Teambook Sindicato de Transporte 2880

Universidad Mayor de San Simón

March 3, 2023

Contents

Chapter 1

Introduction

The following document represents the Teambook for the team Sindicato de Transporte 2880. This version was elaborated for the Latin American Regional phase of 2022's ICPC.

The template for the C++ code is presented:

```
#include <bits/stdc++.h>
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
// #include <ext/rope>
#define int ll
#define mp        make_pair
#define pb        push_back
#define all(a)      (a).begin(), (a).end()
#define sz(a)      (int)a.size()
#define eq(a, b)       (fabs(a - b) < EPS)
#define md(a, b)    ((a) % b + b) % b
#define mod(a)      md(a, MOD)
#define _max(a, b) ((a) > (b) ? (a) : (b))
#define srt(a)      sort(all(a))
#define mem(a, h)    memset(a, (h), sizeof(a))
#define f         first
#define s         second
#define forn(i, n)      for(int i = 0; i < n; i++)
#define fore(i, b, e)  for(int i = b; i < e; i++)
#define forg(i, b, e, m)  for(int i = b; i < e; i+=m)
#define index  int mid = (b + e) / 2, l = node * 2 + 1, r = l + 1;
#define DBG(x) cerr<<#x<<"␣=␣"<<(x)<<endl
#define RAYA cout<<"=============================="<<'\n'
// int in(){int r=0,c;for(c=getchar();c<=32;c=getchar());if(c=='-')
     return -in();for(;c>32;r=(r<<1)+(r<<3)+c-'0',c=getchar());
    return r;}


using namespace std;
// using namespace __gnu_pbds;
// using namespace __gnu_cxx;

// #pragma GCC target ("avx2")
// #pragma GCC optimization ("O3")
// #pragma GCC optimization ("unroll-loops")

typedef long long     ll;
typedef long double ld;
typedef unsigned long long     ull;
typedef pair<int, int>  ii;
typedef pair<pair<int, int>, int> iii;
typedef vector<int>     vi;
typedef vector<ii>      vii;
typedef vector<ll>      vll;
// typedef tree<int,null_type,less<int>,rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;
// find_by_order kth largest  order_of_key <
// mt19937 rng(chrono::steady_clock::now().time_since_epoch().count
    ());
// rng
const int tam = 200010;
const int MOD = 1000000007;
```

```cpp
const int MOD1 = 998244353;
const double DINF=1e100;
const double EPS = 1e-9;
const long double PI  = acos(-1.0L);



signed main()
{
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    // freopen("asd.txt", "r", stdin);
    // freopen("qwe.txt", "w", stdout);


    return 0;
}
```

../Template.cpp

Chapter 2

Mathematics

This chapter is about some useful mathematical tools needed in order to solve problems.

2.1 GCD and LCM

In order to find the greatest common divisor (GCD) of two numbers, the Euclidean algorithm can be used. The implementation is as follows:

```
ll gcd(ll a, ll b){return b==0? a:gcd(b,a%b);}

int x, y, d;
void extendedEuclid(int a, int b)//ecuacion diofantica ax + by = d
{
   if(b==0) {x=1; y=0; d=a; return;}
   extendedEuclid(b,a%b);
   int x1=y;
   y = x-(a/b)*y;
   x=x1;
}
```

../Mathematics/Euclid.cpp

Another (and faster) way to find the GCD is by using the following code:

```
int gcd(int a, int b) {
    if (!a || !b)
        return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
    do {
        b >>= __builtin_ctz(b);
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b);
    return a << shift;
}
```

../Mathematics/FastGCD.cpp

The way Halim suggests to find the GCD and the LCM is given by the following code:

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

../Mathematics/HalimGCD.cpp

2.2 Prime Numbers

The fastest way to check the primality of a number is by using Erathostenes' sieve. The typical implementation is as follows:

```
bitset<100000> bi;
vi primos;  //primos
vector<ll> pric;  //primos al cuadrado
void criba()
{
   bi.set();
```

```
   for(int i=2;i<100000;i++)
      if(bi[i])
      {
         for(int j=i+i;j<100000;j+=i)
            bi[j]=0;
         primos.push_back(i);
         pric.push_back((ll)i*(ll)i);
      }
}
int euler(int n)
{
   int res=n;
   for(int i=0;pric[i]<=n;i++)
   {
      if(n%primos[i]==0)
      {
         res-= res/primos[i];
         while(n%primos[i]==0) n/=primos[i];
      }
   }
   if(n!=1) res-=res/n;
   return res;
}
```

