SUMMER OF SCIENCE -MIDTERM REPORT

# GRAPH THEORY


PURANJAY DATTA

19D070048

MENTOR-Adwait Godbole

# *INTRODUCTION*

I Puranjay Datta would like to give a brief introduction to my project report on graph theory.

Graph theory in mathematics means the study of graphs Graphs are therefore mathematical structures used to model pairwise relations between objects. They are found on road maps, constellations, when constructing schemes and drawings. Graphs underlie many computer programs that make modern communication and technological processes possible. They contribute to the development of thinking, both logical and abstract. For example, maybe you remember this game from your childhood: connect the dots on the piece of paper to make a figure, a dog or a cat – those connections are also graphs.

We all know the saying: "Mathematics is not needed in real life", but graph theory is actually applicable in real life like In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another.

My report covers some basic algorithms of graph theory like shortest path finding algorithm, depth first search, finding cycles in a graph . The google maps that we use has more sophisticated algorithm which arises from these basics.

# *INDEX*

# INTRODUCTION

## 1.GRAPH REPRESENTATION

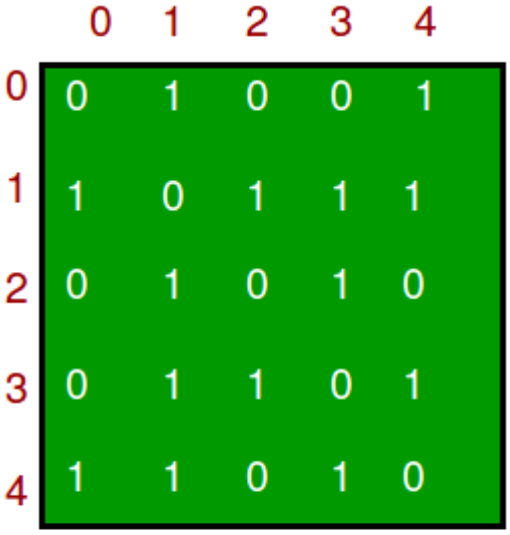
Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components:

- Nodes: These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- Edges: Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

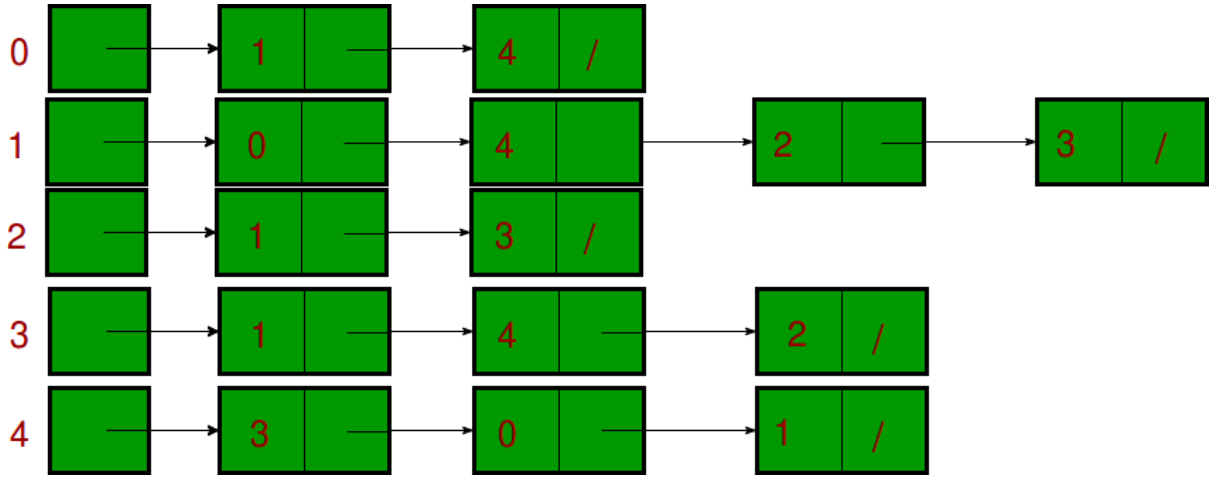
Following two are the most commonly used representations of a graph.
**1.** Adjacency Matrix
**2.** Adjacency List
There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

- **Adjacency Matrix:**
  Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

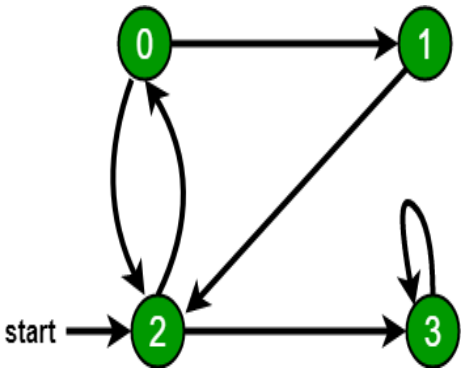



- **Adjacency List:**
  An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.



# Breadth First Search or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). 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.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



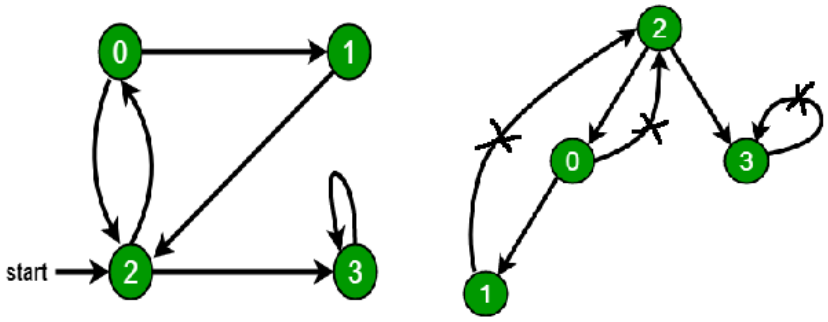

## PSEUDO CODE-

```
while(!queue.empty())
   {
      // Dequeue a vertex from queue and print it
      s = queue.front();
      cout << s << " ";
      queue.pop_front();

      // Get all adjacent vertices of the dequeued
      // vertex s. If a adjacent has not been visited,
      // then mark it visited and enqueue it
      for (i = adj[s].begin(); i != adj[s].end(); ++i)
      {
         if (!visited[*i])
         {
            visited[*i] = true;
            queue.push_back(*i);
         }
      }
   }
}
```

