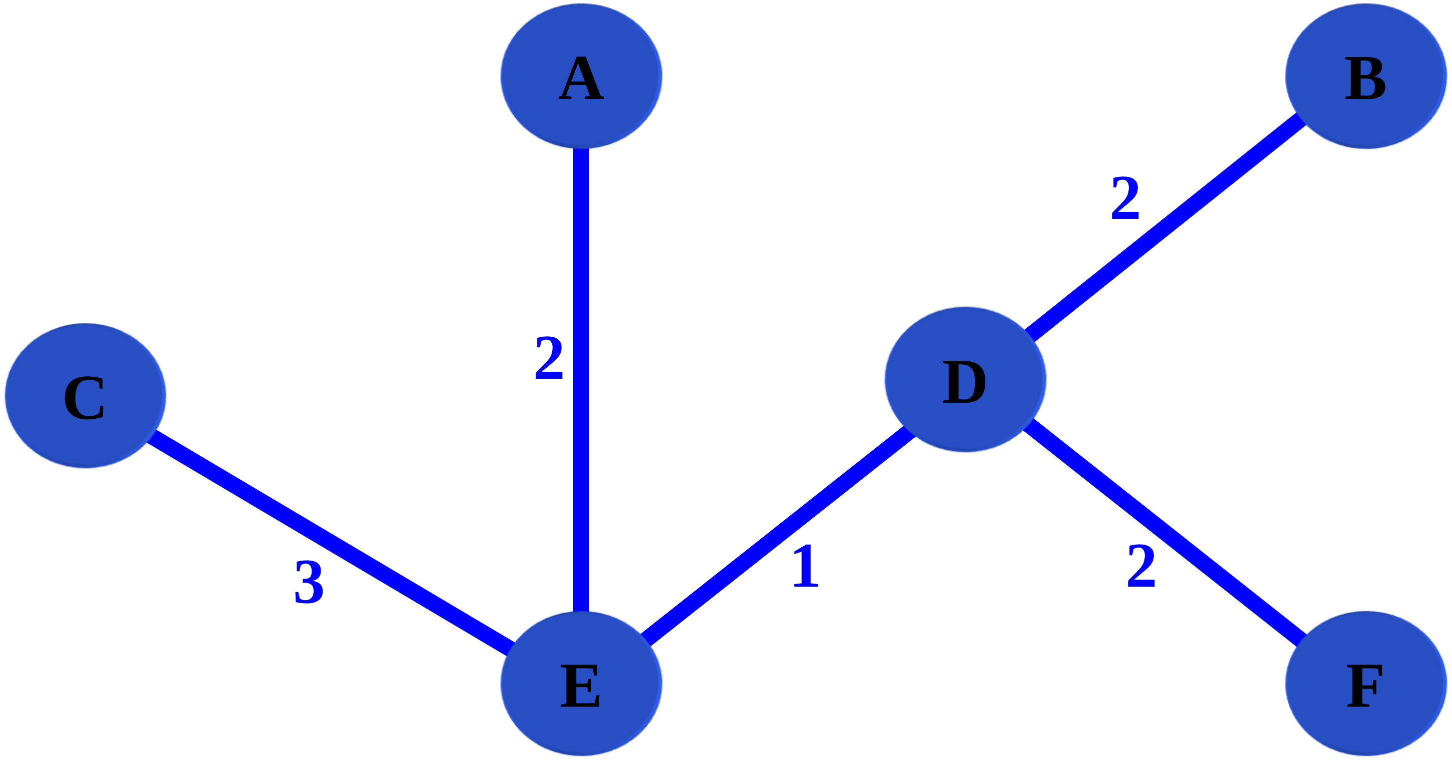
A
B
2
2
C
D
3
3
1
2
E
F


A
B
C
D
E
F
2
2
3
3
1
2


A
B
2
2
C
D
3
1
2
E
F


A
B
C
D
E
F
2
2
1
2
3

minimum- spanning tree



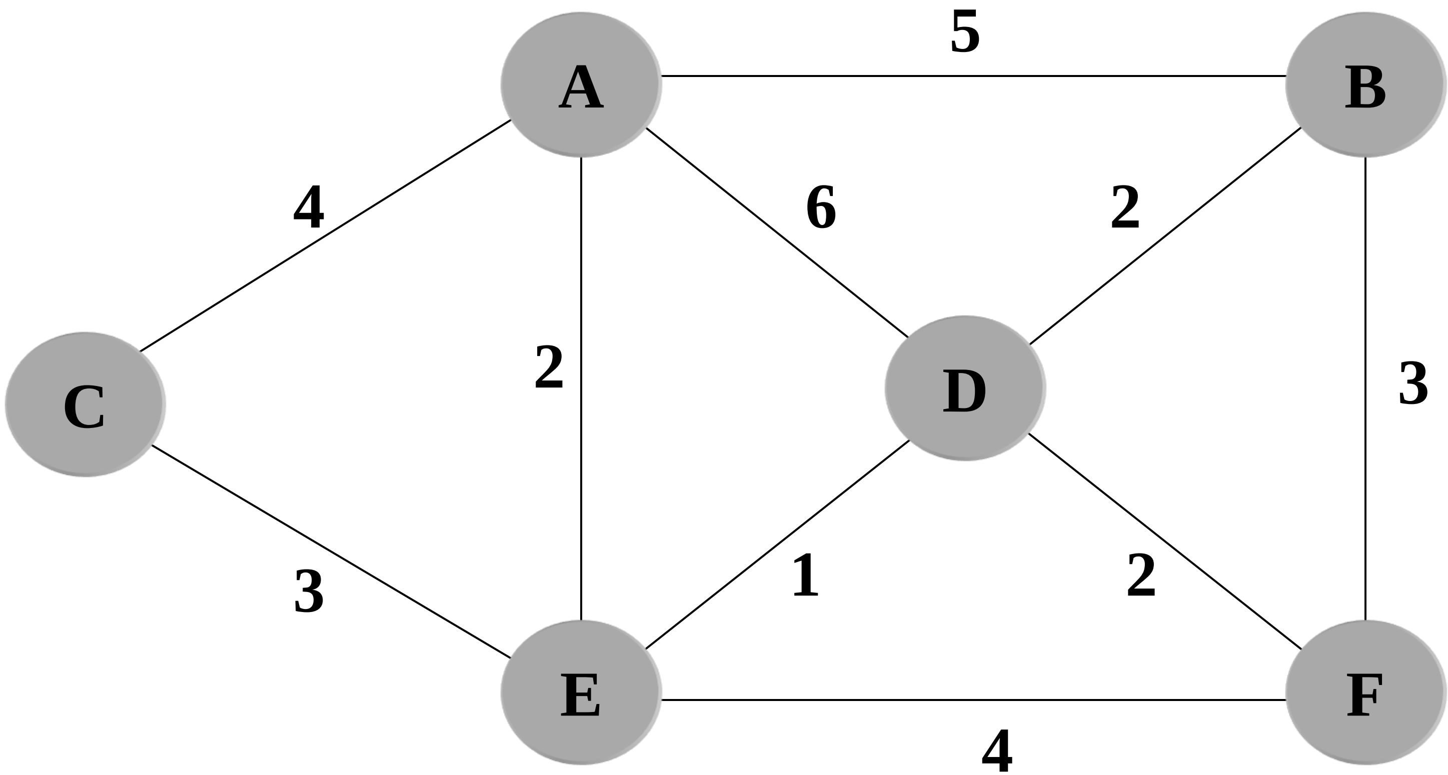
Running time: $O(n^2)$

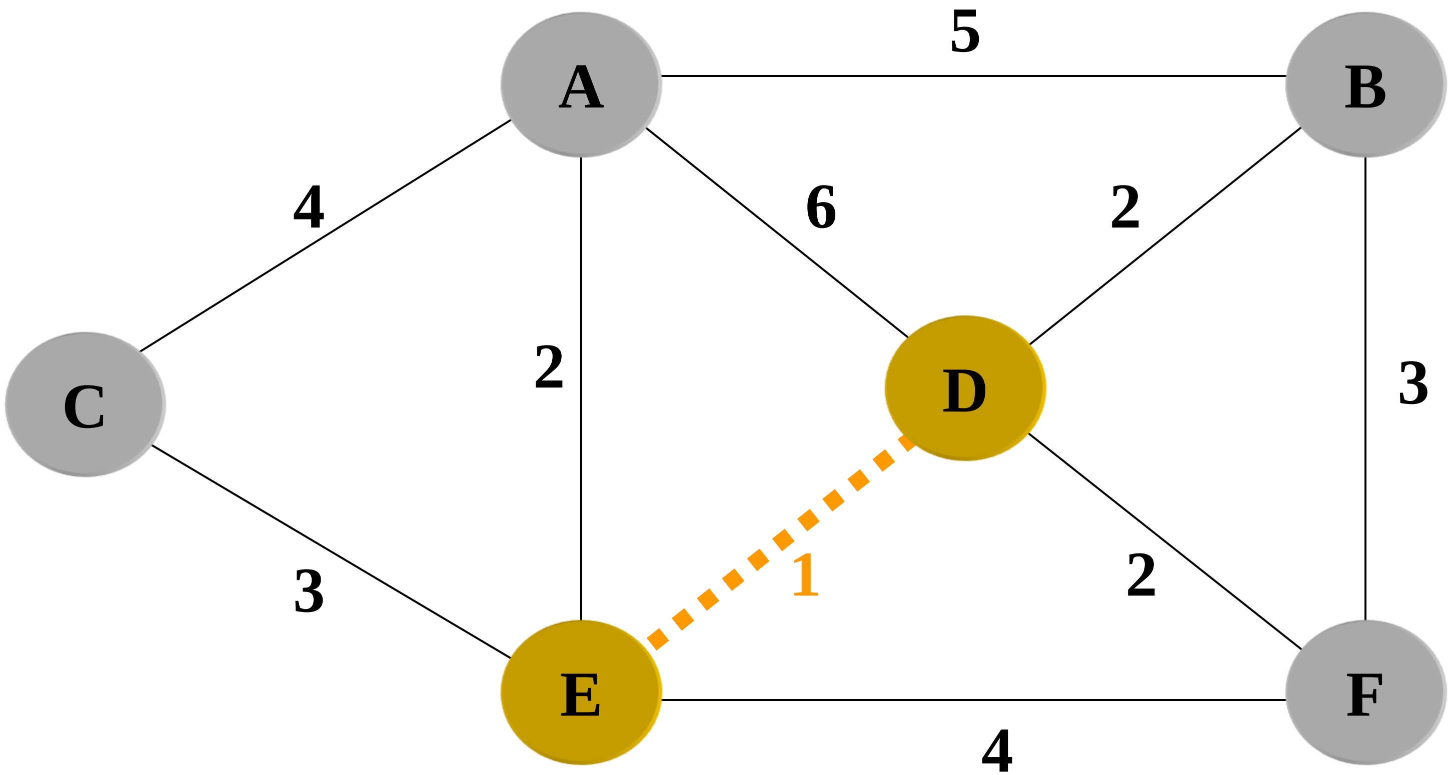
