Competitive Programming Week 6

Introduction to graphs

Membership sign up:

JUNIOR EXEC APPLICATIONS OPEN

Go to link below or scan QR code to apply.
The deadline is OCTOBER 31st at midnight.

https://forms.gle/8dkGw1Zezfm4BVnRA

SCAN ME

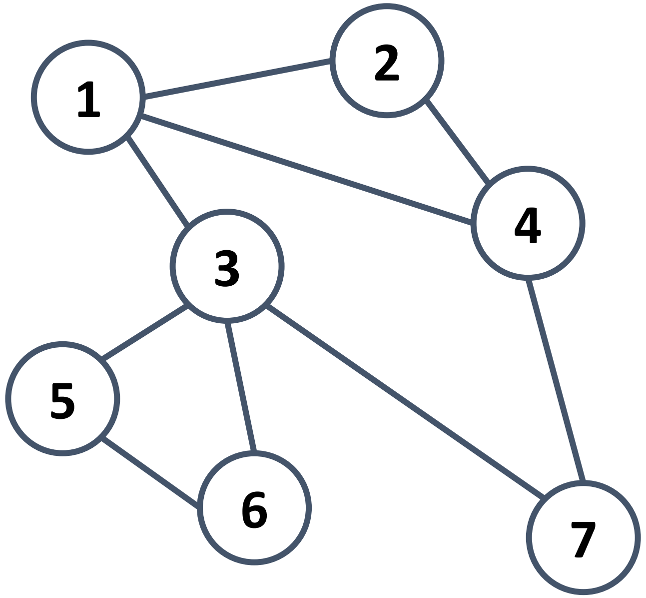
Alberta Collegiate Programming Contest (ACPC):

This year ACPC will take place on November 25th
We will be hosting a location here at U of C
Sign up begins next week! November 1st

Practice Contests Leaderboard

- Tied for first place with 37 problems solved:
  - **Alex Chen, Martin L**
- In third place with 35 problems solved:
  - **Quwsar Ohi**
- In fourth place with 34 problems solved:
  - **Nathan Weiss**
- In fifth place with 32 problems solved:
  - **Jimmy Xu**
- In sixth place with 30 problems solved:
  - **Max McEvoy**
- In seventh place with 29 problems solved:
  - **Mikhail Singh**
- Tied for eighth place with 22 problems solved:
  - **Anh Nguyen, Khoa Nguyen, Hy Huynh, Nguyễn Bá Khánh Tùng**

Graphs



A Graph consists of a set of nodes/vertices and a set of edges connecting said nodes. Mathematically written as G = (V, E) where V is the vertex set and E is the edge set

A graph can be directed or undirected (default):

- In an **undirected** graph edges are bidirectional i.e. you can go from vertex i to j if there exists an edge (i, j) or (j, i) in E.
- In a **directed** graph edges are directional/“one way” i.e. you can go from vertex i to j iff there is an edge (i, j) in E.

A **path** from vertex i to vertex j is a sequence of edges
$(i, k_1), (k_1, k_2), \dots, (k_n, j)$

A graph is **connected** iff there exists a path from any vertex to any other vertex in the graph

A subset U of V is a **component** iff there exists a path from any vertex to any other vertex in U

Graphs

A path is a **cycle** if it starts at a vertex v and ends back at v

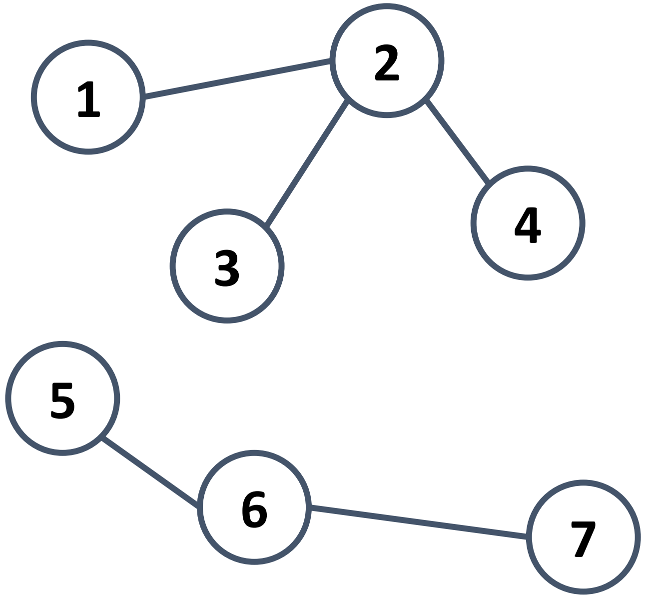
A graph is **acyclic** if there are no cycles in it

