Labs PGX

The report of graph algorithms task

Name: TAUIL Youssef
Mail: Tauilyoussef1996@gmail.com

Contents

Chapter I

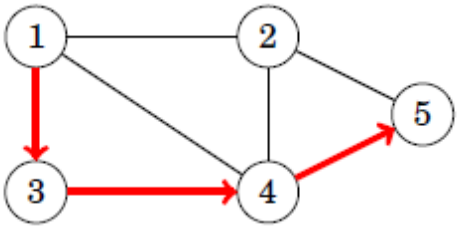
Introduction

Graphs are everywhere; many of us use them or come across them daily. A graph utilizes the basic idea of using vertices to establish relationships between pairs of nodes. In terms of applications, many real world relationships are best modeled using graph structures. A social network such as Facebook, a highway system connecting different cities..., in this introduction, we will discuss concepts and basics of graphs, after we go through their representation and their necessary traversal algorithms using C++ as our language to model them.

I.1 Graph terminology:

• A graph is a non-linear data structure consisting of nodes (vertices) and edges which connect a pair of nodes .

For example, the following consists of 5 nodes and 7 edges:



• A path leads from node a to node b through edges of the graph.

• The length of the path is the number of edges in it. For example the above graph contains a path 1 –> 3 –> 4 –> 5 of length 3 from node 1 to node 5.

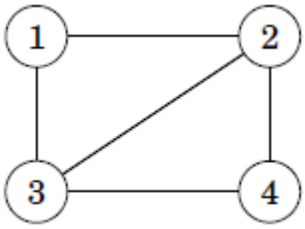
• A path is cycle if the first and last node is the same (ex: 1 –> 3 –> 4 –> 1).

• A path is simple if each node appears at most once in the path.

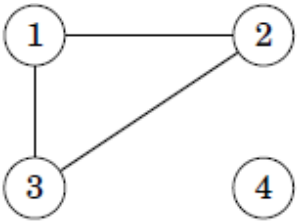
• There are several fundamental properties of graphs witch impact the choice of data structures used to represent them and algorithms available to analyze them , the first step in any graph problem is determining which flavor of graph you are dealing with:

a- Connectivity :

- A graph is connected if there is a path between any two nodes.



(a) Connected

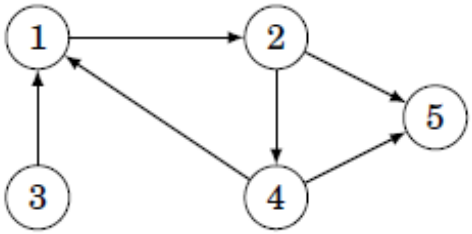


(b) Not connected (forest)

- The connected parts of a forest are called its components.

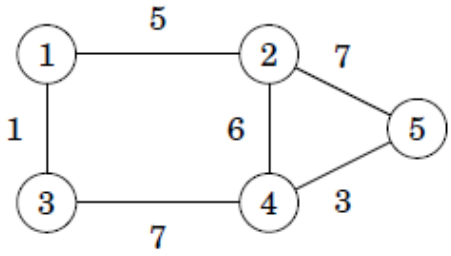
b- Edge directions :

A graph is **directed** if the edges can be traversed in one direction only,All the above examples are for **undirected** graphs, for example the following graph is **directed**:



c- Edge weights :

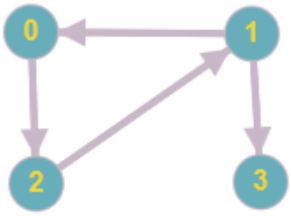
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths, for example the following graph is **weighted**:



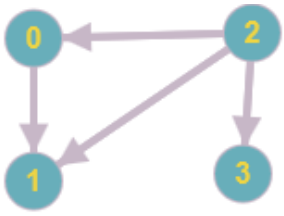
For example, the length of the path 1 –> 2 –> 5 is 12.

d- Cyclic Vs. Acyclic :

An **acyclic** graph does not contain any cycles, Trees are connected **acyclic** undirected graphs, let treat those examples:



(a) Cyclic



(b) Acyclic

The first graph is cyclic because we have a cycle path between the nodes 0,1 and 2. But on the second graph we can't find a cycle path.

Note : if the graphs are undirected the both become cyclic .

