CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25) GRAPH ALGORITHMS

Chittaranjan Hota, PhD
Sr. Professor of Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

GRAPH ALGORITHMS

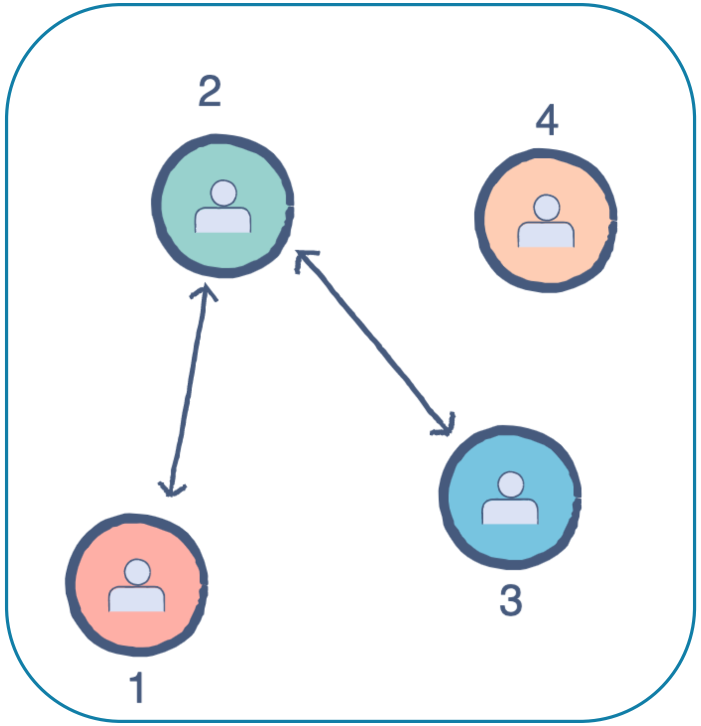
A graph is a way of representing relationships that exist between pairs of objects.

A graph is a pair (***V***, ***E***), where:

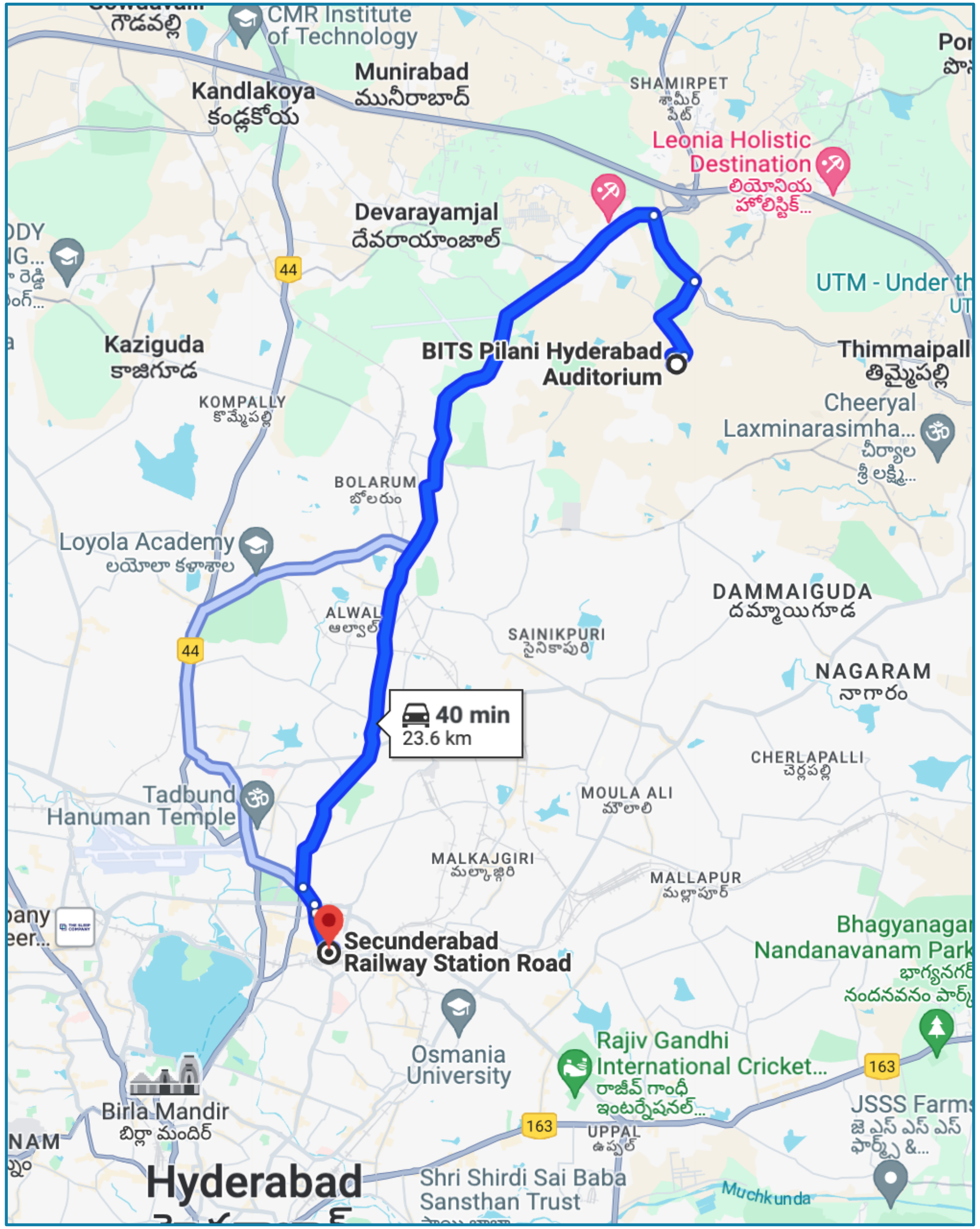
V is a set of nodes, called vertices.

E is a collection of pairs of vertices, called edges.

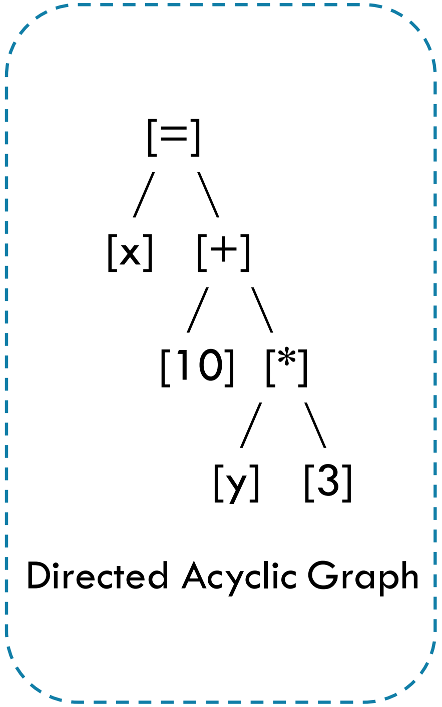
<u>Vertices and edges are positions and store elements.</u>



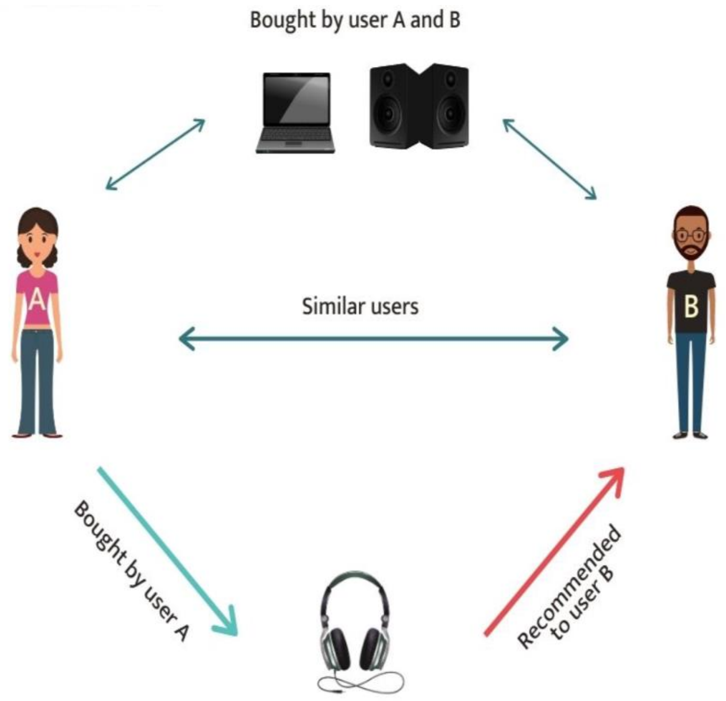
(Social Network)



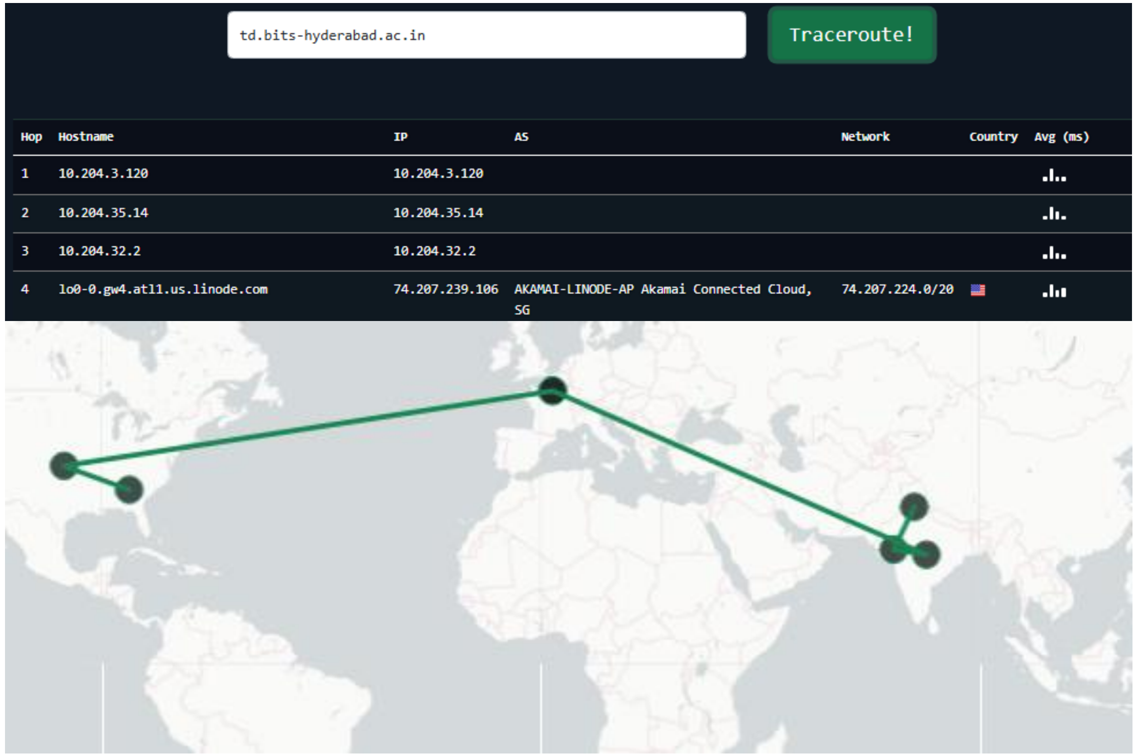
GRAPH APPLICATIONS CONTINUED…



(Abstract Syntax Tree)



(Recommender Systems)



(Internet topology)

TERMINOLOGIES

-End vertices (or endpoints) of an edge

- endpoints of a?

