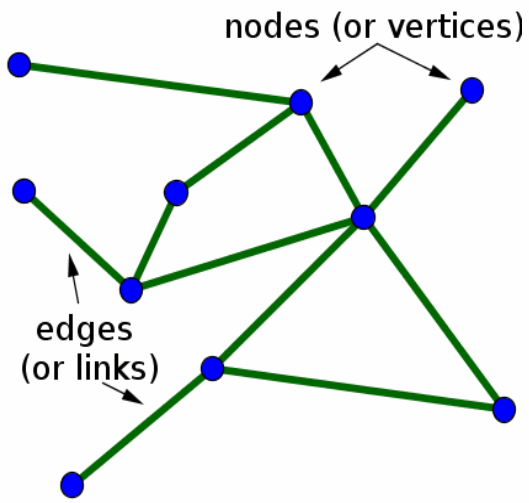
Graph Theory

Topics

1. Graph Definition
2. Graph Representation
3. Graph Traversals
4. Topological Sort
5. Shortest Path Algorithms
6. Graph Coloring
7. Bipartite Graphs
8. Disjoint Set Data Structure [Union-Find (or) Merge-Find]
9. Minimum Spanning Tree
10. C++, Java & Python Programs on Graphs
11. Problems & Solutions
12. Important References

Graph Definition

Informally, a graph is a diagram consisting of points, called **vertices** (aka **nodes**), joined together by lines, called **edges**; each edge joins exactly *two* vertices.



A graph $G = (V,E)$ consists of a (finite) set denoted by V, or by $V(G)$ if one wishes to make clear which graph is under consideration, and a collection E, or $E(G)$, of unordered pairs $\{u,v\}$ of distinct elements from V. Each element of V is called a **vertex** or a **point** or a **node**, and each element of E is called an **edge** or a **line** or a **link**.

Formally, a graph G is an ordered pair of disjoint sets (V,E), where $E \subseteq V \times V$. **Set V** is called the ***vertex*** or ***node set***, while **set E** is the ***edge set*** of **graph G**. Typically, it is assumed that *self-loops* (*i.e.*, edges of the form (u,u), for some $u \in V$) are not contained in a graph.

Walk, Trail & Path

A **Walk** is a list of vertices and edges $(v_0\, e_1\, v_1\, e_2\, \ldots\, e_k\, v_k)$, such that for $1 \le i \le k$, the edge e_i connects v_{i-1} and v_i. A **Trail** is a walk with no repeated edge. A **Simple Path** is a walk with no repeated vertices (hence no repeated edges). A **Cycle** is a path in which no vertex is repeated, except the first and the last vertex, which are same. The length of any of these is equal to the number of edges in it.

Directed and Undirected Graph

A graph $G = (V,E)$ is directed if the edge set is composed of ordered vertex (node) pairs. A graph is undirected if the edge set is composed of unordered vertex pairs.

Vertex Cardinality

The number of vertices, the **cardinality** of V, is called the **order of graph** and devoted by $|V|$. We usually use ***n*** to denote the order of G. The number of edges, the **cardinality** of E, is called the **size of graph** and denoted by $|E|$. We usually use ***m*** to denote the size of G.

Neighbor Vertex and Neighborhood

We write $v_iv_j \in E(G)$ to mean $\{v_i,v_j\} \in E(G)$, and if $e = v_iv_j \in E(G)$, we say v_i and v_j are **adjacent** or, we say v_i and v_j are in the **neighbourhood** of each other.

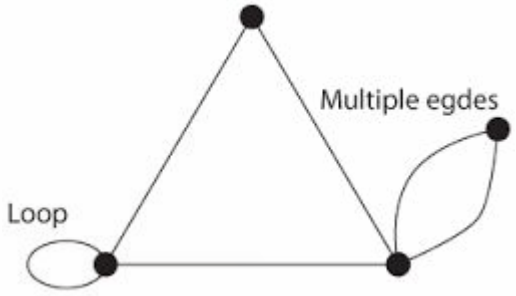
Formally, given a graph $G = (V,E)$, two vertices $v_i, v_j \in V$ are said to be *neighbors*, or *adjacent* nodes, if $(v_i,v_j) \in E$. If G is directed, we distinguish between incoming neighbors of v_i (those vertices $v_j \in V$ such that $(v_j,v_i) \in E$) and outgoing neighbors of v_i (those vertices $v_j \in V$ such that $(v_i,v_j) \in E$).

Vertex Degree

The ***degree of a vertex*** v is the number of edges which contain v *i.e.*, the number of edges that are incident on v. Note: A loop contributes **2** to the degree of the vertex it is incident on. As each edge contributes 2 to the degree of its endpoints, ***the sum of all degrees of vertices equals twice the number of edges.***

In a directed graph, the number of incoming edges to a vertex is called the ***indegree*** of the vertex and the number of outgoing edges from a vertex is its ***outdegree***.

Loop and Multiple Edges



A loop is an edge whose endpoints are equal *i.e.*, an edge joining a vertex to itself is called a **loop**. We say that the graph has **multiple edges** if in the graph $\exists$ two or more edges joining the same pair of vertices.

