Teambook Sindicato de Transporte 2880

Universidad Mayor de San Simón

March 1, 2023

Contents

Chapter 1

Mathematics

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

1.1 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

1.2 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

1.3 Modular Arithmetic

The modular inverse is defined by the following equation:

$$a \cdot a^{-1} \equiv 1 \mod m \tag{1.1}$$

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 of the other useful relationships in mosular arithmetic are:

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

Chapter 2

Graphs

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

2.1 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 ← ∅
    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
            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] ← 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
            DFSUtil(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
#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

2.2 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)
  $visited \leftarrow \emptyset$
  $time \leftarrow 0$
  $parent \leftarrow \emptyset$
  $low \leftarrow \emptyset$
  $disc \leftarrow \emptyset$
  $AP \leftarrow \emptyset$
  $bridge \leftarrow \emptyset$
  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**
      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, $bridge$)
  $visited[v] \leftarrow true$
  $disc[v] \leftarrow time$
  $low[v] \leftarrow time$
  $time \leftarrow time + 1$
  $children \leftarrow 0$
  for all $u \in Adj(v)$ **do**
    if $visited[u] = false$ **then**
      $parent[u] \leftarrow v$
      $children \leftarrow children + 1$
      BFSUtil(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

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

../Graphs/BFS.cpp