Data Structures & Programming

An Introduction to Graph Algorithms

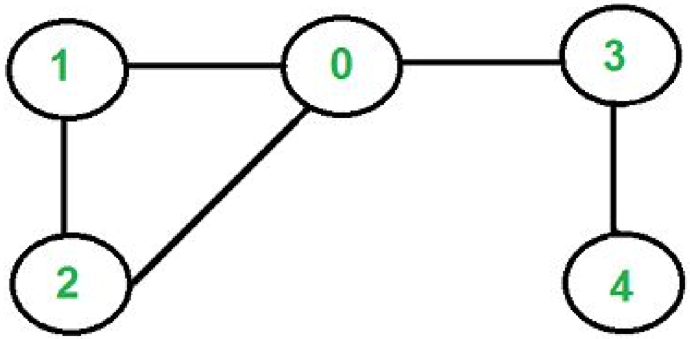
Golnar Sheikhshab

Simple Graphs

A **simple graph.**

Nodes: {0, 1, 2, 3, 4}

Edges: {{0,1},{0,2},{0,3},{1,2},{3,4}}

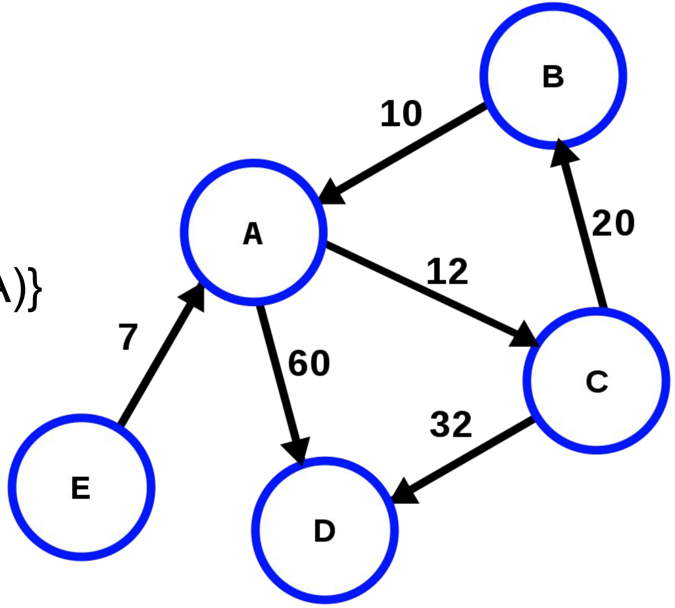


Directed or Weighted Graphs

A **directed** and **weighted** graph.

Nodes: {A, B, C, D, E}

Edges: {(A,C), (A,D), (B,A), (C,B), (C,D), (E, A)}



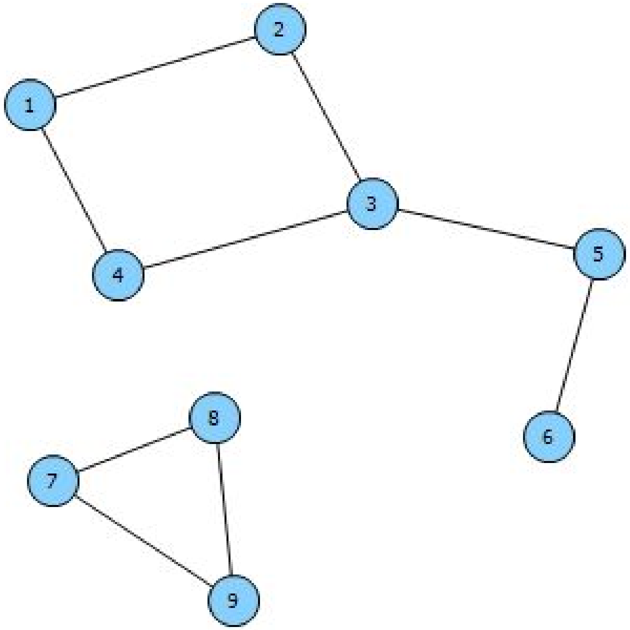
Path, Cycles, and Connected Components

There are two **path**s from 1 to 6:
Path1: ({1,2}, {2,3}, {3,5},{5,6})
Path2: ({1,4}, {4,3}, {3,5},{5,6})
There is no **path** from 3 to 7.