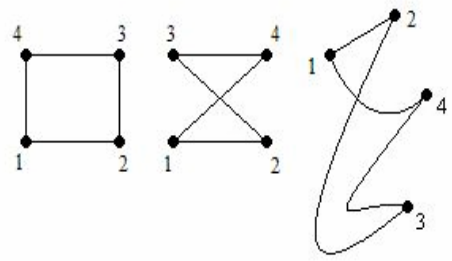
Simple Graph

A graph with no loops or multiple edges is called a simple graph. We specify a simple graph by its set of vertices and set of edges, treating the edge set as a set of unordered pairs of vertices and write $e = uv$ (or $e = vu$) for an edge e with endpoints u and v.

Connected Graph

A graph that is in one piece is said to be connected, whereas one which splits into several pieces is disconnected. A graph G is connected if there is a path in G between any given pair of vertices, otherwise it is disconnected. Every disconnected graph can be split up into a number of connected subgraphs, called **connected components**. A **forest** is a graph where each connected component is a tree.

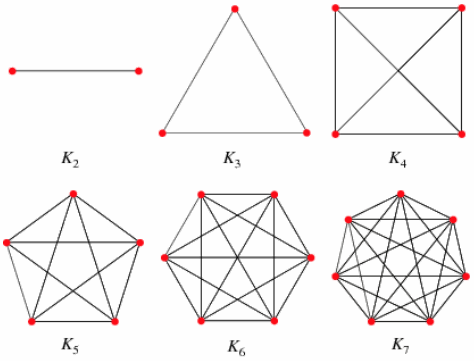
Isomorphic Graphs



Two graph G and H are isomorphic if H can be obtained from G by relabeling the vertices - that is, if there is a one-to-one correspondence between the vertices of G and those of H, such that the number of edges joining any pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H.

Complete Graphs

A complete graph is a graph in which every two distinct vertices are joined by exactly one edge. The complete graph with n vertices is denoted by K_n.

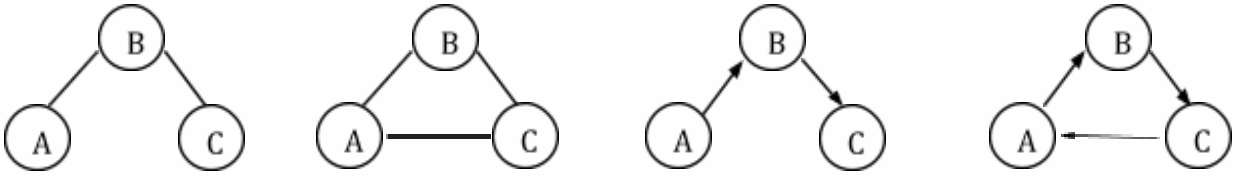


Cyclic and Acyclic Graphs

Cyclic, as the name suggests contains cycles, and the opposite for an acyclic graph.

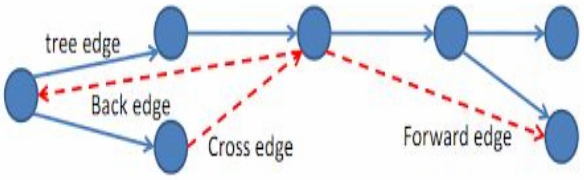
Directed and Undirected Graphs

In a directed graph, edges are unidirectional. If two nodes (say A and B) are connected by a directed edge, then one can traverse from A to B only and not from B to A using this edge. In an undirected graph, all edges are bidirectional.



Undirected Acyclic Graph | **Undirected Cyclic Graph** | **Directed Acyclic Graph (DAG)** | **Directed Cyclic Graph**

DFS Edge Classification

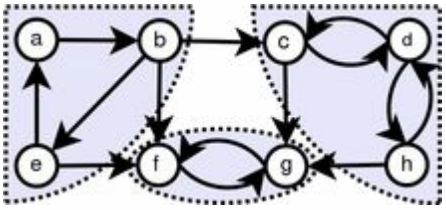


The edges we traverse as we execute a depth-first search can be classified into four edge types. During a DFS execution, the classification of edge (u,v), the edge from vertex u to vertex v, depends on whether we have visited v before in the DFS and if so, the relationship between u and v.

1. If v is visited for the first time as we traverse the edge (u,v), then the edge is a **tree edge**.
2. Else, v has already been visited:
   a. If v is an ancestor of u, then edge (u,v) is a **back edge**.
   b. Else, if v is a descendant of u, then edge (u,v) is a **forward edge**.
   c. Else, if v is neither an ancestor or descendant of u, then edge (u,v) is a **cross edge**.

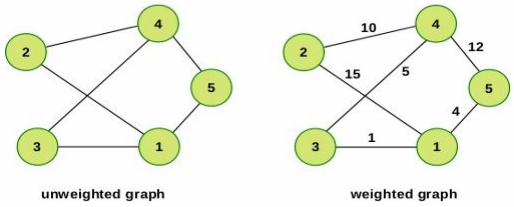
After executing DFS on graph G, every edge in G can be classified as one of these four edge types. We can use edge type information to learn some things about G.
For example, tree edges will form a tree containing each vertex visited during DFS in G.
Also, G has a cycle if and only if DFS edge classification finds at least one back edge.

Strongly Connected Components [SCC]
In the mathematical theory of **directed graphs**, a graph is said to be **strongly connected** if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.
Reference: **DFS based linear-time algorithms to find SCC in a directed graph.**



The **SCC**s in the graph are:
1. (a,b,e,f)
2. (f,g)
3. (c,d,g,h)

Weighted and Unweighted Graphs



Weighted graph: Each edge has a weight.
Unweighted graph: Edges don't have any weight.