-Edges incident on a vertex

- edges incident on Y?

-Adjacent vertices

- Y and V: are they adjacent?

-Degree of a vertex

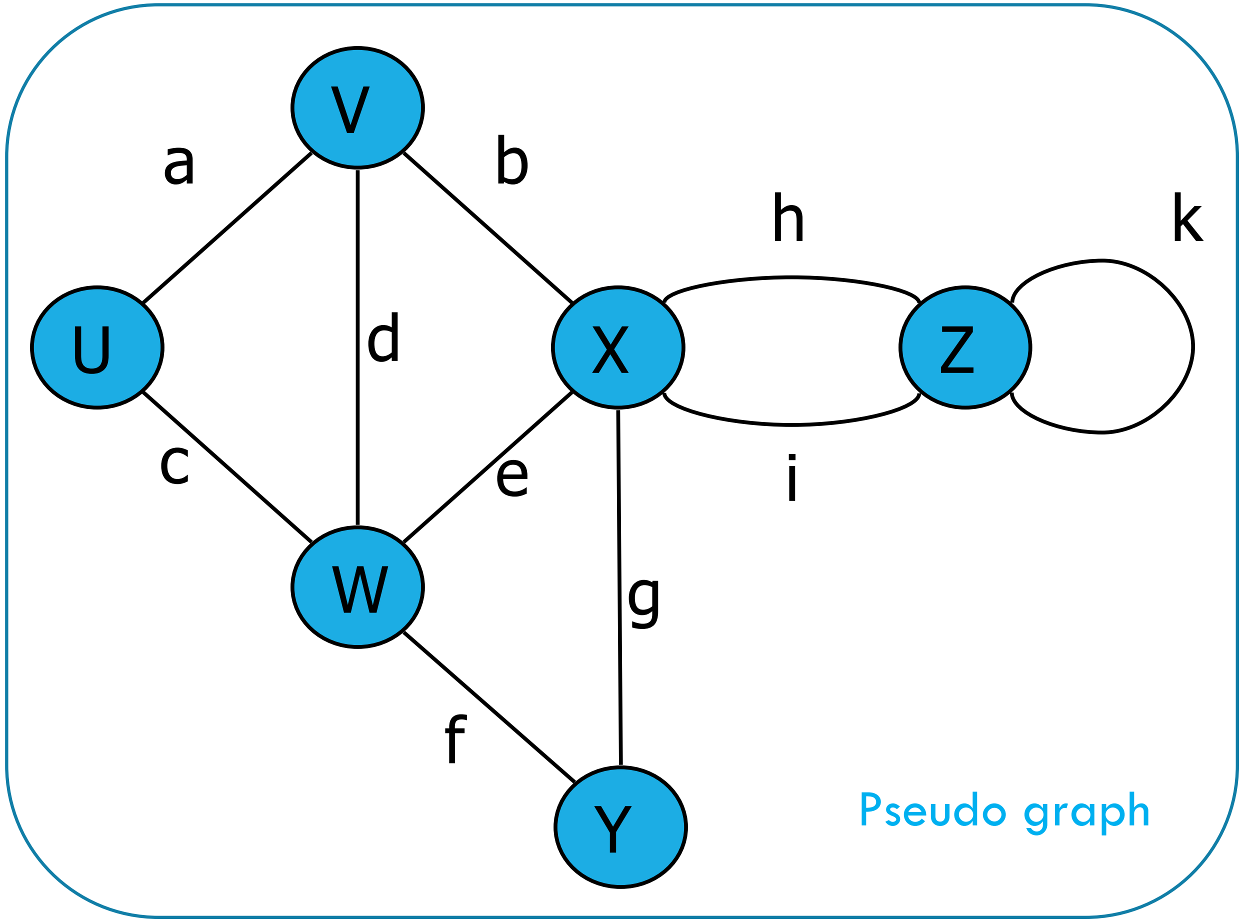
- X has degree how much

-Parallel edges

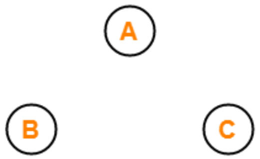
- which are parallel edges here?

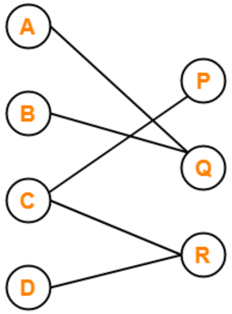
-Self-loop

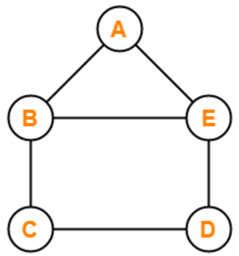
- which one is a self-loop?

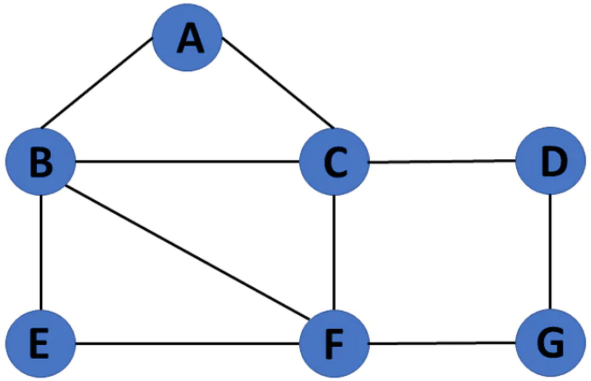


TERMINOLOGIES

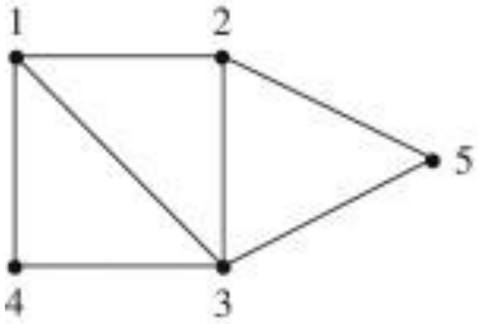




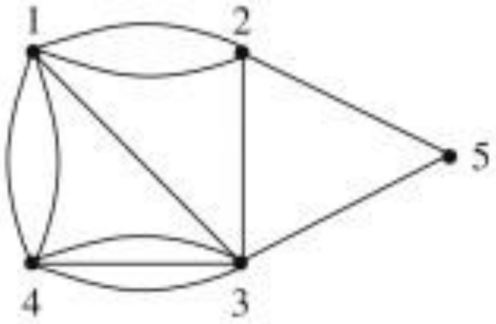




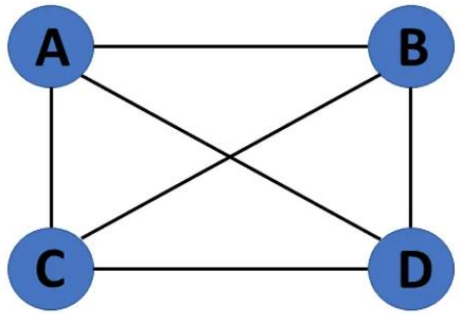
(Finite Graph)



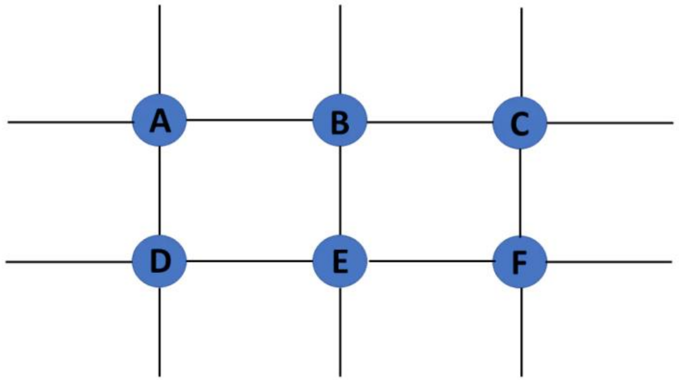
Simple graph



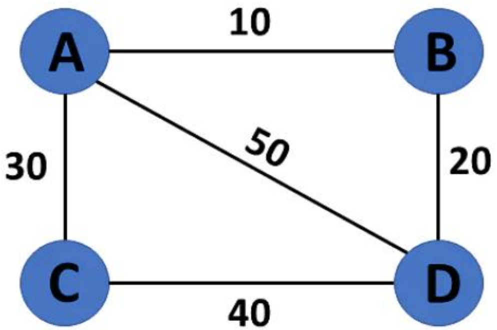
Multi graph



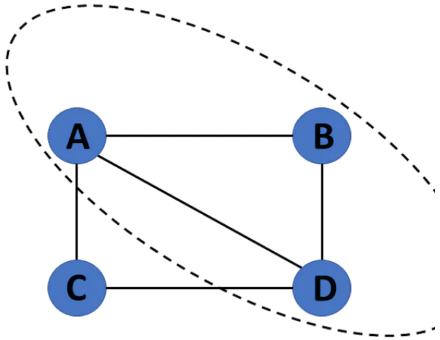
Complete graph



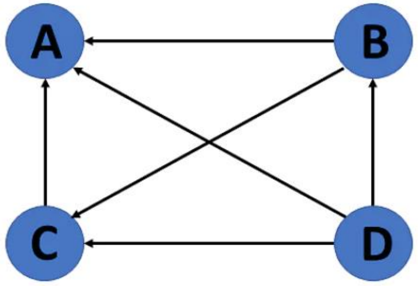
(Infinite Graph)



Weighted graph



Sub-graph

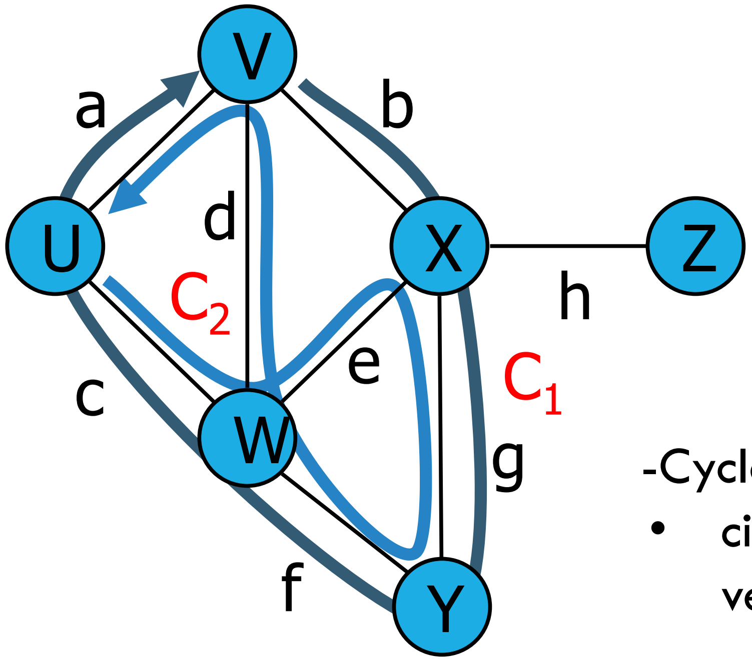


(Directed graph)