# Depth First Search or DFS for a Graph

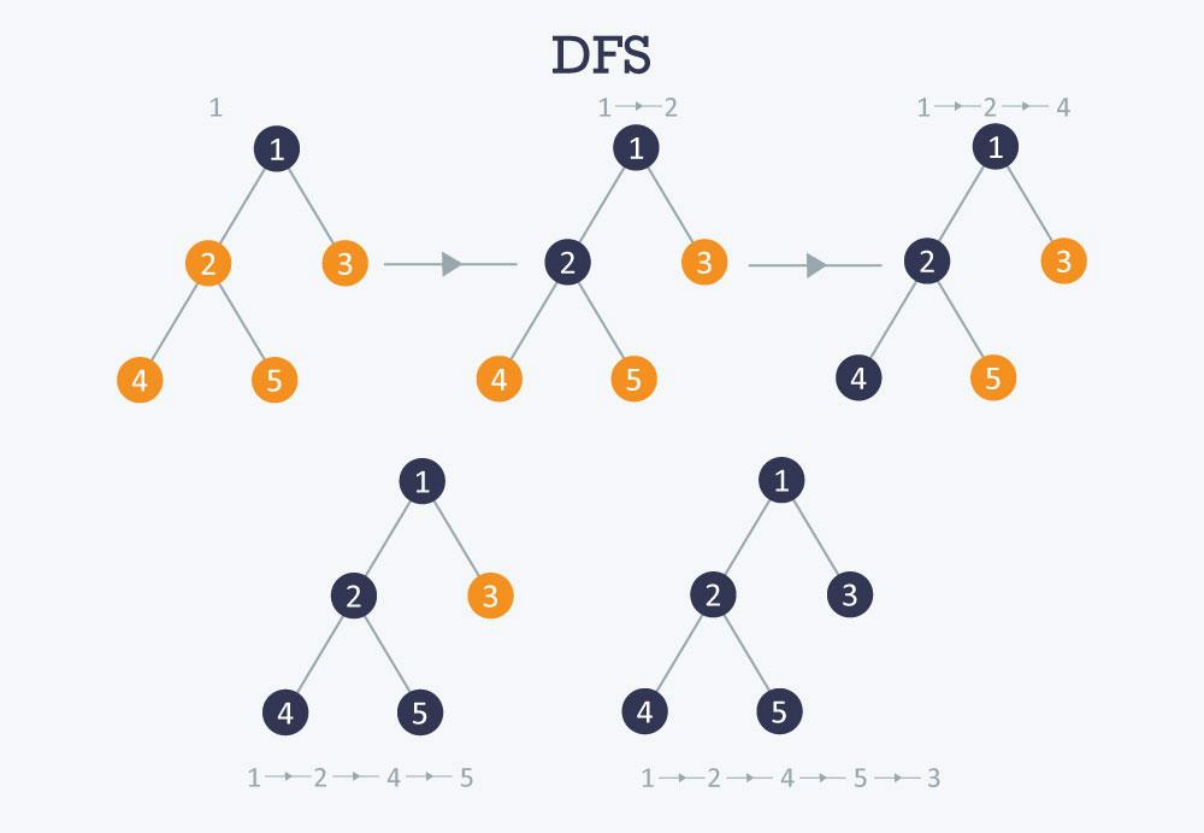
Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.




```
DFS-iterative (G, s):
//Where G is graph and s is source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not
visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                    S.push( w )
                   mark w as visited
```

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```



# CONNECTED COMPONENTS

Given an undirected graph $G$ with $n$ nodes and $m$

edges. We are required to find in it all the connected components, i.e, several groups of vertices such that within a group each vertex can be reached from another and no path exists between different groups.

## An algorithm for solving the problem

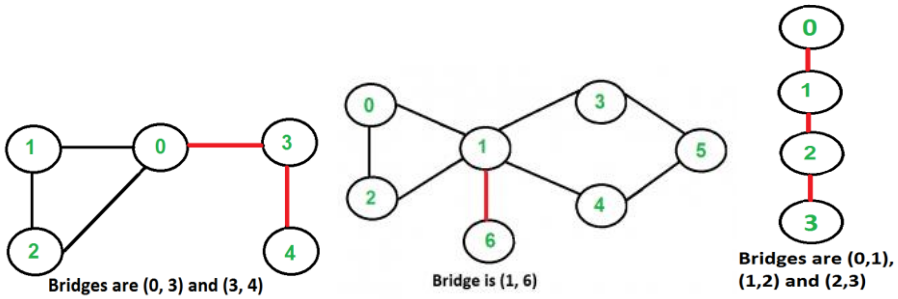
- To solve the problem, we can use Depth First Search or Breadth First Search.
- In fact, we will be doing a series of rounds of DFS: The first round will start from first node and all the nodes in the first connected component will be traversed (found). Then we find the first unvisited node of the remaining nodes, and run Depth First Search on it, thus finding a second connected component. And so on, until all the nodes are visited.
- The total asymptotic running time of this algorithm is $O(n+m)$

- : In fact, this algorithm will not run on the same vertex twice, which means that each edge will be seen exactly two times (at one end and at the other end).

# Bridges in a graph

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.
Like Articulation Points, bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.




**How to find all bridges in a given graph?**
A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every edge (u, v), do following
…..a) Remove (u, v) from graph
..…b) See if the graph remains connected (We can either use BFS or DFS)
…..c) Add (u, v) back to the graph.

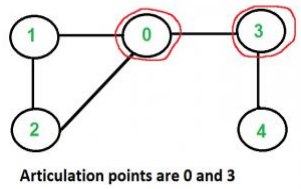
Time complexity of above method is O(E*(V+E)) for a graph represented using adjacency list. Can we do better?