Graph Representation

To represent a graph, all we need is a set of vertices, and for each vertex the neighbors of the vertex (vertices directly connected to it by an edge) and if its a weighted graph, the weight associated with each edge. There are different ways to optimally represent a graph, depending on the density of its edges. A Sparse graph has relatively less number of edges, $m = O(n)$, whereas, a dense graph has significantly many edges, $m = O(n^2)$, where n is the number of vertices in the graph. These are the two following ways of representing a graph in memory:

1. **Adjacency List:** In this representation, for each node in the graph we maintain the list of its neighbours. We have an array of vertices, indexed by the vertex number and for each vertex v, the corresponding array element points to a singly-linked list containing the neighbors of v.

2. **Adjacency Matrix:** In this representation, we construct an $n \times n$ matrix, say A. If there is an edge from a vertex i to vertex j, then the corresponding element of A, $A_{i,j} = 1$, otherwise $A_{i,j} = 0$. Note: Even if the graph of 100 vertices contains only a single edge, we still have to create a 100×100 matrix with lots of zeroes. For a weighted graph, instead of 1's and 0's, we can store the weight of the edge. In that case, to indicate that there is no edge we can put a safely large value (we can call it `INF` (infinity)) or 0, provided the edge wieghts are non-zero.

Adjacency List vs Adjacency Matrix

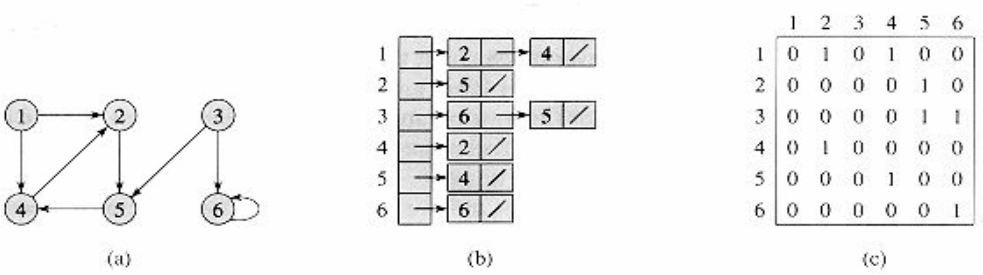


Figure (a) represents a conceptual view of graph. Figure (b) & (c) represent view of a graph in the memory, where (b) & (c) are Adjacency List & Adjacency Matrix respectively.

a. While the adjacency list saves a lot of space for sparse graphs, we cannot instantly access an edge.

b. While we can access any edge instantly in adjacency matrix, it takes a lot of space (space is fine for graphs even with n = 1000) and to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph (the whole row corresponding to the vertex), which takes quite some time.

c. So there is a trade-off between the two representations, and one has to decide depending on the problem. In general, we use adjacency list.

Graph Traversals

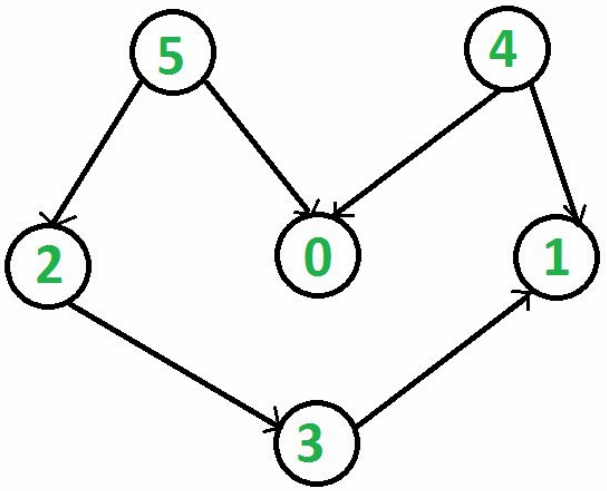
So far we have learned how to represent our graph in memory, but now we need to start doing something with this information. There are two methods for searching in graphs that are extremely prevalent, and will form the foundations for more advanced algorithms later on. These two methods are the **Depth First Search** [DFS] and the **Breadth First Search** [BFS].

Depth First Search

The Depth First Search [DFS] is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. The basic concept is to visit a node, then push all of the nodes to be visited onto the **stack**. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well, and we continue doing this until all nodes are visited. It is a key property of the DFS that we should not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it, using an **array**.

Breadth First Search
The Breadth First Search [BFS] is an extremely useful searching technique. It differs from the DFS *i.e.*, BFS uses a **queue** to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property and that is, if all of the edges in a graph are unweighted (or have the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about "what using a queue means to the search order". When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the **FIFO** property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a BFS is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source. We do this by marking the node as we visit it using an array again.

Topological Sort



Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge `uv`, vertex `u` comes before `v` in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
For example, a topological sorting of the following graph is "`5 4 2 3 1 0`". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "`4 5 2 3 1 0`". The first vertex in topological sorting is always a vertex with in-degree as `0` (a vertex with no incoming edges).

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, `O(|V| + |E|)`.

