Breadth First Search: Shortest Reach

locked

by pranav9413

Editorial by AllisonP

Statistics

Difficulty: Medium
Time Complexity: $O(V+E)$ Required
Knowledge: BFS, Graph Theory
Publish Date: Dec 18 2014

The Challenge

Implement a Breadth-First Search (BFS) shortest-reach algorithm that prints the shortest distance from node s to each of the other respective nodes in a given graph. The distance for any node unreachable from s is considered to be -1.

Approach

1. Implement a graph having the following properties:
   - Some sort of iterable adjacency list of neighbors for each node.
   - An n-element boolean array that tracks whether or not a node has been $visited$. Initialize each element as $false$.
   - An n-element array of integers that tracks the $distance$ from some starting node s. Initialize each element as -1.
2. Perform a simple BFS from node s:
   1. Create a $queue$ of nodes or integer node IDs.
   2. Add starting node s to the queue, set $visited_s = true$, and set $distance_s = 0$. As we've already initialized the *distance* to all nodes as -1, we only need to find the distances for nodes that are connected to s.
   3. Loop through the $queue$ performing the following sequence of actions until it's empty:
      1. Dequeue the first node and call it u. Note that, the first time this loops, this will be node s.
      2. Iterate through each unvisited neighbor (v) of the current node (u) performing the following actions:
         - Add unvisited neighbor v to the $queue$.
         - Set v as visited.
         - Set v's distance from s as the distance from previous node u plus the distance of the edge connecting u and v.

      Because all edge weights are equal (6) and we use a queue to control the sequence in which we visit each node, we ensure that the value being stored as a node's distance from s is always minimal.

It's important to note that this implementation of BFS shortest-reach only works on graphs with *unweighted* or *equally weighted* edges.

Set by Pranav Sachdeva

Problem Setter's code :

C++

```
.#define pb push_back

using namespace std;

vector<'int'>adj[1009];

int main()

{

int t,n,m,x,y,s,i;
cin>>t;
while(t--)
{
    cin>>n>>m;
    for(i=0;i<n+3;i++)
    adj[i].clear();
    for(i=0;i<m;i++)
    {
        cin>>x>>y;
        adj[x].pb(y);
        adj[y].pb(x);
    }
    cin>>s;
    bool vis[n+1];
    memset(vis,false,sizeof(vis));
    int dist[n+1];
    memset(dist,0,sizeof(dist));
    list<int>q;
    q.push_back(s);
    vis[s]=true;
    dist[s]=0;
    while(!q.empty())
    {
        int u=q.front();
        q.pop_front();
        for(i=0;i<adj[u].size();i++)
        {
            int v= adj[u][i];
            if(!vis[v])
            {
                vis[v]=true;
                dist[v]=dist[u]+6;
                q.push_back(v);
            }
        }
    }
    for(i=1;i<=n;i++)
    {
        if(i!=s)
        {
            if(dist[i]==0)
            cout<<"-1 ";
            else
            cout<<dist[i]<<" ";
        }
    }
    cout<<endl;
}
return 0;
}
```

Tested by AllisonP

Problem Tester's code :

Java

```
import java.util.*;

class Graph {
    /** Edge weight **/
    private static int EDGE_DISTANCE;

    /** Track if each node is visited; Java default initialization is false
**/
    public boolean[] visited;
    /** Track distance to 'start' node **/
    public int[] distance;
    /** Graph where index is 0-indexed node ID and each element is the set
 of neighboring nodes. **/
    public ArrayList<HashSet<Integer>> graph;
    /** Starting node number for BFS, default Java initialization is to nod
e 0 **/
    public int start;

    public Graph(int n, int edgeWeight) {
        this.EDGE_DISTANCE = edgeWeight;

        this.visited = new boolean[n];
        this.distance = new int[n];
        Arrays.fill(distance, -1);

        this.graph = new ArrayList<HashSet<Integer>>();
        for(int i = 0; i < n; i++) {
            this.graph.add(new HashSet<Integer>());
        }
    }

    public void shortestReach(int s) {

        this.start = s;

        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(start);
        visited[start] = true;
        distance[start] = 0;

        // BFS from start
        while (queue.size() > 0) {
            int u = queue.remove();

            // for each unvisited neighbor of the current node
            for (int v : graph.get(u)) {

                // Add unvisited neighboring nodes to queue to check its ne
ighbors at next level of the search, set distance
                if (!visited[v]) {
                    queue.add(v);
                    visited[v] = true;
                    distance[v] = distance[u] + EDGE_DISTANCE;
                }
            }
        }

        for (int i : distance) {
            if (i != 0) {
                System.out.print(i + " ");
            }
        }

        System.out.println();

        // Just resets all distances to -1 in the event that this method is
ever called multiple times for some different 's'.
        Arrays.fill(distance, -1);
        Arrays.fill(visited, false);
```

```
    }
}

class Solution {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int queries = scanner.nextInt();

        for(int t = 0; t < queries; t++) {

            // Create a graph of size n where each edge weight is 6:
            Graph bfs = new Graph(scanner.nextInt(), 6);
            int m = scanner.nextInt();

            // read and set edges
            for(int i = 0; i < m; i++) {
                int u = scanner.nextInt() - 1;
                int v = scanner.nextInt() - 1;

                // add each edge to the graph
                bfs.graph.get(u).add(v);
                bfs.graph.get(v).add(u);
            }

            // Find shortest reach from node s
            bfs.shortestReach(scanner.nextInt() - 1);
        }

        scanner.close();
    }
}
```