A **tree** is a connected acyclic graph

A graph is a **forest** iff every component in it is a tree

A graph can be **weighted** i.e. each edge has a cost associated with it

The **degree** of a vertex v is the number of vertices that connect to it



Graph Representations

COMPETITIVE
PROGRAMMING
CLUB

Graph adjacency list

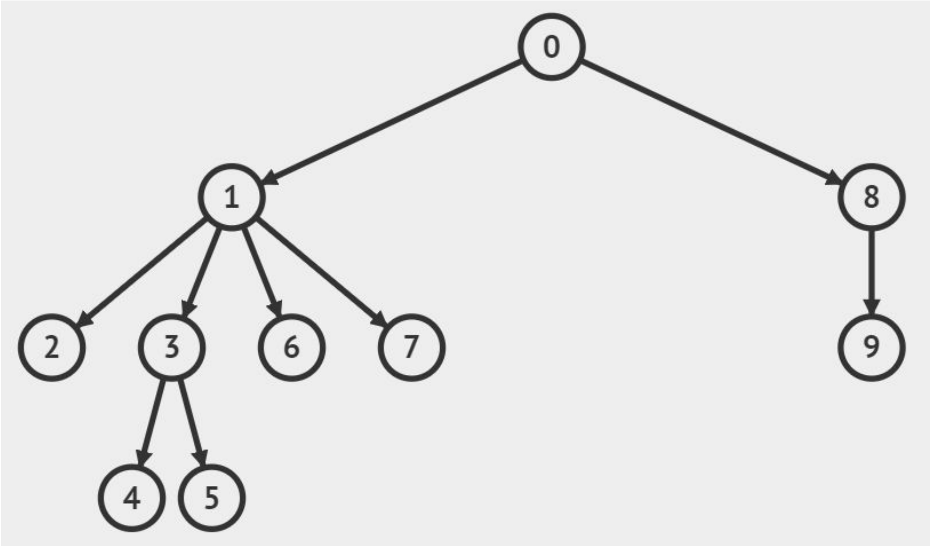
- For each node store each its neighbours in a list called an "adjacency list"
  - List of lists in Python, or vector of vectors in C++

Pros:

- We can easily and efficiently traverse each node's children by traversing its list of neighbours
- Very space efficient. Storage of the vertex and edge data is minimal

Cons:

- Checking if an edge exists is worst case O(n)
- Inverting a directed graph is expensive
- For weighted graphs, the weights will have to be stored elsewhere



Adjacency List				
0:	1	8		
1:	2	3	6	7
2:				
3:	4	5		
4:				
5:				
6:				
7:				
8:	9			
9:				

Graph adjacency matrix

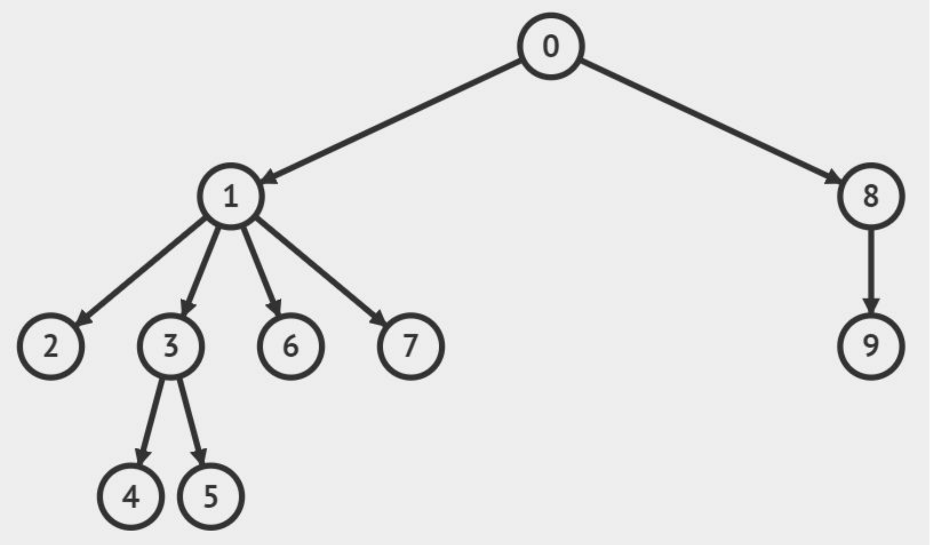
- Define a nxn array A where A[i][j] = 1 if there is an edge from i to j, 0 otherwise.
    - List of lists in Python, or vector of vectors in C++