(1,2,3,4) forms a **cycle**

This graph is not **connected**.

connected components of the graph:
{1,2,3,4,5,6} and
{7,8,9}

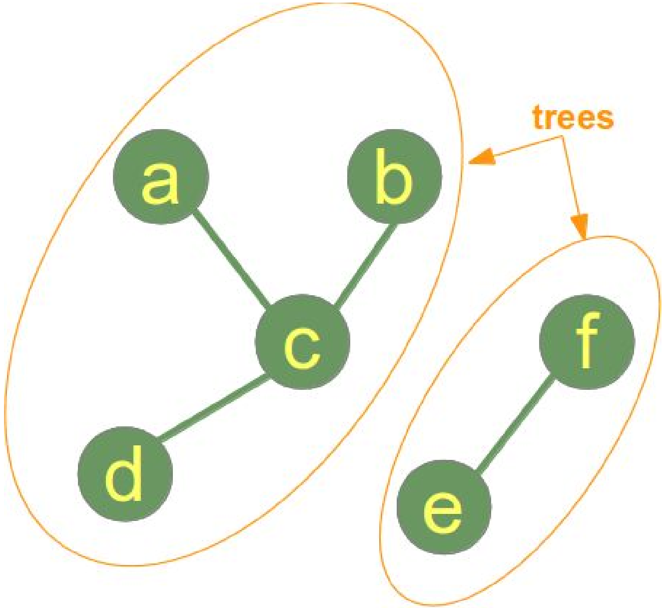


Tree and Forest

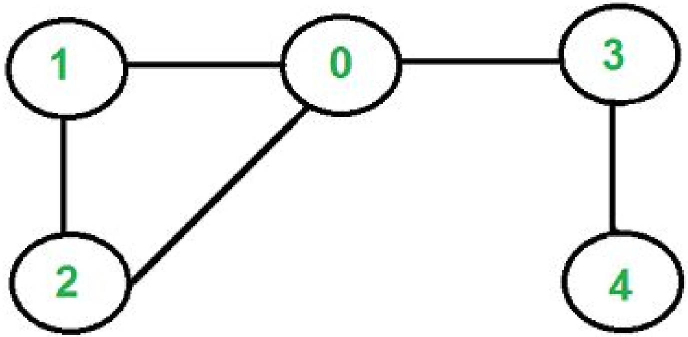
Tree: a simple connected graph with no cycles

Forest: a simple graph whose connected components have no cycles

General trees and binary trees that we have seen previously are rooted trees with directions imposed on them.



Storing the graph



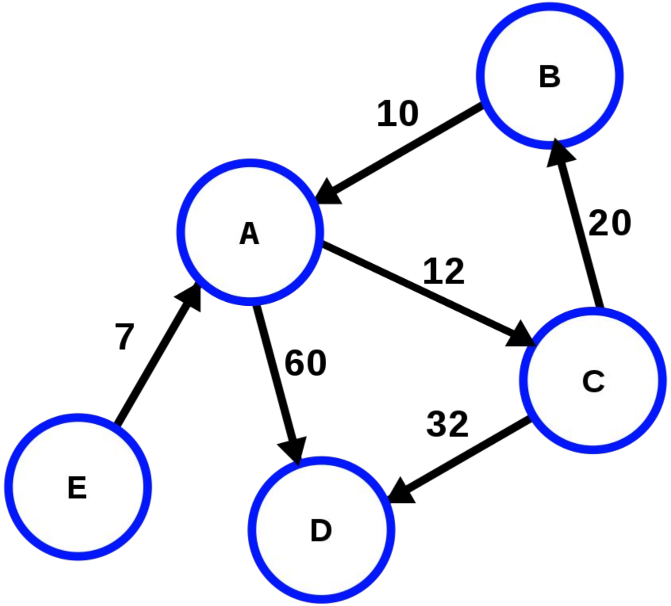
Adjacency list:

0: {1,2,3,}
1: {0,2}
2: {0,1}
3: {0,4}
4:{3}

Adjacency matrix:

0 1 1 1 0
1 0 1 0 0
1 1 0 0 0
1 0 0 0 1
0 0 0 1 0

Storing the graph



Adjacency list:

- A: ((C,12),(D,60))
- B: ((A,10))
- C: ((B,20),(D,32))
- D: ()
- E: ((A,7))

Ids:

- A: 0
- B: 1
- C: 2
- D: 3
- E: 4

Adjacency matrix:

00	00	12	60	00
10	00	00	00	00
00	20	00	32	00
00	00	00	00	00
07	00	00	00	00

labels:

(A, B, C, D, E)

Graph Traversal (or search)

Depth first search (DFS)

Breadth first search (BFS)

We start from a node and try to either visit all nodes or find a specific node.