CONTINUED…

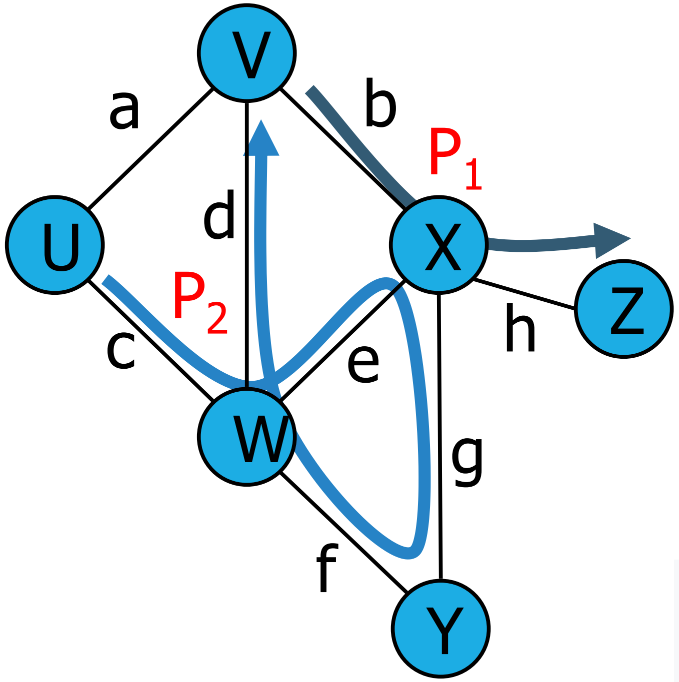
-Path: sequence of alternating vertices and edges

- A simple path, & not a simple path

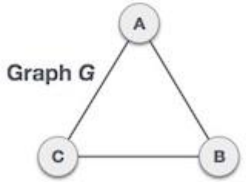


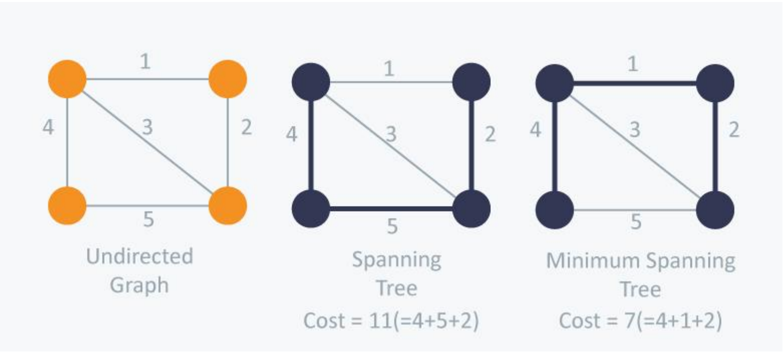
-Cycle

- circular sequence of alternating vertices and edges



A spanning tree: a sub-graph of a graph, which includes all the vertices of the graph with a minimum possible number of edges.





A directed graph can have at most ? edges, where n is the number of vertices. An undirected graph can have at most ? edges. Sparse graph: A graph in which the number of edges is much less than the possible number of edges. Dense graph: A graph in which the number of edges is close to the possible number of edges.

GRAPH ADT

A graph: directed/ undirected, cyclic/ acyclic, connected/ disconnected.

A tree: special type of graph. A connected Acyclic graph. N nodes and N-1 edges.

Vertices and edges

- are positions, store elements

Accessor methods

- e.endVertices(): a list of the two endvertices of e
- e.opposite(v): the vertex opposite of v on e
- u.isAdjacentTo(v): true iff u and v are adjacent
- *v: reference to element associated with vertex v
- *e: reference to element associated with edge e

Update methods

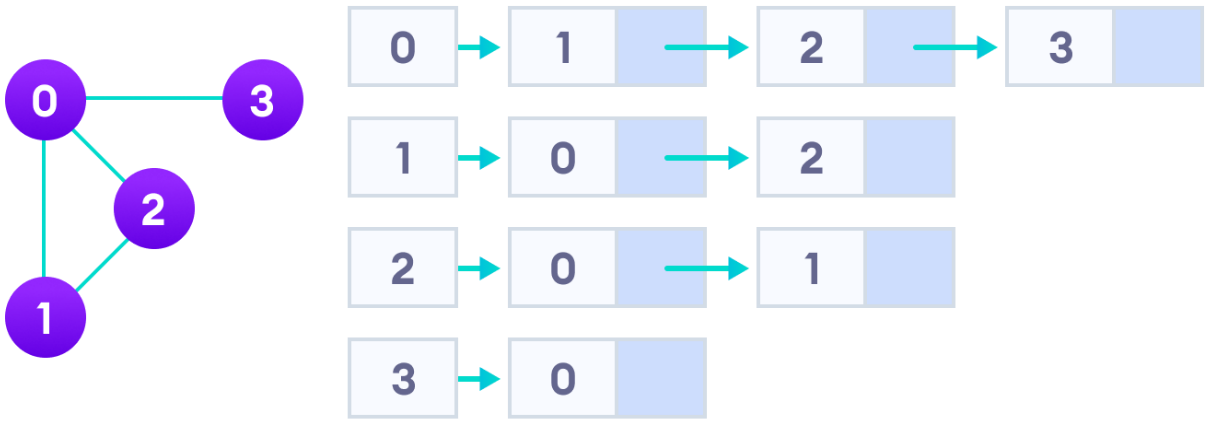
- insertVertex(o): insert a vertex storing element o
- insertEdge(v, w, o): insert an edge (v,w) storing element o
- eraseVertex(v): remove vertex v (and its incident edges)
- eraseEdge(e): remove edge e

Iterable collection methods

- incidentEdges(v): list of edges incident on v
- vertices(): list of all vertices in the graph
- edges(): list of all edges in the graph

Linear or Non-linear ADT?

DATA STRUCTURES FOR GRAPHS



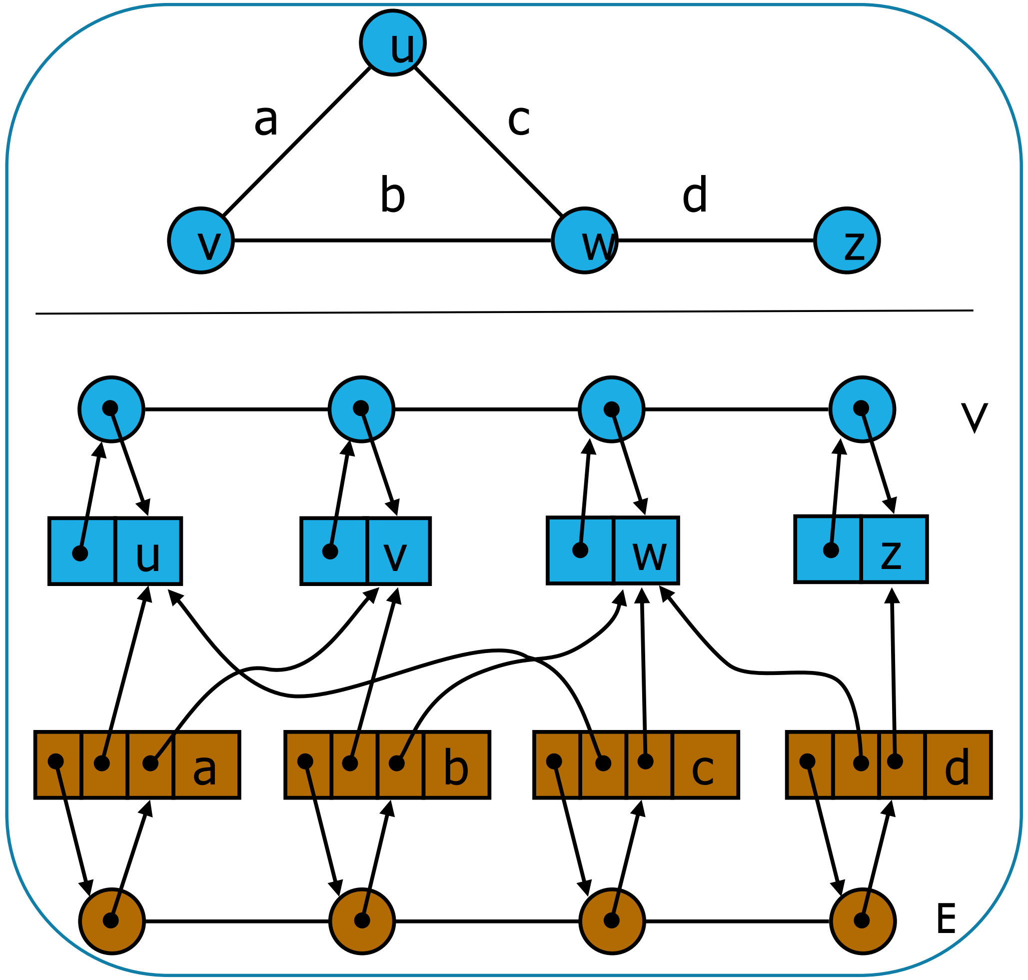
An adjacency list represents a graph as an array of linked lists.

An adjacency list is efficient in terms of storage because we only need to store the values for the edges.

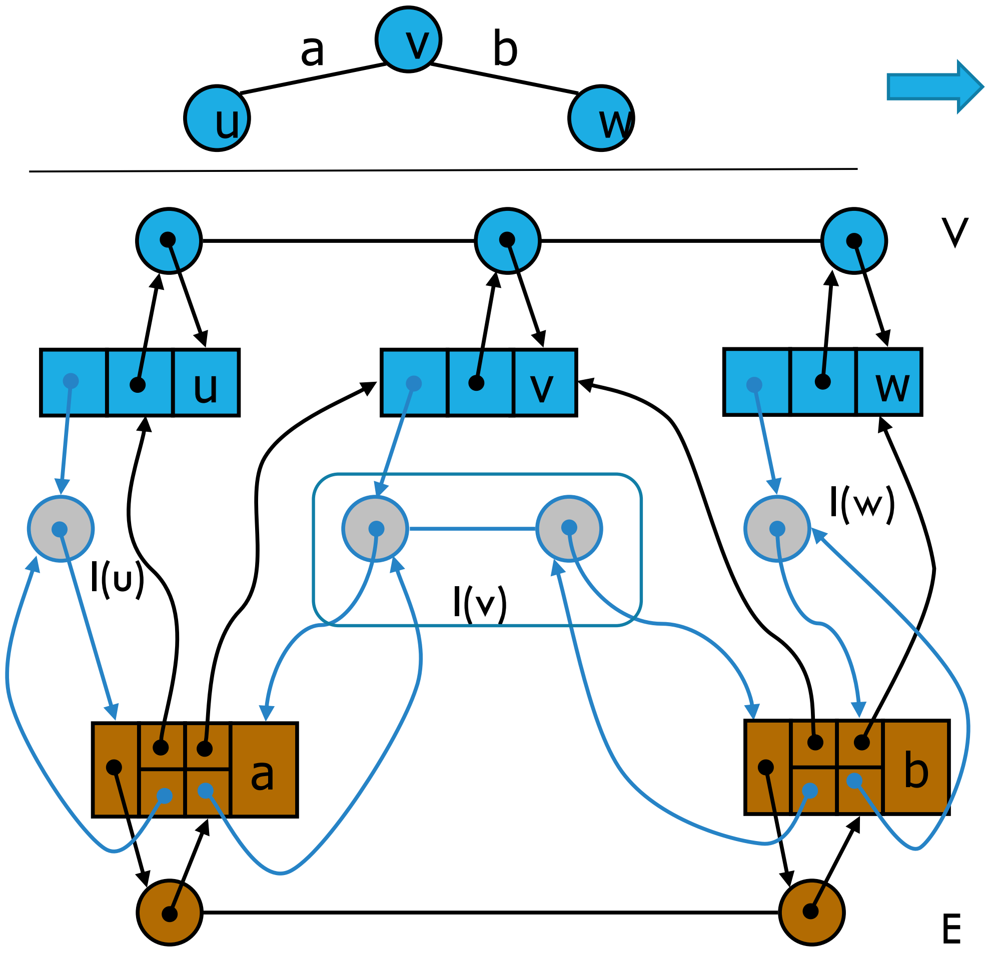
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

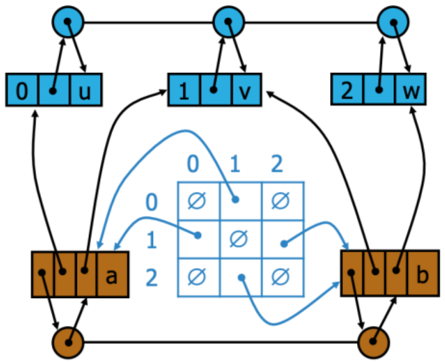
CONTINUED...