Kruskal‘s Algorithm

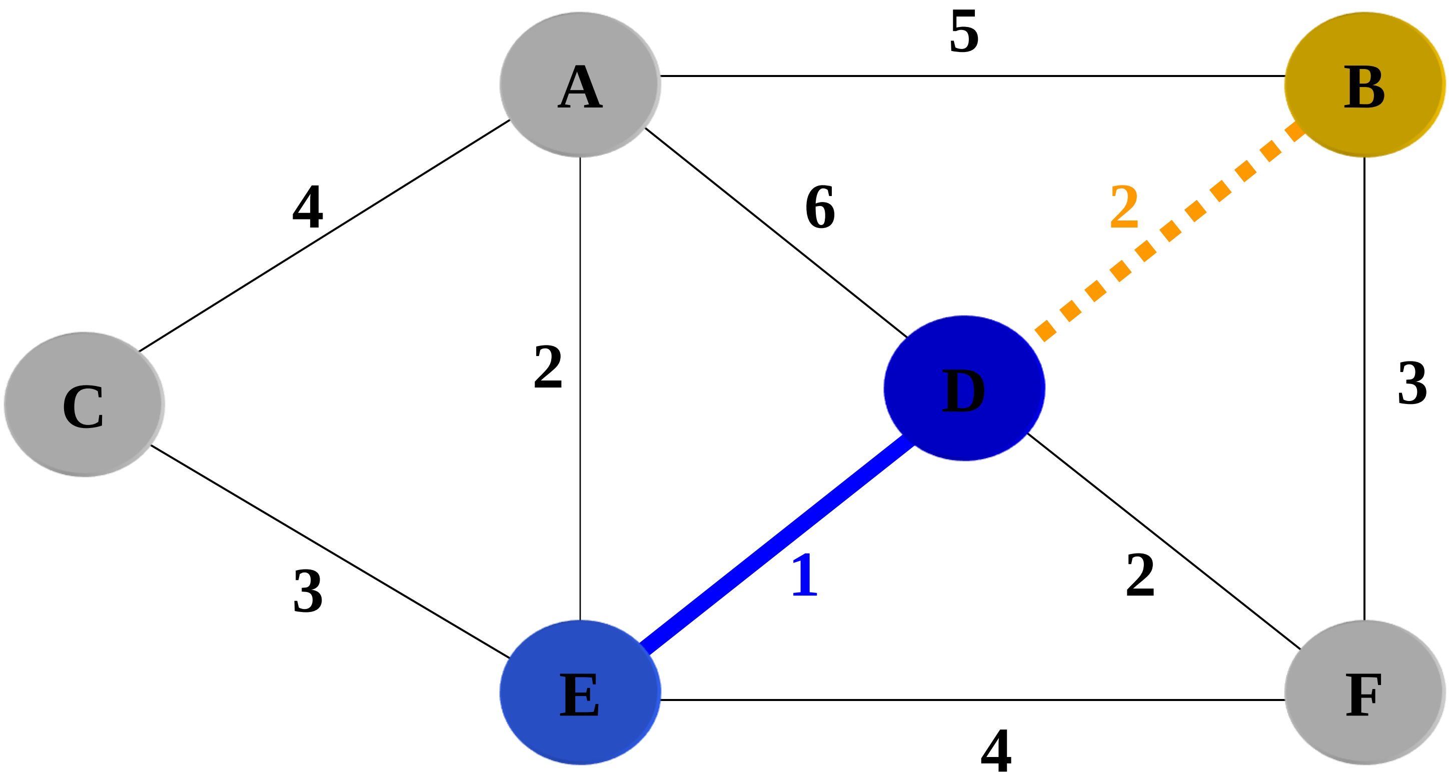
1. Each vertex is in its own cluster

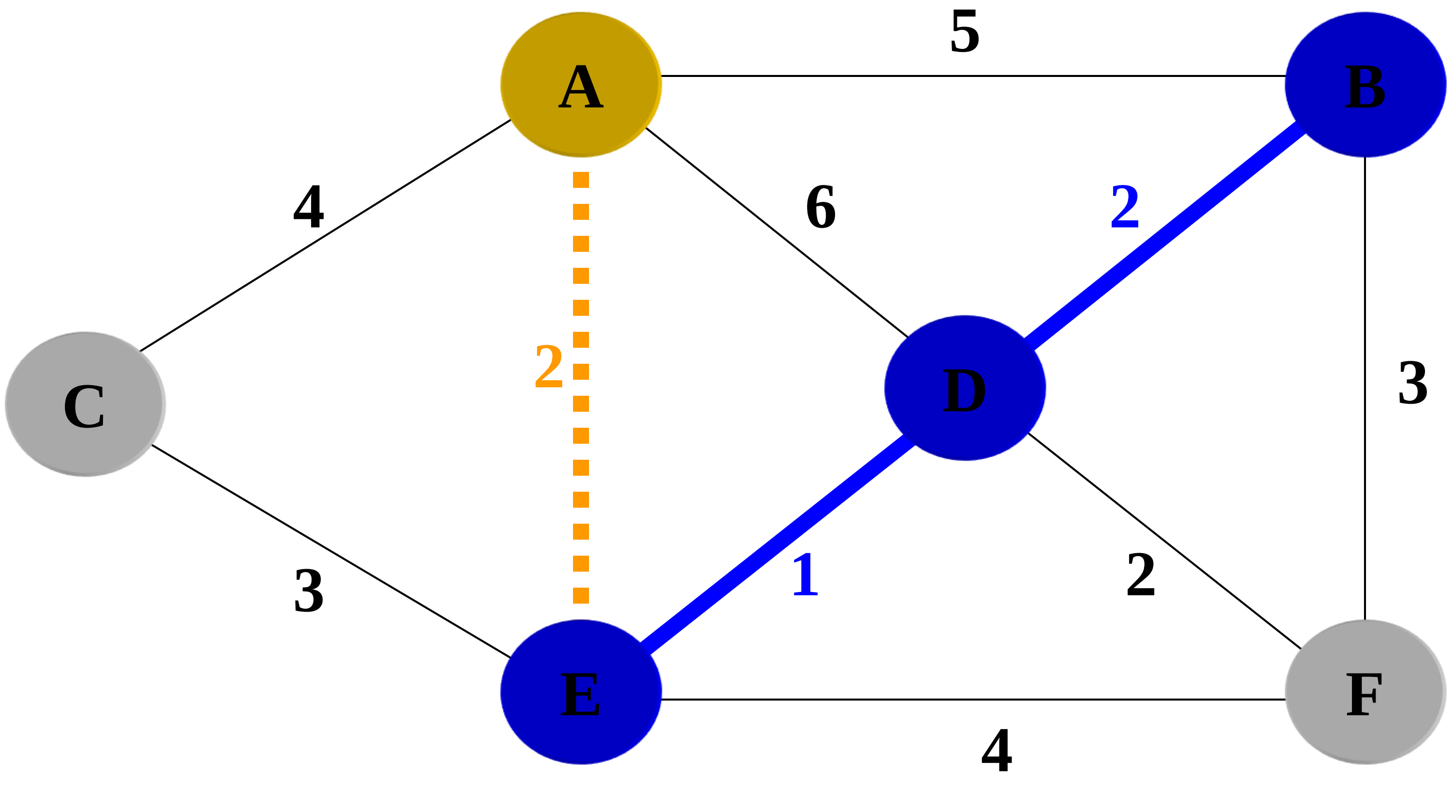
2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

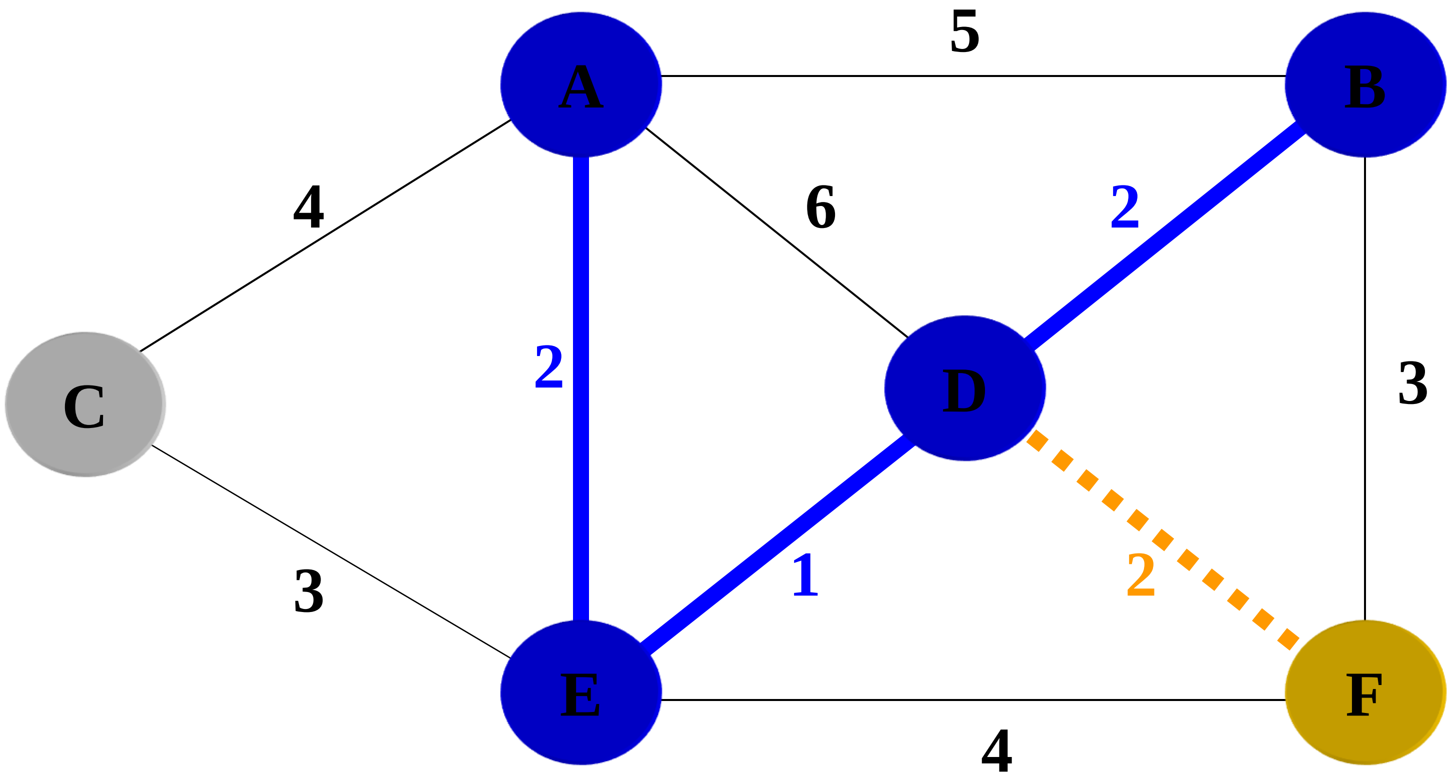
3. Continue until *n-1* edges were selected

