Data Structures (15B11CI311)

Odd Semester 2020

3rd Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Non Linear Data Structures : Graphs

Definition, Applications and Representation

Graphs and Relations

- *What makes graphs so special?*
- *Trees captures relationships too, so why are graphs more interesting?*
- *What do we mean by a "relationship"? Can you think of a mathematical way to represent relationships.*
- *Can you see how to represent a (mathematical) relation with a graph?*

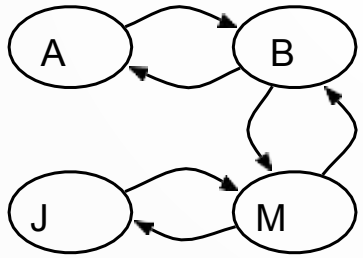


Figure : Friendship relation {*(A, B), (B, A), (B, M), (M, B), (J, M), (M, J}* as a graph.

Defining Graphs

Directed graphs

Formally, a *directed graph* or (*digraph*) is a pair $G = (V, E)$ where

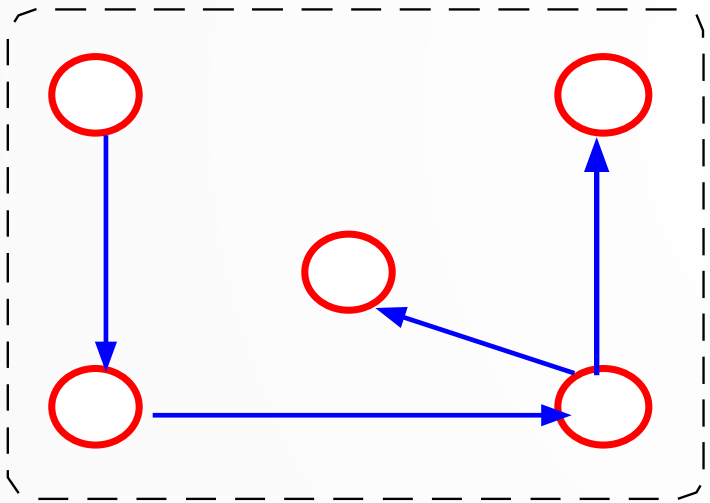
- V is a set of *vertices* (or nodes), and
- $E \subseteq V \times V$ is a set of *directed edges* (or arcs).

Each arc is an ordered pair $e = (u, v)$. A digraph can have *self loops* (u, u). Directed graphs represent asymmetric relationships,

e.g., my web page points to yours, but yours does not necessarily point back.

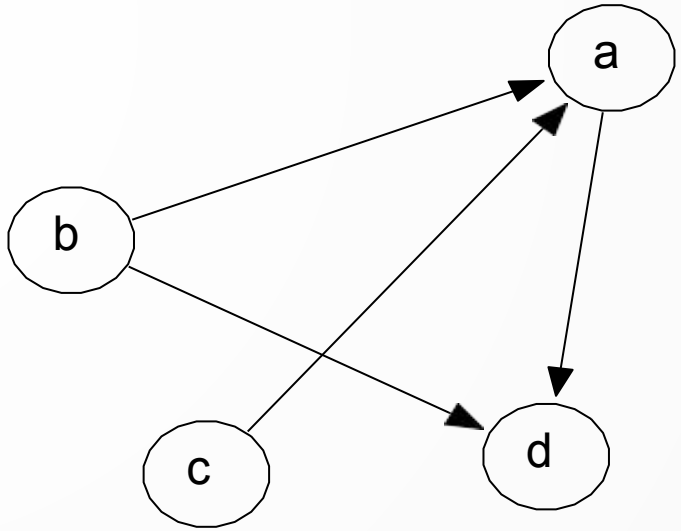
Examples

A graph where edges are directed



V= {a,b,c,d,e}

E= {(b,a),(c,a),(b,d),(a,d)}



Defining Graphs Cond…

Undirected Graph

Formally, an un*directed graph* or (*digraph*) is a pair $G = (V, E)$ where

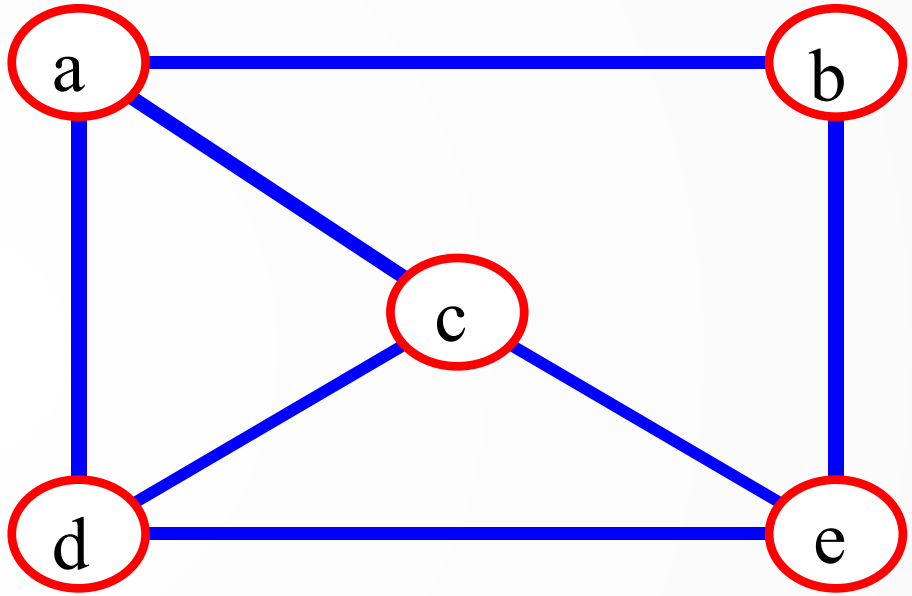
- V is a set of *vertices* (or nodes), and
- $E \subseteq C(V,2)$ is a set of *edges* (or arcs).

- Each edge is an unordered pair $e = \{u, v\}$ (or equivalently $\{v,u\}$).
- Undirected graph can not have self loop since $\{v,v\} = \{v\}$.
- Undirected graphs represent symmetric relationships.

Example

V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}



Applications of Graphs

- *Social network graphs: to tweet or not to tweet.*
- *Transportation networks.* to find the best routes between locations.
- *Utility graphs. To* minimizing the costs to build infrastructure that matches required demands.
- *Document link graphs.* to analyze relevance of web pages, the best sources of information, and good link sites.
- *Protein-protein interactions graphs.* to study molecular pathways.

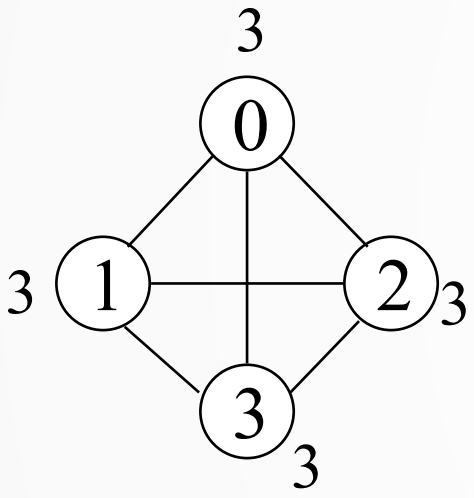
Terminology

- **Neighbors:** A vertex u is a *neighbor* of (or equivalently *adjacent* to) a vertex v in a graph $G = (V, E)$ if there is an edge $\{u, v\} \in E$. For a directed graph a vertex u is an *in-neighbor* of a vertex v if $(u, v) \in E$ and an *out-neighbor* if $(v, u) \in E$.
- **Neighborhood :** For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex v $\in$ is its set of all neighbors of v, i.e., $N_G(v) = \{ u | \{u,v\} \in E \}$. For a directed graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v. If we use $N_G(v)$ for a directed graph, we mean the out neighbors. The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods.

Terminology Cond …

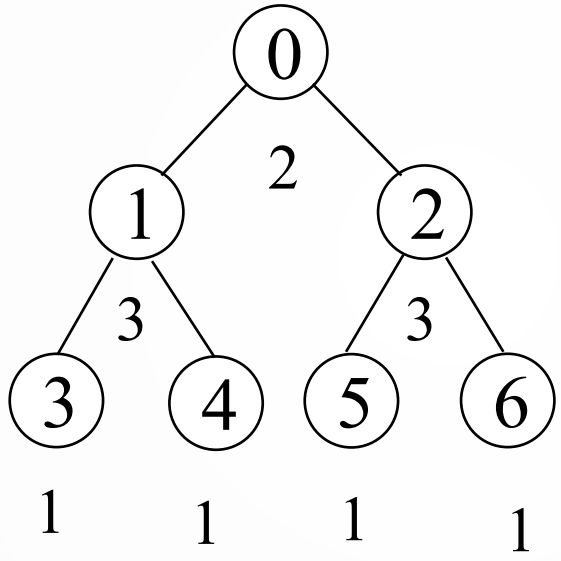
- **Degree:** The *degree* $d_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$ is the size of the neighborhood $|N_G(v)|$. For directed graphs we use *in-degree* $d_G^-(v) = |N_G^-(v)|$ and *out-degree* $d_G^+(v) = |N_G^+(v)|$.

Examples

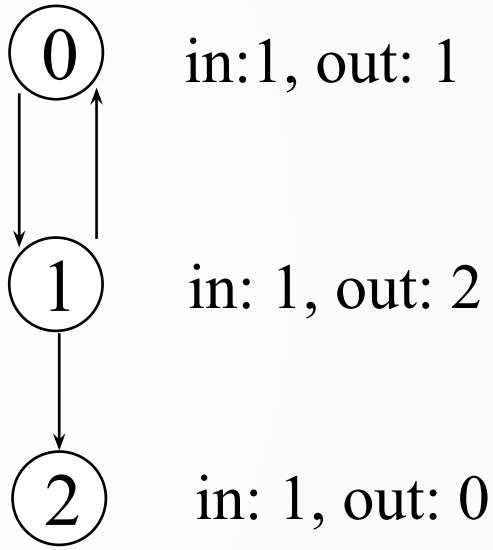


G_1

undirected graph degree



G_2



G_3

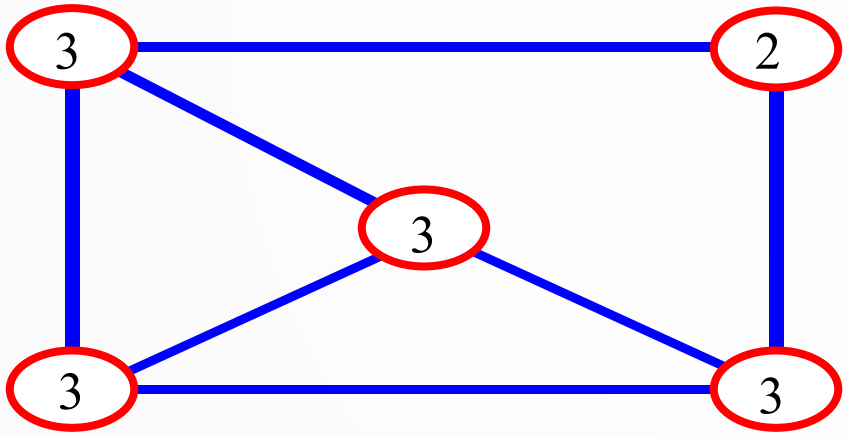
directed graph in and out degree

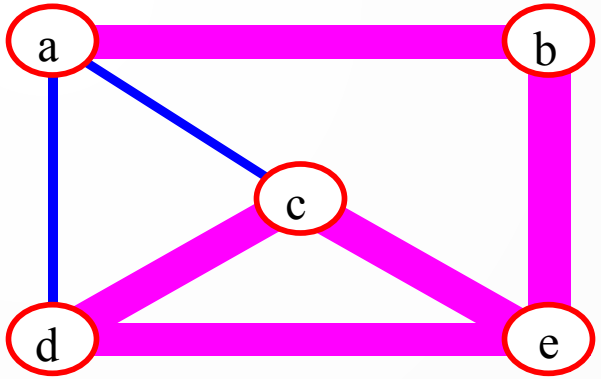
Terminology Cond …

- **Paths:** A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $Paths(G) = \{ P \in V^+ \mid 1 <= i < |P|, (P_i, P_{i+1}) \in E \}$ is the set of all paths in G, where V^+ indicates all positive length sequences of vertices (allowing for repeats). The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path. A path in a finite graph can have infinite length.
- A *simple path* is a path with no repeated vertices.