Snakes and Ladders: The Quickest Way Up

locked

by idlecool

Editorial by Tanzir5

Statistics

Difficulty: Medium
Time Complexity: O(T) Required
Knowledge: Graph Theory, Breadth First Search
Publish Date: Dec 10 2014

Modeling the board as a graph

Let's model the board as a graph. Every square on the board is a node. The source node is square 1. The destination node is square 100. From every square, it is possible to reach any of the 6 squares in front of it in one move. So every square has a directed edge to the 6 squares in front of it.

Handling the snakes and ladders

For a snake starting at square i and finishing at j, we can consider that there is no node with index i in the graph. Because reaching node i is equivalent to reaching node j since the snake at node i will immediately take us down to node j. The same analogy goes for ladders too. To handle the snakes and ladders, let's keep an array **go_immediately_to[]** and initialize it like this.

```
if(no snake or ladder starts at node i)
    go_immediately_to[i] = i.
else
    j = ending node of the snake or the ladder starting at node i.
    go_immediately_to[i] = j.
```

Now let's run a standard Breadth First Search(BFS). Whenever we reach a node i, we will consider that we have reached the node $\mathbf{go\_immediately\_to[i]}$ and then continue with the BFS as usual. The distance of the destination is the solution to the problem.

Time Complexity

The size of the board is always $10 * 10$. You can consider time complexity of each BFS is constant as the number of snakes or ladders won't have much effect. There are T test cases; total complexity is $O(T)$ omitting the constant factor of $10 * 10$.

Editorialist's solution

C++

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

```
int n, m;
queue<int> q;
int go_immediately_to[110], dist[110];
bool vis[110];
bool isValid(int node)
{
    if(node < 1 || node > 100 || vis[node])
        return false;
    else
        return true;
}
int BFS(int source)
{
    memset(vis, 0, sizeof(vis));
    while(!q.empty())
        q.pop();

    vis[source] = 1;
    q.push(source);
    dist[source] = 0;
    while(!q.empty())
    {
        int current_node = q.front();
        q.pop();
        for(int i = 1; i<=6; i++)
        {
            int next_node = go_immediately_to[current_node+i];
            if(isValid(next_node))
            {
                q.push(next_node);
                vis[next_node] = 1;
                dist[next_node] = dist[current_node]+1;
            }
        }
    }
    if(!vis[100])
        return -1;
    else
        return dist[100];
}
int main()
{
    int i, j, cs, t, u, v;
    cin >> t;
    for(cs = 1; cs<=t; cs++)
    {
        cin >> n;

        for(i = 1; i<=100; i++)
            go_immediately_to[i] = i;

        for(i = 1; i<=n; i++)
        {
            cin >> u >> v;
            go_immediately_to[u] = v;
        }

        cin >> m;

        for(i = 1; i<=m; i++)
        {
            cin >> u >> v;
            go_immediately_to[u] = v;
        }

        cout << BFS(1) << endl;
    }

}
```

Matrix

locked

by HackerRank

Statistics

Difficulty: Hard
Time Complexity: $O(n)$ Required
Knowledge: Dynamic Programming on tree
Publish Date: Mar 17 2017

Editorial by jtnydv25

Note that when no path exists between any two machines, for any node, there exists at most one machine in its subtree.

For a node i, let $f(i, j)$ be the the least cost of removing nodes from the subtree rooted at node j, such that there exist exactly i machines in it, which are reachable from j. Notice that $0 \leq i \leq 1$.

Let C_j denote the set of children of node j, and w_{ij} be the edge weight of the edge between i and j.

Also, let $x_j = \sum_{c \in C_j} \min(f(1, c) + w_{cj}, f(0, c))$

Notice that x_j denotes the min cost to remove edges such that no machine in subtree rooted at j, but not in j can reach node j.