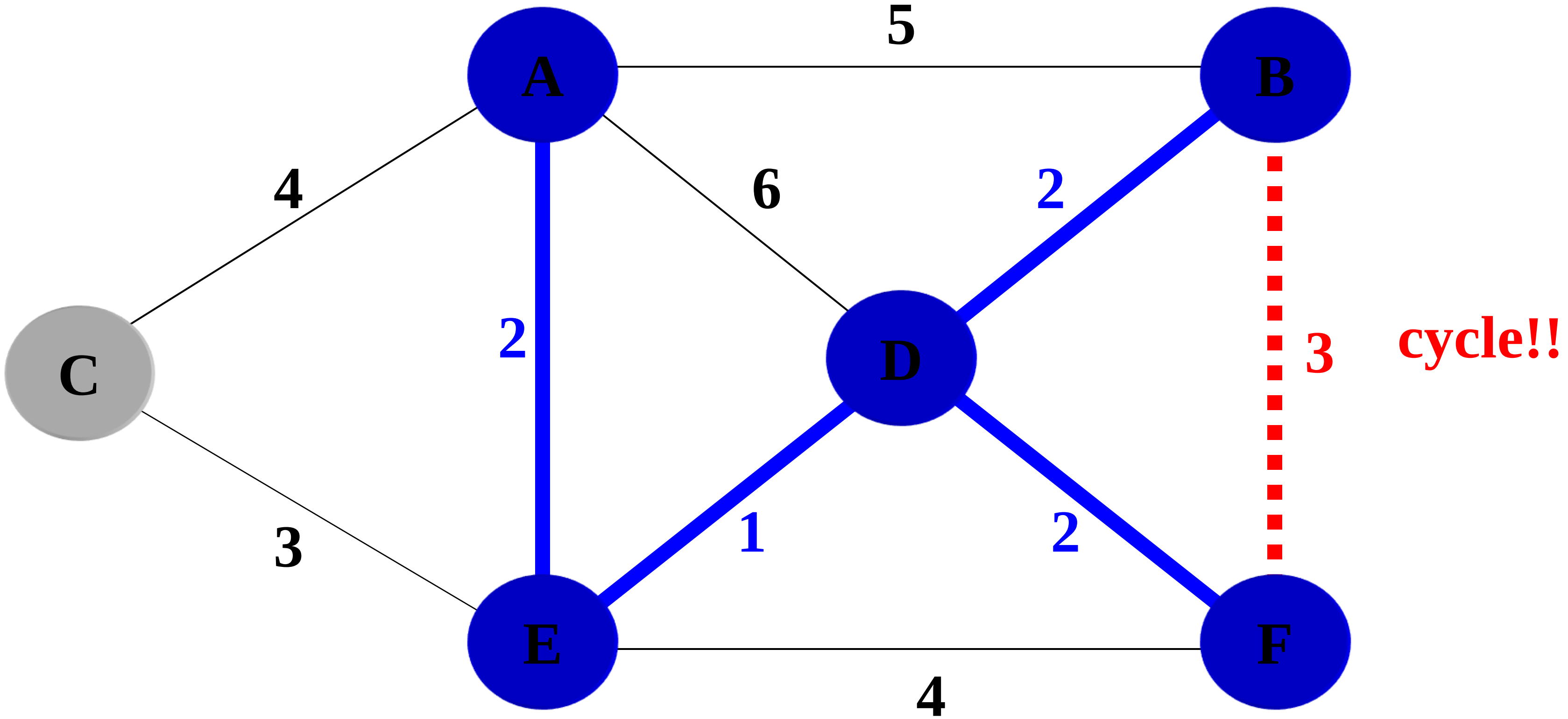

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

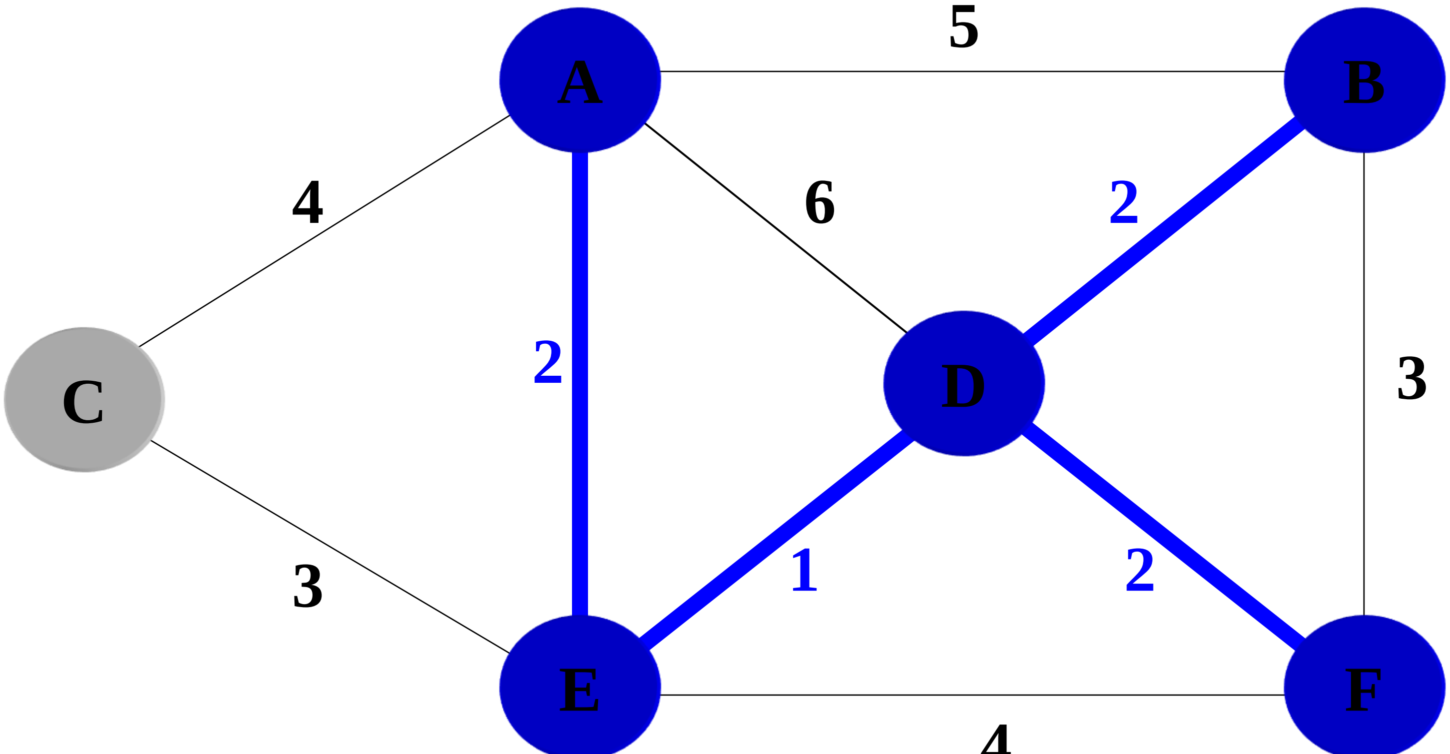

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

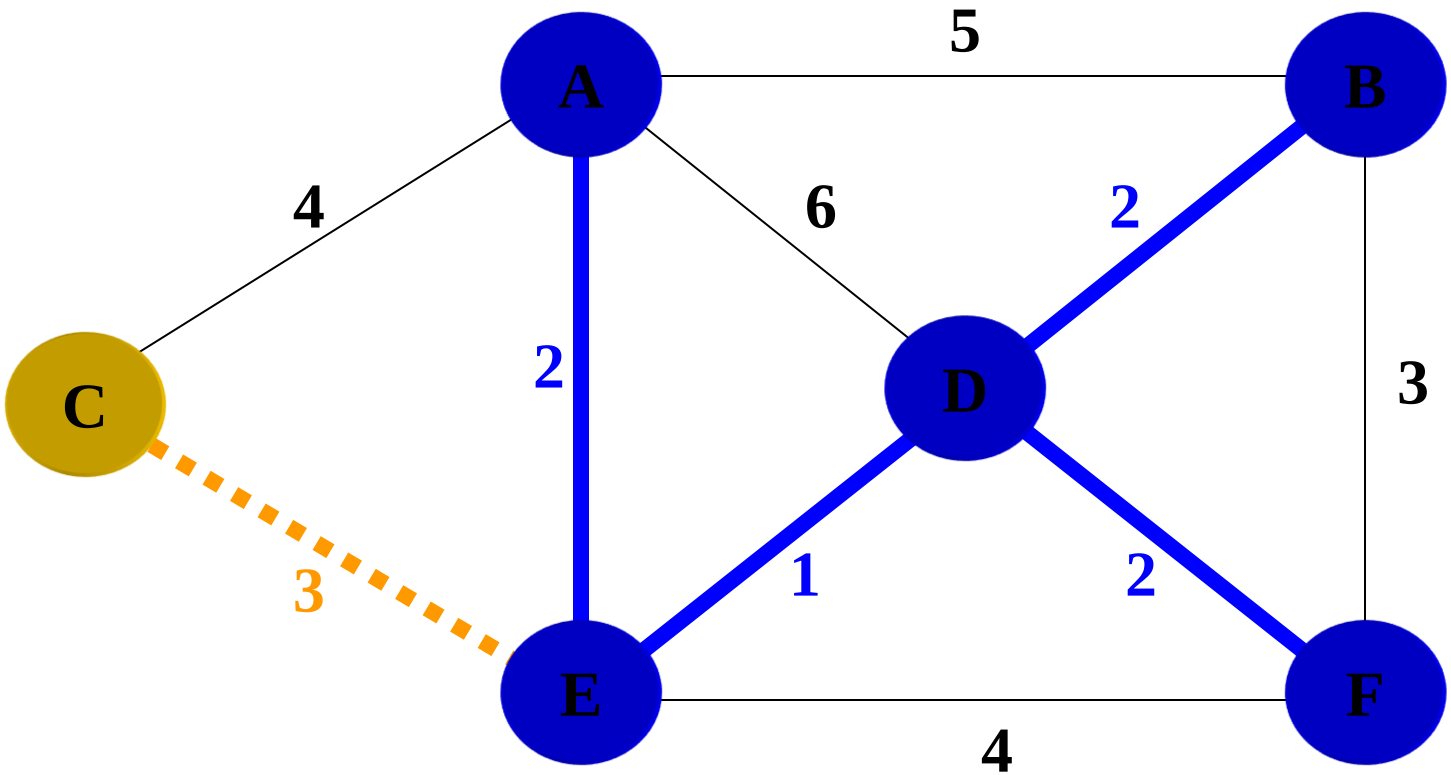

5
A
B
4
6
2
2
C
D
3
3
1
2
E
F
4


A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4


5
A
B
4
6
2
2
C
D
3
1
2
3
E
F
4

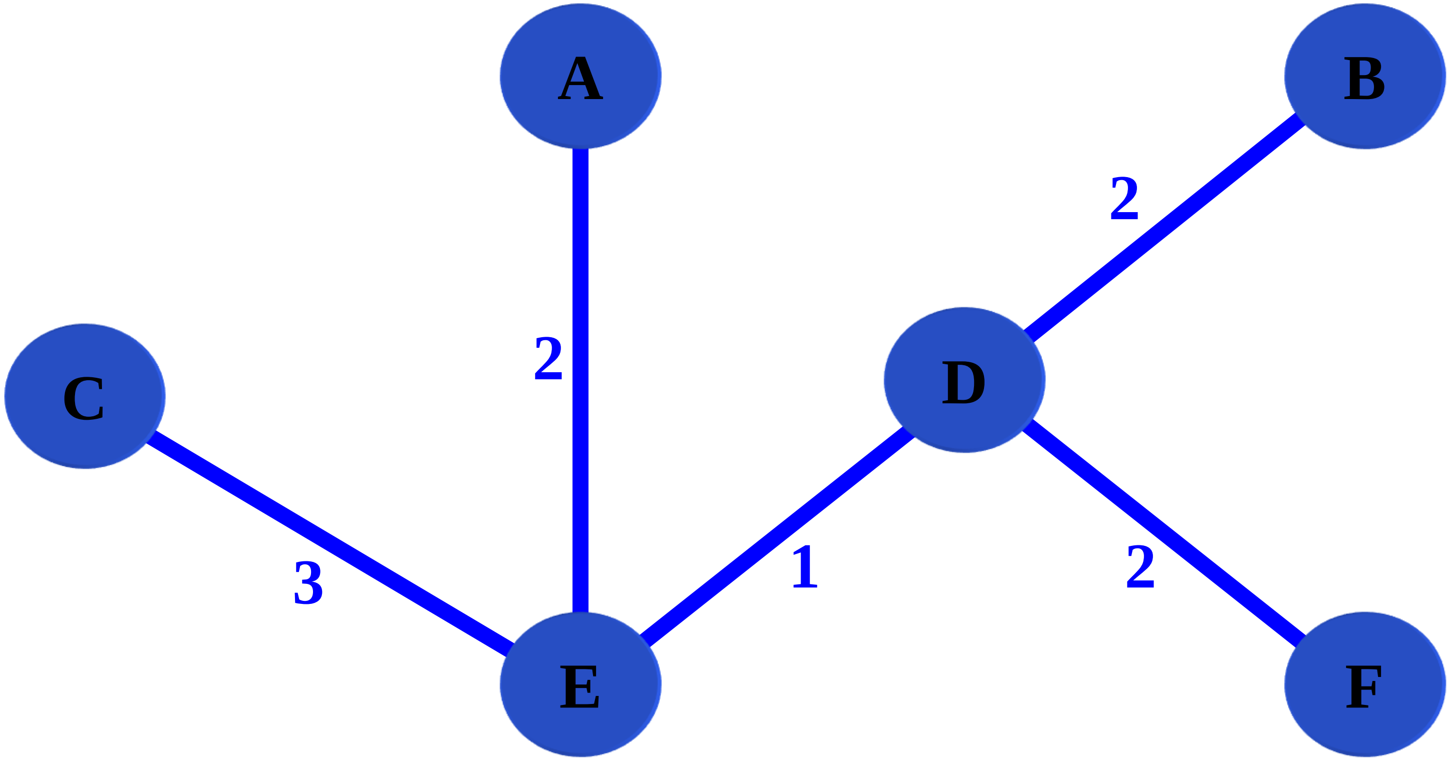

5
A
B
4
6
2
2
3
cycle!!
C
D
1
2
3
E
F
4


5
A
B
4
6
2
2
C
D
3
3
1
2
E
F
4



5
A
B
4
6
2
2
C
D
3
3
1
2
E
F
4

minimum- spanning tree



The correctness of Kruskal‘s Algorithm

⇨ **Crucial Fact about MSTs**

Running time: $O\ (\ m \log n\ )$

$$O\ (\ |E|\ \log |E|\ )$$

By implementing queue Q as a heap, Q could be initialized in $O\ (\ m\ )$ time and a vertex could be extracted in each iteration in $O\ (\ \log n\ )$ time

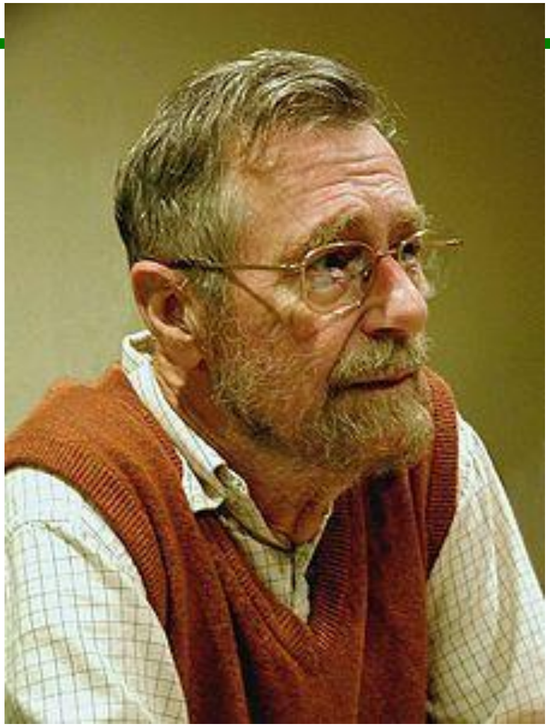
Code Fragment

Input: A weighted connected graph G with n vertices and m edges
Output: A minimum-spanning tree T for G

for each vertex v in G **do**
    Define a cluster $C(v) \leftarrow \{v\}$.
Initialize a priority queue Q to contain all edges in G, using weights as keys.
$T \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
    Extract (and remove) from Q an edge (v,u) with smallest weight.
    Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
    if $C(v) \neq C(u)$ **then**
        Add edge (v,u) to T.
        Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).
return tree T

Dijkstra's algorithm

The author: Edsger Wybe Dijkstra



"Computer Science is no more about computers than astronomy is about telescopes."

http://www.cs.utexas.edu/~EWD/

Edsger Wybe Dijkstra

-May 11, 1930 – August 6, 2002

 - Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science.

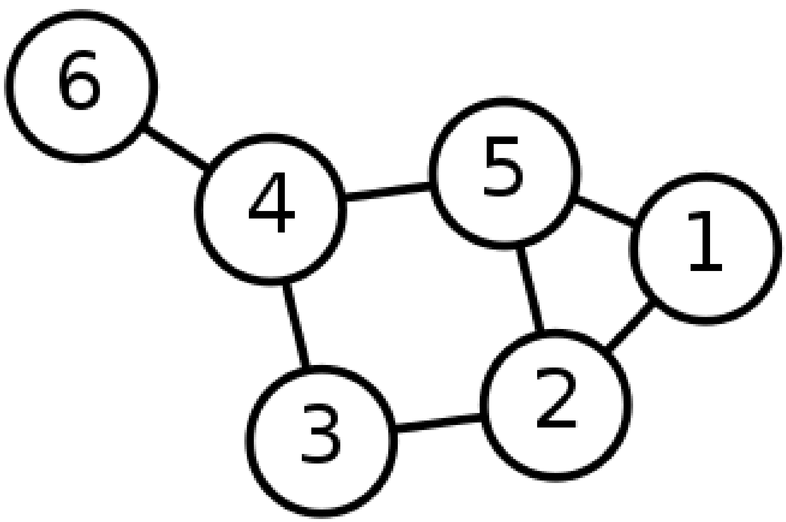
-The Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000

- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.

- Known for his many essays on programming.