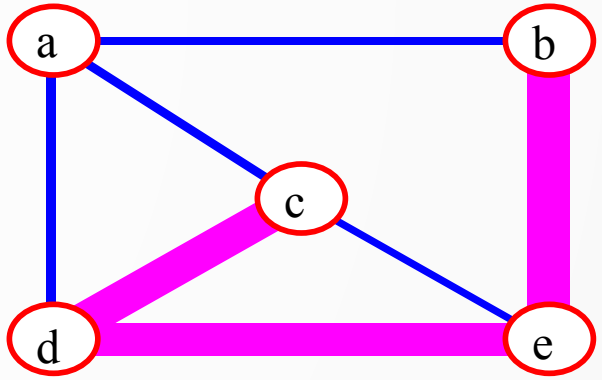
Examples

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





a b e d c

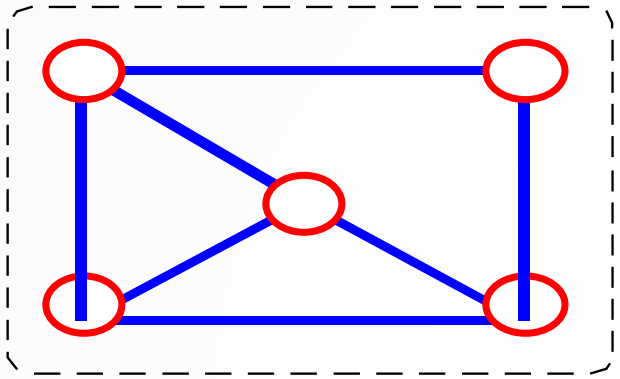


b e d c

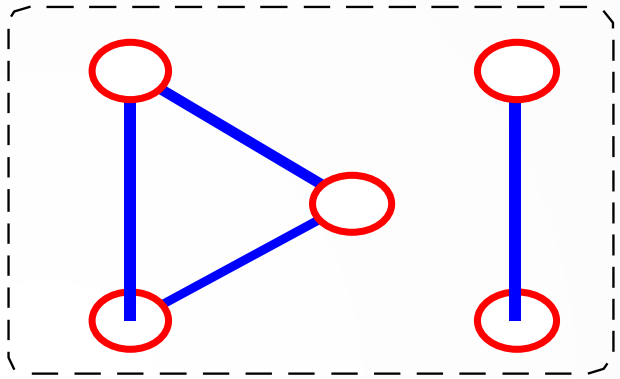
Terminology Cond …

- **Reachability and Connectivity:** A vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. We use $R_G(v)$ to indicate the set of all vertices reachable from v in G. An undirected graph is *connected* if all vertices are reachable from all other vertices. A directed graph is *strongly connected* if all vertices are reachable from all other vertices.
- **Subgraph:** subset of vertices and edges forming a graph.
- **Connected component:** maximal connected subgraph.

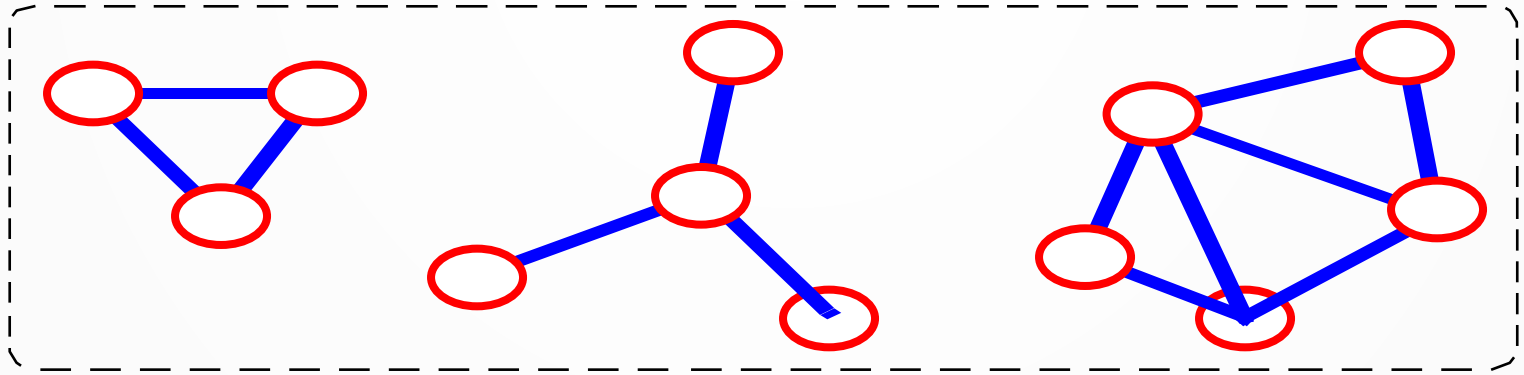
Examples



connected



not connected

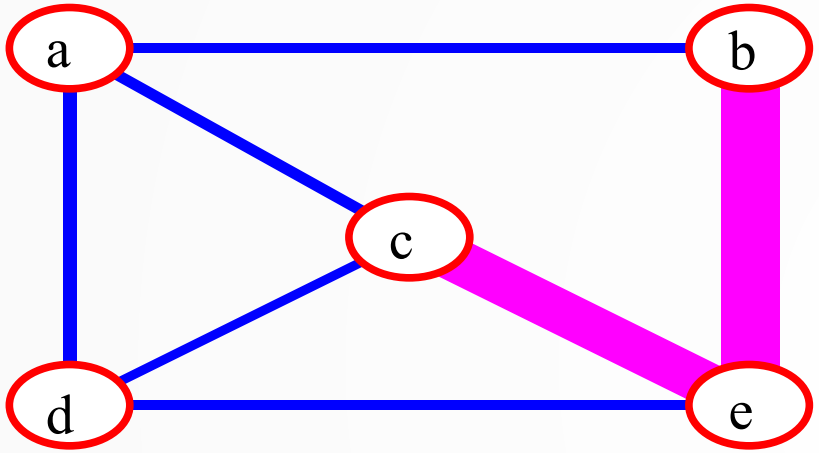


the graph above has 3 connected components.

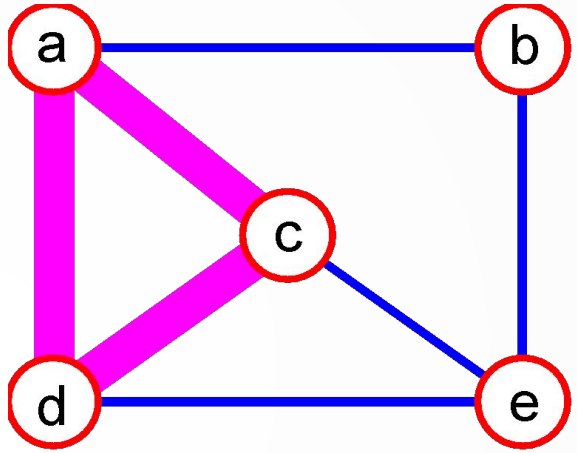
Terminology Cond …

- **Cycles:** In a directed graph a *cycle* is a path that starts and ends at the same vertex. A cycle can have length one (i.e. a *self loop*). A *simple cycle* is a cycle that has no repeated vertices other than the start and end vertices being the same. In an undirected graph a (simple) *cycle* is a path that starts and ends at the same vertex, has no repeated vertices other than the first and last, and has length at least three.

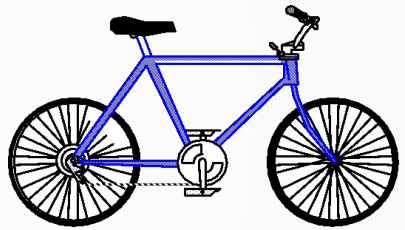
Example



b e c



a c d a



simple path: no repeated vertices

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

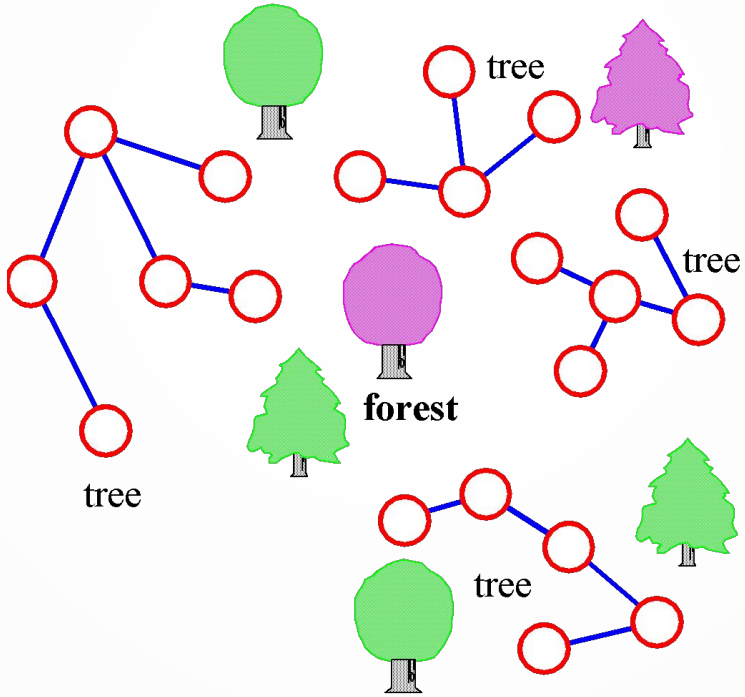
Terminology Cond …

- **Trees and forests:** An undirected graph with no cycles is a *forest* and if it is connected it is called a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