../Mathematics/Erathostenes.cpp

Nevertheless, the following implementation is faster, since the statement if

```
if (i % prime[j] == 0) break;
```

terminates the loop when p divides i. The inner loop is executed only once for each composite. Hence, the code performs in O(n) complexity, resulting in the 'linear' sieve:

```
// This algorithm allows to find Eratosthenes sieve in O(n logn)
    time.

std::vector <int> prime;
bool is_composite[MAXN];

void sieve (int n) {
   std::fill (is_composite, is_composite + n, false);
   for (int i = 2; i < n; ++i) {
      if (!is_composite[i]) prime.push_back (i);
      for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
         is_composite[i * prime[j]] = true;
         if (i % prime[j] == 0) break;
      }
   }
}

// An application of this lineatr sieve is to find the Euler
    totient function of a number in O(n logn) time.
std::vector <int> prime;
bool is_composite[MAXN];
int phi[MAXN];

void sieve (int n) {
   std::fill (is_composite, is_composite + n, false);
   phi[1] = 1;
   for (int i = 2; i < n; ++i) {
      if (!is_composite[i]) {
         prime.push_back (i);
         phi[i] = i - 1;               //i is prime
      }
      for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
         is_composite[i * prime[j]] = true;
         if (i % prime[j] == 0) {
            phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
    divides i
            break;
         } else {
            phi[i * prime[j]] = phi[i] * phi[prime[j]];  //prime[j]
     does not divide i
         }
      }
   }
}
```

../Mathematics/LinearSieve.cpp

2.3 Modular Arithmetic

The modular inverse is defined by the following equation:

$$a \cdot a^{-1} \equiv 1 \mod m \tag{2.1}$$

The following code shows how to find the modular inverse of a number:

```
int ModPow(int a, int b, int m) {
  int res = 1;
  while (b > 0) {
    if (b & 1) res = (res * a) % m;
    a = (a * a) % m;
    b >>= 1;
  }
  return res;
}

// Language: java
public static int modPow(int a, int b, int m) {
  int res = 1;
  while (b > 0) {
    if ((b & 1) == 1) res = (res * a) % m;
    a = (a * a) % m;
    b >>= 1;
  }
  return res;
}

int ModInverse(int a, int m) {
  return ModPow(a, m - 2, m);
}
```

../Mathematics/ModularInverse.cpp

Some other useful relationships in modular arithmetic are:

- $(a+b) \mod m = (a \mod m + b \mod m) \mod m$
- $(a-b) \mod m = (a \mod m - b \mod m) \mod m$
- $(a*b) \mod m = (a \mod m * b \mod m) \mod m$
- $(a/b) \mod m = (a \mod m * b^{-1} \mod m) \mod m$
- $(a^b) \mod m = (a \mod m)^b \mod m$
- $(a^b) \mod m = (a \mod m)^{b \mod \phi(m)} \mod m$
- $(a^b) \mod m = (a \mod m)^{b \mod (m-1)} \mod m$
- $\frac{a}{k} \equiv \frac{a}{k} \mod m \iff a \equiv k \mod m$
- $\frac{a}{k} \equiv \frac{a}{k} \left(\mod \frac{n}{\gcd(n,k)} \right)$

2.4 Matrix Exponentiation

The following code shows how to find the nth power of a *mat*, noting that a data structure of type matrix is defined as follows:

```
typedef vector<vector<ll>> mat;
mat ans;
void mult(mat m1, mat m2)
{
   assert(m1[0].size() == m2.size());
   ans.clear();
   ll answer = 0;
   fore(i, 0, m1.size())
   {
      vector<ll> fila;
      fore(j, 0, m2[0].size())
      {
         answer = 0;
         fore(k, 0, m2.size())
            answer = (answer + m1[i][k] * m2[k][j]) % MOD;
         fila.pb(answer);
      }
      ans.pb(fila);
   }
}
void pot(mat base, ll exp)
{
   mat res(base.size(), vector<ll>(base.size(), 0));
   fore(i, 0, base.size())
   res[i][i] = 1;
   while(exp)
   {
      if(exp & 1)
      {
         mult(res, base);
```

```
            res = ans;
        }
        mult(base, base);
        base = ans;
        exp /= 2;
    }
    ans = res;
}
```

../Mathematics/MatrixPower.cpp

Chapter 3

Graphs

This chapter shows some of the basec algorithms and implementations required to solve problems that include graphs.