I.2 Graph representation:

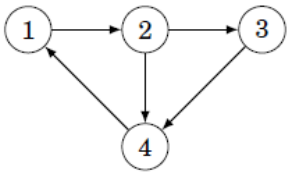
Graph is a data structure that consists of finite set of vertices, and a finite set of ordered pair of the form (u,v) as edges ((u,v) is not same as (v,u) in case of directed graph).

For the choice of my system, I'm going to use the language $C++$ and their standard libraries to implement our graph structure and algorithms.

Next we will go through two common representations, and explain our choice of the better representation we need to implement those algorithms.

a- Adjacency matrix :

An adjacency matrix is a two-dimensional array that indicates witch edges the graph contains.

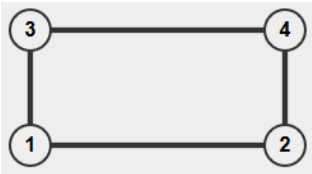


	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

If the graph is weighted, we replace the ones by the weights. Even this representation is easier to implement and take $O(1)$ time to check or update a given edge, the representation cannot be used if the graph is large because of the waste memory (most of the cases are zero).

b- Adjacency list :

In the adjacency list representation, each node x is assigned an adjacency list that consists of nodes to which there is an edge from x. A convenient way to store the adjacency list is to declare a vector of vectors.



Adjacency List		
0:		
1:	2	3
2:	1	4
3:	1	4
4:	2	3

If the graph is undirected, we add each edge in both directions. For the weighted graph, the structure can be extended to vector of vectors of pair. The adjacency list of node a contains the pair (b,w) when there is an edge from node a to node b with weight w .

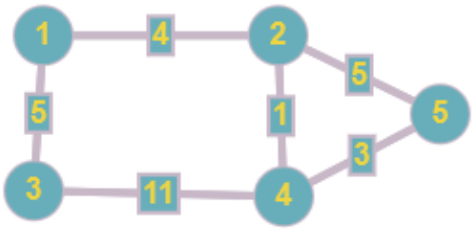
Now we will see the code in $C{+}{+}$ for the adjacency list that we are going to use on our algorithms because of his minimum memory and best complexity on implementing the inputs $O(N+M) < O(N^2)$ (when N number of nodes and M number of edges).

Let implement an example of a weighted undirected graph:

```
#include <bits/stdc++.h>
using namespace std;

int n,m;  // n: number of nodes ; m: number of edges
vector<vector< pair<int,int> > > graph; // adjacency list of weighted graph

int main() {
        cout << " number of nodes : "; cin >> n  ;
        cout << " number of edges : "; cin >> m ;
        graph.clear();
        graph.resize(n);
        // adjacency list inputs, let suppose the nodes from 1 to n
        cout << " let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b,w;
                cin >> a >> b >> w ; // edge from a to b with weight w
                a--; b--;
                // we use the case memory [0] for node 1, 1 for 2 ... and so on,
                // to minimize the memory
                // we can declare resize(n+1) but it waste
                //of case graph[0] from the vector .
                graph[a].push_back(make_pair(b,w));
                graph[b].push_back(make_pair(a,w));// indirected graph
        }
        cout << "The adjacency list is :" << endl;
```

```
        for(int i=0;i<n;i++){ // loop for nodes
                int node=i;
                cout << node +1 << " : ";
                for(int j=0;j<graph[i].size();j++){// loop for connected edges
                        cout << "(" << graph[i][j].first + 1 << " , " << graph[i][j].sec
                        <<((j==graph[i].size()-1)?"":";") ;
                }
                cout << endl ;
        }

        return 0;
}
```

- Input and output example:



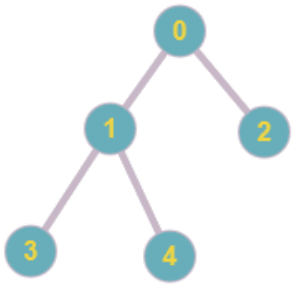

```
number of nodes : 5
number of edges : 6
let create some edges :
1 2 4
1 3 5
3 4 11
2 4 1
2 5 5
4 5 3
The adjacency list is :
1 : (2 , 4);(3 , 5)
2 : (1 , 4);(4 , 1);(5 , 5)
3 : (1 , 5);(4 , 11)
4 : (3 , 11);(2 , 1);(5 , 3)
5 : (2 , 5);(4 , 3)
```