Examples

tree - connected graph without cycles

forest - collection of trees

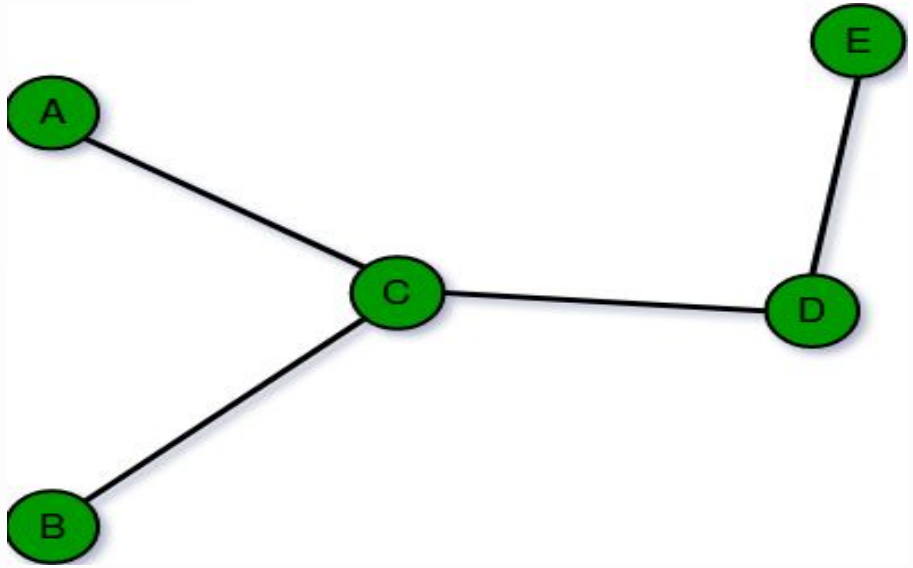


Terminology Cond …

- **Distance.** The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v. It is also referred to as the *shortest path length* from u to v.
- **Diameter.** The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $\text{diam}(G) = \max \{\delta_G(u, v) : u, v \in V\}$.

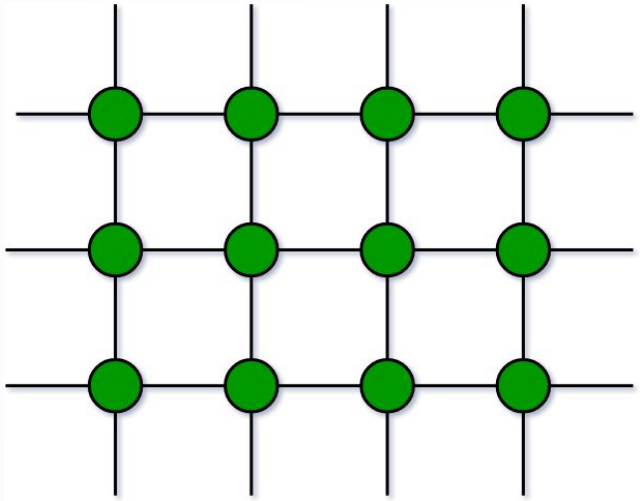
Types of Graphs

Finite Graphs has finite number of vertices and finite number of edges.



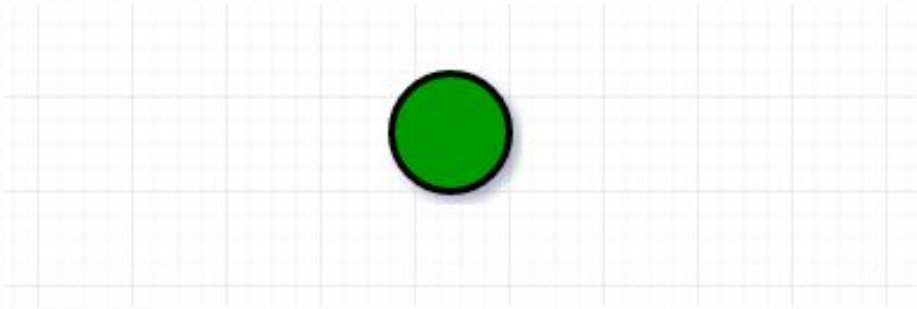
Types of Graphs Cond…

- **Infinite Graph** has infinite number of vertices as well as infinite number of edges.



Types of Graphs Cond…

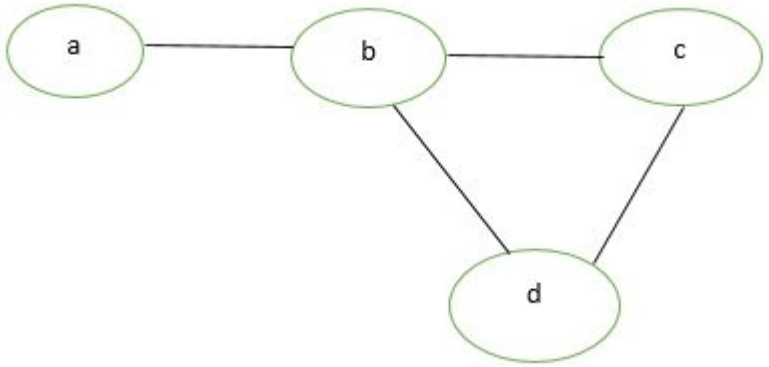
- **Trivial Graph is** a finite graph with only one vertex and no edge.

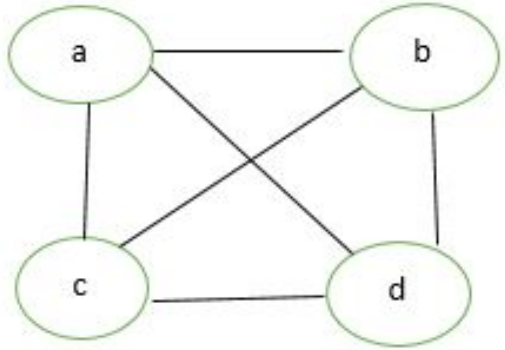


Types of Graphs Cond…

- **Simple Graph** does not contains more than one edge between the pair of vertices.

Example : A railway tracks connecting different cities





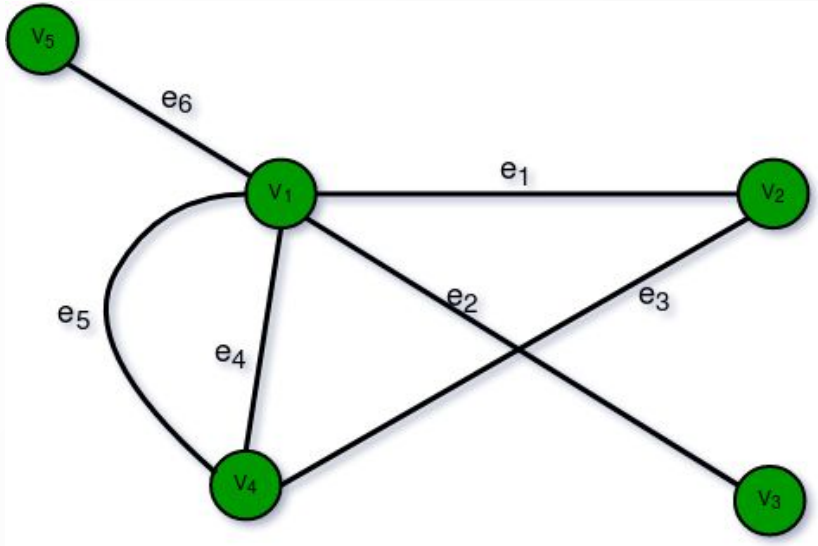
Types of Graphs Cond…

- **Multi Graph** contains some parallel edges but doesn't contain any self-loop.

  Example : A Road Map.

Parallel Edges: If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.

Loop: An edge of a graph which join a vertex to itself is called loop or a self-loop.



Types of Graphs Cond…

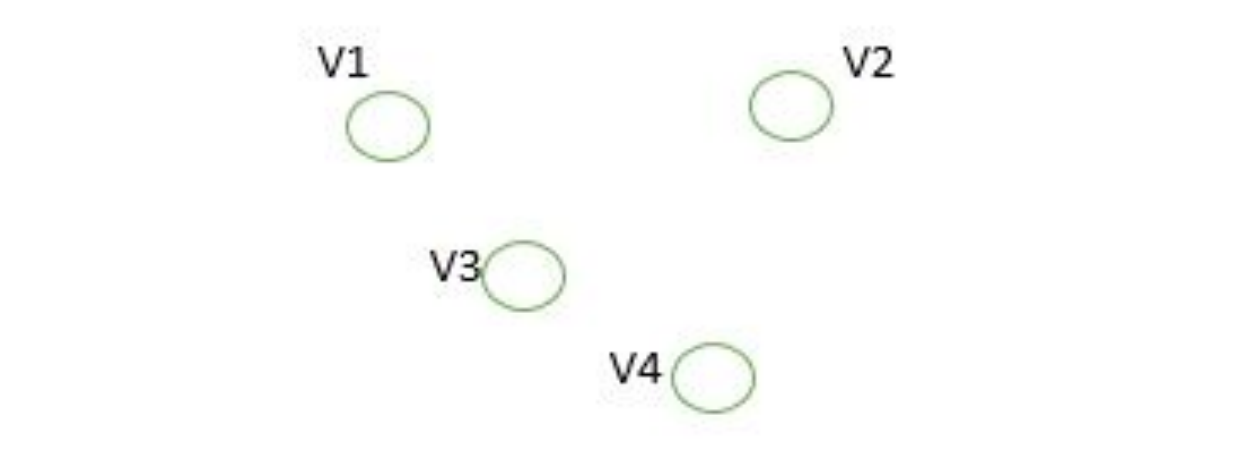
- **Sparse and dense graphs.**

  Let $n = |V|$ and $m = |E|$ then , a graph is *sparse* if $m << n^2$ and *dense* otherwise.

Types of Graphs Cond…

- **Null Graph** contains n number of vertices but do not contain any edge.

  Example : A graph of order n and size zero



Types of Graphs Cond…

Complete graph is one in which all pairs of vertices are adjacent.

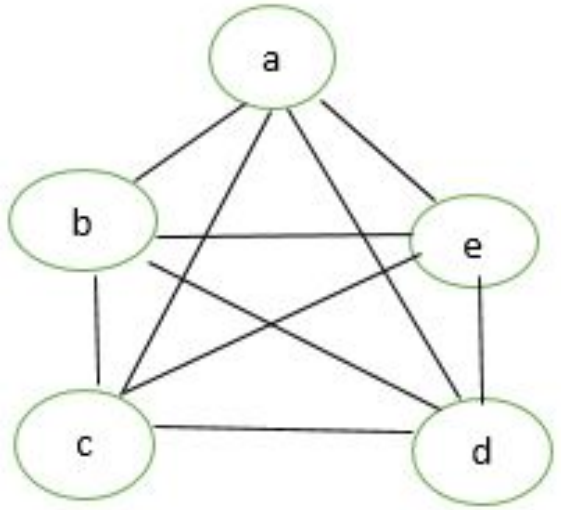
- How many *total edges in a complete graph?*
  - Each of the n vertices is incident to n-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = n(n -1)/2.