1. **Topological Sort using Kahn's Algorithm:** One of these algorithms works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges (*i.e.*, `indegree=0`) and insert them into a set `S`; at least one such node must exist in a non-empty acyclic graph. Then,

```
L ← Empty list that will contain the sorted elements.
S ← Set of all nodes with no incoming edges.
while S is non-empty do:
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do:
        remove edge e from the graph
        if m has no other incoming edges, i.e., indegree(m)=0 then:
            insert m into S

if graph has edges then:
    //used to detect cycle in a directed graph
    return error (graph has at least one cycle)
else:
    return L (a topologically sorted order)
```

2. **Topological Sort using Depth First Search:** An alternative algorithm for topological sorting is based on depth-first search. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (*i.e.*, a leaf node):

```
L ← Empty list that will contain the sorted nodes
while there are unmarked nodes do:
    select an unmarked node n
    visit(n)

function visit(node n):
    // detect cycle in a directed graph
    if n has a temporary mark then stop (not a DAG):
        if n is not marked (i.e., has not been visited yet) then:
            mark n temporary
            for each node m with an edge from n to m do:
                visit(m)

            mark n permanently
            unmark n temporarily
            add n to head of L
```

Each node `n` gets prepended to the output list `L` only after considering all other nodes which depend on `n` (all descendants of `n` in the graph). Specifically, when the algorithm adds node `n`, we are guaranteed that all nodes which depend on `n` are already in the output list `L`: they were added to `L` either by the recursive call to `visit()` which ended before the call to visit `n`, or by a call to `visit()` which started even before the call to visit `n`. Since each edge and node is visited once, the algorithm runs in linear time.

Shortest Path Algorithms

An extremely common problem on graphs is to find the path from one position to another. In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map (the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment) may be modeled by a special case of the shortest path problem in graphs.

There are a few different ways for going about this, each of which has different uses. We will discuss few of them here.

Single-Source Shortest Path

Given a graph and a source vertex **S**, find the shortest path to all other vertices.

1. **Unweighted Graph -** Simply use **Breadth First Search**.
   Algorithm:
   a. Store **S** in a queue and set the distance to ***s*** to be 0 in a distance array, d.
   b. Initialize the distances to all other vertices as "not computed", say -1.
   c. While there are vertices in the queue:
      Read a vertex **v** from the queue
      For all edges vertices **v->w**:
        If distance to **w** is -1 (not computed) do:
          Make distance to **w** equal to d[**v**] + 1, *i.e.*, d[w] = d[v] + 1
          // Store path from **S** to w **
          Make path to **w** equal to **v**, *i.e.*, path[w] = v
          Append **w** to the queue

   Complexity: O(|E| + |V|)

   ** To store the path from node A to node B, we only need to store the previous node and we can print the path by traversing on the path array from B to A (path[*u*] holds the predecessor of *u* in the shortest path from ***s*** to ***u***)

2. **Weighted Graph – Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
   Algorithm: Use priority queue based on the distance of current node from S.
   Let the node at which we are starting be called the **source node (S)**. Let d[w] be the distance from the **S** to w. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

a. Store **S** in a priority queue with distance as 0. Set d[S] = 0.
b. Initialize the distances to all other vertices as "not computed", sentinel value infinity.
c. while there are vertices in the queue:
   DeleteMin a vertex v from the queue
   For all adjacent vertices w do:
     if d[v] + weight(v,w) < d[w]:
       d[w] = d[v] + weight(v,w); // update distance to w
       insert w in the priority queue with distance d[w]
       //Store path from S to w **
       Make path to w equal to v, ie, path[w] = v

Complexity: $O(|E| \times \log_2|V| + |V| \times \log_2|V|) = O((|E|+|V|) \times \log_2(|V|))$

Note: Dijkstra's algorithm can be used for both weighted directed and undirected graphs. However, for a weighted DAG, topological ordering can also be used to quickly compute shortest paths.

All-Pairs Shortest Path

Find the shortest paths between every pair of vertices `u,v` in the graph.

Floyd–Warshall is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices.

Consider a graph `G` with vertices `V` numbered `1` through `N`. Further consider a function `shortestPath(i,j,k)` that returns the shortest possible path from `i` to `j` using vertices only from the set `{1,2,...,k}` as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each `i` to each `j` using only vertices `1` to `k+1`. For each of these pairs of vertices, the true shortest path could be either:

a. A path that only uses vertices in the set `{1,...,k}` or,
b. A path that goes from `i` to `k+1` and then from `k+1` to `j`.

We know that the best path from `i` to `j` that only uses vertices `1` through `k` is defined by `shortestPath(i,j,k)`, and it is clear that if there were a better path from `i` to `k+1` to `j`, then the length of this path would be the concatenation of the shortest path from `i` to `k+1` (using vertices in `{1,...,k}`) and the shortest path from `{k+1}` to `j` (also using vertices in `{1,...,k}`). If `w(i,j)` is the weight of the edge between vertices `i` and `j`, we can define `shortestPath(i,j,k+1)` in terms of the following recursive formula:

```
shortestPath(i,j,k+1) = min(shortestPath(i,j,k), shortestPath(i,k+1,k) +
shortestPath(k+1,j,k))
Base case: shortestPath(i,j,0) = w(i,j)
```

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing `shortestPath(i,j,k)` $\forall$`(i,j)` pairs for `k=1`, then `k=2`, etc. This process continues until `k=N`, and finds the shortest path $\forall$`(i,j)` pairs using any intermediate vertices.