**A O(V+E) algorithm to find all Bridges**
The idea is similar to O(V+E) algorithm for Articulation Points. We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the previous post, the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, "low[v] > disc[u]".*

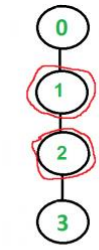
# Articulation Points (or Cut Vertices) in a Graph

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.
For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

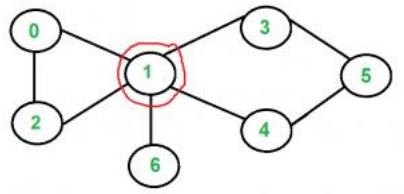
Following are some example graphs with articulation points encircled with red color.



Articulation points are 0 and 3



Artiulation
Points are 1 & 2



Articulation Point is 1

**How to find all articulation points in a given graph?**
A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

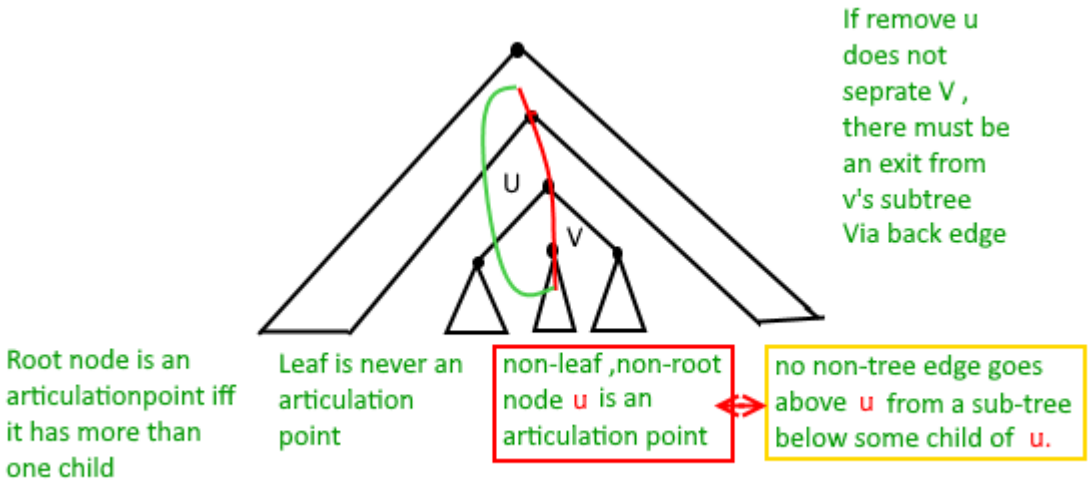
1) For every vertex v, do following
…..a) Remove v from graph
..…b) See if the graph remains connected (We can either use BFS or DFS)
…..c) Add v back to the graph

Time complexity of above method is O(V*(V+E)) for a graph represented using adjacency list. Can we do better?

**A O(V+E) algorithm to find all Articulation Points (APs)**
The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one

of the following two conditions is true.
**1)** u is root of DFS tree and it has at least two children.
**2)** u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

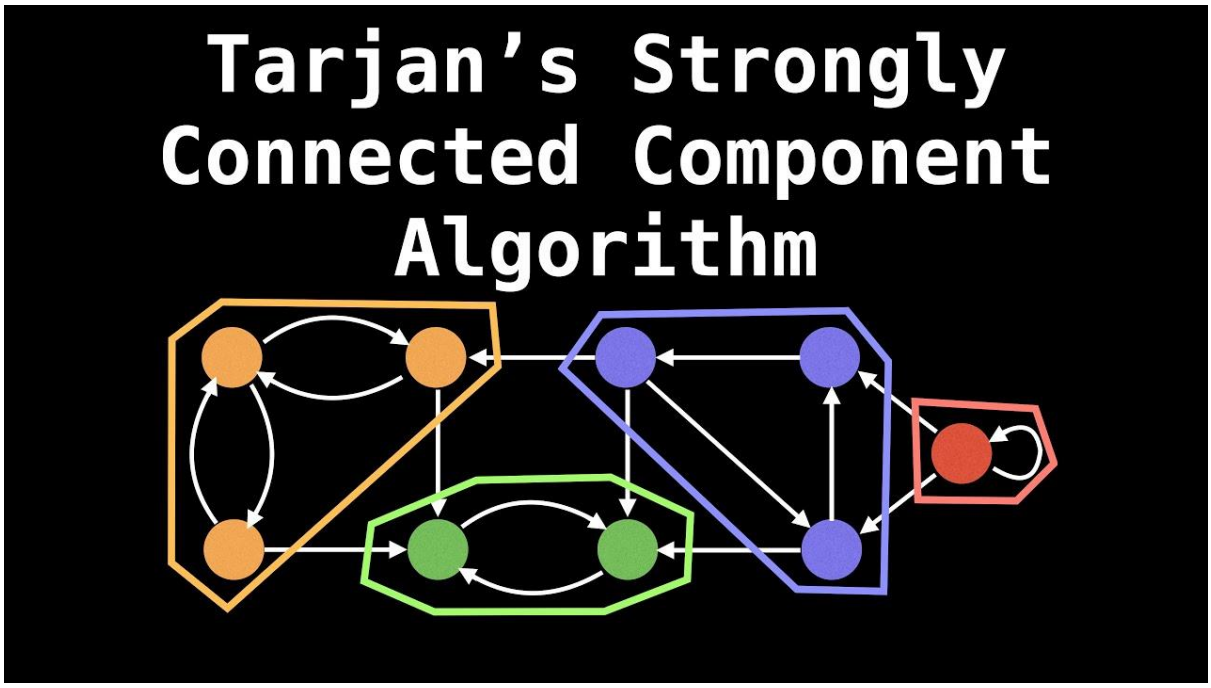
Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.




We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.
How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

# TARJAN'S ALGORITHM




MAIN IDEA BEHIND TARJAN'S ALGORITHM ARE THE FOLLOWING

We maintain a array of low value ie low[] and discovery time ie id[] and onstack [] array which tells us whether the node is there on stack or not.