- Therefore, if a graph is not complete,
  if m < n(n -1)/2

Let n = #vertices, and m = #edges

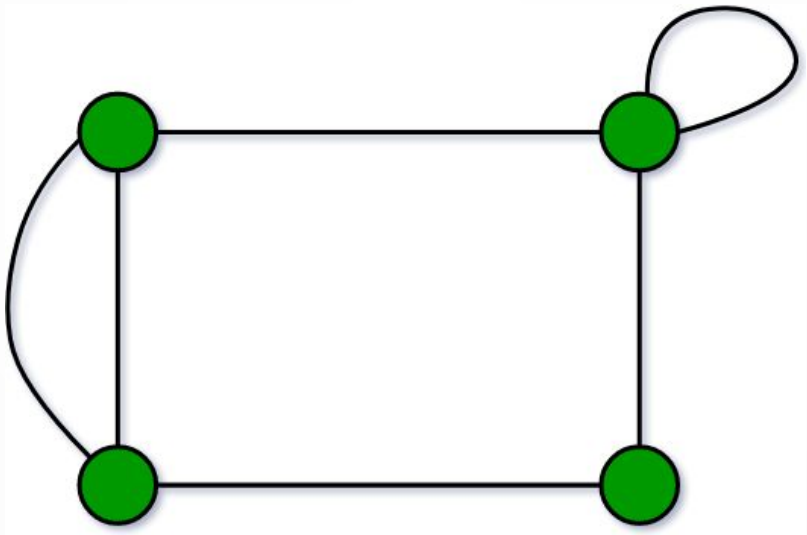
If n= 5

Then m = n*(n-1)/2 = 10



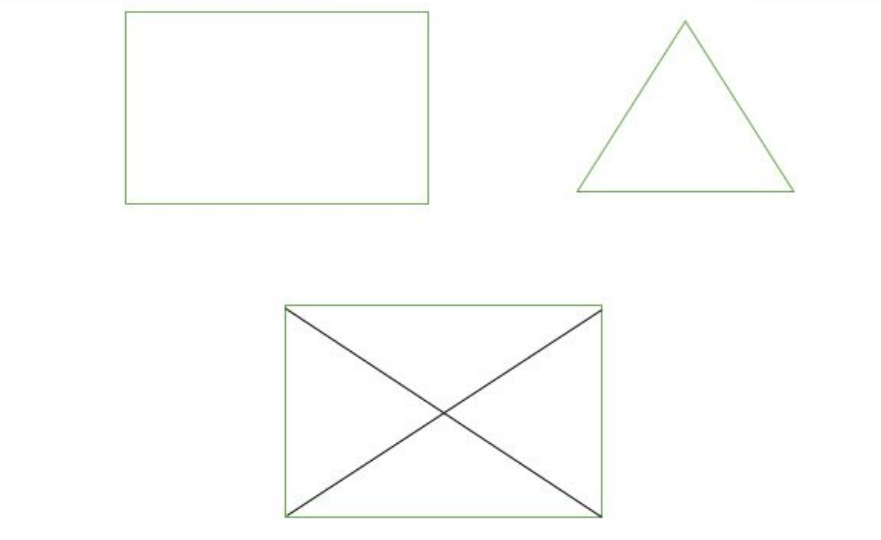
Types of Graphs Cond…

- **Pseudo Graph:** A graph G with a self loop and some multiple edges is called pseudo graph.



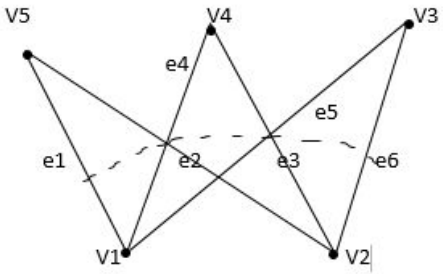
Types of Graphs Cond…

- **Regular Graph:** A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.



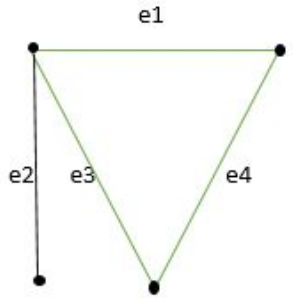
Types of Graphs Cond…

- **Bipartite Graph:** A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has its one end in V1(G) and other end in V2(G).
  The partition V1 U V2 = V is called Bipartite of G.
  Here in the figure:
  V1(G)={V5, V4, V3}
  V2(G)={V1, V2}



Types of Graphs Cond…

- **Labelled Graph:** If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called *Weighted Graph*.



Types of Graphs Cond…

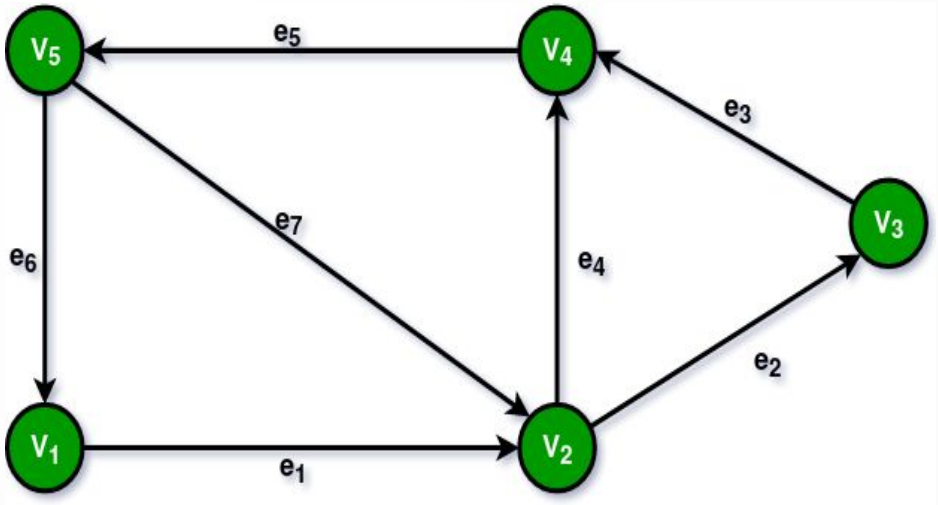
- **Digraph Graph:** A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.

  Here in the figure:

  e1 = (V1, V2)

  e2 = (V2, V3)

  e4 = (V2, V4)



Types of Graphs Cond…

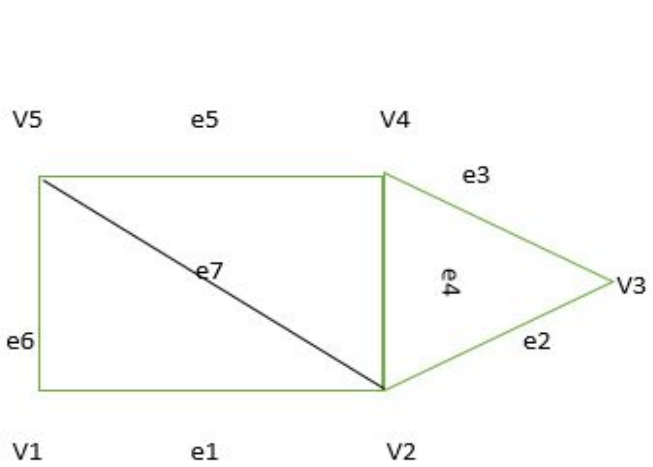
- **Subgraph:** A graph G = (V1, E1) is called subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.

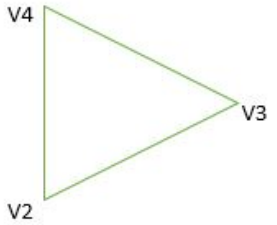
Types of Subgraph:

- **Vertex disjoint subgraph:** Any two graph G1 = (V1, E1) and G2 = (V2, E2) are said to be vertex disjoint of a graph G = (V, E) if V1(G1) intersection V2(G2) = null. In figure there is no common vertex between G1 and G2.
- **Edge disjoint subgraph:** A subgraph is said to be edge disjoint if E1(G1) intersection E2(G2) = null. In figure there is no common edge between G1 and G2.

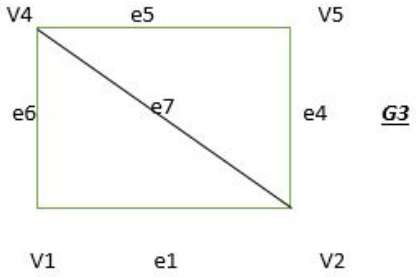
  Note: Edge disjoint subgraph may have vertices in common but vertex disjoint graph cannot have common edge, so vertex disjoint subgraph will always be an edge disjoint subgraph.

Example : Subgraph



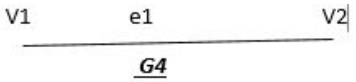


G1



G3

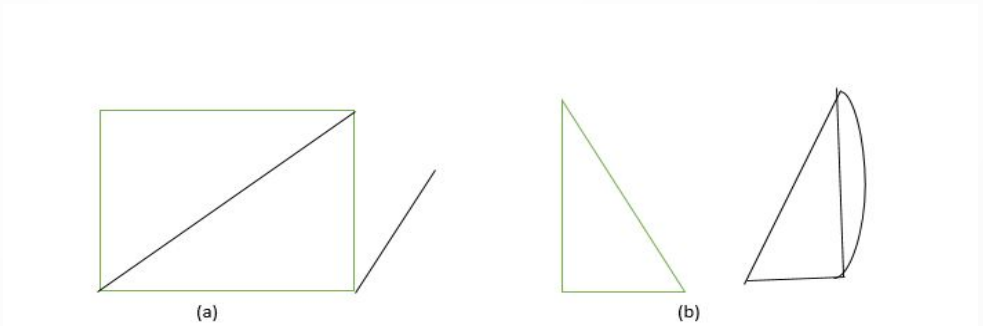
G2



G4

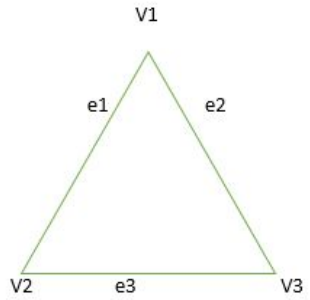
Types of Graphs Cond…

- **Connected or Disconnected Graph:** A graph G is said to be connected if for any pair of vertices (Vi, Vj) of a graph G are reachable from one another. Or a graph is said to be connected if there exist atleast one path between each and every pair of vertices in graph G, otherwise it is disconnected. A null graph with n vertices is disconnected graph consisting of n components. Each component consist of one vertex and no edge.



Types of Graphs Cond…

- **Cyclic Graph:** A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – – – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – – – – – – — -(Vn, V1) are called cyclic graph.