3.1 Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points. The algorithm is as follows:

The implementation can be done as follows:

```
vector<vector<int>> g(tam);
vector<bool> vis(tam);

void dfs(int u){
    vis[u]=true;
    ans++;
    for(int v: g[u]){
        if(!vis[v]){
            dfs(v);
        }
    }
}

signed main()
{
    int n,m;
    cin>>n>>m; // n nodes, m edges
    g.assign(tam,vector<int>());
    vis.assign(tam, false);
    for(int i=0; i<m;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    ll res = 0;
    for(int i=1; i<=n;i++){
        if(!vis[i]){
            ans=0;
            dfs(i);
            res = max(res,ans);
        }
    }
    g.clear();
    vis.clear();
    return 0;
}
```

../Graphs/DFS.cpp

An application of this algorithm in order to find the shorteast path between two nodes can be done as follows:

```
// The following code represents the implementation of a DFS
    algorithm
// to find the shortest path between two nodes in a graph.
// The graph is represented as an adjacency list.
// The algorithm is implemented using a stack.
```

Algorithm 1 Depth First Search (DFS)

procedure DFS(G)
  $visited \leftarrow \emptyset$
  $time \leftarrow 0$
  $parent \leftarrow \emptyset$
  $low \leftarrow \emptyset$
  $disc \leftarrow \emptyset$
  $AP \leftarrow \emptyset$
  $bridge \leftarrow \emptyset$
  for all $v \in V$ **do**
    $visited[v] \leftarrow false$
    $parent[v] \leftarrow -1$
    $low[v] \leftarrow \infty$
    $disc[v] \leftarrow \infty$
  end for
  for all $v \in V$ **do**
    if $visited[v] = false$ **then**
      DFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
    end if
  end for
end procedure
procedure DFSUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
  $visited[v] \leftarrow true$
  $disc[v] \leftarrow time$
  $low[v] \leftarrow time$
  $time \leftarrow time + 1$
  $children \leftarrow 0$
  for all $u \in Adj(v)$ **do**
    if $visited[u] = false$ **then**
      $parent[u] \leftarrow v$
      $children \leftarrow children + 1$
      DFSUtil(G, u, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
      $low[v] \leftarrow min(low[v], low[u])$
      if $parent[v] = -1$ and $children > 1$ **then**
        $AP[v] \leftarrow true$
      end if
      if $parent[v]! = -1$ and $low[u] \geq disc[v]$ **then**
        $AP[v] \leftarrow true$
      end if
      if $low[u] > disc[v]$ **then**
        $bridge[v][u] \leftarrow true$
      end if
    else
      $low[v] \leftarrow min(low[v], disc[u])$
    end if
  end for

```cpp
#include <bits/stdc++.h>

using namespace std;


vector<int> DFS(vector<vector<int>> &adj, int s, int t) {
  stack<vector<int>> path_stack;
  vector<int> path;
  vector<int> visited(adj.size(), 0);
  path_stack.push({s});
  while (!path_stack.empty()) {
    path = path_stack.top();
    path_stack.pop();
    int last = path[path.size() - 1];
    if (last == t) {
      return path;
    }
    if (visited[last] == 0) {
      visited[last] = 1;
      for (int i = 0; i < adj[last].size(); i++) {
        if (visited[adj[last][i]] == 0) {
          vector<int> new_path(path);
          new_path.push_back(adj[last][i]);
          path_stack.push(new_path);
        }
      }
    }
  }
  return {};
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> adj(n, vector<int>());
  for (int i = 0; i < m; i++) {
    int x, y;
    cin >> x >> y;
    adj[x - 1].push_back(y - 1);
    adj[y - 1].push_back(x - 1);
```

```
  }
  int x, y;
  cin >> x >> y;
  x--, y--;
  vector<int> path = DFS(adj, x, y);
  for (int i = 0; i < path.size(); i++) {
    cout << path[i] + 1 << " ";
  }
}
```