(Edge list)



(Adjacency list)



Quick insertions and deletions of edges?

Finding an edge in the graph?

If it is a sparse graph, which one is better?

(Adjacency matrix)

COMPLEXITY

(Edge list)

Operation	Time
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
incidentEdges, isAdjacentTo	$O(m)$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex	$O(m)$

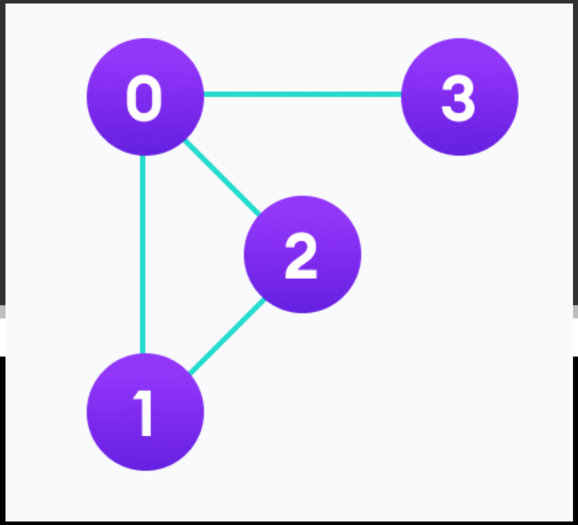
(Adjacency list)

Operation	Time
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
v.incidentEdges()	$O(\deg(v))$
v.isAdjacentTo(w)	$O(\min(\deg(v), \deg(w)))$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex(v)	$O(\deg(v))$

(Adjacency matrix)

Operation	Time
vertices	$O(n)$
edges	$O(n^2)$
endVertices, opposite	$O(1)$
isAdjacentTo, isIncidentOn	$O(1)$
incidentEdges	$O(n)$
insertEdge, eraseEdge,	$O(1)$
insertVertex, eraseVertex	$O(n^2)$

```cpp
#include <bits/stdc++.h>
using namespace std;
void addEdge(vector<int> adj[], int s, int d) {
  adj[s].push_back(d);
  adj[d].push_back(s);
}
void printGraph(vector<int> adj[], int V) {
  for (int d = 0; d < V; ++d) {
    cout << "\n Vertex " << d << ":";
    for (auto x : adj[d])
      cout << "-> " << x;
    printf("\n");
  }
}
int main() {
  int V = 5;
  vector<int> adj[V];
  addEdge(adj, 0, 1);
  addEdge(adj, 0, 2);
  addEdge(adj, 0, 3);
  addEdge(adj, 1, 2);
  printGraph(adj, V);
}
```




```
Vertex 0:-> 1-> 2-> 3

Vertex 1:-> 0-> 2

Vertex 2:-> 0-> 1

Vertex 3:-> 0

Vertex 4:
```

GRAPH TRAVERSALS: DEPTH FIRST (DFS)

A graph traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.

Source: https://levelup.gitconnected.com/ (A path in a Maze)

can be used for:

1. Testing whether the graph G is connected?
2. Computing a spanning tree if exists
3. Computing a path between two vertices
4. Finding out if there exists a cycle in G
5. Finding strongly connected components (if each vertex has a path to every other vertex)

Algorithm DFS(G, v):

  Input: A graph G and a vertex v of G

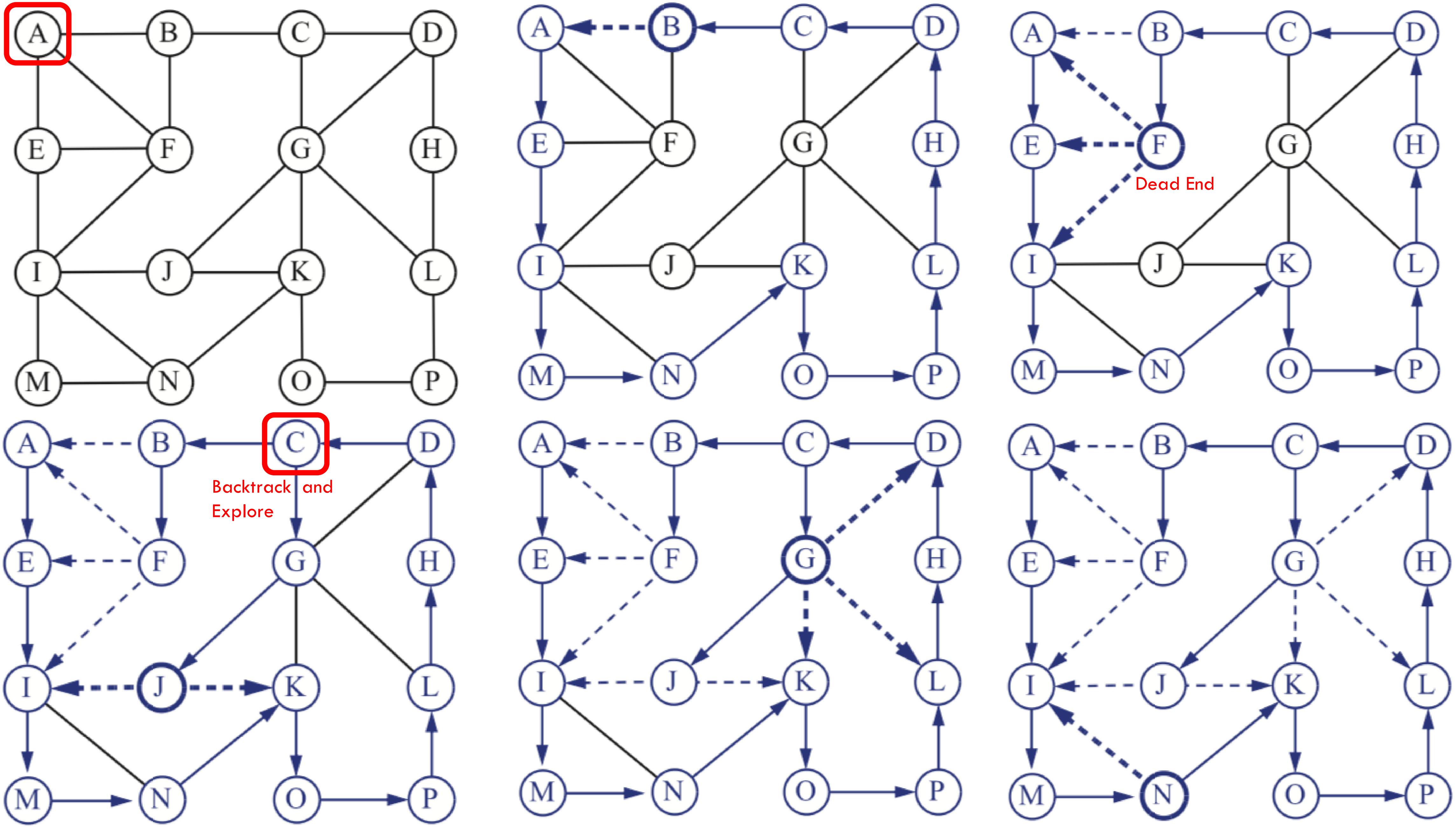
  Output: A labeling of the edges in the connected edges and back edges

label v as visited
for all edges e in v.incidentEdges() **do**
  if edge e is unvisited **then**
    $w \leftarrow e$.opposite(v)
    if vertex w is unexplored **then**
      label e as a discovery edge
      recursively call DFS(G, w)
    else
      label e as a back edge

Recursive and Backtracking algo.

What is the complexity?

AN EXAMPLE DFS



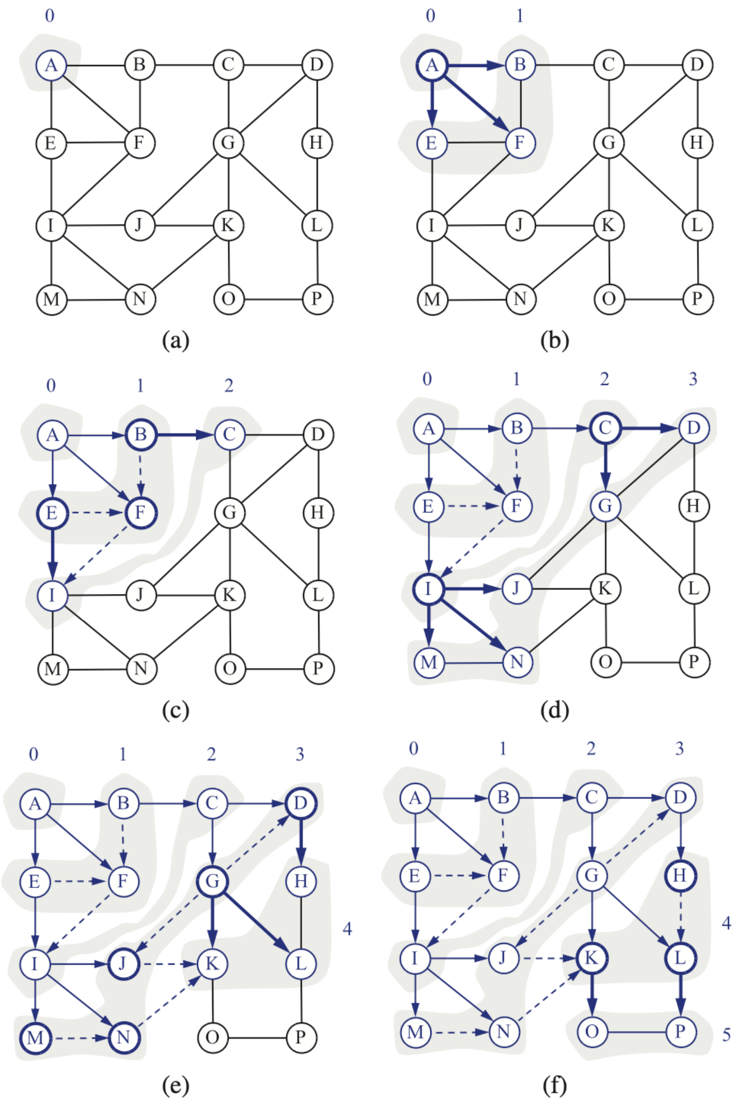
DFS IMPLEMENTATION

```cpp
#include <iostream>
#include <list>
using namespace std;
class DFSGraph {
  int V;    // No. of vertices
  list<int> *adjList;    // adjacency list
  void DFS_util(int v, bool visited[]);
  public:
    DFSGraph(int V) {
      this->V = V;
      adjList = new list<int>[V];
    }
  void addEdge(int v, int w){
    adjList[v].push_back(w); // Add w to v’s list.
  }
  void DFS();    // DFS traversal function
};

void DFSGraph::DFS_util(int v, bool visited[]) {
  // current node v is visited
  visited[v] = true;
  cout << v << " ";
  // recursively process all the adjacent vertices
  list<int>::iterator i;
  for(i = adjList[v].begin(); i != adjList[v].end(); ++i)
    if(!visited[*i])
      DFS_util(*i, visited);
}
void DFSGraph::DFS() {
  bool *visited = new bool[V];
  for (int i = 0; i < V; i++)
    visited[i] = false;
  for (int i = 0; i < V; i++)
    if (visited[i] == false)
      DFS_util(i, visited);
}

int main() {
  DFSGraph gdfs(6);
  gdfs.addEdge(0, 1);
  gdfs.addEdge(0, 2);
  gdfs.addEdge(0, 3);
  gdfs.addEdge(1, 4);
  gdfs.addEdge(2, 4);
  gdfs.addEdge(4, 5);
  gdfs.DFS();
  return 0;
}
```