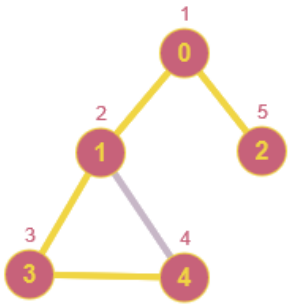
I.3 Traversal graph:

This chapter discusses two fundamental graph algorithms: depth-first search (DFS) and breadth-first search (BFS). Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from a starting node. The difference in the algorithms is the order in which they visit the nodes.

a- Depth-first search:

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. Let consider how depth-search process that graph:





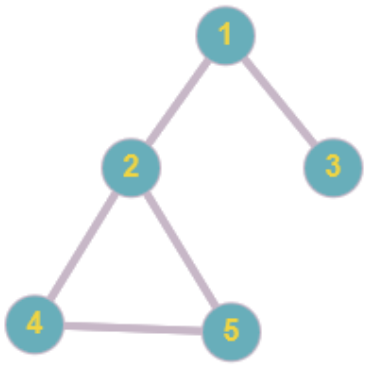
As we see on the two figures, the algorithm keeps track of visited nodes, so that it processes each node only once.

There are two common implementations for DFS, recursive version and stack version. Next we will go through the code on $C++$ and explain our example:

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int> > graph;
const int N = 1e5;
int n,m;
bool visited[N]; // for avoiding process the same node ( if vis[i]=true we continue )
// visited[N]={ 0 or false } because it's a global declaration for the variable

void dfs(int u) {
        visited[u] = 1;
        cout <<  u + 1 <<"  ";
        for (int i = 0; i < graph[u].size(); i++)
                if (!visited[graph[u][i]]){
                        dfs(graph[u][i]);
                }
}

void dfs_stack(int u) {
        stack <int>  s ;
        visited[u] = 1 ; s.push(u) ;
        while (!s.empty()) {
                int v = s.top() ;
                cout << v + 1 << "  " ;
                s.pop() ;
                for (int i = 0; i < graph[v].size(); i++)
                        if (!visited[graph[v][i]]) {
                                visited[graph[v][i]] = 1;
                                s.push(graph[v][i]);
                        }
```

```
        }
}

int main() {
        cout << "number of nodes : ";
        cin >> n ;
        cout << "number of edges : ";
        cin >> m ;
        graph.clear();
        graph.resize(n);
        // adjacency list inputs, let suppose the nodes from 1 to n
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b ;
                cin >> a >> b ;
                a-- ; b-- ;
                graph[a].push_back(b) ;
                graph[b].push_back(a) ;// indirected graph
        }
        cout << "Begin a DFS from the node : " << endl ;
        int node ; cin >> node ; node-- ;
        memset(visited, 0, n) ;// we fill the visited array with 0 of memory n
        dfs(node) ;

        memset(visited, 0, n) ;// we refill the visited array with 0 of memory n
        dfs_stack(node) ;

        return 0 ;
}
```

- Input and output example:




```
number of nodes : 5
number of edges : 5
let create some edges :
1 2
1 3
2 4
2 5
4 5
Begin a DFS from the node :
1
the recursive version :
1  2  4  5  3
the stack version :
1  3  2  5  4
```

Recursive version:

Every time we find not visited node from the root we call our function dfs("processed node") until there is no nodes to processed, it like Fibonacci calls, we execute from last call to the first and on the way we explore other calls for not visited nodes. Every call for dfs(node) mean that we visit a new node and visited[node] become true.

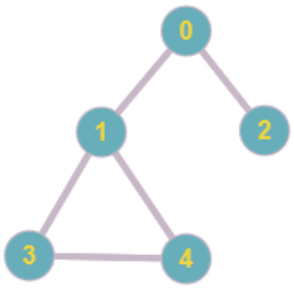
Stack version:

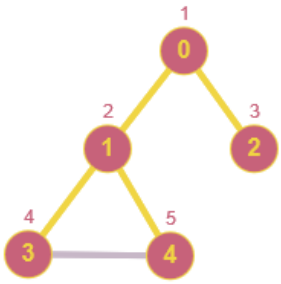
For the utility of the stack, we push all the accessible nodes from the root to the stack and they become visited, after we use the property of the stack (last in first out) to process the last node has been pushed, and so on we go deep on the path until there is no nodes to be processed (already visited or we reach the leaf).

For the two implementations, the complexity still the same $O(N + M)$ (N number of nodes and M number of edges) because the algorithm process each node and edge once on the traversal.

b- Breadth-first search:

Breadth-first search visits the nodes in increasing order of their distance from the starting node. It goes through the nodes one level after another until all the nodes have been visited. Let consider how depth-search process that graph:



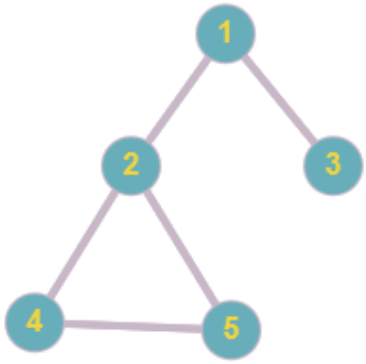


As we see on the two figures, the algorithm goes through the nodes level by level until they become all visited.

There is only one implementation for BFS when we use a queue to process our nodes. Next we will see the code on $C++$ and explain our example:

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<vector<int> > graph;
const int N = 1e5;
int n,m;
bool visited[N];

void bfs(int u) {
        queue <int>  s ;
        visited[u] = 1 ;
        s.push(u) ;
        while (!s.empty()) {
                int v = s.front() ;
                cout << v + 1 << " " ;
                s.pop() ;
                for (int i = 0; i < graph[v].size(); i++)
                        if (!visited[graph[v][i]]) {
                                visited[graph[v][i]] = 1 ;
                                s.push(graph[v][i]) ;
                        }
        }
}

int main() {
        cout << "number of nodes : "; cin >> n ;
        cout << "number of edges : "; cin >> m ;
        graph.clear() ;
        graph.resize(n) ;
        // adjacency list inputs, let suppose the nodes from 1 to n
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b ;
                cin >> a >> b ;
                a-- ; b-- ;
                graph[a].push_back(b) ;
                graph[b].push_back(a) ;// indirected graph
        }
        cout << "Begin a BFS from the node : " << endl;
        int node ; cin >> node ; node-- ;
        memset(visited, 0, n) ;
        bfs(node) ;

        return 0;
}
```

- Input and output example:




```
number of nodes : 5
number of edges : 5
let create some edges :
1 2
1 3
2 4
2 5
4 5
Begin a BFS from the node :
1
1 2 3 4 5
```

The complexity is the same as DFS $O(N + M)$ because it's just difference on the way of visiting nodes(visit each node and edge once).

Now after we set up our graph analytics system, we will try to answer the given algorithms, each one on a chapter, let's begin by finding path.

Chapter II

Finding path

Path finding is a huge topic to be answered on one algorithm; there are several path algorithms for different purposes, but the most useful and practical ones are for finding the shortest path between two nodes and they depend on the restrictions of the problem, I will treat three algorithms of finding those shortest paths, each one is the best solution or even the unique one for the type of problem, let's see together how we process those algorithms.

II.1 Unweighted graphs:

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find the shortest path (the easier problem), first of all, we will define the restrictions of the algorithm and after explain the solution code.

For that algorithm, it need just to be unweighted , there is no problem with cycles or directions choice.

While we use a BFS approach, it's obvious that the complexity still the same $O(N + M)$ (N number of nodes and M number of edges).

For the process, we just need to add a distance array that begin with 0 on the root and infinite value for the rest nodes, we begin a BFS with an addition of 1 at every time we pass from level to another.

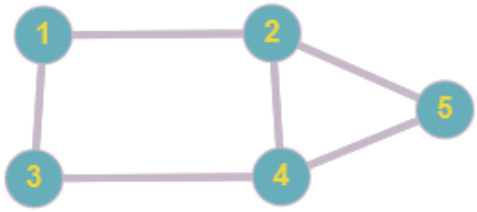
Now, let's see the code on $C++$ and explain our example:

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int> > graph;
const int N = 1e5;
int n,m;
bool visited[N];
int dist[N]={0}; // array of distance for each node
int path[N];// array of stocking the paths
```