Single-Source Shortest Path Problem

<u>Single-Source Shortest Path Problem</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

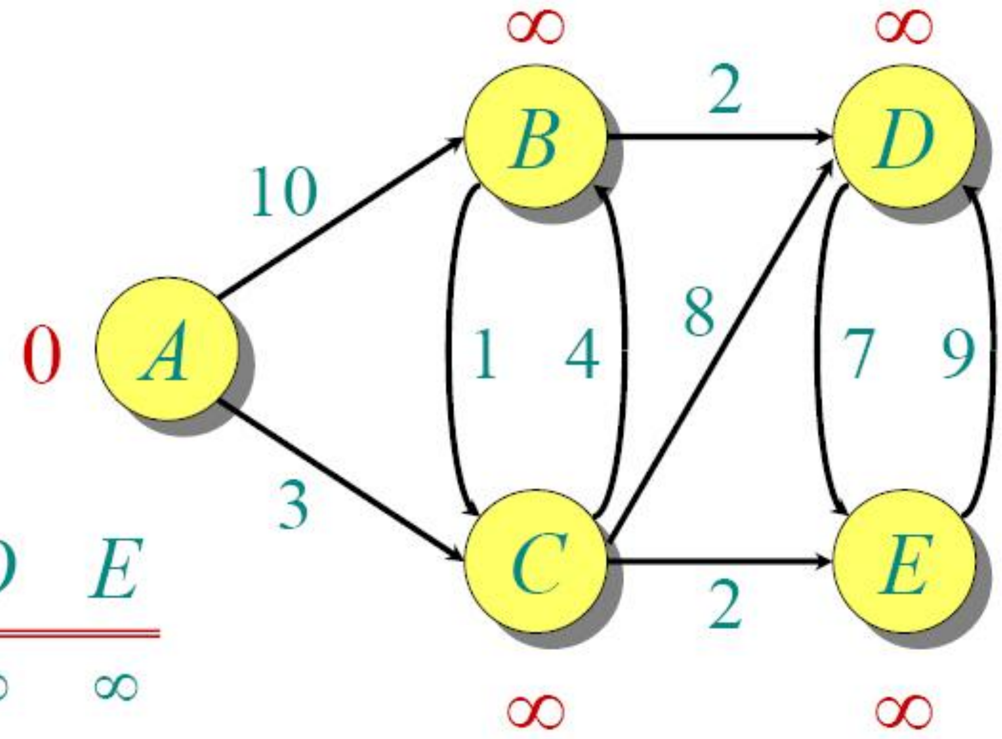
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's algorithm - Pseudocode

```
dist[s] ←o                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                        (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all
vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min.
distance)
     S←S∪{u}                                (add u to list of visited vertices)
      for all v ∈ neighbors[u]
           do  if   dist[v] > dist[u] + w(u, v)            (if new shortest path found)
                    then      d[v] ←d[u] + w(u, v)     (set new value of shortest path)
                         (if desired, add traceback code)
return dist
```

Dijkstra Animated Example

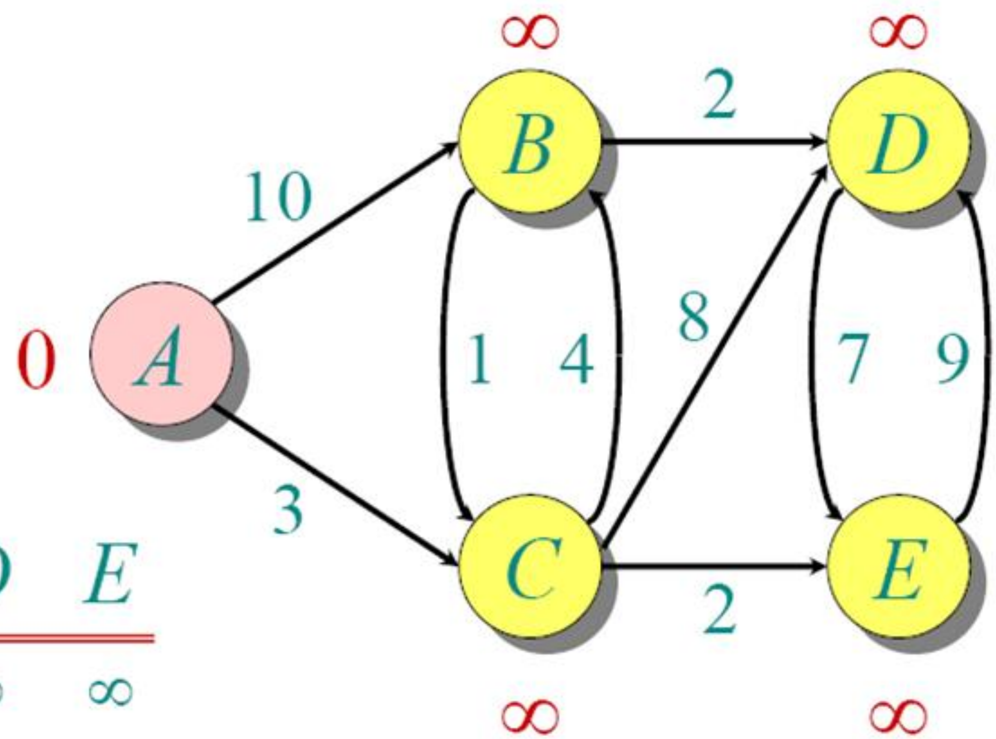
- **Initialize:**



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

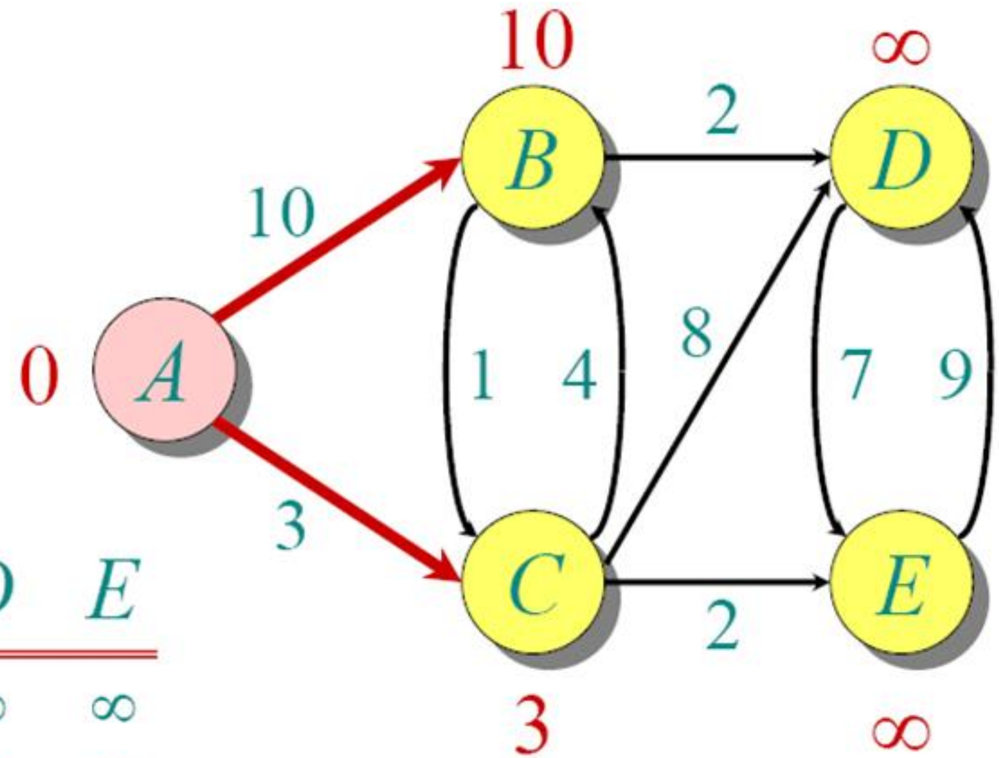
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

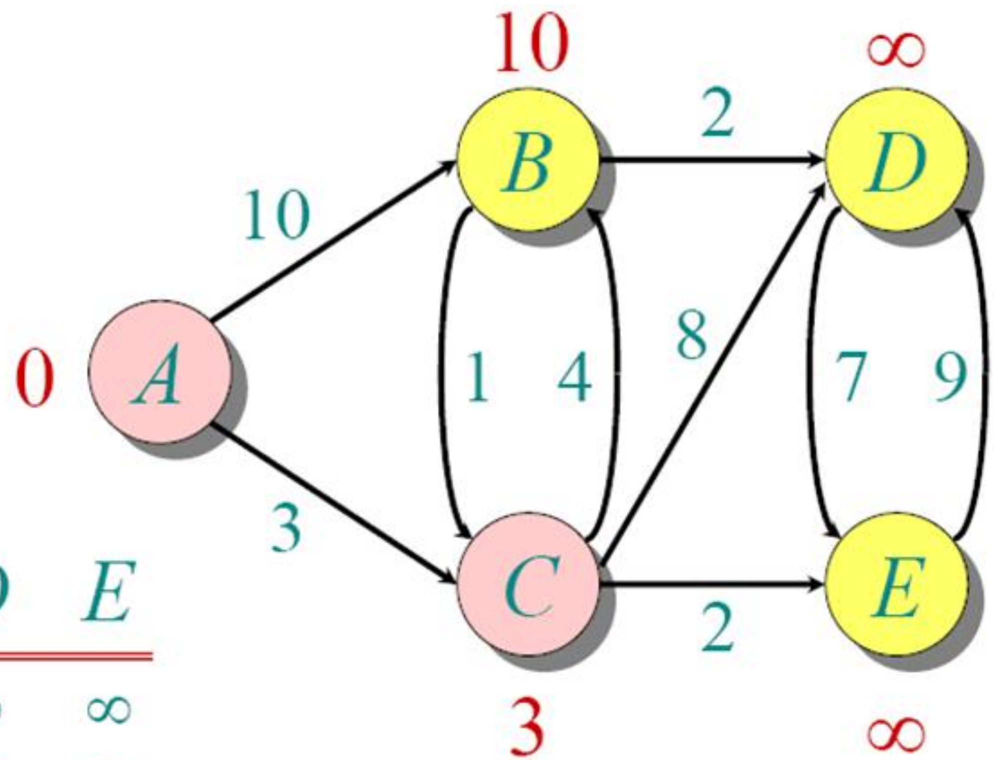
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Dijkstra Animated Example



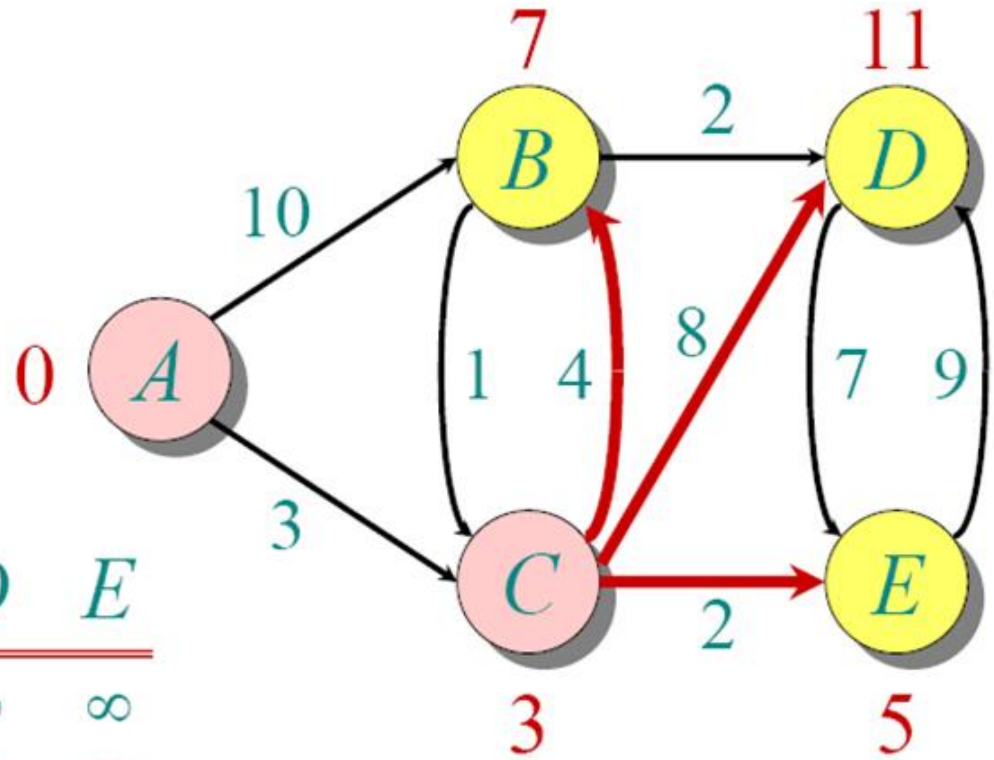
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