While we encounter a node that is already visited and on the stack currently then we find the minimum of low[current node] and id[node visited and on stack].

On call back minimize the low value and if the low[node]=id[node] it means there is scc present and start popping till we find the current node from the stack. This comprises of one scc.

# KOSARAJU'S ALGORITHM

**The algorithm in steps can be described as below:**

STEP 1) Do a DFS on the original graph, keeping track of the finish times of each node. This can be done with a **stack**, when some DFS finishes put the source vertex on the stack. This way node with highest finishing time will be on top of the stack.

```
stack STACK
void DFS(int source) {
   visited[s]=true
   for all neighbours X of source that are not visited:
      DFS(X)
   STACK.push(source)
}
```

**STEP 2) Reverse the original graph**, it can be done efficiently if data structure used to store the graph is an adjacency list.

```
CLEAR ADJACENCY_LIST
for all edges e:
   first = one end point of e
   second = other end point of e
   ADJACENCY_LIST[second].push(first)
```

STEP 3) Do DFS on the reversed graph, with the source vertex as the vertex on top of the stack. When

finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Component's, so pop vertices from top of the stack until a valid unvisited node is found. This will have the highest finishing time of all currently unvisited nodes. This step is repeated until all nodes are visited.

```
while STACK is not empty:
   source = STACK.top()
   STACK.pop()
   if source is visited :
      continue
   else :
      DFS(source)
```

# DIJKSTRA'S ALGORITHM

Here is an algorithm described by the Dutch computer scientist Edsger W. Dijkstra in 1959.Let's create an array $d[]$

where for each vertex $v$ we store the current length of the shortest path from $s$ to $v$ in $d[v]$. Initially $d[s]=0$

, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v]=\infty,\ v\neq s$$

In addition, we maintain a Boolean array $u[]$

which stores for each vertex $v$

whether it's marked. Initially all vertices are unmarked:

$$u[v]=\text{false}$$

The Dijkstra's algorithm runs for $n$

iterations. At each iteration a vertex $v$ is chosen as unmarked vertex which has the least value $d[v]$

:

Evidently, in the first iteration the starting vertex $s$

will be selected.

The selected vertex $v$

is marked. Next, from vertex $v$ **relaxations** are performed: all edges of the form $(v,\text{to})$ are considered, and for each vertex $\text{to}$ the algorithm tries to improve the value $d[\text{to}]$. If the length of the current edge equals $len$

, the code for relaxation is:

$$d[\text{to}]=\min(d[\text{to}],d[v]+len)$$

After all such edges are considered, the current iteration ends. Finally, after $n$

iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from $s$ to all vertices $v$

.

Note that if some vertices are unreachable from the starting vertex $s$

, the values $d[v]$ for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.

PSEUDO CODE

```
void dijkstra()
{    long long int source=1;

     priority_queue< iPair, vector <iPair> , greater<iPair> > pq;
    vector<long long int> d(n+1, INF);
         pq.push(make_pair(0, source));
         d[source]=0;

     while (!pq.empty())
    {
         long long int u=pq.top().second;
        pq.pop();

         marked[u]=true;

         for(auto to:adj[u])
         {
             if(marked[to]==false && d[to]>d[u]+weight[{u,to}])
             {
                 d[to]=d[u]+weight[{u,to}];
                 parent[to]=u;
                 pq.push(make_pair(d[to], to))
             }
         }

    }
}
```

# Bellman-Ford Algorithm

Let us assume that the graph contains no negative weight cycle. The case of presence of a negative weight cycle will be discussed below in a separate section.

We will create an array of distances $d[0...n-1]$

, which after execution of the algorithm will contain the answer to the problem. In the beginning we fill it as follows: $d[v]=0$, and all other elements $d[]$ equal to infinity $\infty$

.

The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce **relaxation** along each edge $(a,b)$

having weight $c$. Relaxation along the edges is an attempt to improve the value $d[b]$ using value $d[a]+c$. In fact, it means that we are trying to improve the answer for this vertex using edge $(a,b)$ and current response for vertex $a$

.

It is claimed that $n-1$

phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph (again, we believe that the cycles of negative weight do not exist). For unreachable vertices the distance $d[]$ will remain equal to infinity $\infty$.

# FLOYD WARSHALL'S ALGORITHM

## Description of the algorithm

The key idea of the algorithm is to partition the process of finding the shortest path between any two vertices to several incremental phases.

Let us number the vertices starting from 1 to $n$

. The matrix of distances is $d[][]$

.

Before $k$

-th phase ($k=1...n$), $d[i][j]$ for any vertices $i$ and $j$ stores the length of the shortest path between the vertex $i$ and vertex $j$, which contains only the vertices $\{1,2,...,k-1\}$

as internal vertices in the path.

In other words, before $k$

-th phase the value of $d[i][j]$ is equal to the length of the shortest path from vertex $i$ to the vertex $j$, if this path is allowed to enter only the vertex with numbers smaller than $k$

(the beginning and end of the path are not restricted by this property).

It is easy to make sure that this property holds for the first phase. For $k=0$

, we can fill matrix with $d[i][j]=w_{ij}$ if there exists an edge between $i$ and $j$ with weight $w_{ij}$ and $d[i][j]=\infty$ if there doesn't exist an edge. In practice $\infty$

will be some high value. As we shall see later, this is a requirement for the algorithm.

Suppose now that we are in the $k$

-th phase, and we want to compute the matrix $d[][]$ so that it meets the requirements for the $(k+1)$-th phase. We have to fix the distances for some vertices pairs $(i,j)$

. There are two fundamentally different cases:

- The shortest way from the vertex $i$

to the vertex $j$ with internal vertices from the set $\{1,2,...,k\}$ coincides with the shortest path with internal vertices from the set $\{1,2,...,k-1\}$

.

In this case, $d[i][j]$

- will not change during the transition.
- The shortest path with internal vertices from $\{1,2,...,k\}$

is shorter.