Pseudo code

```
a. let dist be a |V| × |V| array of minimum distances initialized to ∞
   (infinity)
b. for each vertex v:
     dist[v][v] ← 0
c. for each edge (u,v):
     dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
d. for k from 1 to |V|:
     for i from 1 to |V|:
       for j from 1 to |V|:
         if dist[i][j] > dist[i][k] + dist[k][j]:
           dist[i][j] ← dist[i][k] + dist[k][j]
```

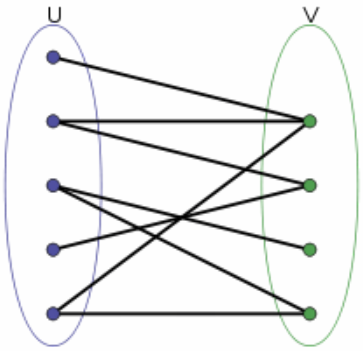
Complexity: $O(N^3)$

Graph Coloring

Graph coloring is a special case of graph labeling; it is an ***assignment of labels*** traditionally called "colors" to elements of a graph subject to certain *constraints*. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a **vertex coloring**. Since a vertex with a loop (*i.e.*, a connection directly back to itself) could never be properly colored, it is understood that graphs in this context are loopless.

A coloring using at most k colors is called a **k-coloring**. The smallest number of colors needed to color a graph G is called its **chromatic number**. A graph that can be assigned a k-coloring is **k-colorable**, and it is **k-chromatic** if its chromatic number is exactly k. A subset of vertices assigned to the same color is called a **color class**; every such class forms an independent set. Thus, *a **k-coloring** is the same as a partition of the vertex set into k independent sets, and the terms **k-partite** and **k-colorable** have the same meaning*.

Bipartite Graphs



A bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V (*i.e.*, U and V are each independent sets) such that every edge connects a vertex in U to one in V. Vertex set U and V are usually called the parts of the graph. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

The two sets U and V may be thought of as a **coloring of the graph with two colors**: if one colors all nodes in U **blue** and all nodes in V **green**, each edge has endpoints of differing colors, as is required in the **graph coloring problem**. In contrast, such a coloring is impossible in the case of a non-bipartite graph. So, every **2-colorable** graph is a bipartite graph.

Disjoint Set Data Structure [Union-Find (or) Merge-Find]

A **disjoint-set data structure** (also called a **union–find** data structure or **merge–find** set) is a data structure that tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set. Union-find plays a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph. Union-find is also used to detect cycle in both directed and undirected graphs.

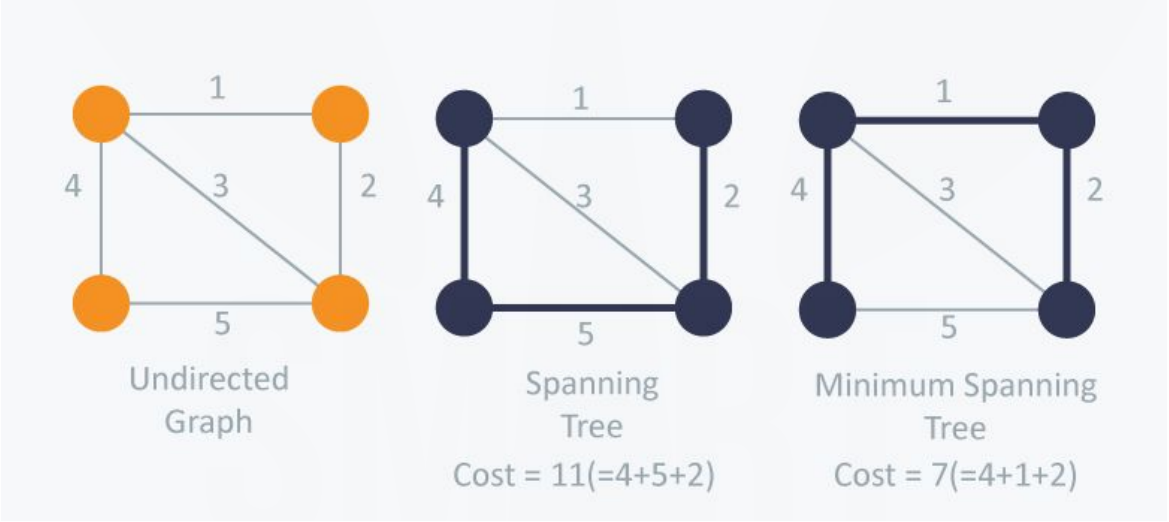
Must Read Tutorial to fully understand the Union-Find Data Structure @HackerEarth

Minimum Spanning Tree

Given an undirected and connected graph `G = (V,E)`, a spanning tree of the graph `G` is a tree that spans `G` (*i.e.,* it includes every vertex of `G`) and is a subgraph of `G` (every edge in the tree belongs to `G`)

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. **Minimum spanning tree** is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

MST has applications in computer networks, electrical grids, cluster analysis, image segmentation, etc.



There are two popular algorithms to find the MST of a graph, and they're:

1. Kruskal's Algorithm.
2. Prim's Algorithm.

MST Tutorial @HackerEarth is a **must read to fully understand this topic**.