Representing Graphs

How we want to represent a graph largely depends on the operations we intend to support. For example we might want to do the following on a graph $G = (V, E)$:

1. Map over the vertices $v \in V$.
2. Map over the edges $(u, v) \in E$.
3. Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors. Return the degree of a vertex $v \in V$.
4. Determine if the edge (u, v) is in E.
5. Insert or delete vertices.
6. Insert or delete edges.

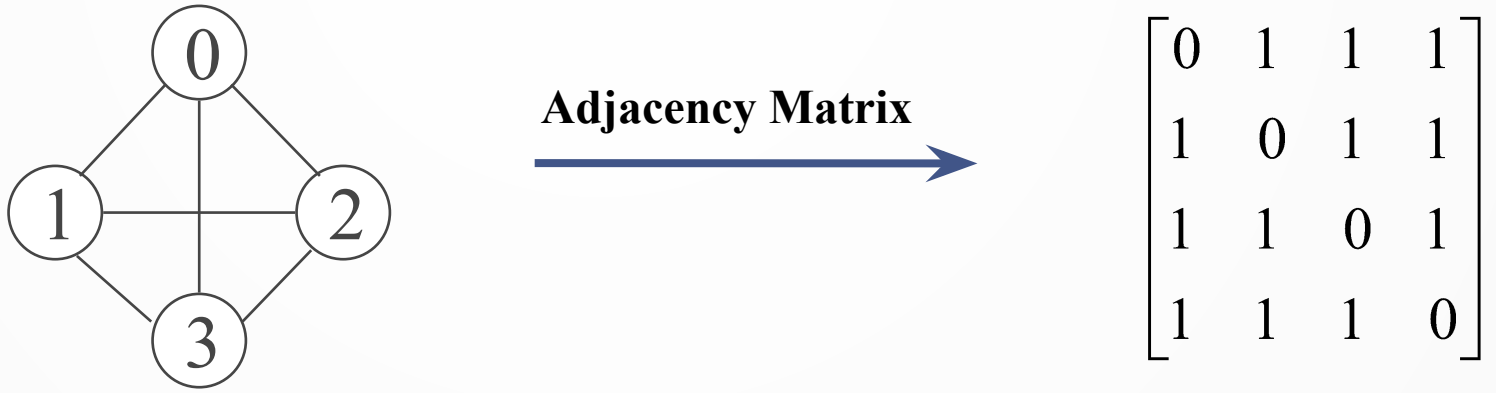
Representing Graphs Cond…

Traditionally, there are different standard representations –

1. Adjacency Matrix
2. Adjacency List

Representing Graphs Cond…

- **Adjacency Matrix :** An n *X* n matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise.
- Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.
- The main problem with adjacency matrices is their space demand of $\Theta(n^2)$. Graphs are often sparse, with far fewer edges than $\Theta(n^2)$.



Representing Graphs Cond… 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, the row sum is the out_degree, while the column sum is the in_degree

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

Representing Graphs Cond…Adjacency List

- **Adjacency List.** An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{ u, v \}$ the edge will appear in the adjacency list for both u and v.
- Adjacency lists are not well suited for parallelism since the lists require that we traverse the neighbors of a vertex sequentially.

Representing Graphs Cond…Adjacency List

Data Structure Declaration

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Graph Traversal and Graph Algorithms

- Traversal
  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
  - Breadth First Search (BFS)
    level order tree traversal
  - Depth First Search (DFS)
    preorder tree traversal
- Shortest Path Finding
- Spanning Trees

Breadth First Search (BFS)

- Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
- Breadth-first search is a way to find all the vertices reachable from a given source vertex, *s*.
- As the name BFS suggests, you are required to traverse the graph breadth wise : First move horizontally and visit all the nodes of the current level then move to the next level

Breadth First Search (BFS) : Search Procedure

1. Start at a given vertex *s*, which is at level 0.
2. Visit all the vertices that are at the distance of one edge away(adjacent), paint them as "visited," vertices and placed into level 1.
3. Visit all the new vertices we can reach at the distance of two edges away from the source vertex s. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on.
4. The BFS traversal terminates when every vertex has been visited.

The state of a vertex, *u*, is stored in a color variable as follows:

color[*u*] = White - for the "undiscovered" state,
color [*u*] = Gray - for the "discovered but not fully explored" state, and
color [*u*] = Black - for the "fully explored" state.

Breadth-First Search Traversal : Algorithm

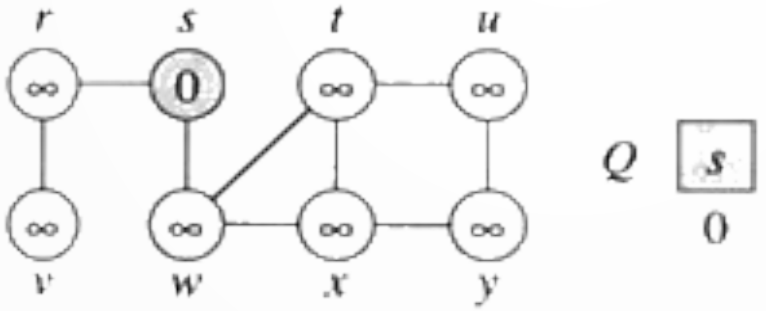
BFS(V, E, *s*) // develops a breadth-first search tree with the source vertex, *s*, as its root.

```
1.      for each u in V − {s}          ▷ for each vertex u in V[G] except s.
2.       do color[u] ← WHITE
3.          d[u] ← infinity
4.          π[u] ← NIL
5.     color[s] ← GRAY                ▷ Source vertex discovered
6.     d[s] ← 0                       ▷ initialize
7.     π[s] ← NIL                     ▷ initialize
8.     Q ← {}                         ▷ Clear queue Q
9.     ENQUEUE(Q, s)
10     while Q is non-empty
11.        do u ← DEQUEUE(Q)                ▷ That is, u = head[Q]
12.           for each v adjacent to u      ▷ for loop for every node along with edge.
13.                 do if color[v] ← WHITE  ▷ if color is white you've never seen it before
14.                       then  color[v] ← GRAY
15.                             d[ v] ← d[u] + 1
16.                             π[ v] ← u
17.                             ENQUEUE(Q,  v)
18.           DEQUUE(Q)
19.      color[u] ← BLACK
```

Breadth-First Search Traversal : Example

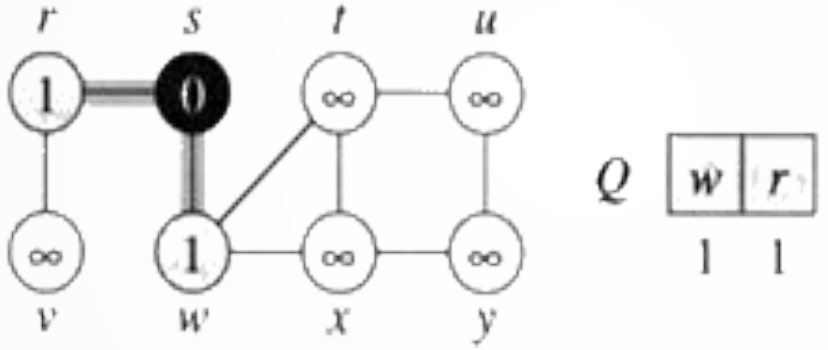
The following figure illustrates the progress of breadth-first search on the undirected sample graph.

a. After initialization (paint every vertex white, set d[*u*] to infinity for each vertex *u*, and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex *s*.

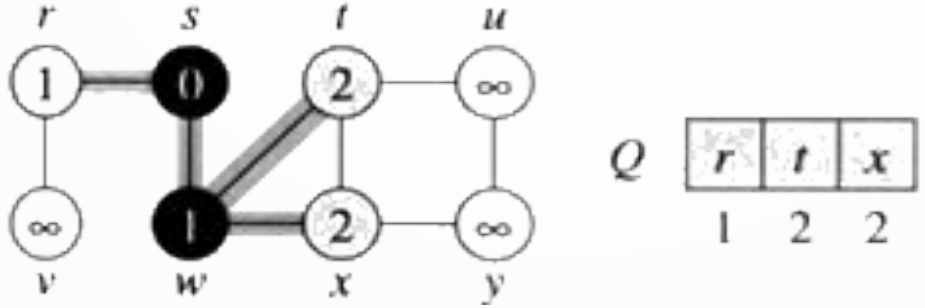


Breadth-First Search Traversal : Example Cond…

b. The algorithm discovers all vertices 1 edge from *s* i.e., discovered all vertices (*w* and *r*) at level 1.

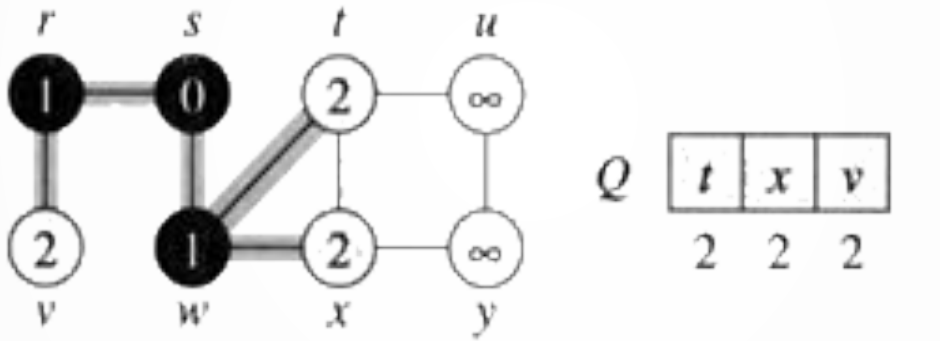


c.

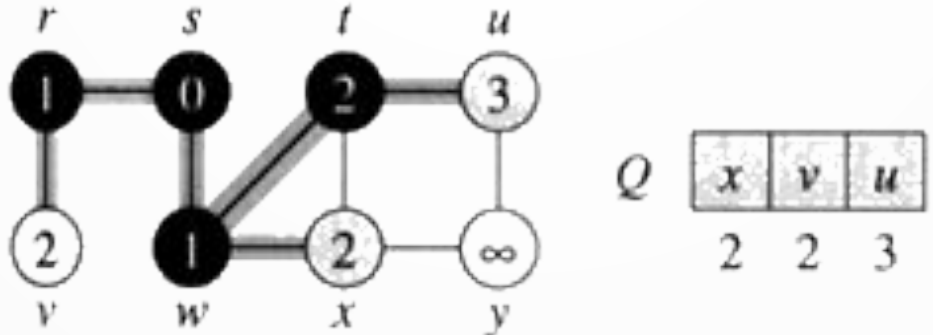


Breadth-First Search Traversal : Example Cond…

d. The algorithm discovers all vertices 2 edges from *s* i.e., discovered all vertices (*t*, *x*, and *v*) at level 2.