Dijkstra Animated Example

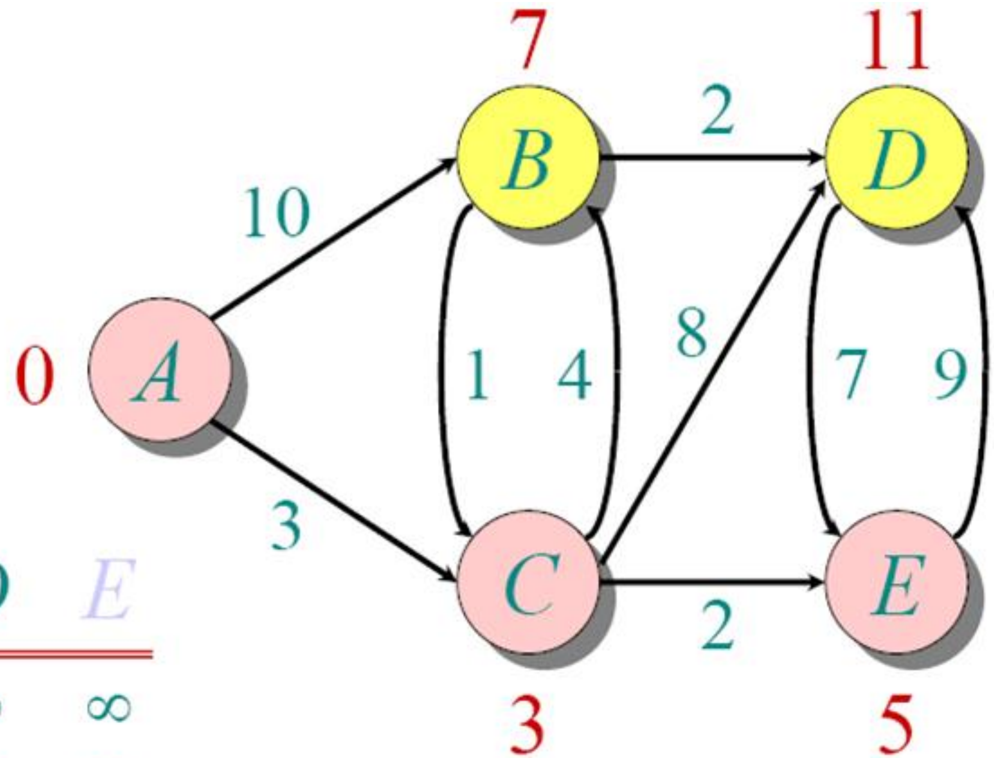
-

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5



S: { A, C }

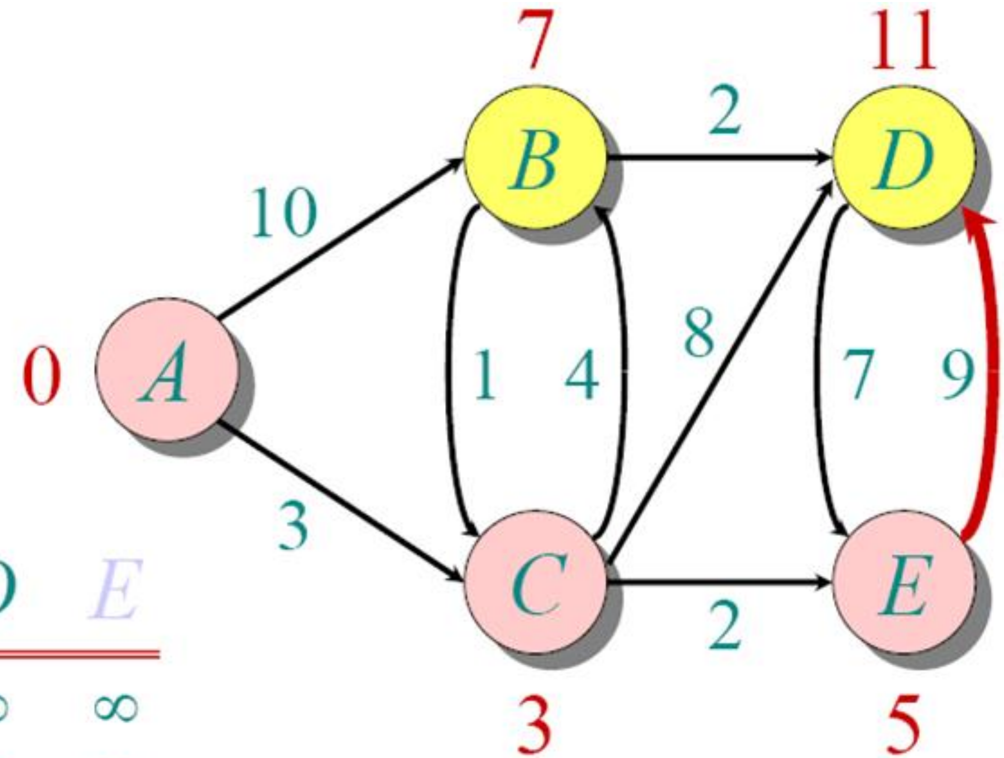
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

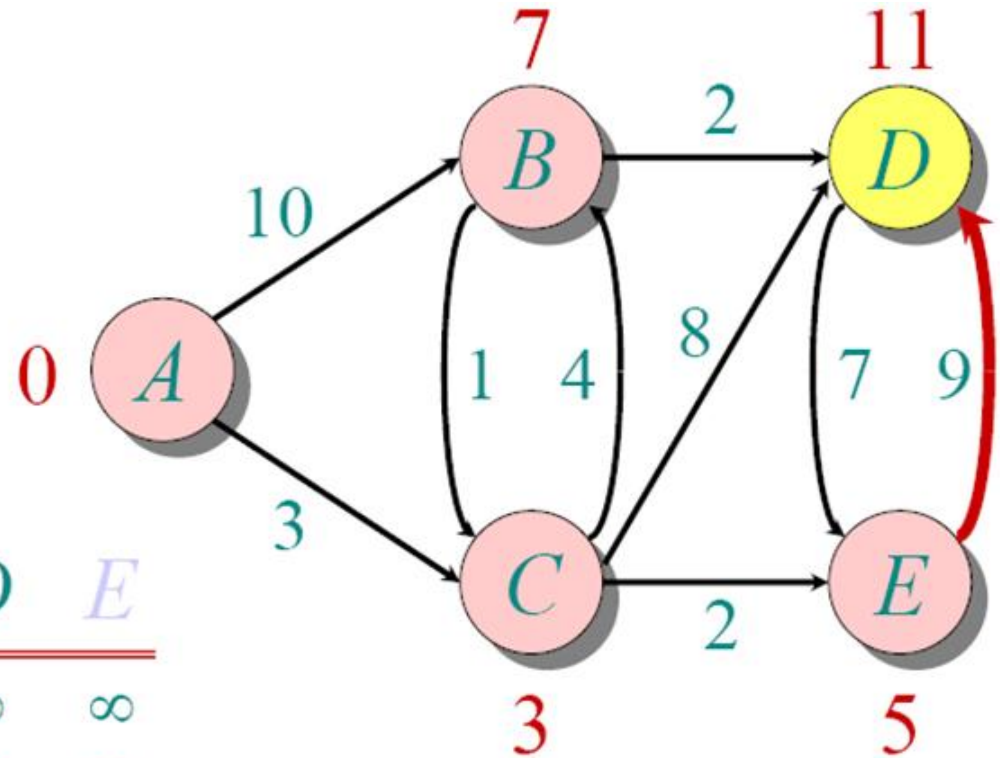
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

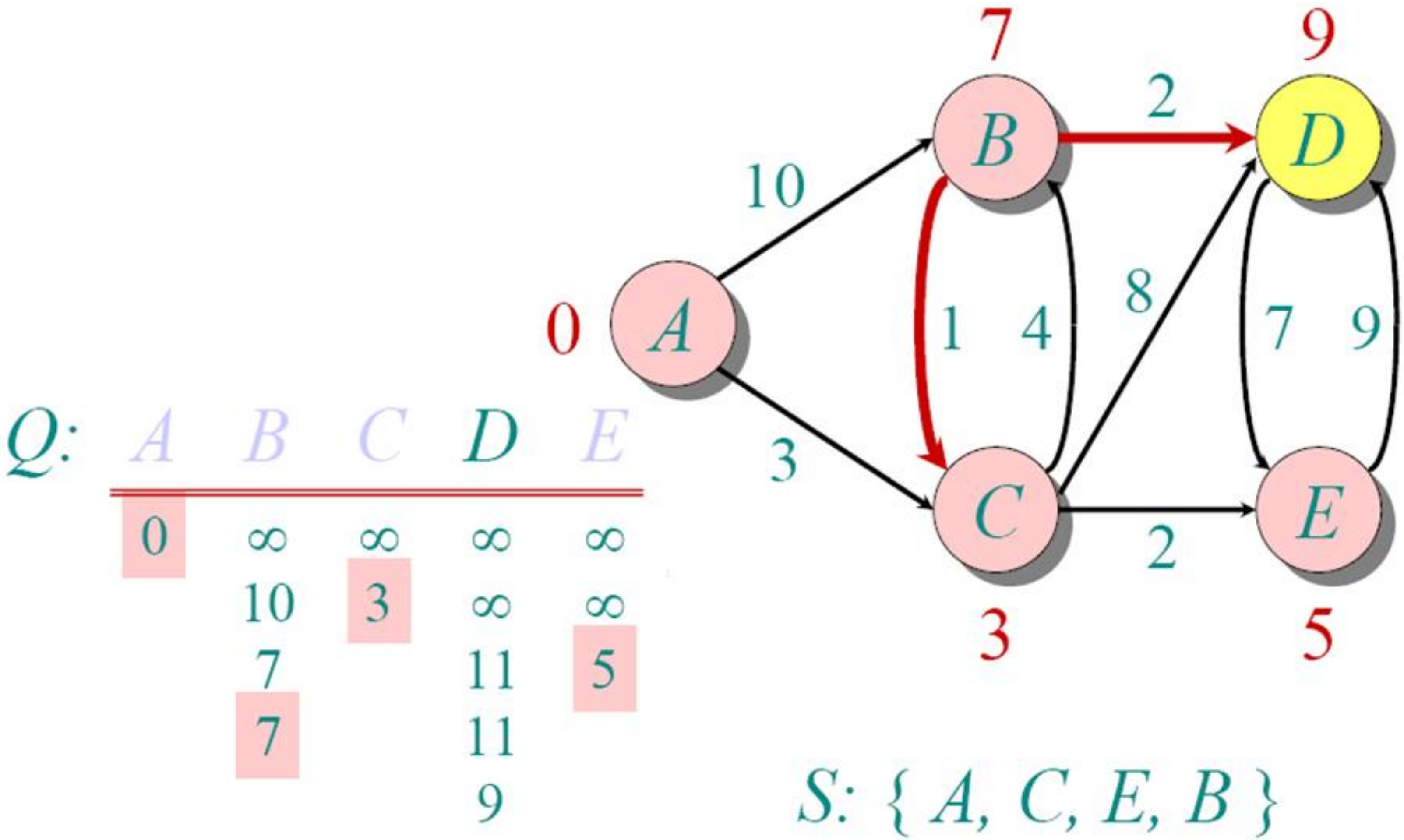
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

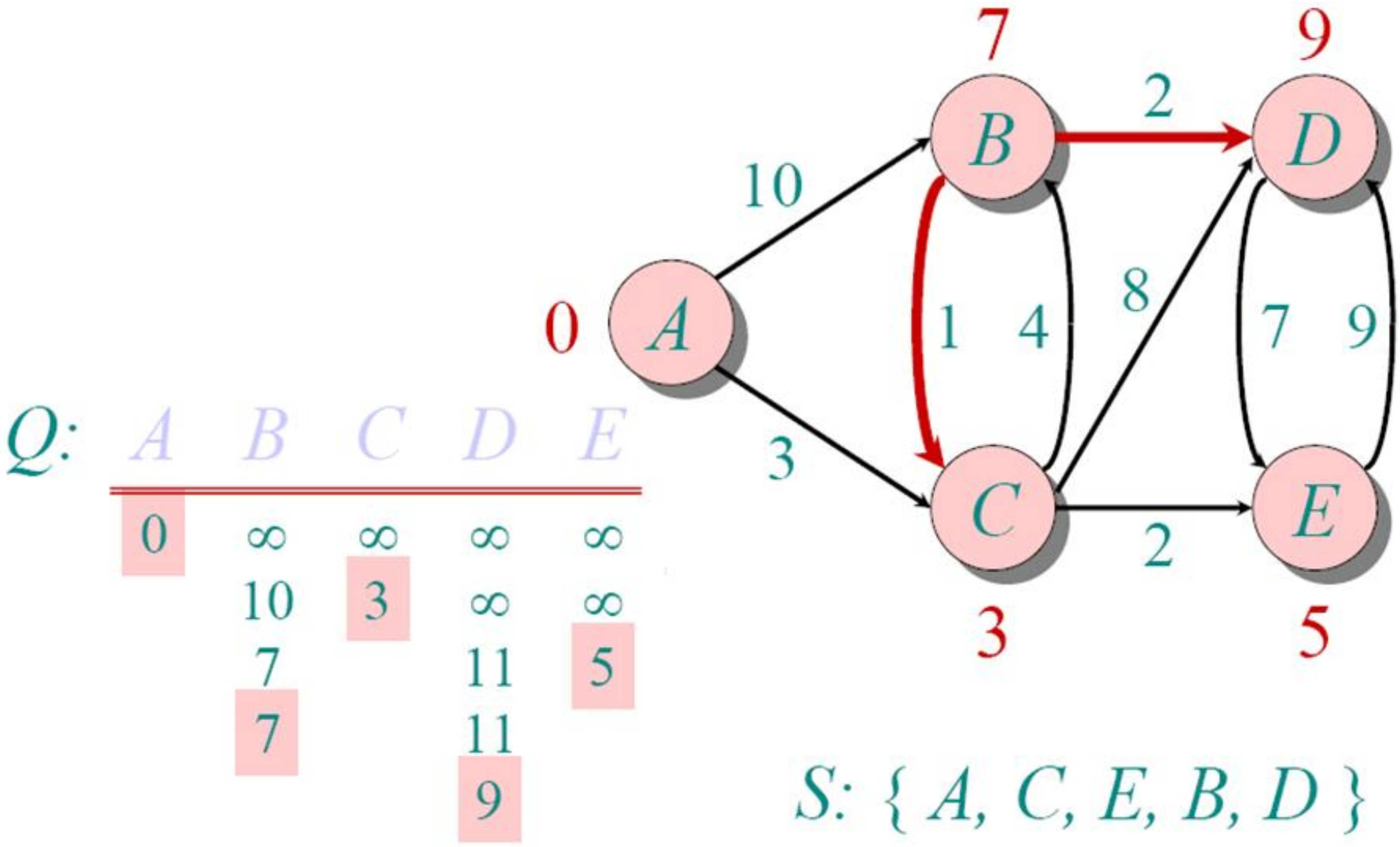
S: { *A*, *C*, *E*, *B* }

Dijkstra Animated Example



Dijkstra Animated Example

-



Implementations & Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

O(|V|^2 + |E|)

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can’t sleep unless you see a proof, see the second reference or ask us where you can find it.

Dijkstra's Algorithm - Why It Works

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

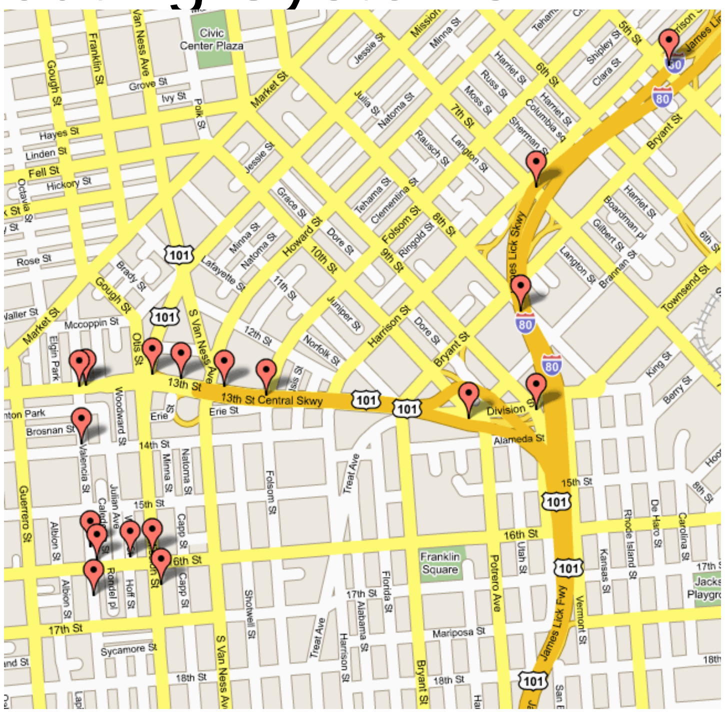
- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

Dijkstra's Algorithm - Why use it?