C++, Java & Python Programs on Graphs

1. **Graph Representation using Adjacency matrix.**

```
int n,m;                // vertices and edges
cin>>n>>m;

int graph[n+1][n+1];  // vertices are generally numbered starting
with 1
int u,v,w;              // w is the weight of the edge
for(int i=0; i<m; i++) {
    cin>>u>>v>>w;
    ar[u][v] = w;      // assign with 1 if the graph is unweighted
    ar[v][u] = w;      // if the graph is bidirectional
}
```

- Graph Representation using Adjacency Matrix in Java & Python

2. **Graph Representation using Adjacency list.**

```
int n,m;    //vertices and edges
cin>>n>>m;

// can use vector<vector<int>> graph(n+1) if the graph is unweighted
vector<vector<pair<int,int>>> graph(n+1);
int u,v,w;    // w is the weight of the edge
for(int i=0; i<m; i++) {
    cin>>u>>v>>w;
    // graph[u].push_back(v) - for unweighted graphs
    graph[u].push_back(make_pair(v,w));
    graph[v].push_back(make_pair(u,w));   // if the graph is
bidirectional
}
```

- Graph Representation using Adjacency List in Java & Python

3. **Depth First Search traversal of a graph.**

```
void dfs(int u, bool vis[], vector<vector<int>> graph) {
    if(vis[u])   return;
    cout<<u<<" ";
    vis[u] = true;
    for(int i=0; i<graph[u].size(); i++)
        dfs(graph[u][i], vis, graph);
}
main() function:
    bool vis[n+1];
    fill(vis, vis+n+1, false);    // #include<algorithm>
    for(int i=1; i<=n; i++)
        if(!vis[i]) {
            dfs(i, vis, graph);
            cout<<endl;
    }
```

- Depth First Search Implementation in Java & Python

4. **Breadth First Search traversal of a graph.**

```
void bfs(int u, bool vis[], vector<vector<int>> graph) {
    queue<int> q;
    q.push(u);
    vis[u] = true;

    while(!q.empty()) {
        u = q.front();
        q.pop();
        cout<<u<<" ";
        for(int i=0; i<graph[u].size(); i++)
            if(!vis[graph[u][i]]) {
                q.push(graph[u][i]);
                vis[graph[u][i]] = true;
            }
    }
}
main() function:
    bool vis[n+1];
    fill(vis, vis+n+1, false);      // #include<algorithm>
    for(int i=1; i<=n; i++)
        if(!vis[i]) {
            bfs(i, vis, graph);
            cout<<endl;
        }
```

- Breadth First Search Implementation in Java & Python

5. **DFS Tree Edge Classification using Start/Finish Time**

```
void dfsEdgeClassification(int u, int v, int start[], int finish[])
{
    if(finish[v] != -1 && start[v]<start[u])
        cout<<"Edge "<<u<<"->"<<v<<" is a cross edge"<<endl;
    else if(finish[v] != -1 && start[u]<start[v])
        cout<<"Edge "<<u<<"->"<<v<<" is a forward edge"<<endl;
    //cycle exits
    else if(finish[v] == -1 && start[v] != -1 && start[v]<start[u])
        cout<<"Edge "<<u<<"->"<<v<<" is a back edge"<<endl;
}
void dfs(int u, bool vis[], vector<vector<int>> graph, int start[],
int finish[]) {
    static int timer = 1;
    if(vis[u])    return;
    cout<<u<<" ";
    vis[u] = true;
    start[u] = timer++;
    for(int i=0; i<graph[u].size(); i++) {
        dfsEdgeClassification(u, graph[u][i], start, finish);
        dfs(graph[u][i], vis, graph, start, finish);
    }
    finish[u] = timer++;
}
```

```
main() function:
    int start[n+1], finish[n+1];
    bool vis1[n+1];
    fill(vis1, vis1+n+1, false);
    fill(start, start+n+1, -1);
    fill(finish, finish+n+1, -1);

    for(int i=1; i<=n; i++)
        if(!vis1[i]) {
            dfs(i, vis1, graph, start, finish);
    }
```

- **Edge Classification Implementation in Python**
- **Key Insight for classifying Forward Edge & Back Edge**

6. **Topological Sorting using Kahn's Algorithm**

```
void topoSort(vector<vector<int>> graph, int inDegree[], int
outDegree[], int n) {
    queue<int> q;
    for(int i=1; i<=n; i++)
        if(inDegree[i] == 0)
            q.push(i);

    queue<int> topologicalSorting;
    int u, v;
    while(!q.empty()) {
        u = q.front();
        q.pop();
        topologicalSorting.push(u);
        for(int i=0; i<graph[u].size(); i++) {
            v = graph[u][i];
            inDegree[v]--;
            if(inDegree[v] == 0)
                q.push(v);
        }
    }

    if(topologicalSorting.size() != n)
        cout<<"Given graph is not a DAG!!";
    else
        while(!topologicalSorting.empty()) {
            cout<<topologicalSorting.front()<<" ";
            topologicalSorting.pop();
        }
}

main() function:
    vector<vector<int>> graph(n+1);
    int inDegree[n+1], outDegree[n+1];
    fill(inDegree, inDegree+n+1, 0);
    fill(outDegree, outDegree+n+1, 0);
```

```
    int u,v;
    for(int i=0; i<m; i++) {
        cin>>u>>v;
        graph[u].push_back(v);
        inDegree[v]++;
        outDegree[u]++;
    }
    topoSort(graph, inDegree, outDegree, n);
```

- Topological Sort using Kahn's Algorithm in Java
- Topological Sort using Kahn's Algorithm in Python