Depth First Search (DFS)

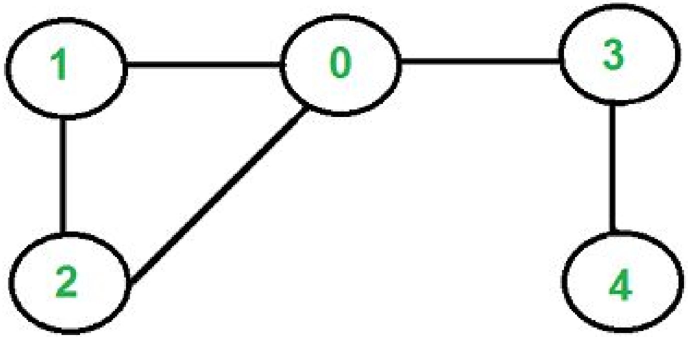
Algorithm DFS_Traversal(G,v):

Input: A graph G (stored as adjacency list) and a vertex v of G
Output: A sequence of vertices in dfs traversal order started at v

label v as visited
ret ← concatenate(ret,(v))
for all **unvisited** vertices w in G[v] do
    DFS(G,w)

DFS(G,0):

ret (global variable)	v	w	visited (global variable)
(0)	0	1	{0}
(0,1)	1	2	{0,1}
(0,1,2)	2	-	{0,1,2}
(0,1,2)	1	-	{0,1,2}
(0,1,2)	0	3	{0,1,2}
(0,1,2,3)	3	4	{0,1,2,3}
(0,1,2,3,4)	4	-	{0,1,2,3,4}
(0,1,2,3,4)	3	-	{0,1,2,3,4}
(0,1,2,3,4)	0	-	{0,1,2,3,4}

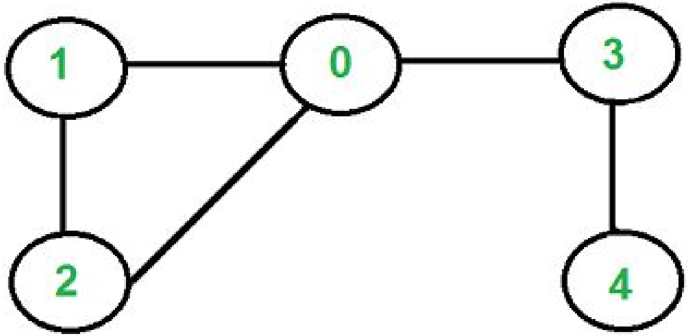


Breadth First Search (BFS)

```
Algorithm BFS_Traversal(G,v):
    Input: A graph G (stored as adjacency list) and a vertex v of G
    Output: A sequence of vertices in bfs traversal order started at v

    declare q as a queue of vertices
    q.enqueue(v)
    label v as visited
    while q is non-empty
        u ← q.front()
        q.dequeue()
        ret ← concatenate( ret, (u) )
        for all unvisited vertices w in G[u] do
            q.enqueue(w)
            label w as visited
```

BFS example



BFS(G,0):

ret	q	visited
()	(0)	{0}
(0)	(1,2,3)	{0,1,2,3}
(0,1)	(2,3)	{0,1,2,3}
(0,1,2)	(3)	{0,1,2,3}
(0,1,2,3)	(4)	{0,1,2,3,4}
(0,1,2,3,4)	()	{0,1,2,3,4}

Do It Yourself

DFS(G,1)?

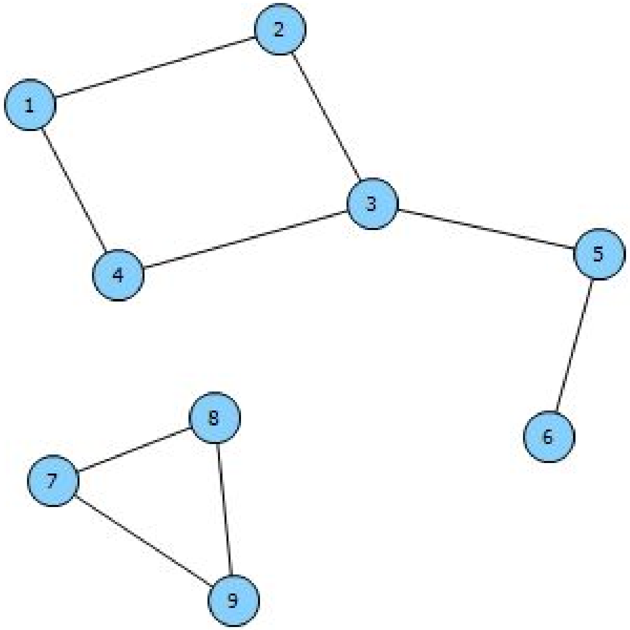
DFS(G,3)?