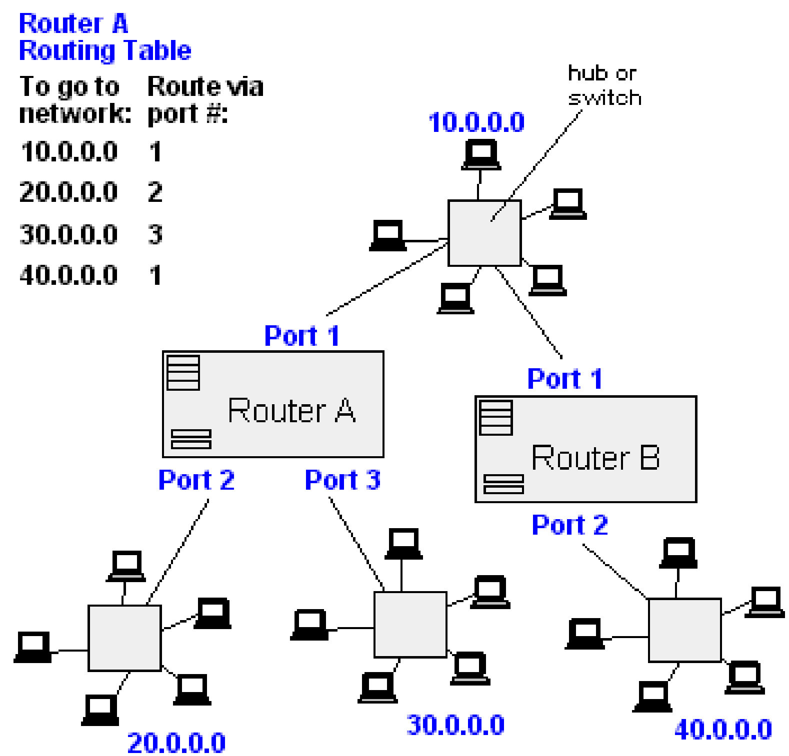
- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



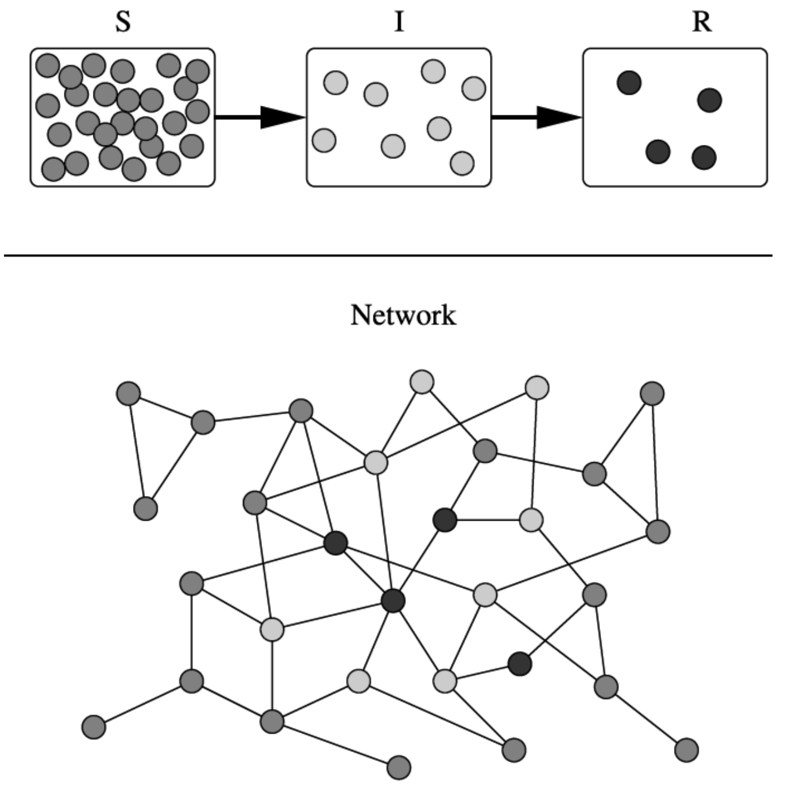
Applications of Dijkstra's Algorithm

- One particularly relevant this week: epidemiology

- Prof. Lauren Meyers (Biology Dept.) uses networks to model the spread of infectious diseases and design prevention and response strategies.

- Vertices represent individuals, and edges their possible contacts. It is useful to calculate how a particular individual is connected to others.

- Knowing the shortest path lengths to other individuals can be a relevant indicator of the potential of a particular individual to infect others.



Graph Implementation

Create_graph using Adjacency Matrix

```
int create_graph(int G[15][15])
{
  Accept the no. of vertices  and edges  as n & e
  for(i=0;i<e;i++)
  {
    printf("Enter the adjacent vertices : ");
    scanf("%d%d",&vi,&vj);
    G[vi][vj] = 1;
    G[vj][vi] = 1;
  }
  return(n);
}
```

Display_graph using Adjacency Matrix

```
void display_graph(int G[15][15],int n)
{
   int i,j;
   printf("\nAdjacency Matrix : \n");
   printf("\n      ");
   for(i=1;i<=n;i++)
     printf("V%d  ",i);

   for(i=1;i<=n;i++)
   {
      printf("\nV%d    ",i);
      for(j=1;j<= n;j++)
      {
       printf("%d   ",G[i][j]);
      }
   }
}
```

Graph using Adjacency list

```
struct adj_node
{
   int vertext;
   struct adj_node *next;
};

Struct adj_node *G[MAX];
int n;
```

Create_graph using Adjacency List

```
int create_graph(struct adj_node
     *G[])
{
  Accept the no. of vertices  and edges
     as n & e
  for(i=0;i<e;i++)
  {
    printf("Enter the adjacent vertices
    : ");
    scanf("%d%d",&v1,&v2);
    add_into_adj_list(G,v1,v2);
    add_into_adj_list(G,v2,v1);
  }
  return(n);
}
```

```
Void add_into_adj_list(G , v1, v2)
{
    node = getnode(v2);
    if(G[v1] == NULL)
        G[v1] = node;
    else
    {
        last = G[v1];
        while(last->next != NULL)
            last = last->next;
      last->next = node;
    }
}
```

Display_graph using Adjacency List

```
void display_graph(struct adj_node  *G[], int n)
{
  Hnode *temp;
  Anode *node;
  printf("\nAdjacency List : \n\n");
  for(i = 1; i<=n; i++)
  {
    printf("\n\tV%d ==>  “,i);
    for(node = G[i]; node != NULL; node = node->next)
    {
          printf(“V%d --> ",node->ver);
    }
    printf("NULL");
  }
}
```

BFS Traversal

```
void bfs(int G[15][15],int n)
{
    Accept the starting vertex as sv
    Initialize visit array  to    0
     printf("\nBFS traversal is : ");
     printf("%d  ",sv);
     visit[sv] = 1;
     enqueue(sv);
     while(Queue is not empty)
     {
         v = dequeue();
         for(w=1;w<=n;w++)
         {
             if(G[v][w]== 1 && visit[w] == 0)
             {
                 printf("%d  ",w);
                 visit[w] = 1;
                 enqueue(w);
             }
         }
     }
}
```

BFS Traversal using adj_list

```
void bfs(struct adj_list *G[],int n)
{
    Accept the starting vertex as sv
    Initialize visit array  to    0
     printf("\nBFS traversal is : ");
     printf("%d  “,sv);
     visit[sv] = 1;
     enqueue(sv);
     while(Queue is not empty)
     {
          v = dequeue();
          for(temp = G[v]; temp!= NULL; temp=temp->next)
         {
              w = temp->vertex;
               if(visit[w] == 0)
              {
                     printf("%d  “,w);
                     visit[w] = 1;
                     enqueue(w);
              }
         }
     }
}
```

DFS Traversal (Recursive)

```
void dfs_rec(int G[15][15],int n,int Visit[],int v)
{
  int w;
  printf("%d  ",v);
  Visit[v] = 1;
  for(w = 1; w<= n;w++)
  {
     if(G[v][w] == 1 && Visit[w] != 1)
          dfs_rec(G,n,Visit,w);
  }
}
```

DFS Traversal (Recursive) using list

```
void dfs_rec(struct adj_list  *G[],int n, int Visit[],int v)
{
  int w;
  printf("%d  ",v);
  Visit[v] = 1;
  for(temp = G[v]; temp != NULL; temp = temp->next)
  {
     w = temp->vertex;
    if(Visit[w] != 1)
        dfs_rec(G,n,Visit,w);
  }
}
```

DFS Traversal (Non rec)

```
void  dfs(int G[15][15],int n)
{
    Accept the starting vertex as sv
    Initialize visit array  to    0
    printf("\nDFS traversal is : %d  ",sv);
     Push(sv);
     printf("%d ", sv);
     Visit[sv] = 1;
      While (top != -1)
    {
          v = stack[top];
          w = find_unvisited_adjacentnodes_to_v(G,n,v,Visit);
          if (w == -1)
               pop();   // backtrack
         else
         {
             push(w);
              printf("%d ", w);
             Visit[w] = 1;
         }
    }
}
```

```
Int find_unvisited_adjacentnodes_to_v(G,n,v,Visit)
{
      for(i=1;i<=n;i++)
      {
          if(G[v][i] == 1 && Visit[i] == 0)
              return(i);
      }
       return(-1);
}
```

DFS Traversal (Non rec using list)

```
void  dfs(struct adj_list *G[],int n)
{
    Accept the starting vertex as sv
    Initialize visit array  to    0
    printf("\nDFS traversal is : %d  ",sv);
     Push(sv);
     printf("%d ", sv);
     Visit[sv] = 1;
      While (top != -1)
    {
          v = stack[top];
          w = find_unvisited_adjacentnodes_to_v(G,v,Visit);
          if (w == -1)
               pop();   // backtrack
         else
         {
             push(w);
             printf("%d ", w);
            Visit[w] = 1;
         }
    }
}
```

```
Int find_unvisited_adjacentnodes_to_v(G,v,Visit)
{
        for(temp = G[v]; temp != NULL; temp = temp->next)
        {
            if(Visit[temp->vertext] == 0)
                return(temp->vertex);
        }
        return(-1);
}
```

Graph using Adjacency list

```
struct adjacent_node
{
   char ver;
   struct adjacent_node *next;
};

struct header_node
{
  char ver;
  char tag;
  struct header_node *down;
  struct adjacent_node *next;
};

typedef struct adjacent_node Anode;
typedef struct header_node Hnode;
```

Prim's algo

```
void primsalgo(int s,int G[15][15],int n)
{
  int V[15],T[15],nt=1,i,e,j,v1,v2,min,min_cost = 0;
  Initialize V array to 0
  V[s] = 1;  T[0] = s;
  for(e=1;e<n;e++)  // Choose n-1 edges
  {
    min = Infinity;
    for(i=0;i<nt;i++)
    {
      s = T[i];
      for(j=1;j<=n;j++)
      {
        if(V[j] == 0 && min > G[s][j])
        {
          min = G[s][j];
          v1 = s,v2 = j;
        }
      }
    }
    printf("\nV%d ----- V%d : %d",v1,v2,min);
    min_cost += min;
    V[v2] = 1;     T[nt] = v2;
    nt++;
  }
  printf("\nMinimum Cost = %d",min_cost);
}
```

Kruskal's Algo

```
void kruskals(int G[15][15],int n)
{
 for(i=1;i<=n;i++)
   Set[i] = i;
 for(i=1;i<=n;i++)
 {
   for(j=i;j<=n;j++)
   {
    if(G[i][j] != Infinity)
      Insert_Min_Heap(i,j,G[i][j]);
   }
 }
 e=0;
 while(e<n-1 && hs != 0)
 {
   h = Extract_Min_Heap();
   j = Set[h.vi];    k = Set[h.vj];
   if(j != k)
   {
    e++;
    printf("\nV%d ----- V%d : %d",h.vi,h.vj,h.wt);
    cost_mst += h.wt;
    Union_set(Set,n,j,k);
   }
 }
 if(e!=n-1)
   printf("\nNo Spanning Tree");
 else
   printf("\n MST wt = %d",cost_mst);
}
```

```
void Union_set(int S[],int n,int j,int k)
{
 int i;
 for(i=1;i<=n;i++)
 {
   if(S[i] == k)
     S[i] = j;
 }
}
```

Topological Sort

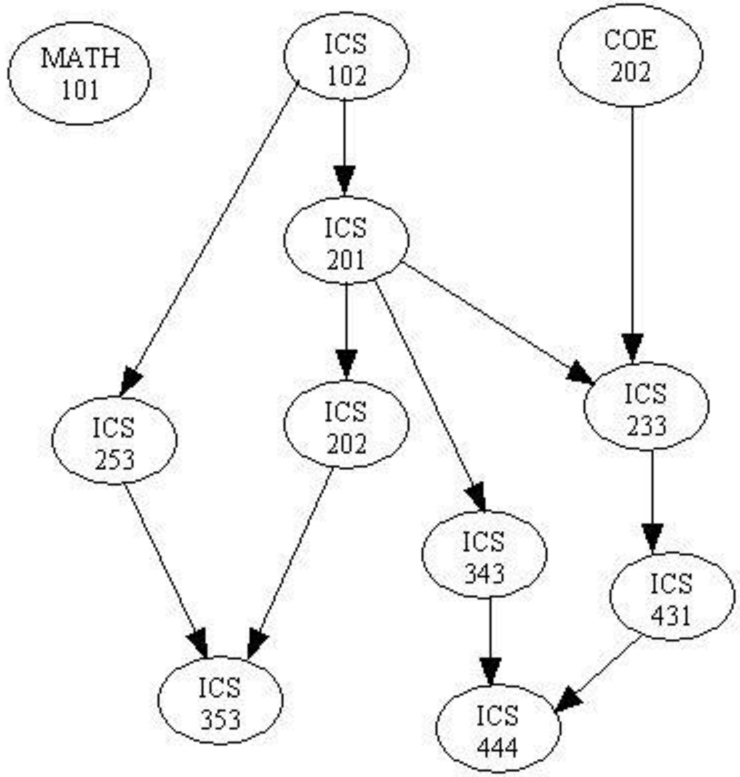
- Introduction.
- Definition of Topological Sort.
- Topological Sort is Not Unique.
- Topological Sort Algorithm.
- An Example.
- Implementation.
- Review Questions.