This means that the new, shorter path passes through the vertex $k$

. This means that we can split the shortest path between $i$ and $j$ into two paths: the path between $i$ and $k$, and the path between $k$ and $j$. It is clear that both this paths only use internal vertices of $\{1,2,...,k-1\}$ and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between $i$ and $j$ as $d[i][k]+d[k][j]$

- .

Combining these two cases we find that we can recalculate the length of all pairs $(i,j)$

in the $k$

-th phase in the following way:

$$d_{\text{new}}[i][j]=min(d[i][j],d[i][k]+d[k][j])$$

Thus, all the work that is required in the $k$

-th phase is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the $n$-th phase, the value $d[i][j]$ in the distance matrix is the length of the shortest path between $i$ and $j$, or is $\infty$ if the path between the vertices $i$ and $j$

does not exist.

A last remark - we don't need to create a separate distance matrix $d_{\text{new}}[][]$

for temporarily storing the shortest paths of the $k$-th phase, i.e. all changes can be made directly in the matrix $d[][]$ at any phase. In fact at any $k$-th phase we are at most improving the distance of

any path in the distance matrix, hence we cannot worsen the length of the shortest path for any pair of the vertices that are to be processed in the $(k+1)$

-th phase or later.

The time complexity of this algorithm is obviously $O(n_3)$

.

## Implementation

Let $d[][]$

is a 2D array of size $n \times n$, which is filled according to the $0$-th phase as explained earlier. Also we will set $d[i][i]=0$ for any $i$ at the $0$-th phase.

Then the algorithm is implemented as follows:

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

It is assumed that if there is no edge between any two vertices $I$ and $j$, then the matrix at $d[i][j]$ contains a large number (large enough so that it is greater than the length of any path in this graph). Then this edge will always be unprofitable to take, and the algorithm will work correctly.

However if there are negative weight edges in the graph, special measures have to be taken. Otherwise the resulting values in matrix may be of the form $\infty-1$, $\infty-2$

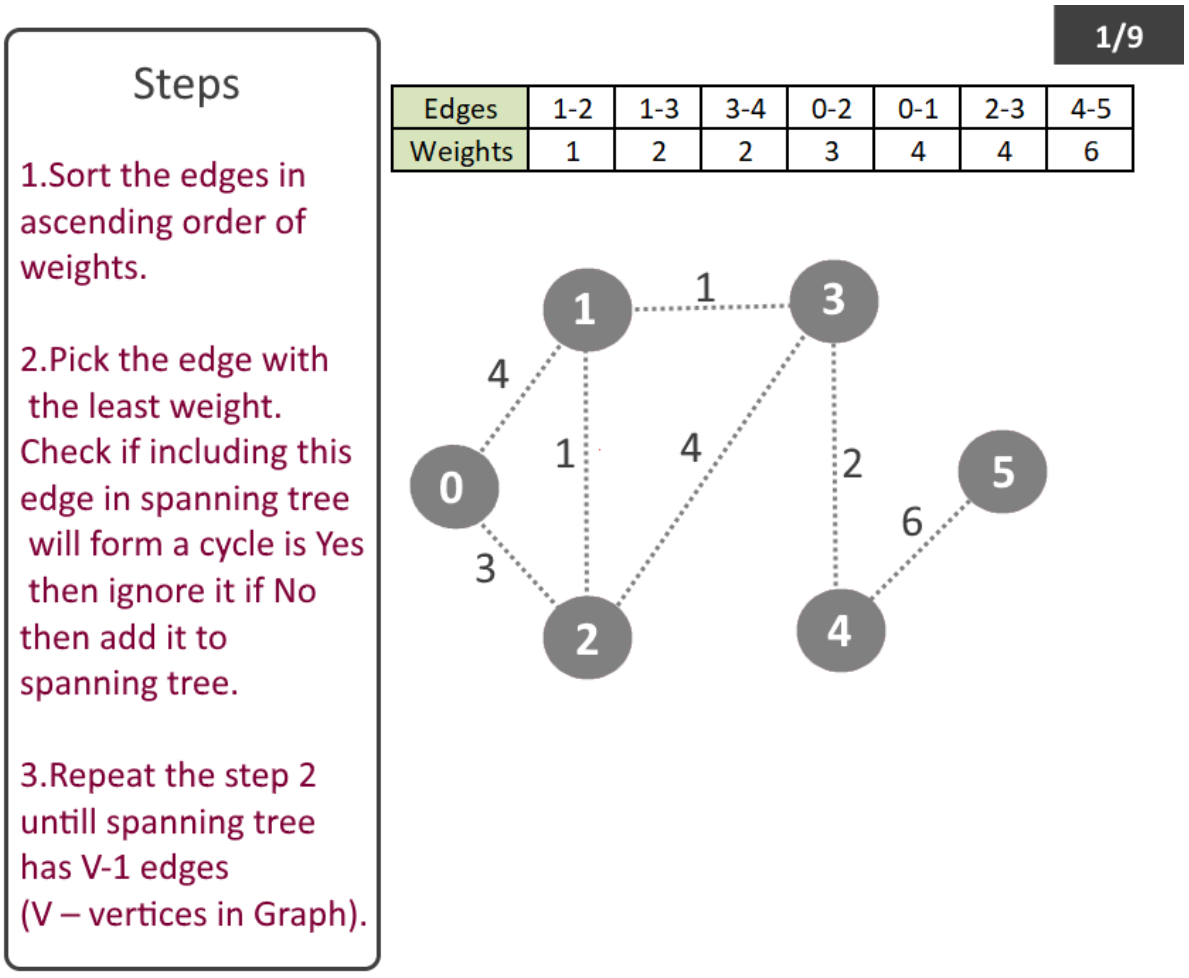
, etc., which, of course, still indicates that between the respective vertices doesn't exist a path. Therefore, if the graph has negative weight edges, it is better to write the Floyd-Warshall algorithm in the following way, so that it does not perform transitions using paths that don't exist.

# KRUSKAL'S MINIMUM SPANNING TREE

## Description

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the `make_set` function - it will take a total of $O(N)$

. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two `find_set` calls in $O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU `union_sets` function will be called - also in $O(1)$. So we get the total time complexity of $O(M\log N+N+M)$ = $O(M\log N)$.