BREADTH FIRST SEARCH (BFS)

Algorithm BFS(s):
  initialize collection L_0 to contain vertex s
  $i \leftarrow 0$
  while L_i is not empty **do**
    create collection L_{i+1} to initially be empty
    for all vertices v in L_i **do**
      for all edges e in v.incidentEdges() **do**
        if edge e is unexplored **then**
          $w \leftarrow e$.opposite(v)
          if vertex w is unexplored **then**
            label e as a discovery edge
            insert w into L_{i+1}
          else
            label e as a cross edge
    $i \leftarrow i+1$

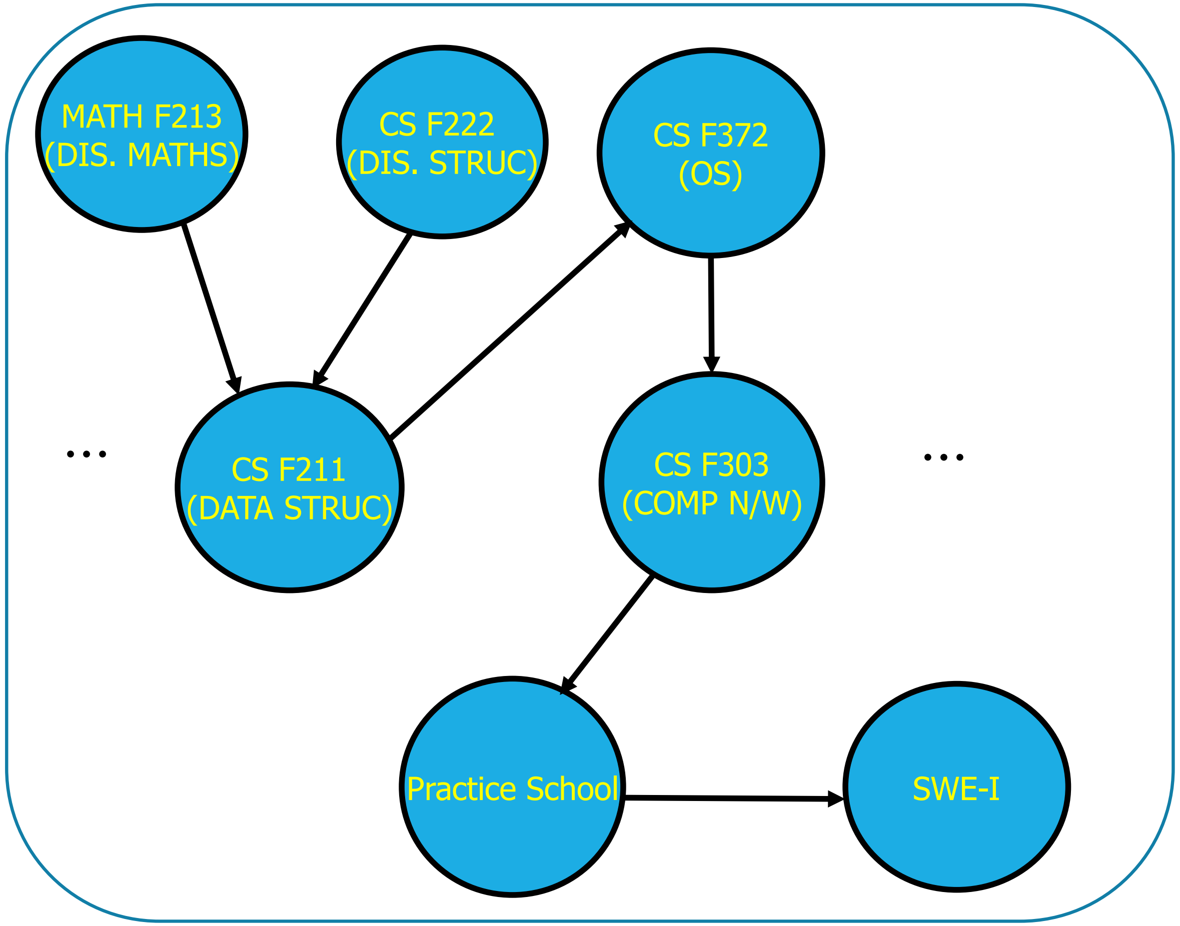
Testing whether the graph G is connected, Computing a spanning tree if exists, Computing a shortest path between two vertices, Finding out if there exists a cycle in G.

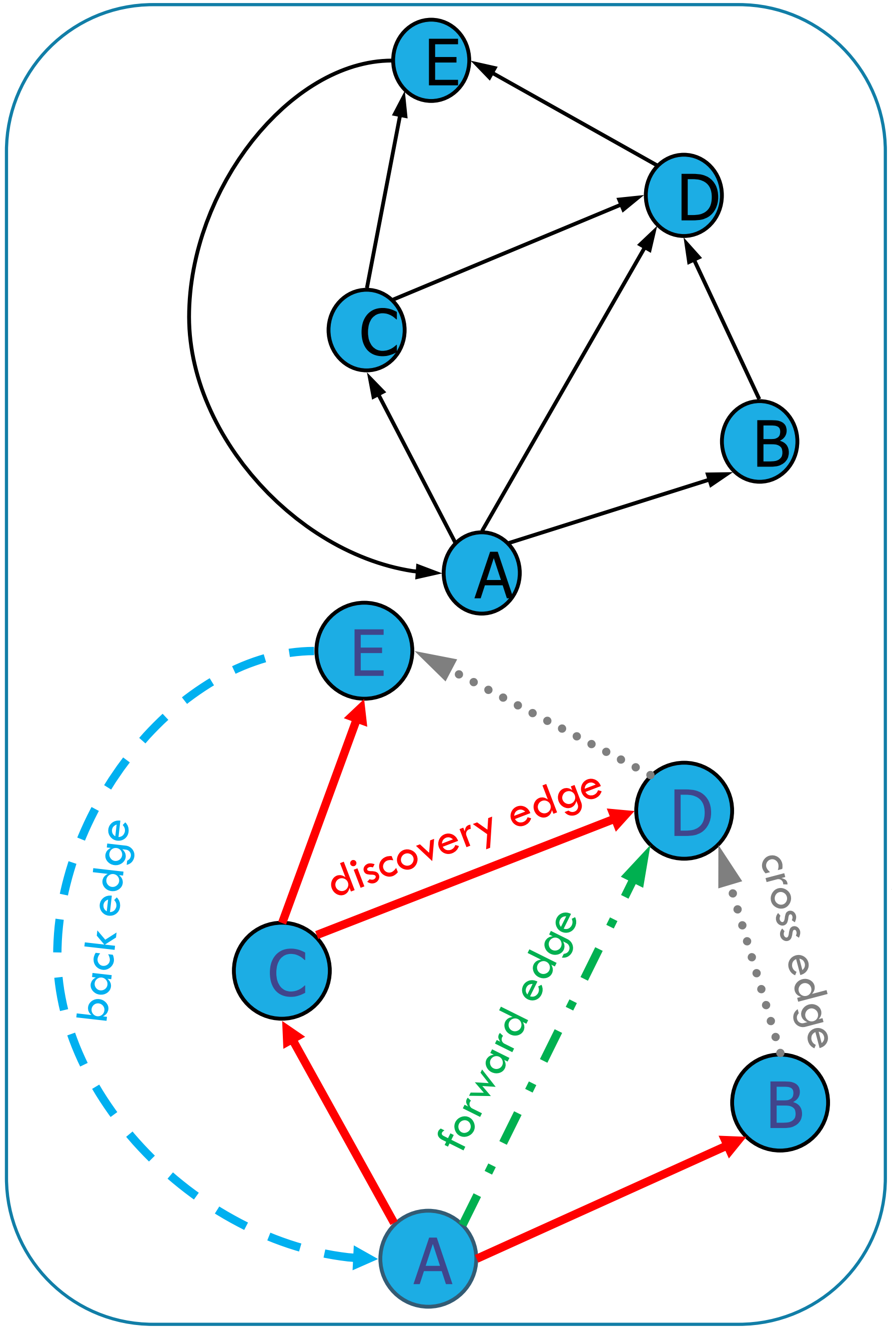


DIRECTED GRAPHS (DIGRAPHS)

- A graph with all of its' edges as directed.