../Graphs/DFS–application.cpp

3.2 Breadth First Search (BFS)

The BFS algorithm is a non-recursive algorithm that visits all the nodes of a graph. It is used to find connected components, topological sorting, and to find bridges and articulation points, to better understand it, a propagating fire can be imagined. The algorithm is as follows:

The implementation can be done as follows:

```
#include <bits/stdc++.h>

using namespace std;
signed main()
{
    vector<vector<int>> adj;  // adjacency list representation
    int n; // number of nodes
    int s; // source vertex

    queue<int> q;
    vector<bool> used(n);
    vector<int> d(n), p(n);

    q.push(s);
    used[s] = true;
    p[s] = -1;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : adj[v]) {
            if (!used[u]) {
```

Algorithm 2 Breadth First Search (BFS)

```
procedure BFS(G)
    visited ← ∅
    time ← 0
    parent ← ∅
    low ← ∅
    disc ← ∅
    AP ← ∅
    bridge ← ∅
    for all v ∈ V do
        visited[v] ← false
        parent[v] ← −1
        low[v] ← ∞
        disc[v] ← ∞
    end for
    for all v ∈ V do
        if visited[v] = false then
            BFSUtil(G, v, visited, time, parent, low, disc, AP, bridge)
        end if
    end for
end procedure
procedure BFSUtil(G, v, visited, time, parent, low, disc, AP, bridge)
    visited[v] ← true
    disc[v] ← time
    low[v] ← time
    time ← time + 1
    children ← 0
    for all u ∈ Adj(v) do
        if visited[u] = false then
            parent[u] ← v
            children ← children + 1
            BFSUtil(G, u, visited, time, parent, low, disc, AP, bridge)
            low[v] ← min(low[v], low[u])
            if parent[v] = −1 and children > 1 then
                AP[v] ← true
            end if
            if parent[v]! = −1 and low[u] ≥ disc[v] then
                AP[v] ← true
            end if
            if low[u] > disc[v] then
                bridge[v][u] ← true
            end if
        else
            low[v] ← min(low[v], disc[u])
        end if
    end for
```

```
                used[u] = true;
                q.push(u);
                d[u] = d[v] + 1;
                p[u] = v;
            }
        }
    }
    return 0;
}
```

../Graphs/BFS.cpp

3.3 Finding Bridges and Articulation Points

The following algorithms are used to find bridges and articulation points in a graph. The implementation of these algorithms is done using DFS and BFS. This algoithms are based on Tarjan's algorithm.

Tarjan's algorithm is an algorithm that is used to find bridges and articulation points in a graph. The algorithm is as follows:

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

vector<bool> visited;
vector<int> tin, low;
int timer;
vector<vectotr<int>> comps; // componentes biconexos
stack<int> stk;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    stk.push(v);
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
```

Algorithm 3 Tarjan's Algorithm

procedure TARJAN(G)
    $visited \leftarrow \emptyset$
    $time \leftarrow 0$
    $parent \leftarrow \emptyset$
    $low \leftarrow \emptyset$
    $disc \leftarrow \emptyset$
    $AP \leftarrow \emptyset$
    $bridge \leftarrow \emptyset$
    for all $v \in V$ **do**
        $visited[v] \leftarrow false$
        $parent[v] \leftarrow -1$
        $low[v] \leftarrow \infty$
        $disc[v] \leftarrow \infty$
    end for
    for all $v \in V$ **do**
        if $visited[v] = false$ **then**
            TarjanUtil(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
        end if
    end for
end procedure
procedure TARJANUTIL(G, v, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
    $visited[v] \leftarrow true$
    $disc[v] \leftarrow time$
    $low[v] \leftarrow time$
    $time \leftarrow time + 1$
    $children \leftarrow 0$
    for all $u \in Adj(v)$ **do**
        if $visited[u] = false$ **then**
            $parent[u] \leftarrow v$
            $children \leftarrow children + 1$
            TarjanUtil(G, u, $visited$, $time$, $parent$, low, $disc$, AP, $bridge$)
            $low[v] \leftarrow min(low[v], low[u])$
            if $parent[v] = -1$ and $children > 1$ **then**
                $AP[v] \leftarrow true$
            end if
            if $parent[v] != -1$ and $low[u] \geq disc[v]$ **then**
                $AP[v] \leftarrow true$
            end if
            if $low[u] > disc[v]$ **then**
                $bridge[v][u] \leftarrow true$
            end if
        else
            $low[v] \leftarrow min(low[v], disc[u])$
        end if
    end for