This article discusses the data structure **Disjoint Set Union** or **DSU**. Often it is also called **Union Find** because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:

- `make_set(v)` - creates a new set consisting of the new element `v`
- `union_sets(a, b)` - merges the two specified sets (the set in which the element `a` is located, and the set in which the element `b` is located)
- `find_set(v)` - returns the representative (also called leader) of the set that contains the element `v`. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after `union_sets` calls). This representative can be used to check if two elements are part of the same set of not. `a` and `b` are exactly in the same set, if `find_set(a) == find_set(b)`. Otherwise they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost $O(1)$

time on average.

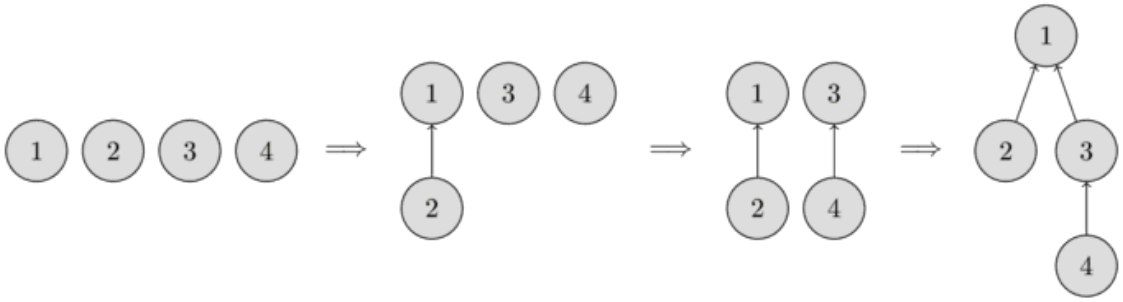
Also in one of the subsections an alternative structure of a DSU is explained, which achieves a slower average complexity of $O(\log n)$

, but can be more powerful than the regular DSU structure.

## Build an efficient data structure

We will store the sets in the form of **trees**: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the following image you can see the representation of such trees.



In the beginning, every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the element 1 and the set containing the element 2. Then we combine the set containing the element 3 and the set containing the element 4. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

For the implementation this means that we will have to maintain an array `parent` that stores a reference to its immediate ancestor in the tree.

# CHECKING A GRAPH FOR ACYCLICITY AND FINDING A CYCLE IN O(M)

## Algorithm

We will run a series of DFS in the graph. Initially all vertices are colored white (0). From each unvisited (white) vertex, start the DFS, mark it gray (1) while entering and mark it black (2) on exit. If DFS moves to a gray vertex, then we have found a cycle (if the graph is undirected, the edge to parent is not considered). The cycle itself can be reconstructed using parent array.

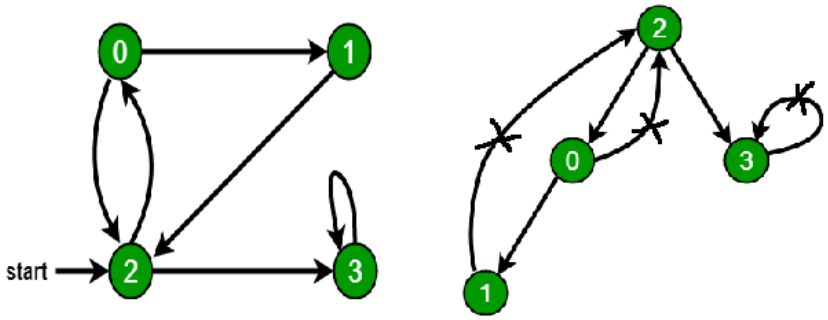
## Implementation

Here is an implementation for directed graph.

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u))
                return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}

void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))
            break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
```