Introduction

- There are many problems involving a set of tasks in which some of the tasks must be done before others.

- For example, consider the problem of taking a course only after taking its prerequisites.

- Is there any systematic way of linearly arranging the courses in the order that they should be taken?

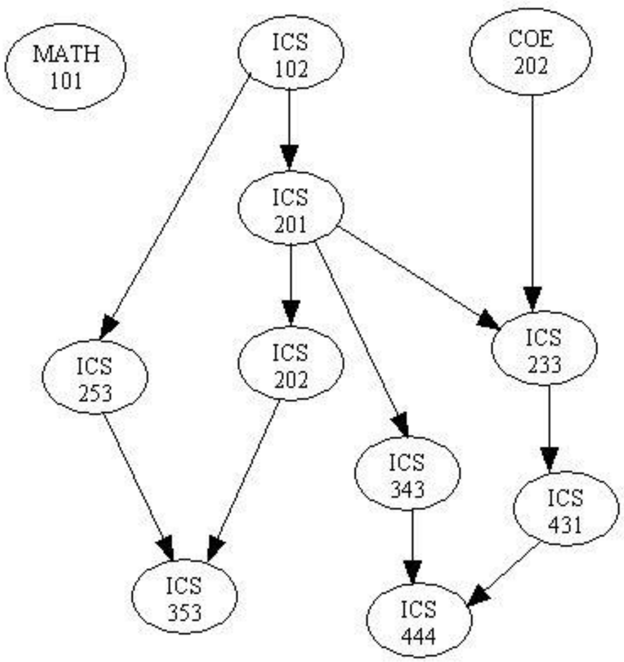


Yes! - Topological sort.

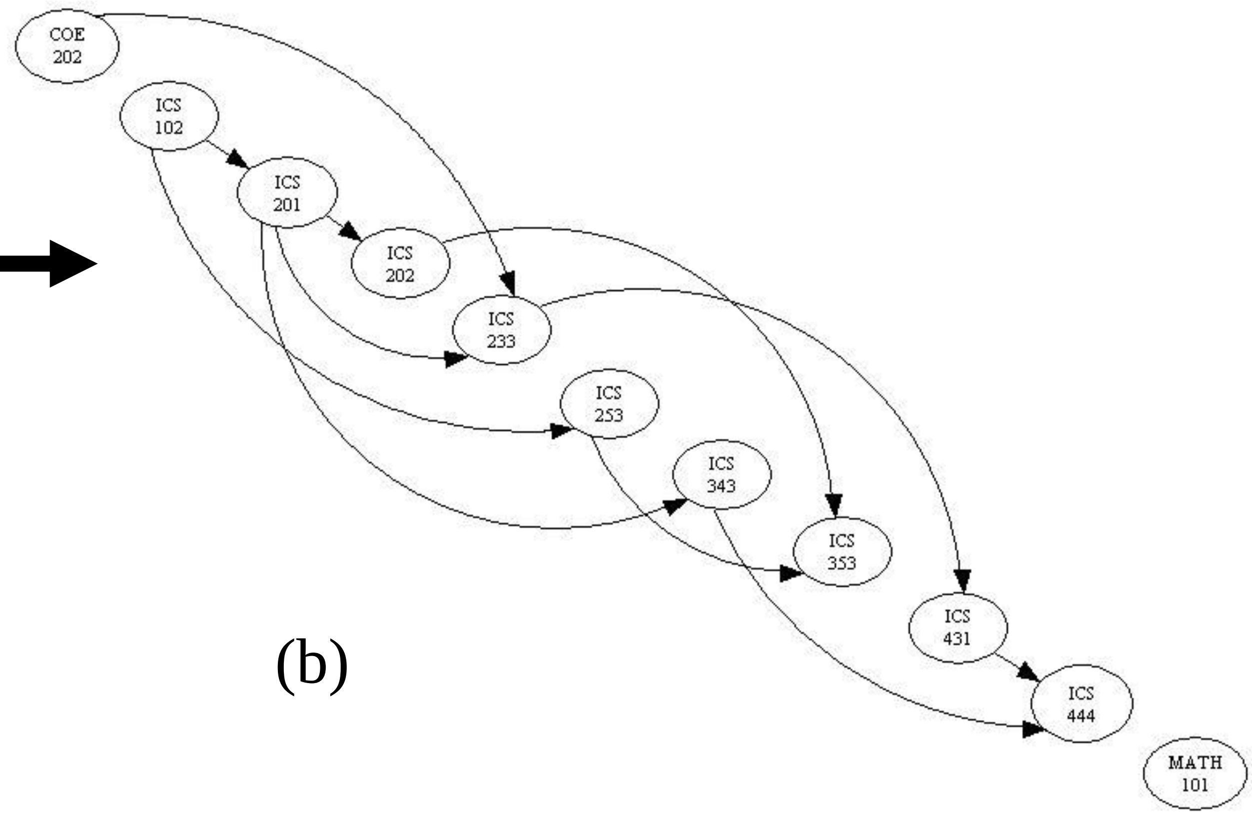
Definition of Topological Sort

- Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appear in the sequence before its predecessor.

- The graph in (a) can be topologically sorted as in (b)



(a)

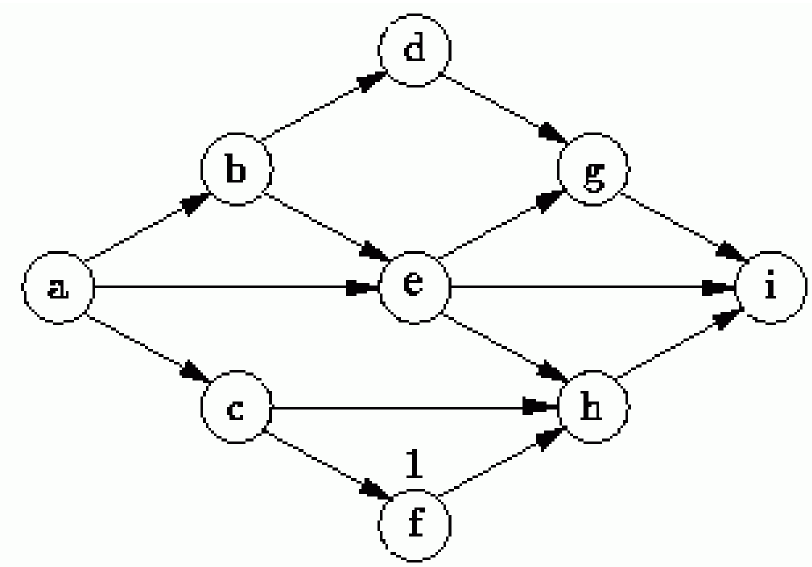


(b)

Topological Sort is not unique

- Topological sort is not unique.

- The following are all topological sort of the graph below:



s1 = {a, b, c, d, e, f, g, h, i}

s2 = {a, c, b, f, e, d, h, g, i}

s3 = {a, b, d, c, e, g, f, h, i}

s4 = {a, c, f, b, e, h, d, g, i}

etc.

Topological Sort Algorithm

- One way to find a topological sort is to consider in-degrees of the vertices.

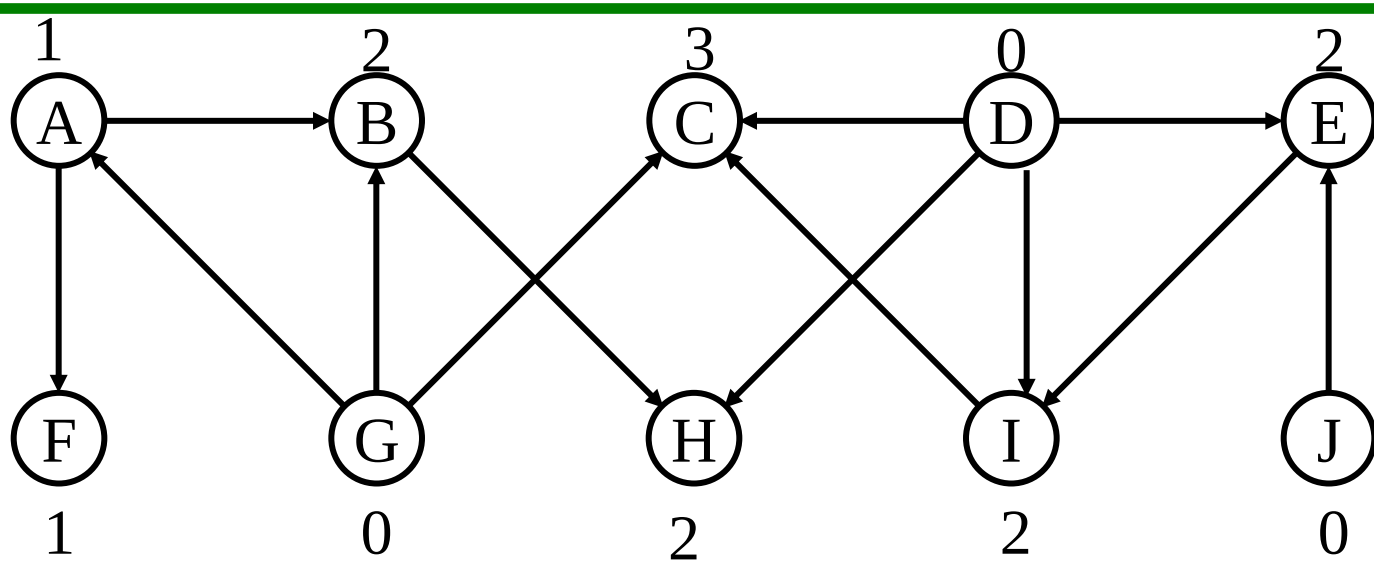
- The first vertex must have in-degree zero -- every DAG must have at least one vertex with in-degree zero.

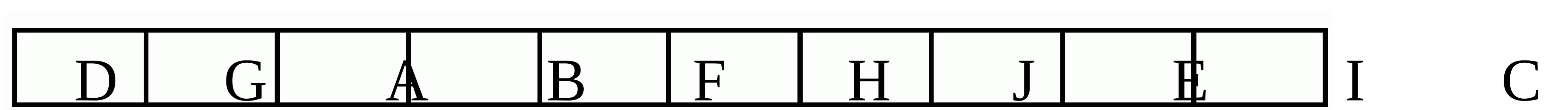
- The Topological sort algorithm is:

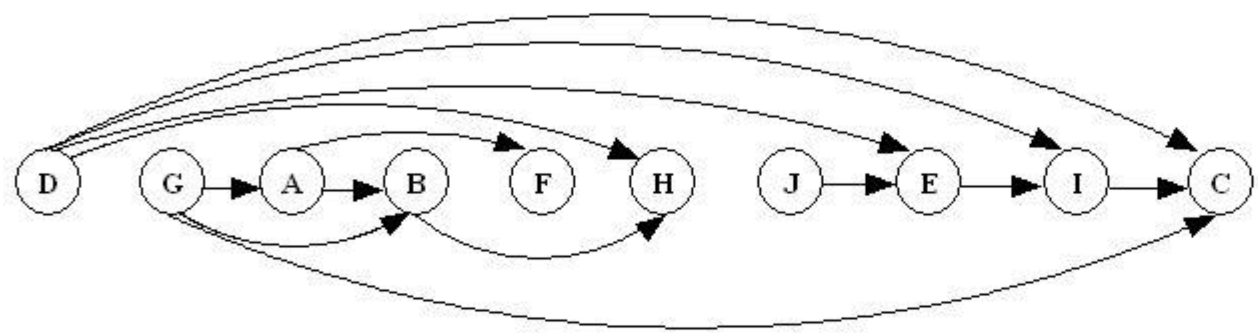
```
int topologicalOrderTraversal( ){
    int numVisitedVertices = 0;
    while(there are more vertices to be visited){
        if(there is no vertex with in-degree 0)
             break;
        else{
          select a vertex v that has in-degree 0;
          visit v;
          numVisitedVertices++;
          delete v and all its emanating edges;
        }
    }

   return numVisitedVertices;
}
```

Topological Sort Example

- **Demonstrating Topological Sort.**







Implementation of Topological Sort

- The algorithm is implemented as a traversal method that visits the vertices in a topological sort order.