Pros:

- Inverting directed graphs is easy (just swap the indices)
- You can store weights right in the matrix (replace 1 with the edge weight be careful if weights can be 0)
- Checking if an edge exists is O(1)

Cons:

- Traversals are slower since we need to check if there is an edge between the current node and every other node
- For sparse graphs (graphs with few edges) a lot of extra information is stored



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	1	1	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	0

Graphs as pointers

- Define a class/struct etc. representing a node with a list of pointers to neighbouring nodes. For undirected connected graphs storing a reference to any node is sufficient

Pros:

- Can be useful for small trees

Cons:

- Hard/slow to create
- Nodes take a lot of space + time to instantiate
- Can only represent connected graphs

Binary Tree:

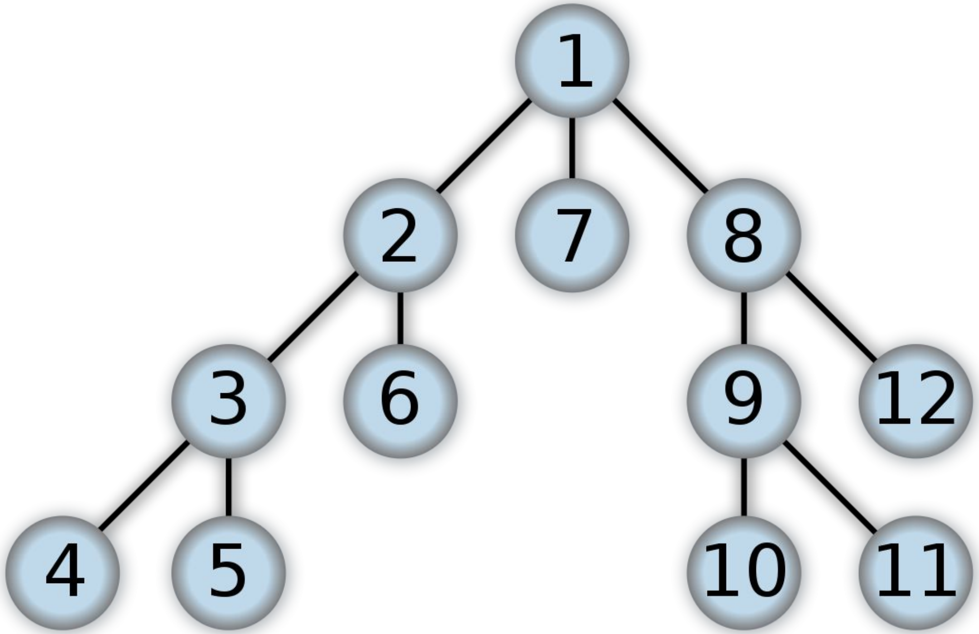
```
struct Node {
	int val;
	Node* left;
	Node* right;
};
```

Graph Traversals

COMPETITIVE
PROGRAMMING
CLUB

Graph Traversals

A traversal is a way of going from node to node in a connected graph/component. Very useful for problems where you have to visit every node or need to search for a node



Binary Tree Traversals

Traversals over binary trees have very simple recursive implementations.
There are 3 traversal patterns:

- **Pre-order:** visit the node, then its left child, then its right child
- **In-order:** visit the node's left child, then it, then its right child
- **Post-order:** visit the node's left child, then its right child, then it