```
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle)
            cout << v << " ";
        cout << endl;
    }
}
```

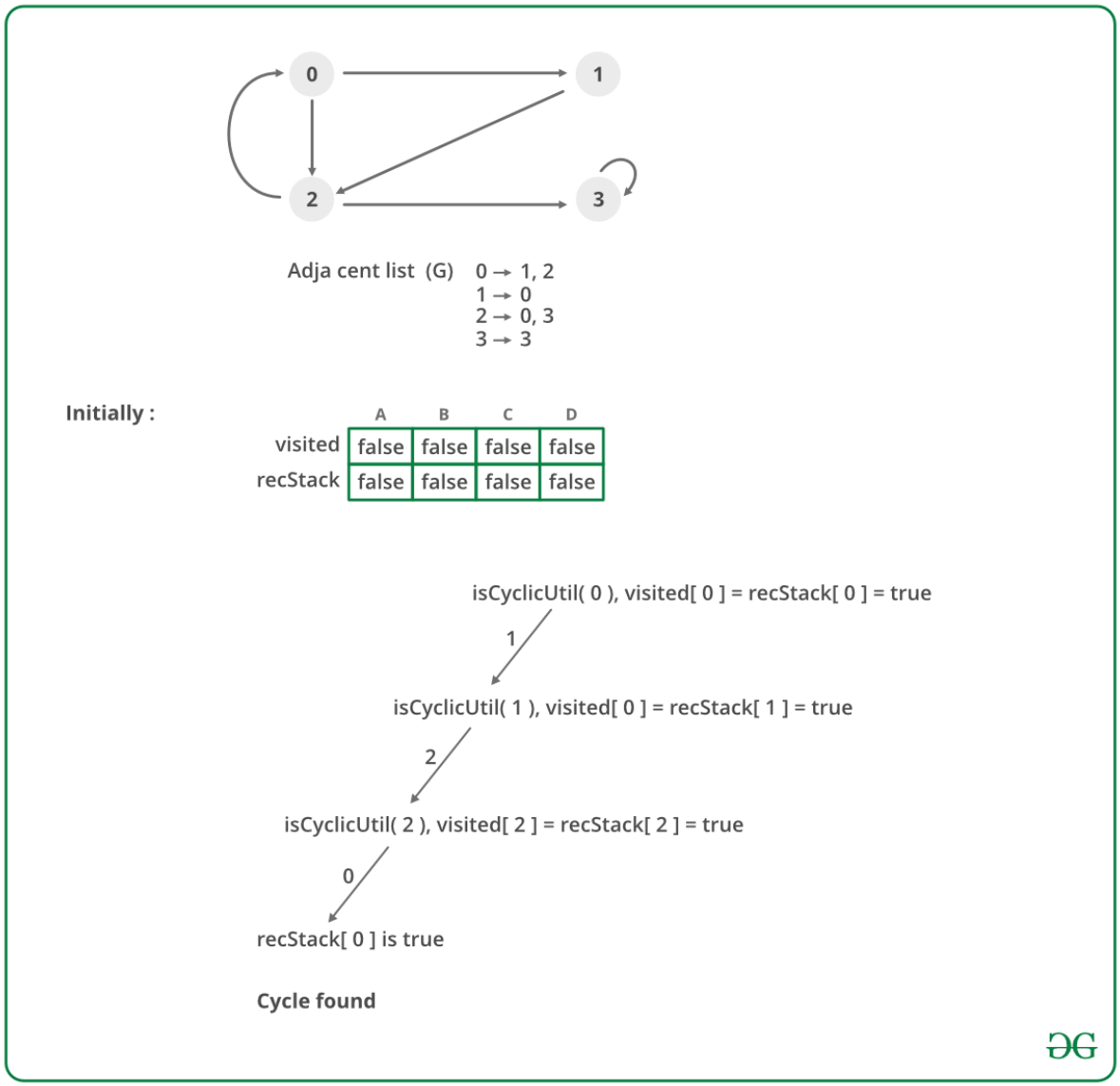
- **Approach:** Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.




For a disconnected graph, Get the DFS forest as output. To detect cycle, check for a cycle in individual trees by checking back edges.

To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. The edge that connects the current vertex to the vertex in the recursion stack is a back edge. Use **recStack[]** array to keep track of vertices in the recursion stack.

**Dry run of the above approach:**

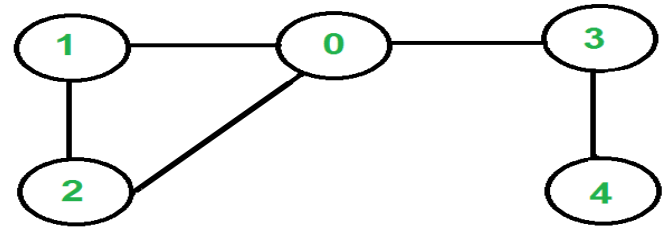



- **Algorithm:**

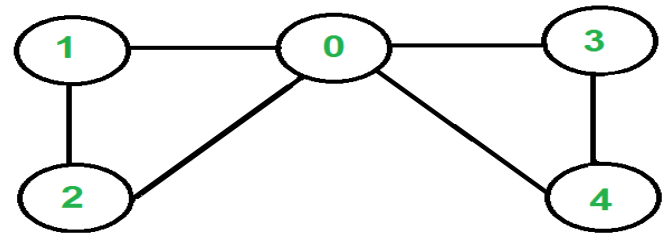
1. Create the graph using the given number of edges and vertices.
2. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
3. Mark the current node as visited and also mark the index in recursion stack.
4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
5. If the adjacent vertices are already marked in the recursion stack then return true.
6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

# *Eulerian path and circuit for undirected graph*

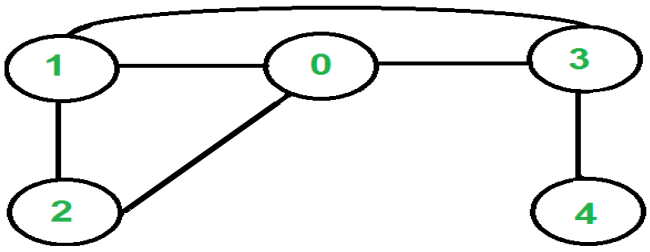
Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.




**The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)**




**The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2" Note that all vertices have even degree**




**The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)**

**How to find whether a given graph is Eulerian or not?**
The problem is same as following question. "Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once".

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to Hamiltonian Path which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

**Eulerian Cycle**
An undirected graph has Eulerian cycle if following two conditions are true.
….a) All vertices with non-zero degree are connected. We don’t care about vertices with zero degree because they don’t belong to Eulerian Cycle or Path (we only consider all edges).
….b) All vertices have even degree.

**Eulerian Path**
An undirected graph has Eulerian Path if following two conditions are true.
….a) Same as condition (a) for Eulerian Cycle
….b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

**How does this work?**
In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

# *Lowest Common Ancestor - Farach-Colton and Bender Algorithm*

Let $G$ be a tree. For every query of the form $(u,v)$ we want to find the lowest common ancestor of the nodes $u$ and $v$, i.e. we want to find a node $w$ that lies on the path from $u$ to the root node, that lies on the path from $v$ to the root node, and if there are multiple nodes we pick the one that is farthest away from the root node. In other words the desired node $w$ is the lowest ancestor of $u$ and $v$. In particular if $u$ is an ancestor of $v$, then $u$

is their lowest common ancestor.

The algorithm which will be described in this article was developed by Farach-Colton and Bender. It is asymptotically optimal.

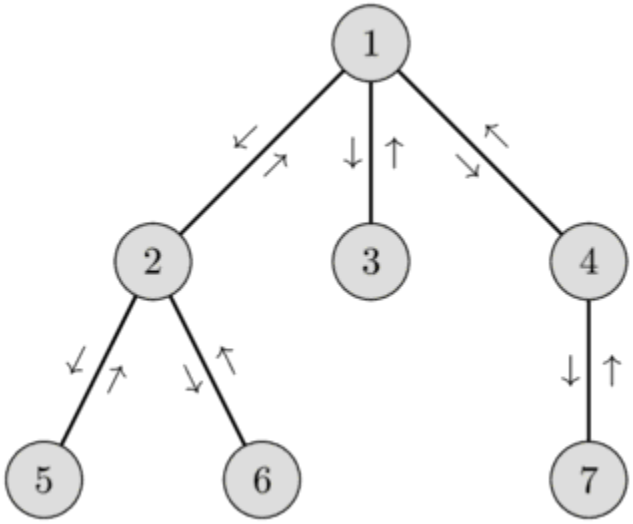
## Algorithm

We use the classical reduction of the LCA problem to the RMQ problem. We traverse all nodes of the tree with DFS and keep an array with all visited nodes and the heights of these nodes. The LCA of two nodes $u$

and $v$ is the node between the occurrences of $u$ and $v$

in the tour, that has the smallest height.

In the following picture you can see a possible Euler-Tour of a graph and in the list below you can see the visited nodes and their heights.




Nodes:Heights:1122532263221132114273421 1

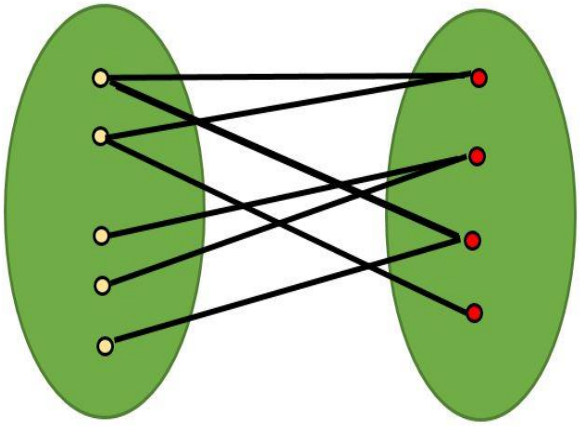
You can read more about this reduction in the article Lowest Common Ancestor. In that article the minimum of a range was either found by sqrt-decomposition in $O(N--\sqrt{})$

or in $O(\log N)$ using a Segment tree. In this article we look at how we can solve the given range minimum queries in $O(1)$ time, while still only taking $O(N)$ time for preprocessing.

# Check whether a graph is bipartite

A bipartite graph is a graph whose vertices can be divided into two disjoint sets so that every edge connects two vertices from different sets (i.e. there are no edges which connect vertices from the same set). These sets are usually called sides.

You are given an undirected graph. Check whether it is bipartite, and if it is, output its sides.



## Algorithm

There exists a theorem which claims that a graph is bipartite if and only if all its cycles have even length. However, in practice it's more convenient to use a different formulation of the definition: a graph is bipartite if and only if it is two-colorable.

Let's use a series of breadth-first searches, starting from each vertex which hasn't been visited yet. In each search, assign the vertex from which we start to side 1. Each time we visit a yet unvisited neighbor of a vertex assigned to one side, we assign it to the other side. When we try to go to a neighbor of a vertex assigned to one side which has already been visited, we check that is has been assigned to the other side; if it has been assigned to the same side, we conclude that the graph is not bipartite. Once we've visited all vertices and successfully assigned them to sides, we know that the graph is bipartite and we have constructed its partitioning.

## Implementation

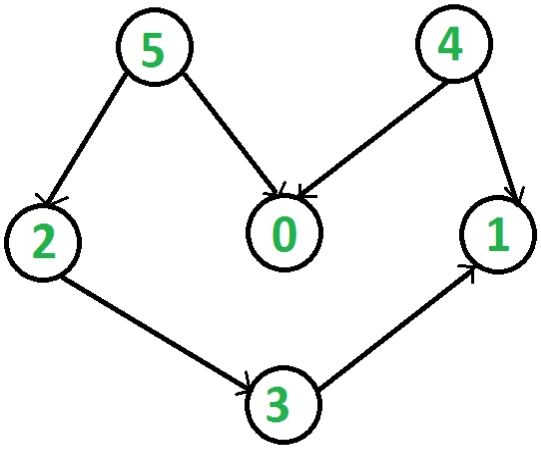
```
int n;
vector<vector<int>> adj;