```
            if (low[to] >= tin[v])
            {
               if (p != -1) IS_CUTPOINT(v);

               comps.push_back({v});
               while (comps.back().back() != to)
               {
                  comps.back().push_back(stk.top());
                  stk.pop();
               }

            }
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}

vector<int> id;
int curBCTNode;
vector<vector<int> > tree;
vector<bool> isAP;

void buildTree() {
    curBCTNode = 0;
    id.assign(n, -1);
    tree.clear();
    isAP.clear();
    fore(v, 0, n) {
        if (cutpoint[v]) {
            id[v] = tree.size();
            tree.pb({});
            isAP.pb(true);
        }
    }
    for (auto comp : comps) {
        int v = tree.size();
        tree.pb({});
        isAP.pb(false);
        for (int x : comp) {
            if (cutpoint[x]) {
                tree[v].pb(id[x]);
                tree[id[x]].pb(v);
            }
            else {
                id[x] = v;
            }
        }
    }
}
```

../Graphs/Tarjan_y_BlockCutTree.cpp

3.3.1 Bridges

A bridge is an edge that if it is removed, the graph will be divided into two or more components. The implementation is:

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
```

```
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

../Graphs/FindBridges.cpp

3.3.2 Articulation Points

An articulation point is a node that if it is removed, the graph will be divided into two or more components. The implementation is:

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

../Graphs/FindArticulationPoints.cpp

3.4 Flows

The flow is a concept that is used in many algorithms, it is used to find the maximum flow that could go through a system of nodes.

3.4.1 Dinic

The Dinic algorithm is a useful algorithm to find the maximum flow that could go through a system of nodes. The implementation of this algorithm is:

```
struct flowEdge
{
    int to, rev, f, cap;
```

```
};

vector<vector<flowEdge> > G;


void addEdge(int st, int en, int cap) {
    // Anade arista (st --> en) con su capacidad
    flowEdge A = {en, (int)G[en].size(), 0, cap};
    flowEdge B = {st, (int)G[st].size(), 0, 0};
    G[st].pb(A);
    G[en].pb(B);
}

int nodes, S, T; // asignar estos valores al armar el grafo G
                 // nodes = nodos en red de flujo. Hacer G.clear();
     G.resize(nodes);
vi work, lvl;
int Q[200010];

bool bfs() {
    int qt = 0;
    Q[qt++] = S;
    lvl.assign(nodes, -1);
    lvl[S] = 0;
    for (int qh = 0; qh < qt; qh++) {
        int v = Q[qh];
        for (flowEdge &e : G[v]) {
            int u = e.to;
            if (e.cap <= e.f || lvl[u] != -1) continue;
            lvl[u] = lvl[v] + 1;
            Q[qt++] = u;
        }
    }
    return lvl[T] != -1;
}

int dfs(int v, int f) {
    if (v == T || f == 0) return f;
    for (int &i = work[v]; i < G[v].size(); i++) {
        flowEdge &e = G[v][i];
        int u = e.to;
        if (e.cap <= e.f || lvl[u] != lvl[v] + 1) continue;
        int df = dfs(u, min(f, e.cap - e.f));
        if (df) {
            e.f += df;
            G[u][e.rev].f -= df;
            return df;
        }
    }
    return 0;
}

int maxFlow() {
    int flow = 0;
    while (bfs()) {
        work.assign(nodes, 0);
        while (true) {
            int df = dfs(S, INF);
            if (df == 0) break;
            flow += df;
        }
    }
    return flow;
}
```

../Graphs/Dinic.cpp

This implementation is done in order to do the Dinic algorithm for a graph with a large number of nodes.

This algorithm is based on the idea of the BFS algorithm, it is used to find the shortest path between two nodes, in this case, the shortest path between the source and the sink. The algorithm is as follows:

3.4.2 Ford Fulkerson

The Ford Fulkerson algorithm is a useful algorithm to find the maximum flow that could go through a system of nodes. The implementation of this algorithm is:

```
// This algorithm solves the max flow problem in a directed graph
   in O(max_flow * E)

// Here the graph is represented by an adjacency matrix, but it can
    be easily changed to an adjacency list
```

Algorithm 4 Dinic