```
void bfs_distance(int u) {
        for(int i=0;i<n;i++)
        path[i]=i;
        queue<int> s;
        visited[u] = 1;
        s.push(u);
        while (!s.empty()) {
                int v = s.front();
                s.pop();
                for (int i = 0; i < graph[v].size(); i++)
                        if (!visited[graph[v][i]]) {
                                visited[graph[v][i]] = 1;
                                dist[graph[v][i]]=dist[v]+1;
                                path[graph[v][i]]=v;
                                s.push(graph[v][i]);
                        }
        }
}


int main() {
        cout << "number of nodes : ";
        cin >> n ;
        cout << "number of edges : ";
        cin >> m ;
        graph.clear();
        graph.resize(n);
        // adjacency list inputs, let suppose the nodes from 1 to n
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b;
                cin >> a >> b ; // edge from a to b with weight w
                a--; b--;
                graph[a].push_back(b);
                graph[b].push_back(a);// indirected graph
        }
        cout << "Shortest path from root to given destination : " << endl;
        int root,destination ; cin >> root >> destination ; root--; destination--;
        memset(visited, 0, n);
        bfs_distance(root);
        cout <<"the answer is: "<< dist[destination] << endl;
        cout << "the path taken is through : " <<  endl;
        stack<int> ans;
        ans.push(destination);
        while(path[destination]!=destination){ // every time path[i]!=i there is path
```

```
        ans.push(path[destination]);// we add the node of the path
        destination=path[destination];// continue the process on the new node
    }
    while(!ans.empty()){
        cout<<ans.top()+1<<' ';
        ans.pop();
    }

        return 0;
}
```

- Input and output example:




```
number of nodes : 5
number of edges : 6
let create some edges :
1 2
1 3
3 4
2 4
2 5
4 5
Shortest path from root to given destination :
1 5
the answer is: 2
the path taken is through :
1 2 5
```

II.2 Weighted graphs:

In the weighted graph, we will process two main algorithms for the shortest path, Dijkstra and Bellman-ford, each one of them treat different restrictions. First, let's begin by Dijkstra as simple one.

II.2.1 Dijkstra : (positive edges)

The benefit of dijkstra's algorithm is that it is more efficient than bellman-ford and can be used for processing large graphs; however, the algorithm requires that there are no negative weight edges in the graph.

The algorithm select initially a root node with cost equal 0 and all others nodes with infinite cost, after we begin our process from the root and goes through all edges that start at the node and reduces the distance using their weights on an descendant order , and so on we continue the process for the remaining nodes.

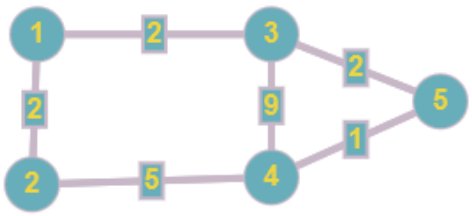
A remarkable note is that whenever a node is selected, its distance is final because we treat all the edges surrounded it on that level.

Next, let's see the code on $C++$ and explain it with given example:

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 ;
vector<vector<pair<int,int> > > g ;
int n,m ;
int cost[N] ;
int path[N] ;// array of stocking the paths

void dijkstra(int s){
    for(int i=0;i<n;i++)
        cost[i]=1e9 ;
    cost[s]=0 ;
    for(int i=0;i<n;i++)
        path[i]=i ; // array for keeping track of the taken paths
        // example : p[1]=2 means that 1 -> 2 and so on...
    priority_queue<pair<int,int> > pq ; // ( default order : ascendant maximum value )
    pq.push(make_pair(0,s)) ;
        //order priority based on the costs.
    while(!pq.empty()){
        int u=pq.top().second , cst=-pq.top().first ;
                //the minus sign stand for returning
        // to the real value, because we store negative costs
                //to reverse the order.
        pq.pop() ;
        if(cost[u]<cst)
            continue ;
        // for that condition , let assume that we have on the queue [ (-3,4),(-7,4)]
                //we begin by treating (-3,4), and because cost[4]=7 > cst=3 (at most 3)
                // there is no need to process the node 4 again with higther cost .
        for(int i=0;i<g[u].size();i++){
            int v=g[u][i].first , c=g[u][i].second;
            if(cost[v]>cst+c){
                cost[v]=cst+c ;
                path[v]=u ;
                pq.push(make_pair(-cost[v],v)) ;
            }
        }
    }
}


int main() {
        cout << "number of nodes : " ; cin >> n ;
        cout << "number of edges : " ; cin >> m ;
```

```
        g.clear();
        g.resize(n);
        // adjacency list inputs, let suppose the nodes from 1 to n
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b,w ;
                cin >> a >> b >> w ; // edge from a to b with weight w
                a-- ; b-- ;
                g[a].push_back(make_pair(b,w)) ;
                g[b].push_back(make_pair(a,w)) ;// indirected graph
        }
        int from,to ;
        cout << "Source : " ; cin >> from ;
        cout << "To : " ; cin >> to ;
        from-- ; to-- ;
    dijkstra(from) ;
    cout <<"Shortest path from "<< from+1 <<" to "<< to+1 <<" is "<< cost[to]<< endl;
    cout << "the path taken is through : " <<  endl;
    stack<int> ans ;
    ans.push(to) ;
    while(path[to]!=to){
        ans.push(path[to]) ;
        to=path[to] ;
    }
    while(!ans.empty()){
        cout<<ans.top()+1<<' ' ;
        ans.pop() ;
    }

        return 0;
}
```