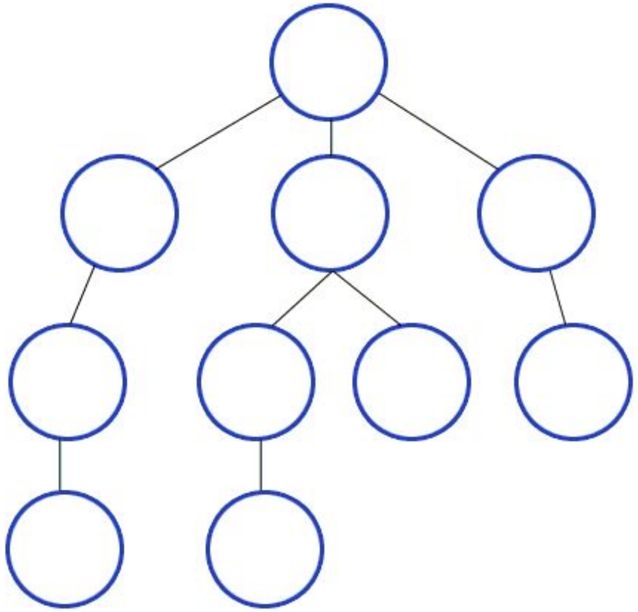
```
def preOrderTraversal(node)
    visit(node)
    preOrderTraversal(node.left)
    preOrderTraversal(node.right)
```

```
def inOrderTraversal(node)
    inOrderTraversal(node.left)
    visit(node)
    inOrderTraversal(node.right)
```

```
def postOrderTraversal(node)
    postOrderTraversal(node.left)
    postOrderTraversal(node.right)
    visit(node)
```

Say you have a binary search tree and want to print the values in sorted order.
What traversal should you use?

Depth-first search



Depth-first search (DFS) is a graph traversal algorithm that acts on an arbitrary connected graph prioritizing depth (i.e. fully explore a path before backtracking)

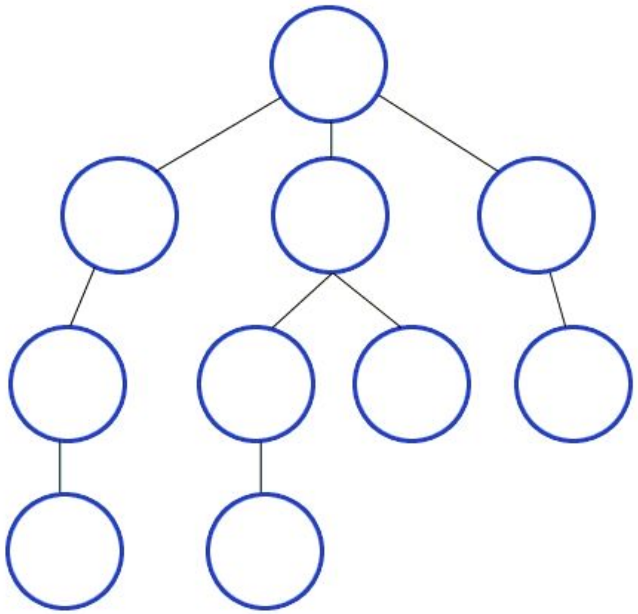
DFS has a simple recursive description:

```
def dfs(node:Node, seen:Set<Node>)
    if (seen.contains(node))
        return
    seen.add(node)
    for (Node n : node.neighbours)
        dfs(n, seen)
    return
```

Alternate Iterative implementation

```
def dfs(start:Node)
    Set<Node> seen
    Stack<Node> toVisit
    toVisit.push(start)
    while (!toVisit.isEmpty())
        Node next = toVisit.pop()
        if (seen.contains(next))
                continue;
        seen.add(next)
        for (node n : next.neighbours)
                toVisit.push(n)
```

Breadth-first search



Breadth-first search (BFS) is a graph traversal algorithm that acts on an arbitrary connected graph prioritizing breadth (i.e. explore paths simultaneously)

```
def bfs(start:Node)
    Set<Node> seen
    Queue<Node> toVisit
    toVisit.enqueue(start)
    while (!toVisit.isEmpty())
        Node next = toVisit.dequeue()
        if (seen.contains(next))
                continue;
        seen.add(next)
        for (node n : next.neighbours)
                toVisit.enqueue(n)
```

Note 1: The only difference in implementation is using a queue instead of a stack
Note 2: The first time we reach a node it will the the shortest path from the starting vertex to it

Today’s Contest:

https://open.kattis.com/contests/c74hx9/

(or look up “CPC Fall 2023 Practice Contest Week 6” in the Kattis contest list)

Feel free to ask questions until 7pm, and then throughout the week on Discord!

COMPETITIVE
PROGRAMMING
CLUB