(Ex: Task Scheduling)





(Directed DFS)

REACHABILITY THROUGH TRANSITIVE CLOSURE

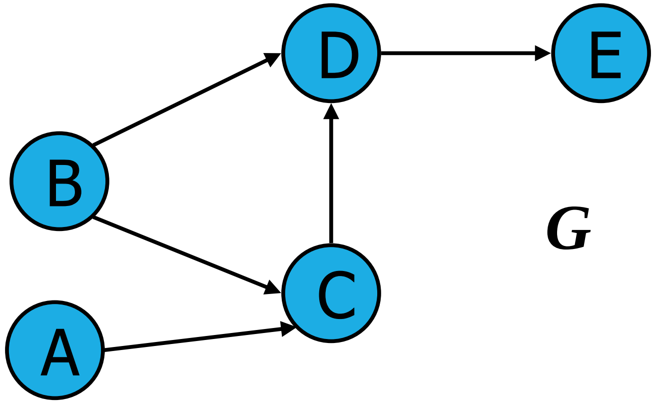
- Gives reachability information

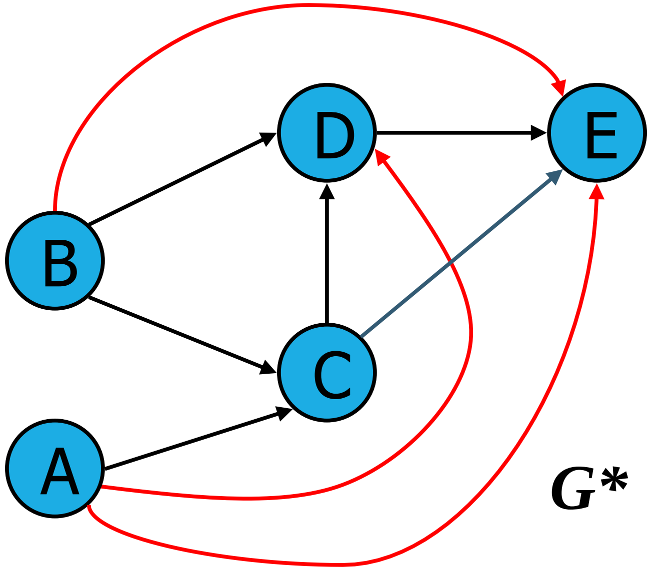
Algorithm FloydWarshall(G):

Input: A digraph G with n vertices
Output: The transitive closure G^* of G

let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of G
$G_0 \leftarrow G$
for $k \leftarrow 1$ to n **do**
  $G_k \leftarrow G_{k-1}$
  for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
    if both edges (v_i, v_k) and (v_k, v_j) are in G_{k-1} **then**
      add edge (v_i, v_j) to G_k (if it is not already present)
return G_n

- Alternatively, we can perform DFS starting at each vertex.

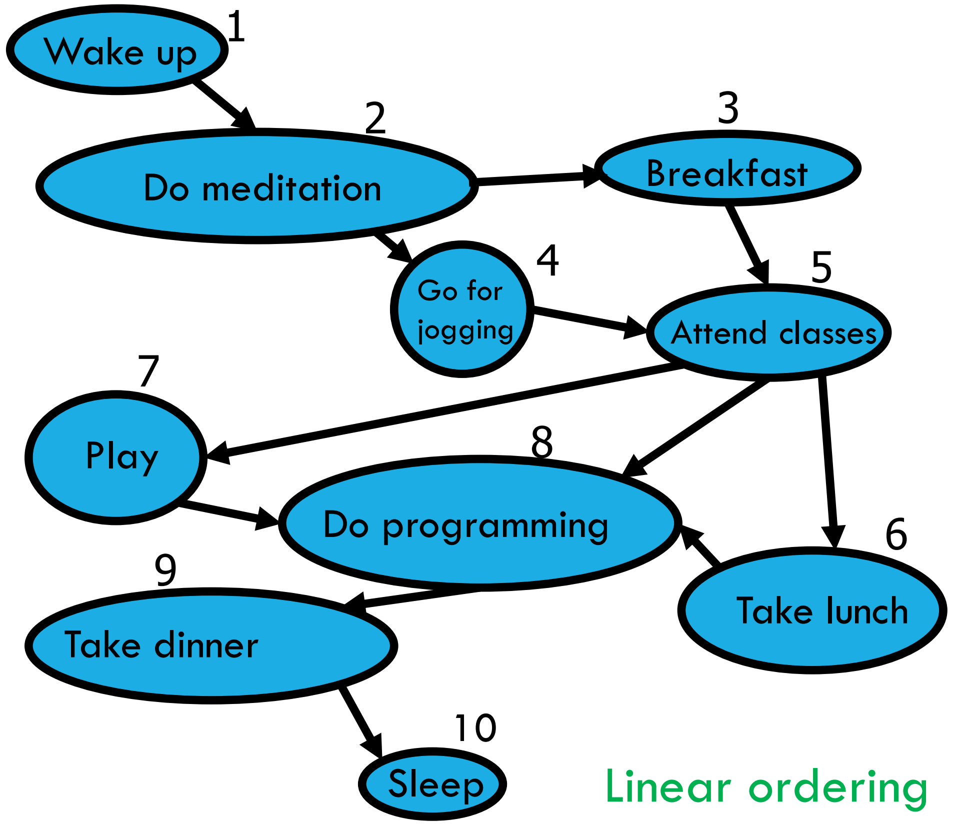




DIRECTED ACYCLIC GRAPHS (DAG)

Algorithm TopologicalSort(G):
  Input: A digraph G with n vertices
  Output: A topological ordering $v_1, \ldots, v_n$ of G
  $S \leftarrow$ an initially empty stack.
  for all u in G.vertices() **do**
    Let incounter(u) be the in-degree of u.
    if incounter$(u) = 0$ **then**
      S.push(u)
  $i \leftarrow 1$
  while !S.empty() **do**
    $u \leftarrow S$.pop()
    Let u be vertex number i in the topological ordering.
    $i \leftarrow i + 1$
    for all outgoing edges (u, w) of u **do**
      incounter$(w) \leftarrow$ incounter$(w) - 1$
      if incounter$(w) = 0$ **then**
        S.push(w)

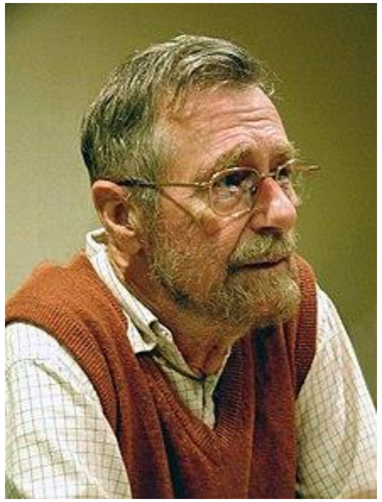
A typical student day: Topological sort

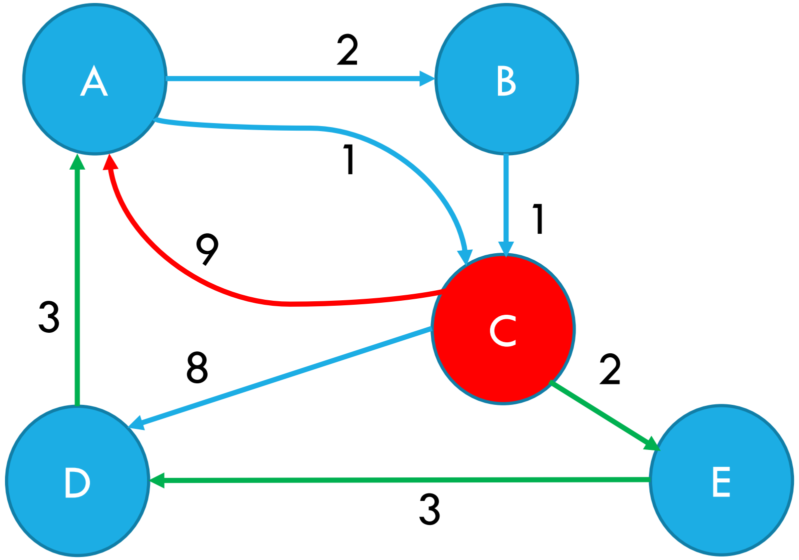


Linear ordering

In order to get a job you need to have work experience, but in order to get work experience you need to have a job.

WHY DIJKSTRA'S ALGO FOR SHORTEST PATH?





Is BFS possible?

Ok for undirected and uniform cost graphs.

If there is a negative weight, will it work properly?

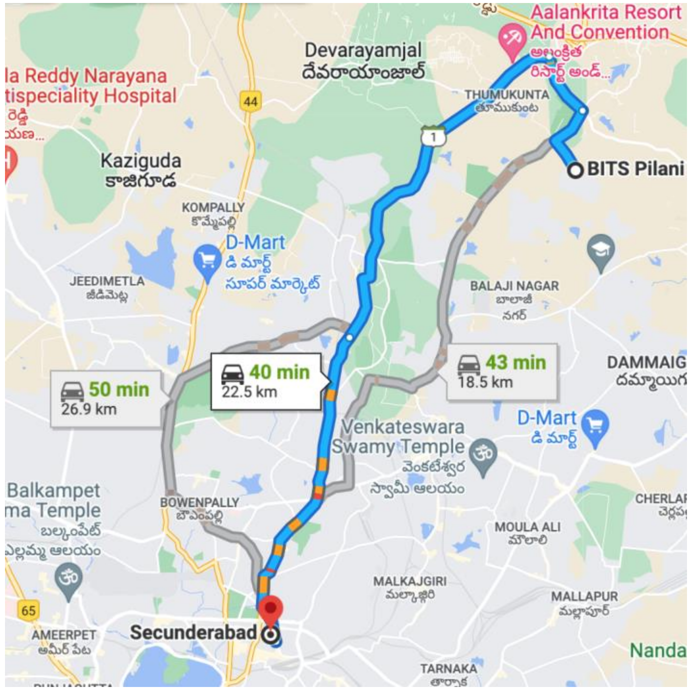
From a source to all other nodes → shortest path tree

Is it not a greedy algorithm?

Applications: Google maps, OSPF: A Link state routing algorithm in the Internet etc.

DIJKSTRA'S SHORTEST PATH EXAMPLES

```
Last login: Wed Apr 27 20:37:04 on ttys000
chittaranjans-MacBook-Pro:~ hota$ netstat -rn
Routing tables

Internet:
Destination        Gateway            Flags        Refs      Use   Netif Expire
default            192.168.0.1        UGSc          350        0     en0
127                127.0.0.1          UCS             0        0     lo0
127.0.0.1          127.0.0.1          UH              1      114     lo0
169.254            link#5             UCS             0        0     en0      !
192.168.0          link#5             UCS             1        0     en0      !
192.168.0.1/32     link#5             UCS             1        0     en0      !
192.168.0.1        e8:48:b8:c1:25:2c  UHLWIir       307      575     en0   1185
192.168.0.111/32   link#5             UCS             0        0     en0      !
192.168.0.215      a4:83:e7:69:89:c9  UHLWI           0        0     en0    486
224.0.0/4          link#5             UmCS            2        0     en0      !
224.0.0.251        1:0:5e:0:0:fb      UHmLWI          0        0     en0
239.255.255.250    1:0:5e:7f:ff:fa    UHmLWI          0       87     en0
255.255.255.255/32 link#5             UCS             1        0     en0      !
255.255.255.255    ff:ff:ff:ff:ff:ff  UHLWbI          0        3     en0      !
```



SHORTEST PATH: DIJKSTRA'S EDGE RELAXATION

Consider an edge **e** = (***u***,***z***) such that

- ***u*** is the vertex most recently added to the cloud
- ***z*** is not in the cloud

The relaxation of edge **e** updates distance ***d***(***z***) as follows:

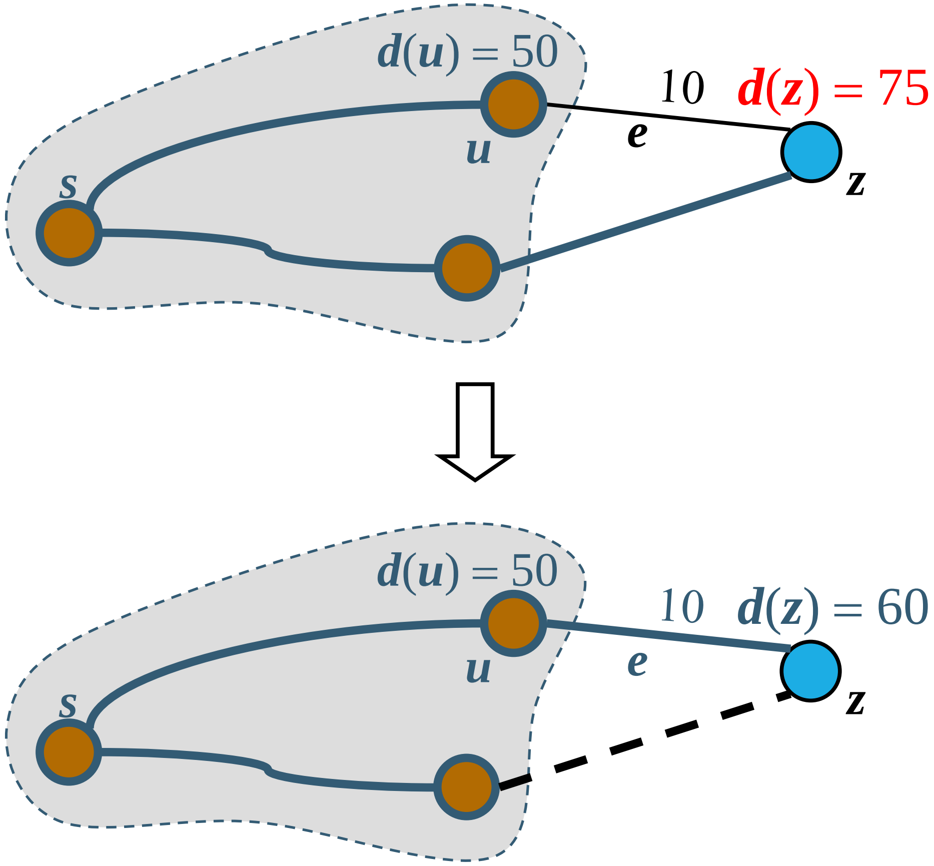
d(***z***) $\leftarrow$ min{***d***(***z***),***d***(***u***) + ***weight***(**e**)}

Algorithm ShortestPath(G, v):

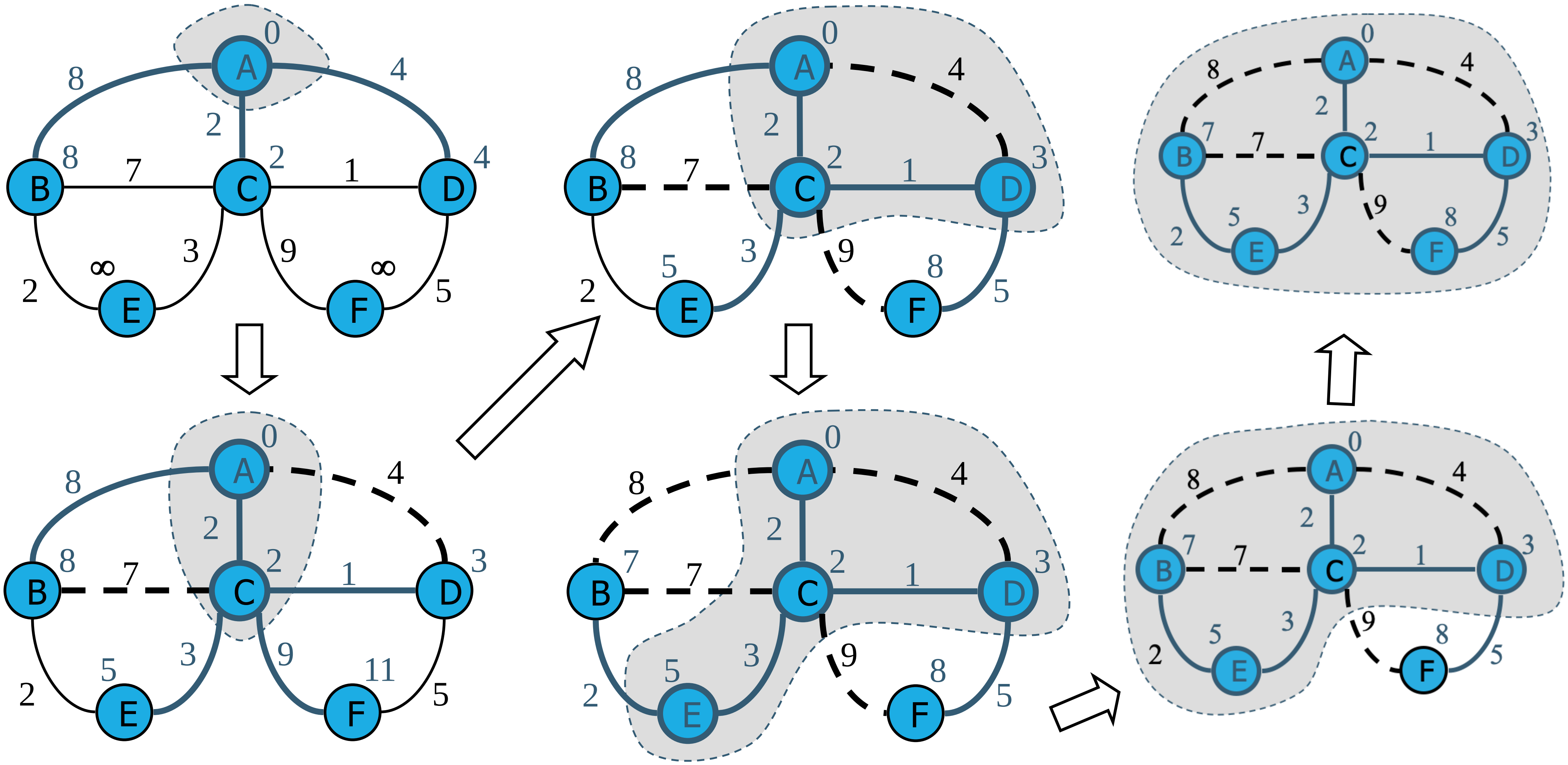
Input: A simple undirected weighted graph G with nonnegative edge weights and a distinguished vertex v of G

Output: A label $D[u]$, for each vertex u of G, such that $D[u]$ is the length of a shortest path from v to u in G

Initialize $D[v] \leftarrow 0$ and $D[u] \leftarrow +\infty$ for each vertex $u \neq v$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  $u \leftarrow Q$.removeMin()
  for each vertex z adjacent to u such that z is in Q **do**
    {perform the ***relaxation*** procedure on edge (u,z)}
    if $D[u] + w((u,z)) < D[z]$ **then**
      $D[z] \leftarrow D[u] + w((u,z))$
      Change to $D[z]$ the key of vertex z in Q.
return the label $D[u]$ of each vertex u



AN EXAMPLE



Dijkstra's algorithm runs in O((n+m)logn) time, using a min priority queue implemented with a heap.

MINIMUM SPANNING TREES (MST)

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

Spanning tree

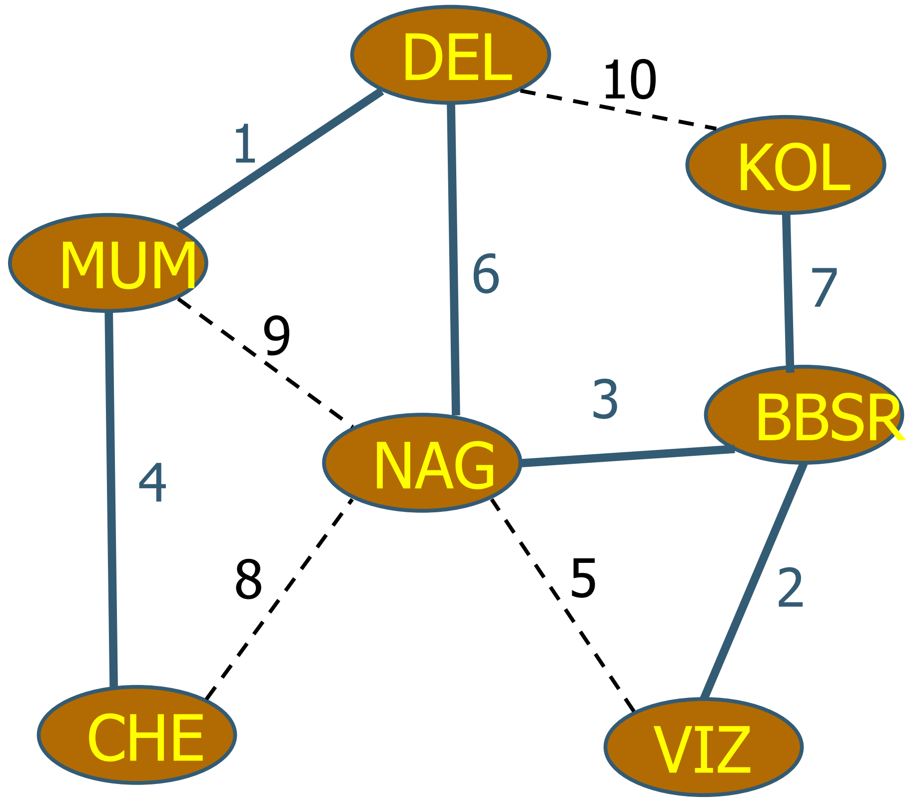
- Spanning subgraph that is itself a tree

Minimum spanning tree (MST)

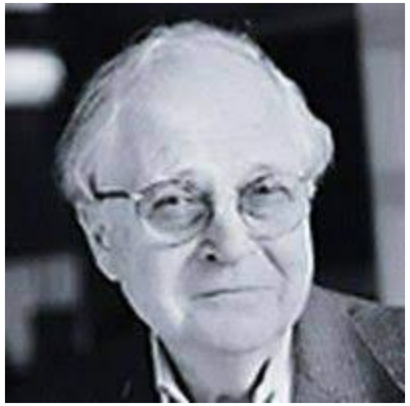
- Spanning tree of a weighted graph with minimum total edge weight