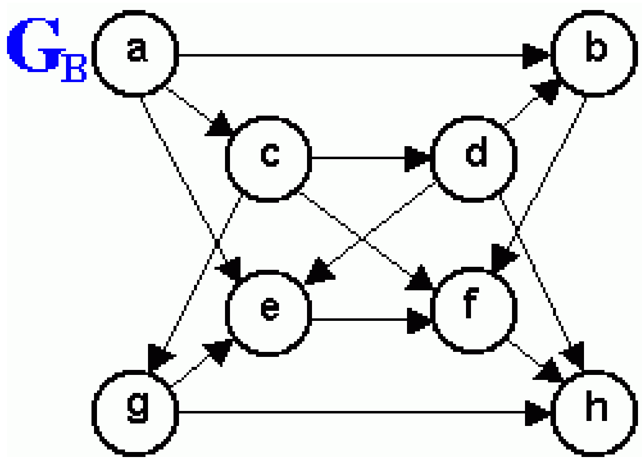
- An array of length |V| is used to record the in-degrees of the vertices. Hence no need to remove vertices or edges.
- A priority queue is used to keep track of vertices with in-degree zero that are not yet visited.

```
public int topologicalOrderTraversal(Visitor visitor){
   int numVerticesVisited = 0;
   int[] inDegree = new int[numberOfVertices];
   for(int i = 0; i < numberOfVertices; i++)
      inDegree[i] = 0;

   Iterator p = getEdges();
   while (p.hasNext()) {
      Edge edge = (Edge) p.next();
      Vertex to = edge.getToVertex();
      inDegree[getIndex(to)]++;
   }
```

Implementation of Topological Sort

```
    BinaryHeap queue = new BinaryHeap(numberOfVertices);
    p = getVertices();
    while(p.hasNext()){
       Vertex v = (Vertex)p.next();
       if(inDegree[getIndex(v)] == 0)
          queue.enqueue(v);
    }

    while(!queue.isEmpty() && !visitor.isDone()){
       Vertex v = (Vertex)queue.dequeueMin();
       visitor.visit(v);
       numVerticesVisited++;
       p = v.getSuccessors();
       while (p.hasNext()){
          Vertex to = (Vertex) p.next();
          if(--inDegree[getIndex(to)] == 0)
             queue.enqueue(to);
       }
    }
    return numVerticesVisited;
}
```

Review Questions

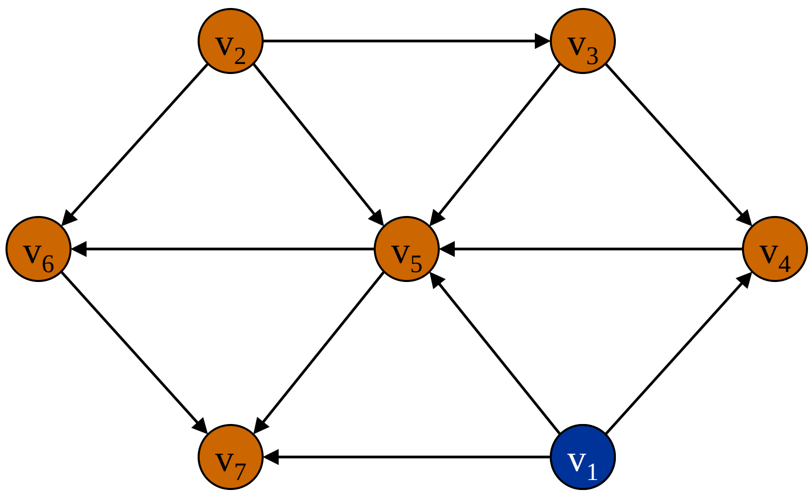


1. List the order in which the nodes of the directed graph GB are visited by topological order traversal that starts from vertex a.

2. What kind of DAG has a unique topological sort?

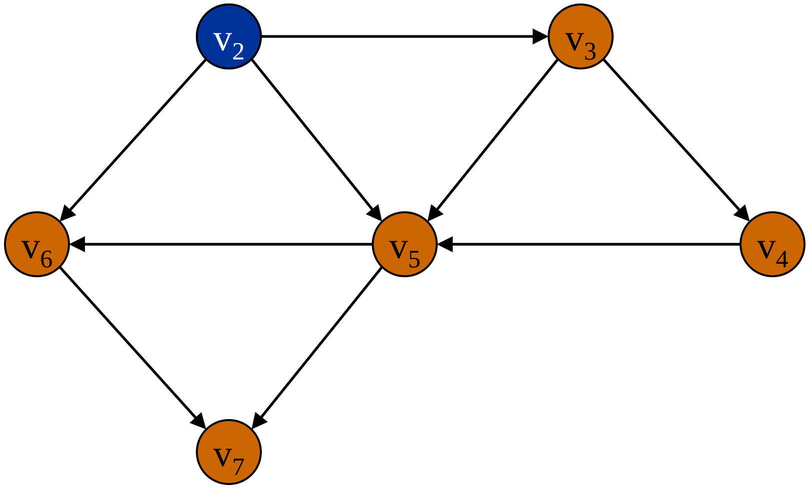
3. Generate a directed graph using the required courses for your major. Now apply topological sort on the directed graph you obtained.

Topological Ordering Algorithm: Example



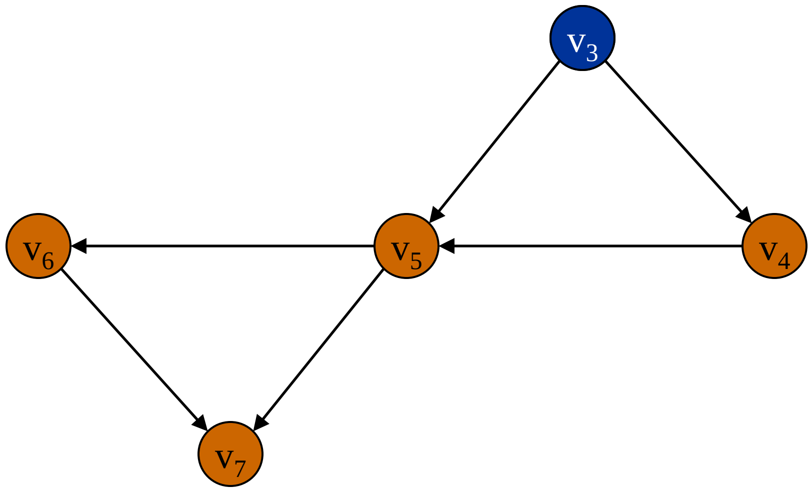
Topological order:

Topological Ordering Algorithm: Example



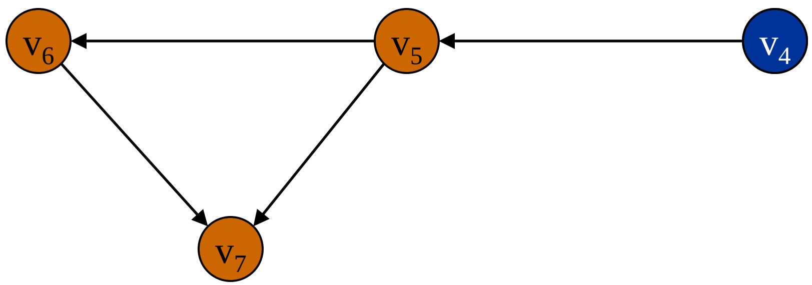
Topological order: v_1

Topological Ordering Algorithm: Example



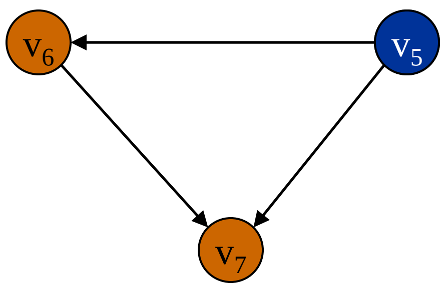
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



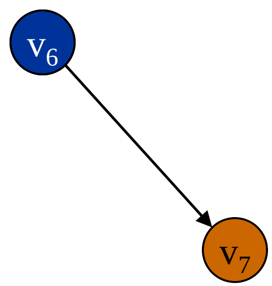
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

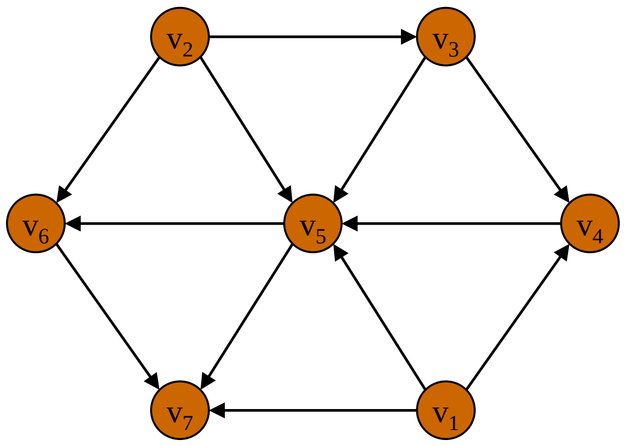


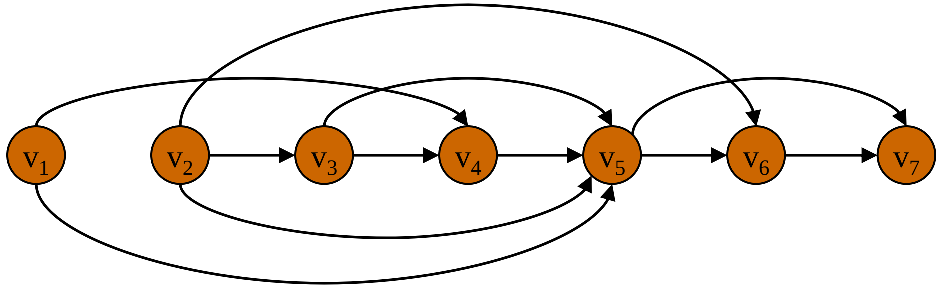
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Topological Sort Example

This job consists of 10 tasks with the following precedence rules:

Must start with 7, 5, 4 or 9.

Task 1 must follow 7.

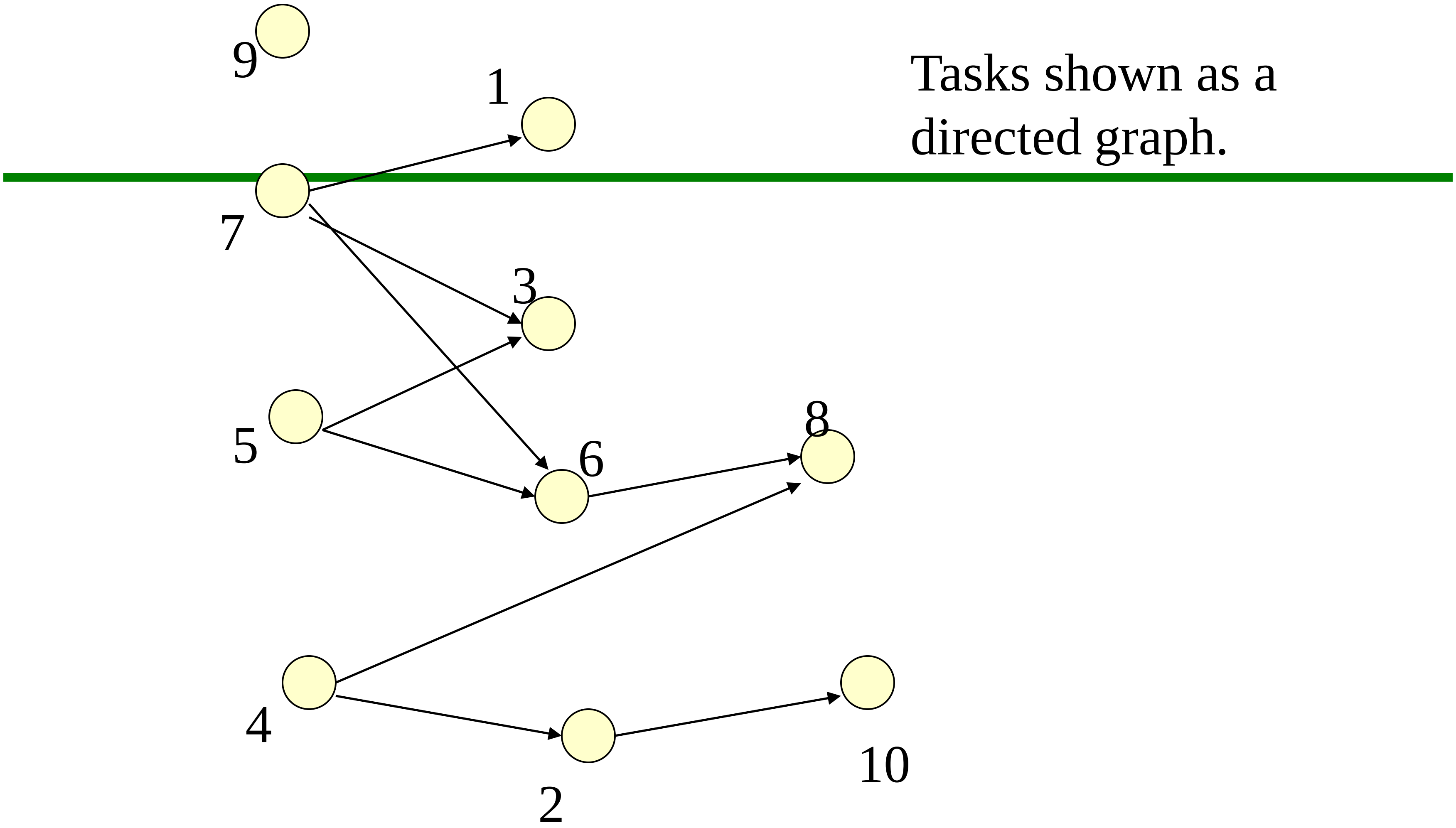
Tasks 3 & 6 must follow both 7 & 5.

8 must follow 6 & 4.

2 must follow 4.

10 must follow 2.

Make a directed graph and then a list of ordered pairs that represent these relationships.



Tasks shown as a directed graph.

Tasks listed as ordered pairs:

7,1 7,3 7,6 5,3 5,6 6,8 4,8 4,2 2,10

Web Graph

- The **webgraph** describes the directed links between pages of the World Wide Web.
- A graph, in general, consists of several vertices, some pairs connected by edges.
- In a directed graph, edges are directed lines or arcs.
- The Web graph relative to a certain set of URLs is a directed graph having those URLs as nodes, and with an arc from x to y whenever page x contains a hyperlink toward page y.
- The webgraph is a directed graph, whose vertices correspond to the pages of the WWW, and a directed edge connects page X to page Y if there exists a hyperlink on page X, referring to page Y.

Applications of Web Graph

- The webgraph is used for computing the PageRank of the WWW pages.
- The webgraph is used for computing the personalized PageRank.
- The webgraph can be used for detecting webpages of similar topics, through graph-theoretical properties only, like co-citation
- The webgraph is applied in the HITS algorithm for identifying hubs and authorities in the web.

Google Map

- Google Maps is a web mapping service developed by Google.
- It offers satellite imagery, street maps, 360° panoramic views of streets (Street View), real-time traffic conditions (Google Traffic), and route planning for traveling by foot, car, bicycle (in beta), or public transportation.
- You can represent the road network as a weighted graph, and finding a route is then just an application of a shortest path algorithm