- Input and output example:

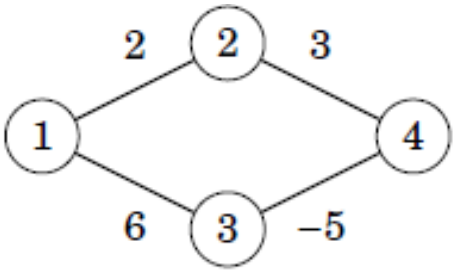



```
number of nodes : 5
number of edges : 6
let create some edges :
1 2 2
1 3 2
2 4 5
3 4 9
4 5 1
3 5 2
Source : 4
To : 1
Shortest path from 4 to 1 is 5
the path taken is through :
4 5 3 1
```

The time complexity of the above implementation is $O(N + M * log(M))$, because the algorithm goes through each node and edge once and add for each node at least one distance to the priority queue who needs $log(M)$ to position the distance on the right place using a binary search.(that $log(M)$ who makes BFS more efficient on the case of unweighted graph)

Why dijkstra doesn't work on negative edges?

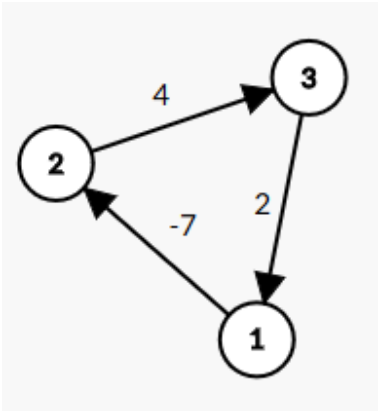
Counter example:



The shortest path from node to node 4 is 1->3->4 with length 1, but dijkstra's algorithm finds the path 1->2->4 (cost[2]=2 < cost[3]=6 and cost[4]=5 < cost[3]=6) by following the minimum weight edges, so the algorithm doesn't take into account that on the other path, the weight -5 compensates the previous large weight 6.

II.2.2 Bellman-ford: (negative edges)

The bellman-ford algorithm finds shortest paths from starting node to all nodes of the graph, and what makes him special than dijkstra is that he treat also the case of negative edges but the graph mustn't contain a cycle with negative length, let's see an example of that type of cycle:



As we see the length of that cycle is negative (-1).

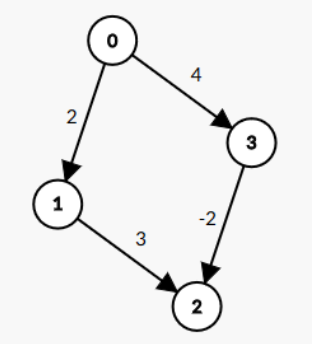
The algorithm consists of n-1 rounds (in case of complete graph we had n-1 possible paths and the shortest could be the last[brute force]), and on each round we go through all the edges with specified order, and we go again until there is no relaxation on the edges (minimisation of the cost).