DFS(G,7)?

BFS(G,1)?

BFS(G,3)?

BFS(G,7)?



Give ideas to answer these questions

A **simple** graph G and two vertices u and v are given.

1. Is there a path between v and u?

2. Are v and u in the same connected components?

3. How many connected components are there?

4. How to find a path between v and u?

5. How to find the shortest path between v and u?

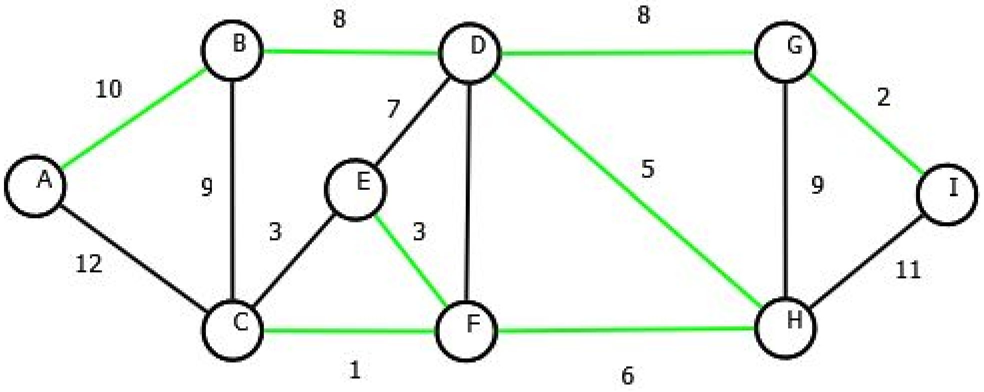
Minimum Spanning Tree

Defined for weighted undirected graphs

It's a tree

It spans all the vertices

Sum of its edge weights is minimum



Minimum Spanning Tree

Which of the following is a spanning tree?

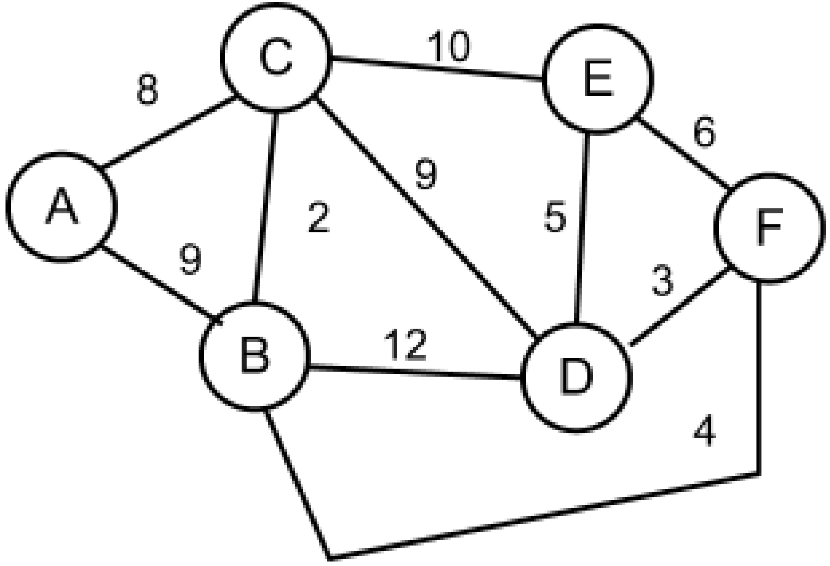
1. {(A,C), (C,E), (C,D), (E,F)}

2. {(A,C), (C,E), (C,D), (E,D)}

3. {(A,B), (A,C), (C,E), (C,D), (E,F)}

4. {(B,C), (D,F), (B,F), (D,E), (A,C)}

5. {(B,C), (D,F), (B,F), (D,E), (E,F)}



Kruskal algorithm for minimum spanning tree

Repeatedly pick the edge with minimum weight unless it causes a cycle

How can we check if an edge will cause a cycle in a graph that doesn't already have one?

Look at the pseudo-code in the next slide and give the time complexity of the algorithm.

Algorithm Kruskal(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

  Define an elementary cluster $C(v) \leftarrow \{v\}$.

Initialize a priority queue Q to contain all edges in G, using the weights as keys.

$T \leftarrow \emptyset$ $\{T$ will ultimately contain the edges of the MST$\}$

while T has fewer than $n-1$ edges **do**

  $(u,v) \leftarrow Q$.removeMin()

  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

Code Fragment 13.25: Kruskal's algorithm for the MST problem.