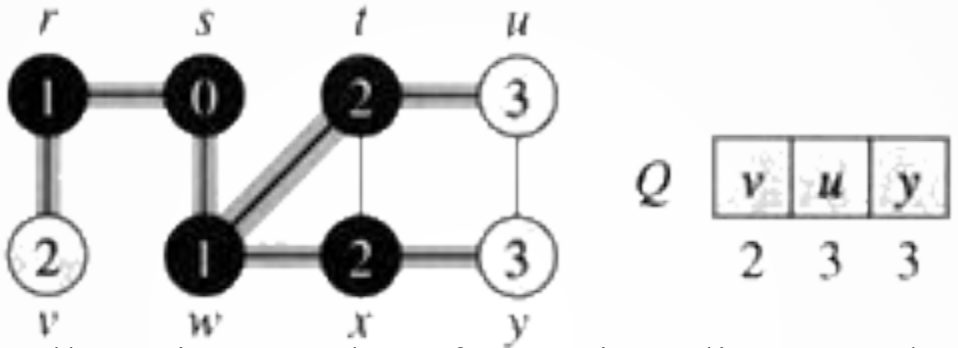


e.

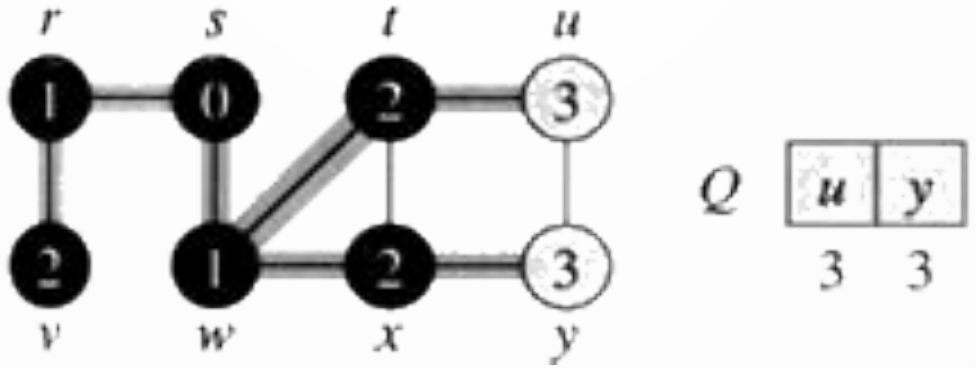


Breadth-First Search Traversal : Example Cond…

f.

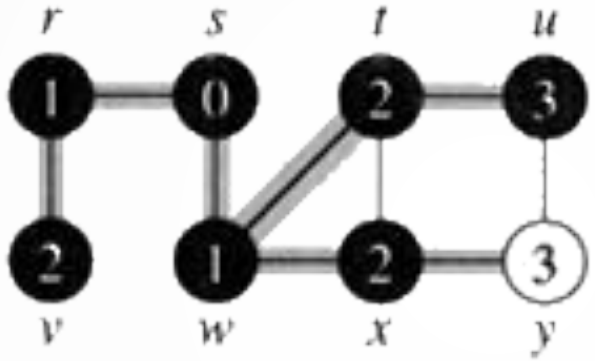


g. The algorithm discovers all vertices 3 edges from *s* i.e., discovered all vertices (*u* and *y*) at level 3.



Breadth-First Search Traversal : Example Cond…

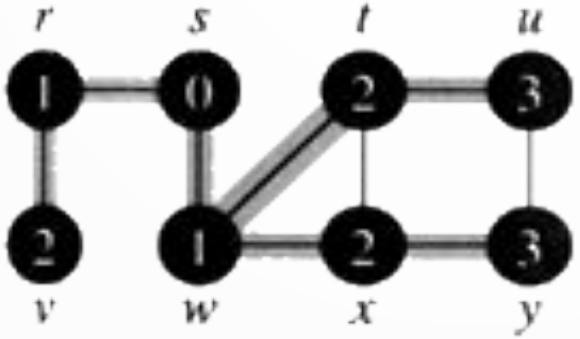
h.



Q [y]

3

i. The algorithm terminates when every vertex has been fully explored.



Q $\varnothing$

Depth First Search (DFS)

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a Boolean visited array.

Like BFS, to keep track of progress depth-first-search colors each vertex. Each vertex of the graph is in one of three states:

- Undiscovered
- Discovered but not finished (not done exploring from it); and
- Finished (have found everything reachable from it) i.e. fully explored.

Depth First Search (DFS) : Search Procedure

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

Depth First Search (DFS) : Search Procedure Cond …

The state of a vertex, *u*, is stored in a color variable as follows:

1. color[u] = White - for the "undiscovered" state,
2. color[u] = Gray - for the "discovered but not finished" state, and
3. color[u] = Black - for the "finished" state.

Like BFS, depth-first search uses π[v] to record the parent of vertex v.

We have π[v] = NIL if and only if vertex v is the root of a depth-first tree.

DFS time-stamps each vertex when its color is changed.

1. When vertex *v* is changed from white to gray the time is recorded in d[*v*].
2. When vertex *v* is changed from gray to black the time is recorded in f[*v*].

The discovery and the finish times are unique integers, where for each vertex the finish time is always after the discovery time. That is, each time-stamp is an unique integer in the range of 1 to 2|V| and for each vertex *v*, d[*v*] < f[*v*]. In other words, the following inequalities hold:

$$1 \leq d[v] < f[v] \leq 2|V|$$

Depth-First Search Traversal : Algorithm

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

DFS (V, E)

```
1.  for each vertex u in V[G]
2.    do color[u] ← WHITE
3.          π[u] ← NIL
4.  time ← 0
5.  for each vertex u in V[G]
6.    do if color[u] ← WHITE
7.          then DFS-Visit(u)
```

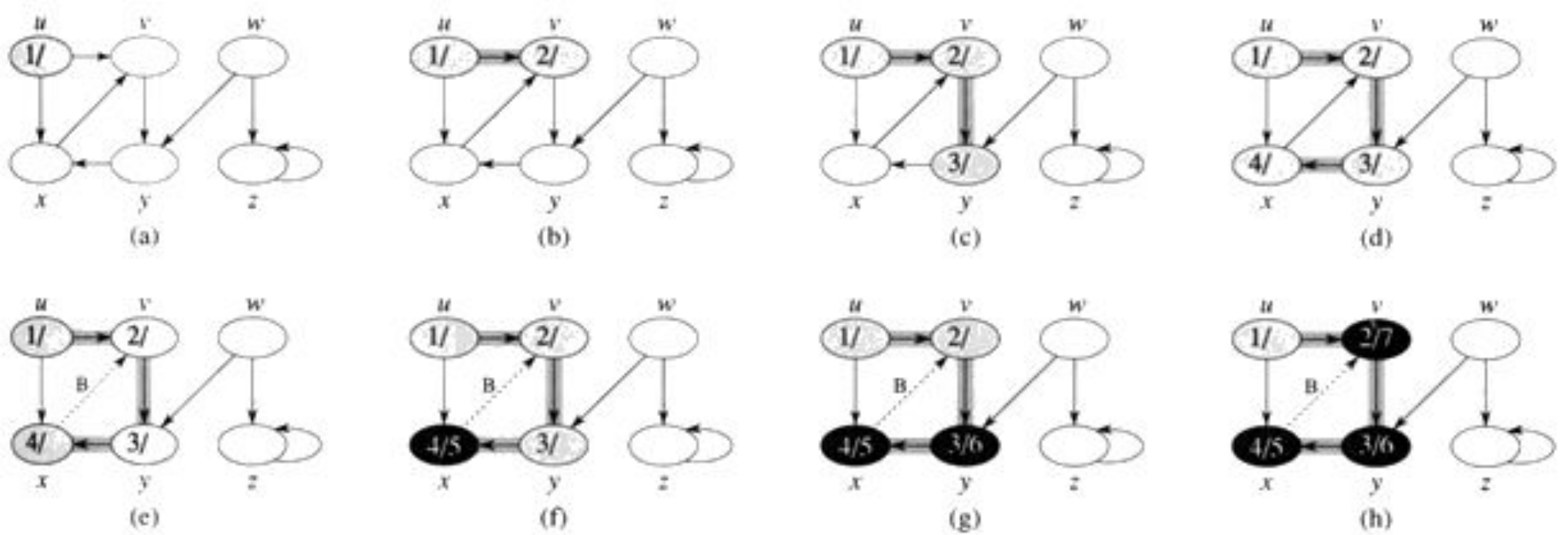
Depth-First Search Traversal : Algorithm Cond…

DFS-Visit(u)

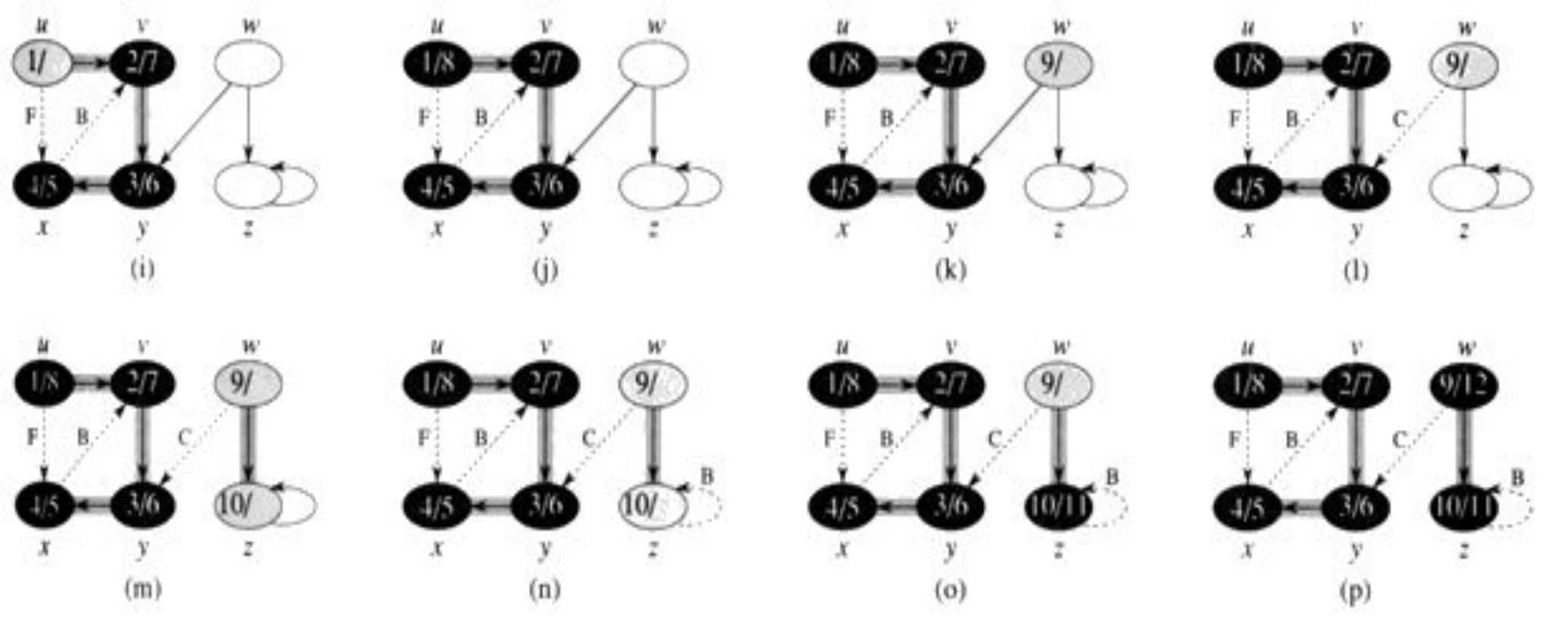
```
1.  color[u] ← GRAY                          ▷ discover u
2.  time ← time + 1
3.  d[u] ← time
4.  for each vertex v adjacent to u     ▷ explore (u, v)
5.     do if color[v] ← WHITE
6.           then π[v] ← u
7.                DFS-Visit(v)
8.  color[u] ← BLACK
9.  time ← time + 1
10. f[u] ← time                              ▷ we are done with u
```

Example

In the following figure, the solid edge represents discovery or tree edge and the dashed edge shows the back edge. Furthermore, each vertex has two time stamps: the first time-stamp records when vertex is first discovered and second time-stamp records when the search finishes examining adjacency list of vertex.