Next, let's see the code on $C++$ and explain it with given example:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5;
vector<vector<pair<int,int> > >g;
int n,m;
int cost[N];
int path[N];

void bellman_ford(int s){

        for(int i=0;i<n;i++)
        cost[i]=1e9;

        cost[s]=0;

    for(int k=0;k<n;++k){
        bool flag = true;
        for(int i=0;i<n;i++){
            for(int j=0;j<g[i].size();j++){
                    int v=g[i][j].first , c=g[i][j].second;// i -> v with cost c
                if(cost[v]>cost[i]+c){
                    cost[v] = cost[i]+c ;
                    path[v]=i;
                    flag = false;
                }
            }
        }
        if(flag)// we finish the process if the costs still the same
            break;
    }

}
```

```
int main() {
        cout << "number of nodes : ";
        cin >> n ;
        cout << "number of edges : ";
        cin >> m ;
        g.clear();
        g.resize(n);
        // edgeslist inputs, let suppose the nodes from 0 to n-1
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b,w;
                cin >> a >> b >> w ; // edge from a to b with weight w
                //a--; b--;
                g[a].push_back(make_pair(b,w));
                //g[b].push_back(make_pair(a,w));// indirected graph
        }
    int from,to ;
    cout << "Source : " ; cin >> from ;
    cout << "To : " ; cin >> to ;
    //from-- ; to-- ;
    bellman_ford(from) ;
    cout <<"Shortest path from " << from << " to " << to << " is "<< cost[to] << endl;
    cout << "the path taken is through : " <<  endl;
    stack<int> ans ;
    ans.push(to) ;
    while(path[to]!=to){
        ans.push(path[to]) ;
        to=path[to] ;
    }
    while(!ans.empty()){
        cout<<ans.top()<<' ' ;
        ans.pop() ;
    }

        return 0;
}
```

- Input and output example:




```
number of nodes : 4
number of edges : 4
let create some edges :
0 1 2
0 3 4
1 2 3
3 2 -2
Source : 0
To : 2
Shortest path from 0 to 2 is 2
the path taken is through :
0 3 2
```

The time complexity of the above implementation is $O(N*M)$ because we had n-1 rounds, and on each round we go through the edges, on the worst case (complete graph) we had all the possible edges so $M = n*(n-1)/2$,then it's become $O(N^3)$.

- Note: (negative cycles)

The Bellman-ford algorithm doesn't work on negative cycles, because the edges keep reducing costs on finite loop that we finish it on the last round, and that useless,but we can benefit the algorithm for checking if there is a negative cycle by going through n rounds and see if the relaxation of the edges continue.

Chapter III

Ranking

In this chapter, we focus on specific type of graphs, the D.A.G directed acyclic graphs, and ranking is a linear ordering of vertices such that for every directed edge u->v, vertex u comes before v in the ordering.

To construct a ranking for vertices, we need to calculate the indegree edges for each vertices and begin with any vertices that had 0 indegree, and when we process their children and minimize their indegree by 1 and push it in the answer if one of those indegrees become 0.

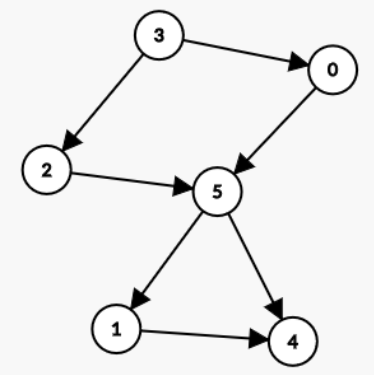
Next, let's see the code on $C++$ and explain it with given example:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5;
vector<vector<int> >g;
vector<int> in,canUse,ans;// in : indegree of nodes
// canuse : the nodes with indegree=0
// ans : flag for cycle
int n,m;

int main() {
        cout << "number of nodes : " ; cin >> n ;
        cout << "number of edges : " ; cin >> m ;
        g.clear();
        g.resize(n);
        in.resize(n);
        // adjacency list inputs, let suppose the nodes from 0 to n-1
        cout << "let create some edges :" << endl;
        for(int i=0;i<m;i++){
                int a,b;
                cin >> a >> b  ; // edge from a to b
                g[a].push_back(b);
                in[b]++; // store the indegree for each vertex
        }
```

```
        // beginig vertices ( indegree[i]=0)
        for(int i=0;i<n;i++)
        if(in[i] == 0)
            canUse.push_back(i);

    while(!canUse.empty()){
        int u = canUse[canUse.size()-1];
        canUse.pop_back();
        ans.push_back(u);
        for(int i=0;i<g[u].size();i++){
            in[g[u][i]]--; // minimize the indegree by 1
            if(in[g[u][i]]==0){
                canUse.push_back(g[u][i]);
            }
        }
    }

    if(ans.size() != n)
        puts("Cycle detected !! ");
    else{
        puts("One of the possible ranking is :");
        for(int i=0;i<ans.size();i++){
            cout<<ans[i]<<((i==ans.size()-1)?"":" -> ");
        }
    }
        return 0;
}
```

The time complexity for the ranking algorithm is $O(N + M)$.