7. **Topological Sorting using Depth First Search Traversal**

```
bool topoSortUsingDFS(int u, int mark[], vector<vector<int>> graph,
stack<int> &topologicalSorting) {
    if(mark[u] == 1)          //there is a cycle in the graph
        return false;
    if(mark[u] == 2)
        return true;

    mark[u] = 1;
    bool retValue = true;
    for(int i=0; i<graph[u].size() && retValue; i++)
        retValue &= topoSortUsingDFS(graph[u][i], mark, graph,
topologicalSorting);

    topologicalSorting.push(u);
    mark[u] = 2;
    return retValue;
}

main:
    int mark[n+1];
    fill(mark, mark+n+1, 0);
    stack<int> topologicalSorting;
    bool noCycle = true;

    for(int i=1; i<=n && noCycle; i++)
        if(mark[i] == 0)
            noCycle = topoSortUsingDFS(i, mark, graph,
topologicalSorting);

    if(noCycle) {
        while(!topologicalSorting.empty()) {
            cout<<topologicalSorting.top()<<" ";
            topologicalSorting.pop();
        }
        cout<<endl;
    } else
        cout<<"Given graph is not a DAG!!"<<endl;
```

Here,

```
mark[u] = 0: unvisited
mark[u] = 1: in current DFS cycle and not completely processed
mark[u] = 2: processed
```

- **Topological Sort using DFS in Java**
- **Topological Sort using DFS in Python**

8. **Dijkstra's Algorithm – Shortest path between 2 nodes in a weighted graph.**

```
int Dijkstra(int source, int destination, vector<vector<pair<int,
int> > > graph) {
    int n = graph.size();
    int distance[n];
    fill(distance, distance + n, INT_MAX);
    distance[source] = 0;

//Declaring a min heap of pair - (distance, nodeNumber)
//Heap property will be maintained using the 1st element of the pair
// - Distance
    priority_queue< pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    pq.push(make_pair(0, source));

    pair<int, int> p;
    int u,d,v,w;

    while(!pq.empty()) {
        p = pq.top();
        pq.pop();
        if(p.second == destination)
            return p.first;

        u = p.second;
        d = p.first;

        if(distance[u] == d) {
            for(int i=0; i<graph[u].size(); i++) {
                v = graph[u][i].first;
                w = graph[u][i].second;
                if(d + w < distance[v]) {
                    distance[v] = d+w;
                    pq.push(make_pair(distance[v], v));
                }
            }
        }
    }

    return distance[destination];
}
```

```
int main() {
    int n,m;      //vertices and edges
    cin>>n>>m;

    vector<vector<pair<int, int>>> graph(n+1);
    int u,v,w;
    for(int i=0; i<m; i++) {
        cin>>u>>v>>w;
        graph[u].push_back(make_pair(v,w));
        graph[v].push_back(make_pair(u,w));
    }

    cout<<Dijkstra(1, 7, graph)<<endl;
    cout<<Dijkstra(3, 9, graph)<<endl;
    return 0;
}
```

- **Implementation of Djikstra's Algorithm in Java**
- **Implementation of Djikstra's Algorithm in Python**

9. **Floyd Warshall – All pair shortest path**

```
void FloydWarshall(vector<vector<int>> &graph) {
    int n = graph.size() - 1;
    for(int k=1; k<=n; k++)
        for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                graph[i][j] = min(graph[i][j], graph[i][k] +
graph[k][j]);
}

int main() {
    int n,m;      //vertices and edges
    cin>>n>>m;

    vector<vector<int>> graph(n+1, vector<int> (n+1, 1e6));
    int u,v,w;
    for(int i=0; i<m; i++) {
        cin>>u>>v>>w;
        graph[u][v] = w;
        graph[v][u] = w;
    }

    FloydWarshall(graph);
    cout<<graph[1][7]<<endl;
    cout<<graph[3][6]<<endl;
    return 0;

}
```

- **Implementation of Floyd Warshall Algorithm in Java @Princeton**
- **Implementation of Floyd Warshall Algorithm in Python**

10. **Check if a given graph is Bipartite - Use 2-colorable property.**

```
int main() {
    // READ THE GRAPH from standard Input
    bool isBipartite = true;
    bool vis[n+1];   fill(vis, vis+n+1, false);
    int color[n+1];  fill(color, color+n+1, 0);

    for(int i=1; i<=n && isBipartite; i++)
        if(!vis[i])
            isBipartite &= isBipartiteUsingBFS(i, vis, color,
graph);

    if(isBipartite)
        cout<<"Given graph is Bipartite"<<endl;
    else
        cout<<"Given graph is not Bipartite"<<endl;

    return 0;
}

bool isBipartiteUsingBFS(int u, bool vis[], int color[],
vector<vector<int>> graph) {
    vis[u] = true;
    color[u] = 1;
    queue<int> q;
    q.push(u);
    int v;

    while(!q.empty()) {
        int u = q.front();
        q.pop();

        for(int i=0; i<graph[u].size(); i++) {
            v = graph[u][i];
            if(color[v] == color[u])
                return false;
            color[v] = (3-color[u]);    //2->1 and 1->2

            if(!vis[v]) {
                q.push(v);
                vis[v] = true;
            }
        }
    }
    return true;
}
```

- Java Program to Check if a Graph is Bipartite @Princeton
- Python Program to Check if a Graph is Bipartite

Problems & Solutions

1. Check if there exists a path between 2 given nodes `A` and `B` of a graph.
   Solution: Simply run BFS/DFS from `A`, if you reach `B` -> path exists

2. Detect Cycle in a Directed Graph.
   Solution:
   a. Using Topo Sort – if Topo Sort fails, cycle exists.
   b. Using Back Edge – if Back Edge exists, cycle exists.
      i. Using Start/Finish Time.
      ii. Using mark array `= {0,1,2}` – code explained above.