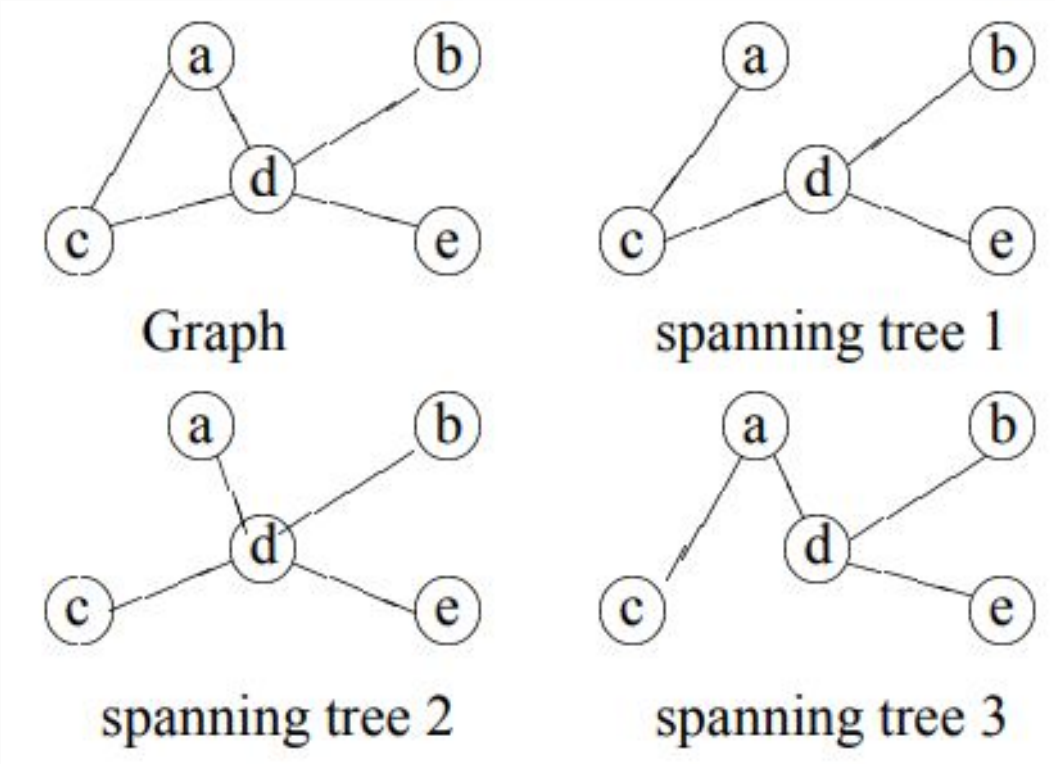


Example Cond…



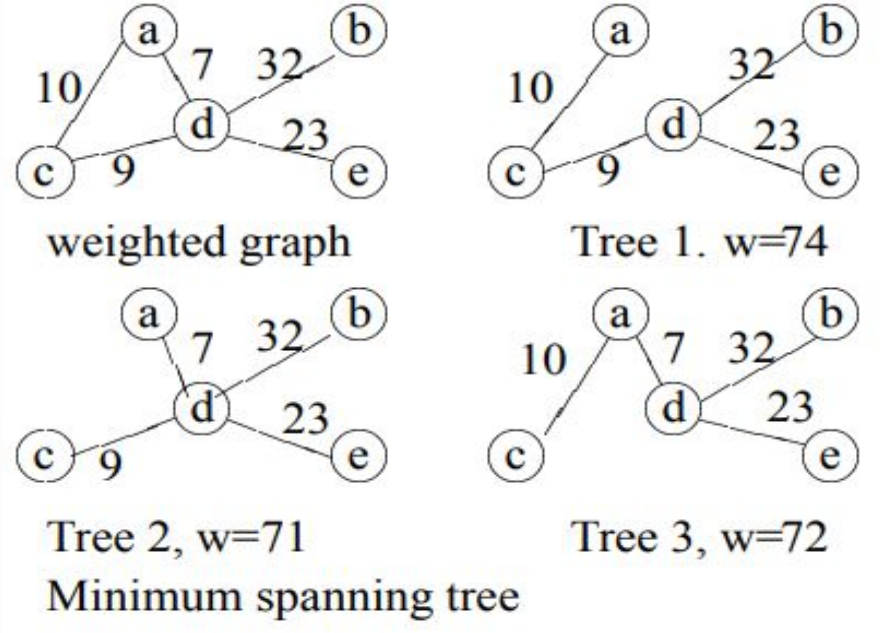
Spanning Tree

- Spanning Trees: A sub graph T of a undirected graph G= {V,E } is a spanning tree if it is a tree and contains every vertex of G.



Minimum Spanning Tree (MST)

- A Minimum Spanning Tree in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).



Remark: The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique.

MST Problem

- MST Problem: Given a connected weighted undirected graph G , design an algorithm that outputs a minimum spanning tree (MST) of G.
- Generic approach: A tree is an acyclic graph. The idea is to start with an empty graph and try to add edges one at a time, always making sure that what is built remains acyclic. And if we are sure every time the resulting graph always is a subset of some minimum spanning tree, we are done.

Generic MST

- Let A be a set of edges such that $A \subseteq T$, where T is a MST. An edge (u,v) is a safe edge for A, if $A \cup \{(u,v)\}$ is also a subset of some MST. If at each step, we can find a safe edge , we can 'grow' a MST. This leads to the following generic approach:

- Generic-MST(G, w) :
  Let A=EMPTY;
  while A does not form a spanning tree find an edge (u, v) that is safe for A add (u, v).
  return A

* Safe edge :

Let G = (V,E) be a connected, undirected graph with a real-valued weight function w defined on E . Let A be a subset of E that is included in some minimum spanning tree for G, let (S, S-V) be any cut of G that respects A , and let (u,v) be a light edge crossing the cut (S,S-V). Then, edge (u,v) is safe for A.

Prim's Algorithm

The generic algorithm gives us an idea how to 'grow' a MST. However the choice of a cut (and hence the corresponding light edge) in each iteration is immaterial. We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed. The Prim's algorithm makes a nature choice of the cut in each iteration – it grows a single tree and adds a light edge in each iteration.

Prim's Algorithm Cond…

Grow a Tree

- Start by picking any vertex r to be the root of the tree.
- While the tree does not contain all vertices in the graph find shortest edge leaving the tree and add it to the tree .

Running time is O((|V|+|E|)log|V|)

Prim's Algorithm Cond…

The idea:

expand the current tree by adding the lightest (shortest) edge leaving it and its endpoints.

- Step 0: Choose any element r ; set S = {r} and A = □

  (Take r as the root of our spanning tree.)
- Step 1: Find a lightest edge such that one endpoint is in S and the other is in V\S . Add this edge to A and its (other) endpoint to S.
- Step 2: If V\S = □ , then stop & output (minimum) spanning tree (S,A) . Otherwise go to Step 1.

Prim's Algorithm Cond…

Remark: G is given by adjacency lists. The vertices in $V \setminus S$ are stored in a priority queue with key=value of lightest edge to vertex in S.

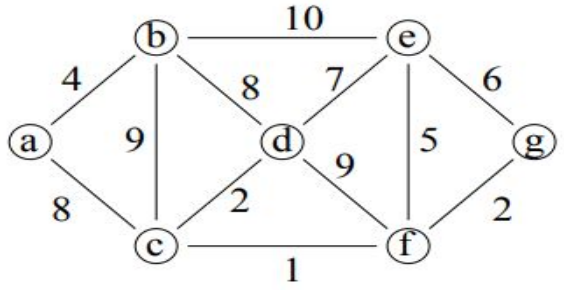
```
Prim(G, w, r)
{   for each u ∈ V                                  initialize
    {   key[u] = +∞;
        color[u] = W;
    }
    key[r] = 0;                                     start at root
    pred[r] = NIL;
    Q = new PriQueue(V);                            put vertices in Q
    while(Q is nonempty)                            until all vertices in MST
    {   u=Q.extraxtMin();                           lightest edge
        for each (v ∈ adj[u])
        {   if ((color[v] == W)&&(w[u, v] < key[v]))
            key[v] = w[u, v];                       new lightest edge
            Q.decreaseKey(v, key[v]);
            pred[v] = u;
        }
        color[u] = B;
    }
}
```

When the algorithm terminates, $Q = \emptyset$ and the MST is

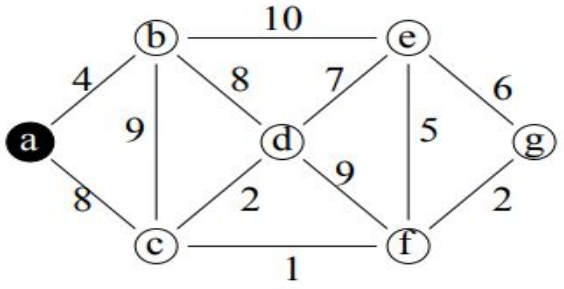
$$T = \{\{v, pred[v]\} : v \in V \setminus \{r\}\}.$$

The pred pointers define the MST as an inverted tree rooted at r.