```
procedure Dinic(G)
    visited ← ∅
    time ← 0
    parent ← ∅
    low ← ∅
    disc ← ∅
    AP ← ∅
    bridge ← ∅
    for all v ∈ V do
        visited[v] ← false
        parent[v] ← −1
        low[v] ← ∞
        disc[v] ← ∞
    end for
    for all v ∈ V do
        if visited[v] = false then
            DinicUtil(G, v, visited, time, parent, low, disc, AP, bridge)
        end if
    end for
end procedure
procedure DinicUtil(G, v, visited, time, parent, low, disc, AP, bridge)
    visited[v] ← true
    disc[v] ← time
    low[v] ← time
    time ← time + 1
    children ← 0
    for all u ∈ Adj(v) do
        if visited[u] = false then
            parent[u] ← v
            children ← children + 1
            DinicUtil(G, u, visited, time, parent, low, disc, AP, bridge)
            low[v] ← min(low[v], low[u])
            if parent[v] = −1 and children > 1 then
                AP[v] ← true
            end if
            if parent[v]! = −1 and low[u] ≥ disc[v] then
                AP[v] ← true
            end if
            if low[u] > disc[v] then
                bridge[v][u] ← true
            end if
        else
            low[v] ← min(low[v], disc[u])
        end if
    end for
```

```cpp
// The algorithm is based on the push-relabel algorithm, which is a
    variant of the relabel-to-front algorithm

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink
  't' in residual graph. Also fills parent[] to store the
  path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
     as visited
    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < V; v++) {
            if (visited[v] == false && rGraph[u][v] > 0) {
                // If we find a connection to the sink node, then
    there is no point in BFS anymore We just have to set its parent
     and can return true
                if (v == t) {
                    parent[v] = u;
                    return true;
                }
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
```

```
    }
    // We didn't reach sink in BFS starting from source, so return
    false
    return false;
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;
    // Create a residual graph and fill the residual graph with
    given capacities in the original graph as  residual capacities
    in residual graph
    int rGraph[V]
              [V]; // Residual graph where rGraph[i][j] indicates
    residual capacity of edge from i to j (if there is an edge. If
    rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially
    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent)) {
        // Find minimum residual capacity of the edges along the
    path filled by BFS. Or we can say find the maximum flow through
     the path found.
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
        // update residual capacities of the edges and reverse
    edges along the path
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
        // Add path flow to overall flow
        max_flow += path_flow;
    }
    // Return the overall flow
    return max_flow;
}

int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V]
        = { { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },
            { 0, 4, 0, 0, 14, 0 },  { 0, 0, 9, 0, 0, 20 },
            { 0, 0, 0, 7, 0, 4 },   { 0, 0, 0, 0, 0, 0 } };

    cout << "The maximum possible flow is "
         << fordFulkerson(graph, 0, 5);

    return 0;
}
```

../Graphs/FordFulkerson.cpp

In order to better understand the adjecency matrix in the code Figure 3.1 shows the graph that is used in the code.

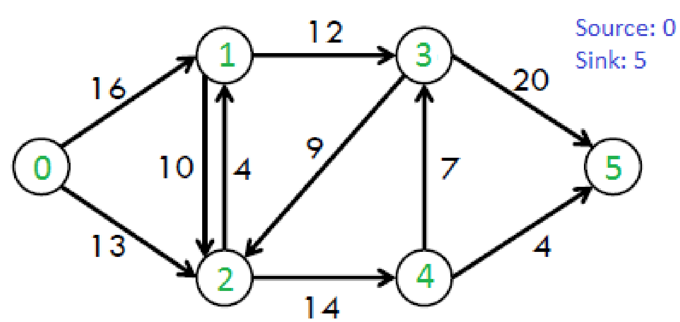

Figure 3.1: Ford Fulkerson

3.5 Dijkstra

The Dijkstra algorithm is a useful algorithm to find the shortest path between two nodes. The implementation of this algorithm is:

```cpp
const int INF = 1e9;
vector<vector<pair<int, int>>> adj;  //To store the node to which
    the edge flows to and the weight of the edge

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}

//In order to restore the path, we need to store the parent of each
     node in the shortest path tree

vector<int> restore_path(int s, int t, vector<int> const& p) {
    vector<int> path;

    for (int v = t; v != s; v = p[v])
        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

../Graphs/Dijkstra.cpp