Basics

Terminology

- nodes
- edges

Path : leads from node a to node b through edges

Length : number of edges in a path

Cycle path : if path has the same first and last node

Simple path : if path has no cycles (each node at most once)

Connected : if there is a path between any two nodes

Component : connected part of a graph

Tree : n nodes and $n - 1$ edges (unique path between any two nodes)

Directed : if edges can be traversed in only one direction

Weighted : each edge has a weight (edge length)

Neighbors / adjacent : there is an edge between two nodes (**degree** is the number of neighbors)

Sum of degrees is always twice the number of edges (always even)

Regular : degree is always the same for all nodes

Complete : every node connects to every other node

In a directed graph:

- indegree is the number of incoming edges
- outdegree is the number of outgoing edges

Colorings

No adjacent node must have the same color.

A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges.

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple.

Representation

Adjacency list representation

Each node stores a list of its neighbors.

An array of vectors (directed) : `vector<int> adj[N];`

where N is the number of nodes.

Undirected : add each edge in both directions

Weighted graph : `vector<pair<int,int>> adj[N];`

Node a contains the pair (b,w) always when there is an edge from node a to node b with weight w.

```
for (auto u : adj[s]) {
    // process node u
}
```

Adjacency matrix representation

Two-dimensional array that indicates which edges the graph contains.

`int adj[N][N];`

We can efficiently check from an adjacency matrix if there is an edge between two nodes.

Not weighted : either 1 or 0

Weighted : contains weight of edge

Cons:

- place (N^2) elements in memory

Edge list representation

An edge list contains all edges of a graph in some order.

`vector<pair<int,int>> edges;`

Good, if the algorithm processes all edges of the graph.

where each pair (a,b) denotes that there is an edge from node a to node b.

Weighted : `vector<tuple<int,int,int>> edges;`

Graph traversal

- depth-first search
- breadth-first search

Both:

- start with a node
- visit all nodes reachable from the start node

Difference is in the order of visited nodes.

Depth-first search

- proceeds to all nodes reachable from the current node in depth-first order.
- always a single path, then backtrack.
- keeps track of visited nodes.

- each node only once
- O(n + m)

Implemented using recursion

```
vector<int> adj[N];
bool visited[N];

void dfs(int s) {
    if (visited[s])
        return;

    visited[s] = true;
    // process node s

    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

Breadth-first search

- visits the nodes in increasing order of their distance from the starting node.
- we can conveniently calculate distances from the starting node.
- O(n + m)

```
queue<int> q;
bool visited[N];
int distance[N];

visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front();
    q.pop();
    // process node s

    for (auto u : adj[s]) {
        if (visited[u])
            continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

Applications

Connectivity check

Run either search and check the visited array.

We can also find all connected components. Iterate through the nodes and start a new depth-first search if the current node does not belong to any component yet.

Finding cycles

A graph contains a cycle if:

- during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited
- a component contains *c* nodes, and contains *c or more* edges

Bipartiteness check

If we can color the graph using two colors, then it is bipartite.

The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite.

General case is difficult (even k=3 is NP-hard).