vector<int> side(n, -1);
bool is_bipartite = true;
queue<int> q;
for (int st = 0; st < n; ++st) {
    if (side[st] == -1) {
        q.push(st);
```

```
        side[st] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int u : adj[v]) {
                if (side[u] == -1) {
                    side[u] = side[v] ^ 1;
                    q.push(u);
                } else {
                    is_bipartite &= side[u] != side[v];
                }
            }
        }
    }
}

cout << (is_bipartite ? "YES" : "NO") << endl;
```

# Topological Sorting

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.

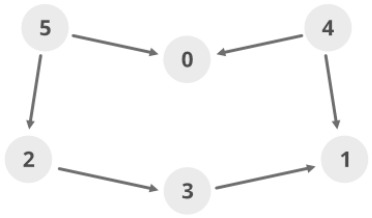
For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).




***Topological Sorting vs Depth First Traversal (DFS)***:
In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

***Algorithm to find Topological Sorting:***
We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Below image is an illustration of the above approach:




Adja cent list (G)
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

visited

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| false | false | false | false | false | false |

Stack( empty )

**Step 1:** Topological Sort( 0 ), visited[ 0 ] = true

↓

List is empty. No more recursion call.

Stack | 0 |

**Step 2:** Topological Sort( 1 ), visited[ 1 ] = true

↓

List is empty. No more recursion call.

Stack | 0 | 1 |

**Step 3:** Topological Sort( 2 ), visited[ 2 ] = true

↓

Topological Sort( 3 ), visited[ 3 ] = true

↓

'1' is already visited. No more recurrsion call

Stack | 0 | 1 | 3 | 2 |

**Step 4:** Topological Sort( 4 ), visited[ 4 ] = true

↓

'0' , '1' are already visited. No more recurrsion call

Stack | 0 | 1 | 3 | 2 | 4 |

**Step 5:** Topological Sort( 5 ), visited[ 5 ] = true

↓

'2' , '0' are already visited. No more recurrsion call

Stack | 0 | 1 | 3 | 2 | 4 | 5 |

**Step 6:** Print all elements of stack from top to bottom