Applications

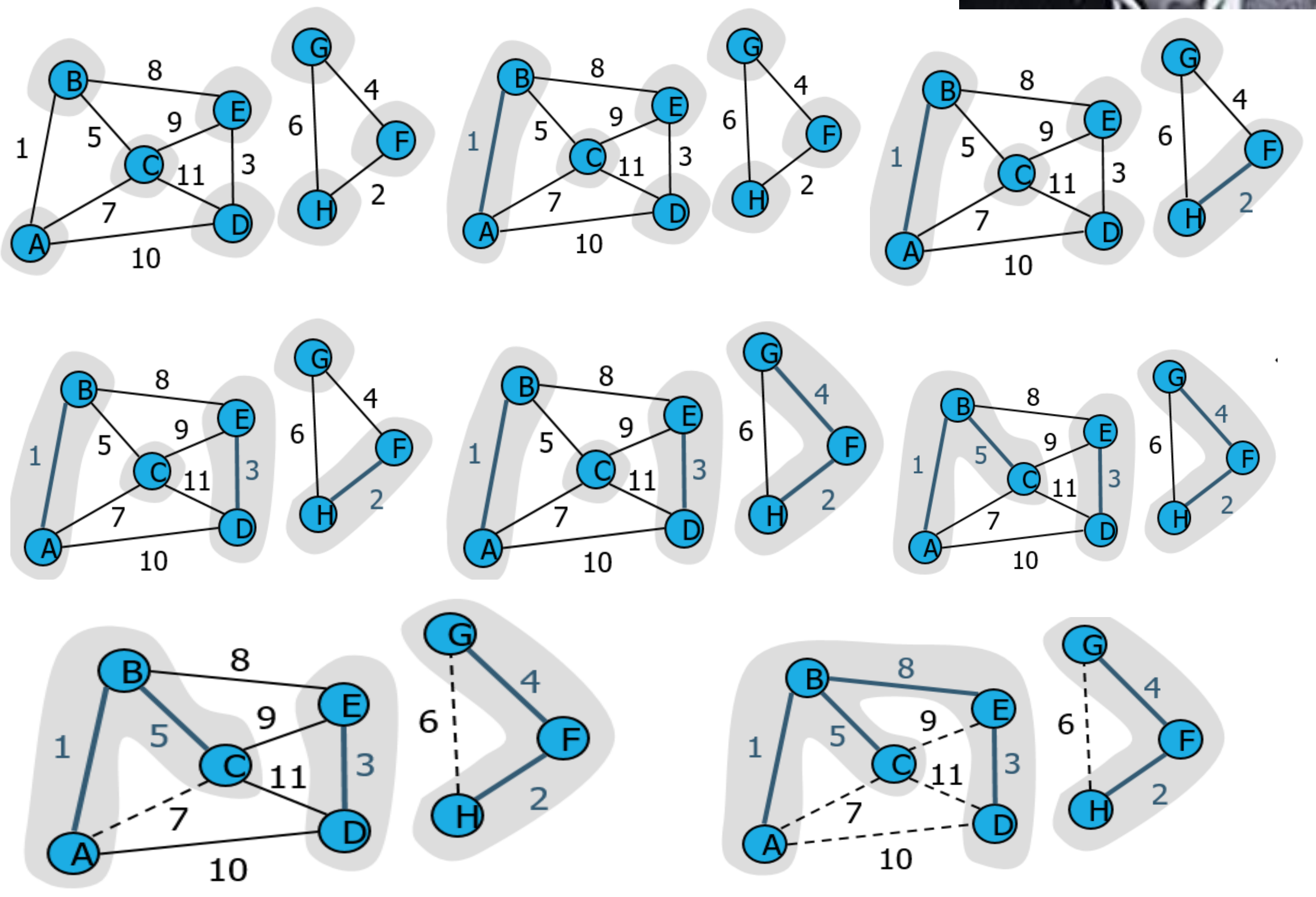
- Communications networks
- Transportation networks



MST: KRUSKAL'S ALGORITHM

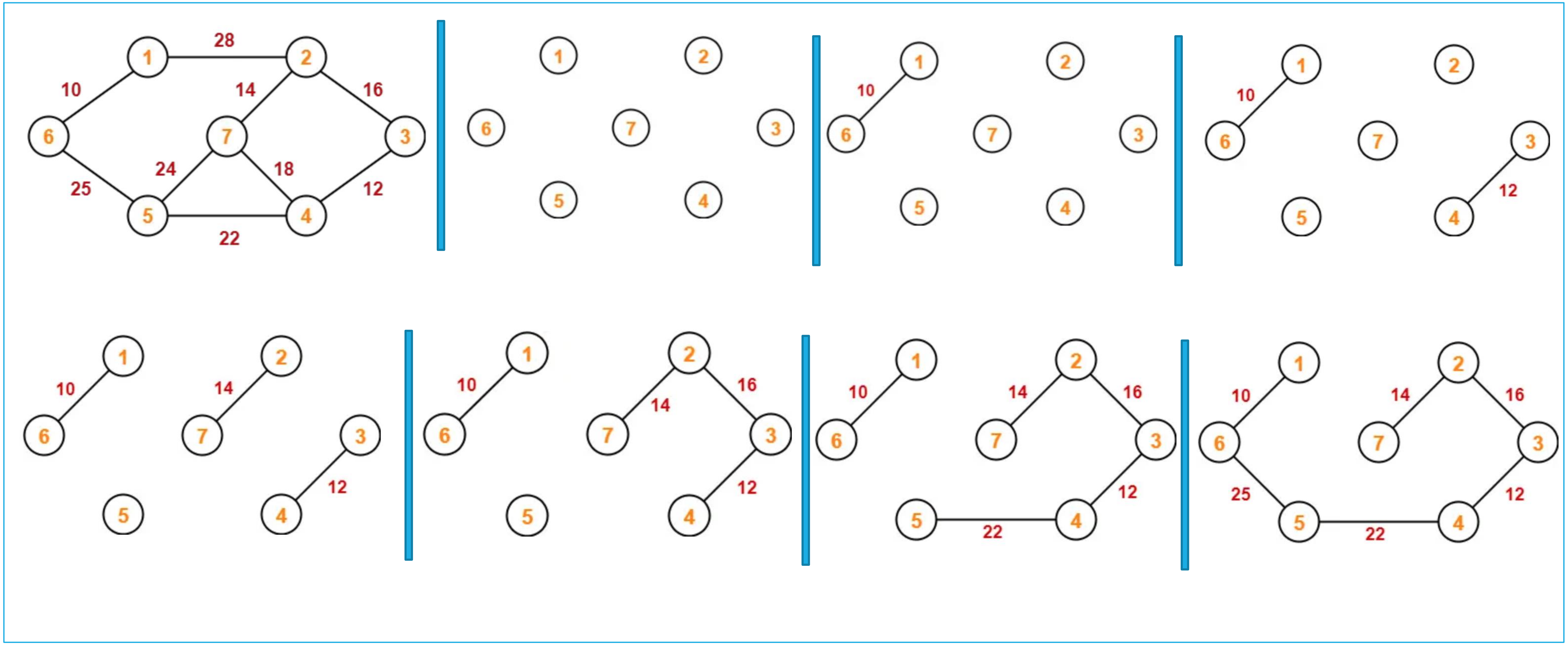


Algorithm *KruskalMST*(**G**)
    for each vertex **v** in **G** **do**
        Create a cluster consisting of **v**
    let Q be a priority queue.
    Insert all edges into **Q**
    $T \leftarrow \emptyset$
    {*T* is the union of the MSTs of the clusters}
    while *T* has fewer than *n* - 1 edges **do**
        e ← ***Q.removeMin*().*getValue*()**
        [*u*, *v*] ← ***G.endVertices*(e)**
        A ← *getCluster*(*u*)
        B ← *getCluster*(*v*)
        if $A \neq B$ **then**
            Add edge **e** to *T*
            mergeClusters(*A*, *B*)
    return *T*



Running time of Kruskal's algorithm is O(|E|.log|V|)

ONE MORE EXAMPLE OF KRUSKAL’S ALGO



PRIM-JARNIK'S ALGORITHM

-Similar to Dijkstra's algorithm

-We pick an arbitrary vertex ***s*** and we grow the MST as a cloud of vertices, starting from ***s***

-We store with each vertex ***v*** label ***d***(***v***) representing the smallest weight of an edge connecting ***v*** to a vertex in the cloud

-At each step:

-We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label

-We update the labels of the vertices adjacent to ***u***

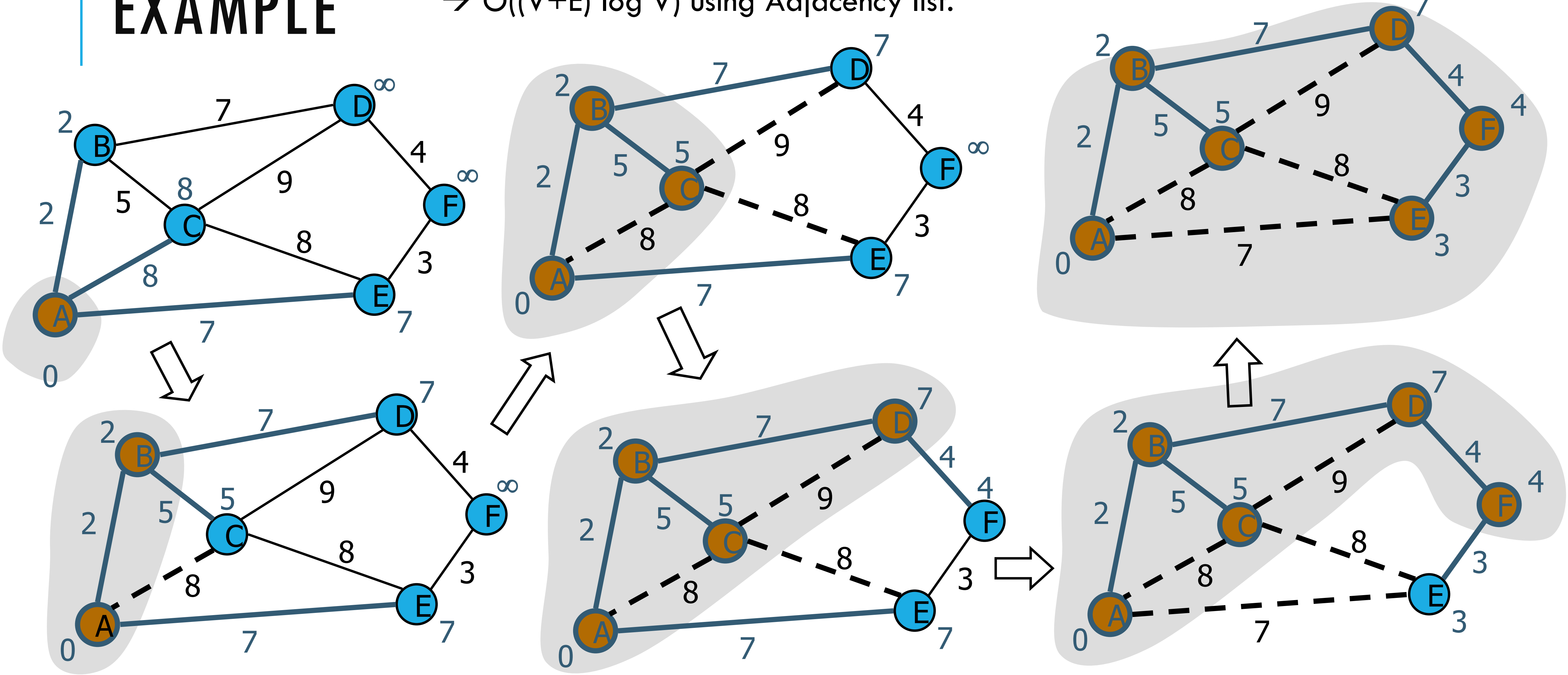
```
Algorithm PrimJarnikMST(G)
   Q ← new heap-based priority queue
   s ← a vertex of G
   for all  v ∈ G.vertices()
      if  v = s
         v.setDistance(0)
      else
         v.setDistance(∞)
      v.setParent(∅)
      l ← Q.insert(v.getDistance(), v)
      v.setLocator(l)
   while  ¬Q.empty()
      l ← Q.removeMin()
      u ← l.getValue()
      for all  e ∈ u.incidentEdges()
         z ← e.opposite(u)
         r ← e.weight()
         if  r < z.getDistance()
            z.setDistance(r)
            z.setParent(e)
            Q.replaceKey(z.getEntry(), r)
```

EXAMPLE

Complexity: Each vertex inserted into priority queue once: O(VlogV)

Each edge can cause decrease key (update) operation: O(E.logV)

→ O((V+E) log V) using Adjacency list.



THANK YOU!

Good luck for Comprehensive exams!