Prim's Algorithm : Example Cond…



Connected graph

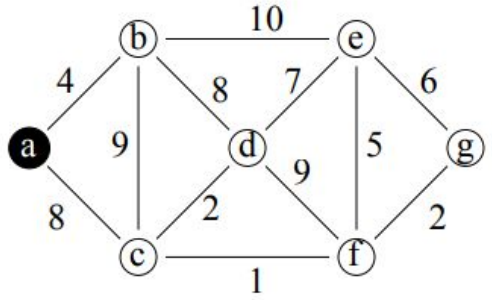


Step 0

S={a}

V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}



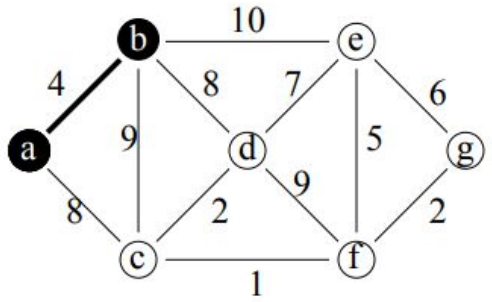
Step 1.1 before

S={a}

V \ S = {b,c,d,e,f,g}

A={}

lightest edge = {a,b}



Step 1.1 after

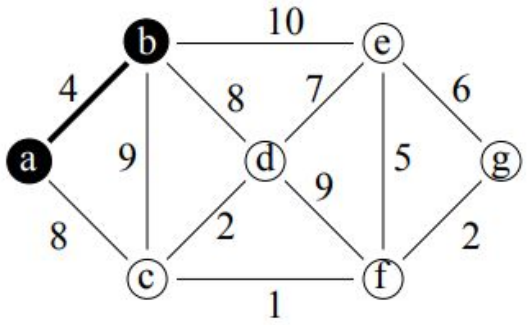
S={a,b}

V \ S = {c,d,e,f,g}

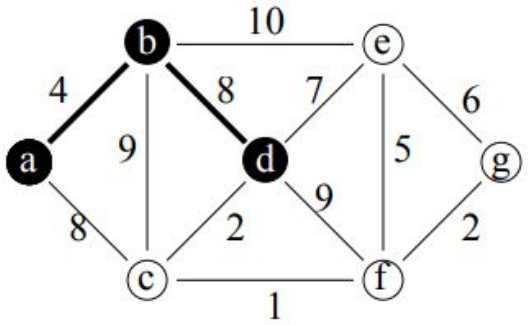
A={{a,b}}

lightest edge = {b,d}, {a,c}

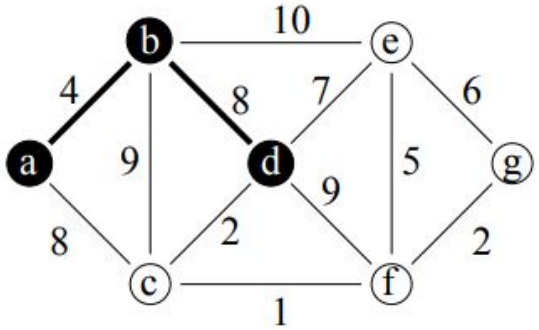
Prim's Algorithm : Example Cond…



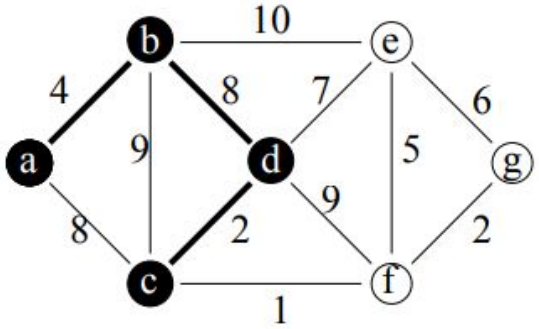
Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}



Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

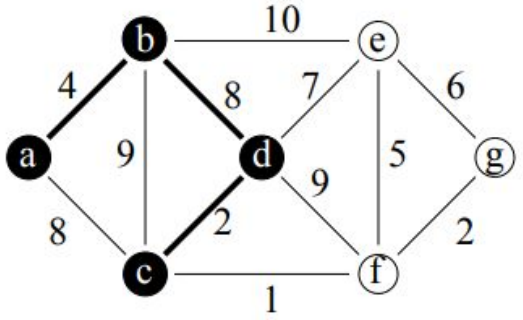


Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

Prim's Algorithm : Example Cond…



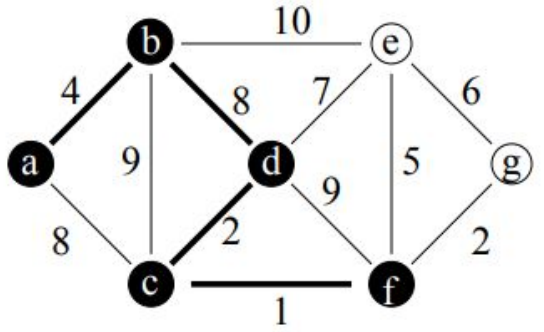
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



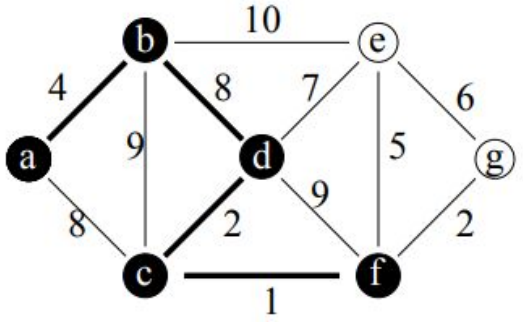
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



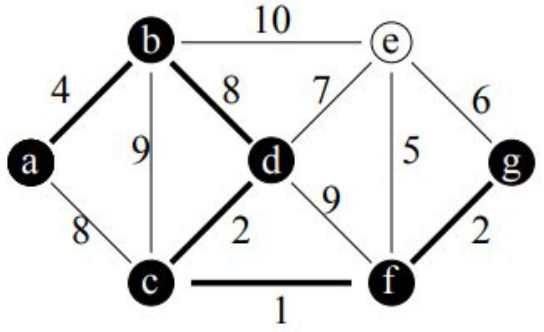
Step 1.4 after

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

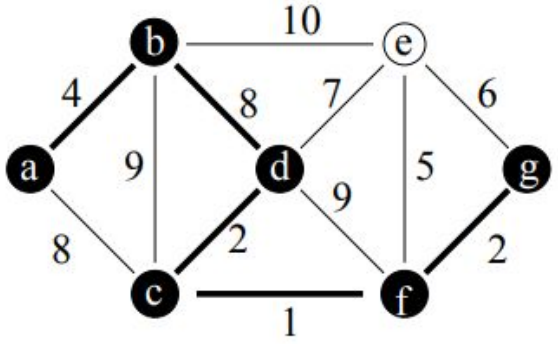
S={a,b,c,d,f,g}

V \ S = {e}

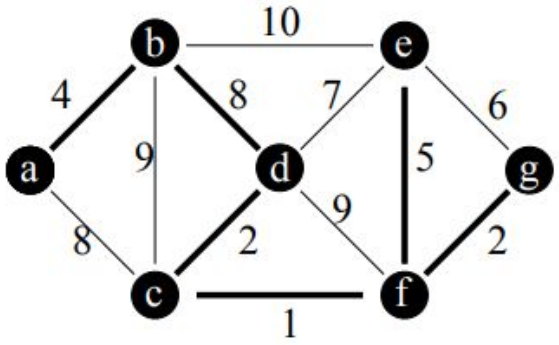
A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

Prim's Algorithm : Example Cond…



Step 1.6 before
S={a,b,c,d,f,g}
V \ S = {e}
A={{a,b},{b,d},{c,d},{c,f},
{f,g}}
lightest edge = {f,e}



Step 1.6 after
S={a,b,c,d,e,f,g}
V \ S = {}
A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}
MST completed

Analysis of Prim's Algorithm

Let n = |V| and e = |E|.

1. O(log n) to extract each vertex from the queue.
   Done once for each vertex = O(n log n).

2. O(log n) to decrease the key value of neighboring vertex.
   Done at most once for each edge = O(e log n)

   Total cost = O ((n+e) log n)

Kruskal's Algorithm

The Kruskal's Algorithm is based directly on the generic algorithm. Unlike Prim's algorithm, we make a different choices of cuts. Initially, trees of the forest are the vertices (no edges). In each step add the cheapest edge that does not create a cycle. Observe that unlike Prim's algorithm, which only grows one tree, Kruskal's algorithm grows a collection of trees (a forest). Continue until the forest 'merge to' a single tree. This is a minimum spanning tree

Outline of Algorithm by Example

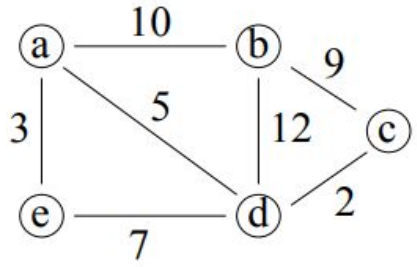
Step 0: Set $A = \emptyset$ and $F = E$, the set of all edges.

Step 1: Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.

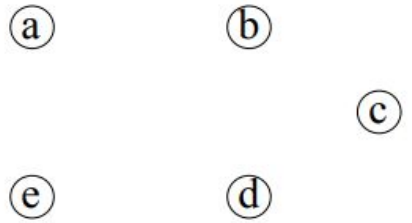
- If "yes", remove e from F.
- If "no", move e from F to A.

Step 2: If $F = \emptyset$, stop and output the minimal spanning tree (V, A). Otherwise go to Step 1.

Remark: Will see later, after each step, (V, A) is a subgraph of a MST.



original graph



forest ⟶ MST

E

edge	weight
{d, c}	2
{a, e}	3
{a, d}	5
{e, d}	7
{b, c}	9
{a, b}	10
{b, d}	12

Forest (V, A)

A={ }

Kruskal's Algorithm : Key Concepts

Question: How does algorithm choose edge e □ F with minimum weight?

Answer: Start by sorting edges in E in order of increasing weight. Walk through the edges in this order. (Once edge causes a cycle it will always cause a cycle so it can be thrown away

Question : How to check for cycles ?

Answer :Observation: At each step of the outlined algorithm, (V,A) is acyclic so it is a forest. If u and v are in the same tree, then adding edge (u,v) to A creates a cycle. If u and v are not in the same tree, then adding edge (u,v) to A does not create a cycle.

Question: How to test whether u and v are in the same tree?

Answer: Use a disjoint-set data structure Vertices in a tree are considered to be in same set. Test if Find-Set(u) = Find-Set(v)?

Kruskal's Algorithm : the details

Sort E in increasing order by weight w; $O(|E| \log |E|)$

/* After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_{|E|}, v_{|E|}\} \rangle$ */

$A = \{\}$;

for (each u in V) CREATE-SET(u); $O(|V|)$

for $e_i = (u_i, v_i)$ from 1 to $|E|$ do $O(|E| \log |V|)$

    if (FIND-SET(u_i) != FIND-SET(v_i))

    { add $\{u_i, v_i\}$ to A;

      UNION(u_i, v_i);

    }

return(A);

Remark: With a proper implementation of UNION-FIND, Kruskal's algorithm has running time O(|E|log|E|

References

1. *Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]*
2. *Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp.*
3. Adam Drozdek, Data Structures and Algorithms in C++ (2nd Edition), 2001
4. Data Structures A Pseudocode Approach with C, Second Edition by Richard F. Gilberg Behrouz A. Forouzan
5. https://www.geeksforgeeks.org/