- Input and output example:




```
number of nodes : 6
number of edges : 7
let create some edges :
3 0
3 2
0 5
2 5
5 1
5 4
1 4
One of the possible ranking is :
3 -> 2 -> 0 -> 5 -> 1 -> 4
```

Chapter IV

Strongly connected components

In this chapter, we will treat the case of directed graphs because any connected undirected graph it already strongly connected.

So what's the definition of strongly connected components?

- Is the maximum of set of vertices such as all the vertices are reachable from the others.

$$\forall (a,b) \in SCC, a \to b \Leftrightarrow b \to a$$

The idea of the algorithm consist of using two arrays the first for indexes and the second for the lowest index of children (if there's a back-edge we take the minimum of the vertex and the children), without forget to push the visited vertices if not already visited (DFS), after we minimize the low of all the vertices, we go back to our visited vertices to give them and index of component (pop the from the last) every time we found the low equal to the given index.

Next, let's see the code on $C++$ and explain it with given example:

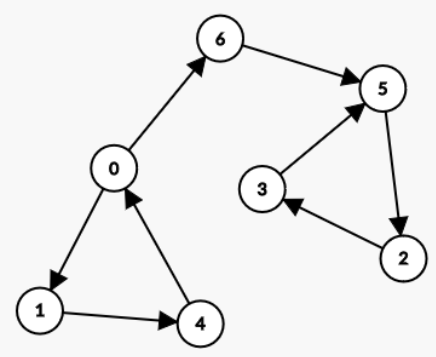
```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int> > g;
const int N = 100000;
int n,m;

int indx[N],low[N],dfsTime;// time of visited node by DFS
// low stand for the indx[lowest children]
vector<int> s;//the visited vertices
bool vis[N];
int compID[N] ,cmp;// components wuth same id
```

```
void DFS(int u){
    indx[u] = low[u] = ++dfsTime;
    s.push_back(u);
    vis[u]=true;

    for(int i=0;i<g[u].size();i++){
        int v=g[u][i];
        if(indx[v]==0)
            DFS(v);
        if(vis[v])
            low[u]=min(low[u],low[v]);
    }
    if(indx[u]==low[u]){
        while(true){
            int v=s.back();
            s.pop_back();
            vis[v]=false;
            compID[v]=cmp;
            if(v==u)
                break;
        }
        ++cmp;
    }
}

int main(){

    cout << "number of nodes : "; cin >> n ;
    cout << "number of edges : "; cin >> m ;
    g.clear();
    g.resize(n);

    cout << "let create some edges :" << endl;
    for(int i=0;i<m;i++){
        int a,b;
                cin>>a>>b;
        g[a].push_back(b);
    }

    for(int i=0;i<n;i++)
        if(indx[i]==0)
            DFS(i);


        vector<vector<int> >res;
```

```
        res.resize(cmp);
        for(int i=0;i<n;i++){
                res[compID[i]].push_back(i);
        }
        cout<<"Their are "<<cmp<<" strongly components :"<<endl;
        for(int i=0;i<cmp;i++){
                cout<<"Elements of component "<<i+1<<" are :"<<endl<<"{";
                for(int j=0;j<res[i].size();j++){
                        cout<<res[i][j]<<((j==res[i].size()-1)?"":",");
                }
                cout<<"}"<<endl;
        }


    return 0;
}
```

The time complexity for the ranking algorithm is O(N+N+M)= O(N+M) because it's an approach of DFS and the process of the vector s (push and pop all vertices).

- Input and output example:




```
number of nodes : 7
number of edges : 8
let create some edges :
0 6
0 1
1 4
4 0
6 5
5 2
2 3
3 5
Their are 3 strongly components :
Elements of component 1 are :
{2,3,5}
Elements of component 2 are :
{6}
Elements of component 3 are :
{0,1,4}
```

Conclusion

On conclusion, the graph theory is huge topic on algorithms,I do have a basic knowledge about several topics, but during this work on the given tasks I applied my theoretical learning and that benefit me a lot on touching the details of each algorithm and implementing the code, I'm really excited about the use of data structures and specially graphs .