3. Detect Cycle in an Undirected Graph.
   Observation: You cannot use the methods explained for a directed graph above, because edges in an undirected graph are bi-directional and any edge will be marked as a back edge if we use Back Edge detection mechanism.
   Solution:
   We can use DFS to detect cycle in an undirected graph in `O(V+E)` time. We do a DFS traversal of the given graph. For every visited vertex '`v`', if there is an adjacent '`u`' such that `u` is already visited and `u` is not parent of `v`, then there is a cycle in the graph.

4. Given an undirected graph, check if it's a tree.
   Solution: A tree can be defined as a **connected acyclic** graph. For the graph to be **acyclic**, given `n` nodes, there should not be more than `n-1` edges in the graph. After the first verification step, for the graph to be a tree, you can simply run a BFS/DFS to check if the graph is **connected**.
   Related problem: **Is it a Tree? @SPOJ**

5. Given a matrix of `0`'s and `1`'s, find the number of islands considering `N4` neighborhood. A group of connected `1`'s forms an island. For example, the below matrix contains `5` islands:

```
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

   Solution: Run BFS/DFS [considering `N4` neighborhood] for every unvisited cell of the matrix which contains a `1`.

6. Given an unweighted, undirected tree, find the length of the longest path in that tree. The length of a path is the number of edges in that path.
   Solution: Use 2 BFS'. At the end of the 1st BFS, you will reach an edge node of the longest path, say `S`. Start another BFS from `S`, the length of the longest path will be the number of BFS iterations as you started from an edge node of the longest path.
   Related problems: **Time to live @SPOJ** & **Longest path in a tree**.

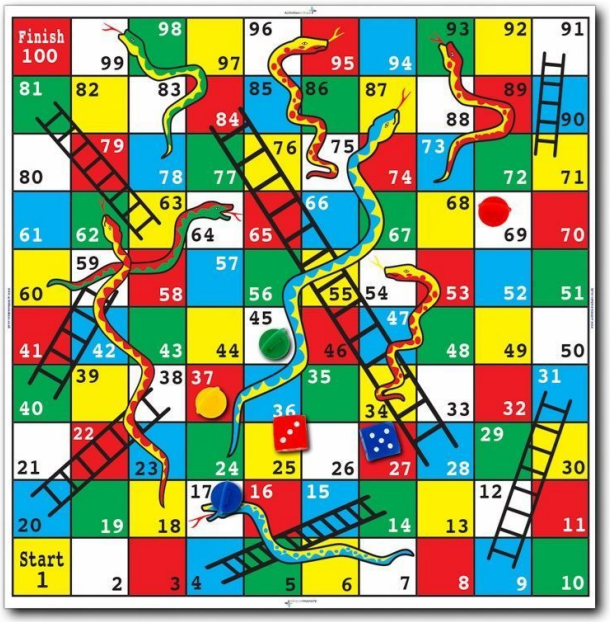
7. There are `N` number of C programming library files. The compiling of a library has the constraint that a library must be compiled after any library it depends on. Given the list of dependencies, find the valid compilation order of the library files, if possible.
Solution: Represent the dependencies as a directed graph, and apply Topological Sorting. If successful, compile the files in reverse order of topological sorting, else print "impossible".
Related Problem: **Find Order in Characters @Geeks4Geeks**

8. Given two nodes `A` and `B` in an unweighted graph, find the length of the shortest path between the given nodes.
Solution: Use **BFS**: Start BFS from `A`, the number of BFS iterations to reach `B` is the length of the shortest path between `A` and `B`.

9. Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell.
Solution:
   a. Convert the board into a Directed Graph, with edges from each `i` to `i+1`, `i+2`, `i+3`, `i+4`, `i+5`, `i+6` – These are possible Dice Rolls. Now, this is Directed Unweighted Graph and your problem has reduced to finding the shortest path from `1` to `100`.

      During your BFS traversal, if you land on a cell where a ladder starts, replace the cell by the ending cell of the ladder, similarly handle snakes as well.

   b. You can also think about a Greedy approach using Graph with Min Heap (similar to Dijkstra).



10. Given a dictionary, and multiple queries of the form:
'`start`' word and '`target`' word (both of same length), find the length of the smallest chain from '`start`' to '`target`', such that adjacent words in the chain only differ by one character and each word in the chain is a valid word *i.e.*, it exists in the dictionary. It may be assumed that both '`start`' and '`target`' words exists in dictionary and length of all dictionary words is same.

Solution: Convert the dictionary of words into an undirected graph, where an edge between two words can be created iff the words only differ by one character. Now, for each query, your problem has reduced to finding the shortest path between '`start`' and '`target`' words – Use BFS. **Related Problem:** **Prime Path @SPOJ**

11. Shortest Path Problems:
    a. **MICEMAZE @SPOJ**.
    b. **HIGHWAYS @SPOJ**.
    c. **Delivery Boy @CodeChef**.
    d. **Problems based on Shortest Path Algorithm @HackerEarth**.

12. Bipartite Graph Problem - **BUGLIFE @SPOJ**.

Important References

1. **Topological Sort's application to find shortest path**
2. **Shortest Path in a DAG**
3. **Graph Algorithms @YaleUniversity**
4. **Tutorials & Problems based on Graphs @HackerEarth**