There exist two cases:

1. j has a machine. Here:
   - $f(0, i) = \infty$
   - $f(1, i) = x_j$
2. j doesn't have a machine. Here:
   - $f(0, i) = x$
   - $f(1, i) = x + \min_{c \in C_j}(x + f(1, c) - \min(f(1, c) + w_{cj}, f(0, c)))$

The implementation is pretty simple, and the complexity is $O(n)$. See the code for more clarity

Tested by jtnydv25

Problem Tester's code :

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define pii pair<int,int>
#define F first
#define S second
#define mp make_pair
#define eb emplace_back
const int N = 1e5+10;
ll f[2][N];
bool machine[N];
vector<pii> con[N];
```

```
const ll inf = 1e18;
void dfs(int s, int p){
        ll x = 0;
        for(auto it: con[s]){
                int v = it.F, w = it.S;
                if(v == p) continue;
                dfs(v, s);
                x += min(w + f[1][v], f[0][v]);
        }
        if(machine[s]){
                f[1][s] = x;
                f[0][s] = inf;
                return;
        }
        f[0][s] = x;
        for(auto it: con[s]){
                int v = it.F, w = it.S;
                if(v != p) f[1][s] = min(f[1][s], x + f[1][v] - min(f[1][v]
+ w, f[0][v]));
        }
}
int main(){
        for(int i = 0;i<N;i++) f[1][i] = f[0][i] = inf;
        int n, k, u, v, w;
        cin >>  n >> k;
        for(int i = 0; i < n - 1; i++){
                cin >> u >> v >> w;
                con[u].eb(mp(v,w));
                con[v].eb(mp(u,w));
        }
        int rt = -1;
        for(int i  = 0; i < k; i++){
                int node;
                cin >> node;
                machine[node] = 1;
                rt = node;
        }
        dfs(rt, -1);
        cout << min(f[0][rt], f[1][rt]) << endl;
}
```

Roads and Libraries

locked

by torquecode

Editorial by torquecode

Observation 1

We can't build new roads in HackerLand, we can only repair existing ones. When repairing roads, we need to be cognizant that there is no guarantee that each and every city is connected, meaning the graph may be disconnected and groups of cities will form. For example, a group of 3 cities might be connected to each other, but not connected to 2 other cities that are only connected to one another. We'll call each of these groups components, and in order to solve the whole problem we'll need to solve the components individually.

Observation 2

Each component needs at least one library. Without a library in one of the component's cities, there is no way for the cities in the component to access a library.

Conclusion

With these observations taken into account, there are two ways to assemble a component:

- A library must exist in at least one city, so $s - 1$ roads must be repaired (where s is the number of cities in the component).
- A library must exist in every city in a component, meaning that no roads need to be repaired. We choose this option when the cost of building a library is less than the cost of repairing a road.

The minimum cost for each component will either be $c_{lib} + (s - 1) \cdot c_{road}$ when we repair roads, or $s \cdot c_{lib}$ when we build a library in each city. Choosing the option that is smallest and summing it with all of the other smallest options for each component yields the value of the cheapest solution. If the cost for repairing a road and building a library are the same, the two approaches will be equal (meaning both options are equally valid).

Set by torquecode

Problem Setter's code :

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

vector<int> adj[100005];
bool visited[100005];
long nodes;
```

Statistics

Difficulty: Medium
Time Complexity: O(N)
Required Knowledge: dfs, components, greedy
Publish Date: Oct 12 2016

```
void DFS(int n)
{
    nodes++;
    visited[n] = true;
    for(vector<int>::iterator i = adj[n].begin(); i != adj[n].end() ; i++)
    {
        if(!visited[*i])
            DFS(*i);
    }
}


int main()
{
    int T;
    cin >> T;
    while(T--)
    {
        int N,M,a,b;
        long X,Y;
        cin >> N >> M >> X >> Y;

        for(int i = 0 ; i < M ; i++)
        {
            cin >> a >> b;
            adj[a].push_back(b);
            adj[b].push_back(a);
        }

        long cost=0;

        for(int i = 1 ; i <= N ; i++)
        {
            if(!visited[i])
            {
                nodes = 0;
                DFS(i);
                cost = cost + X;
                if(X > Y)
                    cost = cost + (Y*(nodes-1));
                else
                    cost = cost + (X*(nodes-1));
            }
        }

        cout << cost << endl;
        for(int i=0 ; i<=N ; i++)
        {
            adj[i].clear();
            visited[i] = false;
        }
    }

    return 0;
}
```

 Tested by allllekssssa

Problem Tester's code :

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<stdio.h>

using namespace std;

int n,m,c,t;
int ob[1000000];
vector <int> v[1000000];
```

```
long long ans,x,y;

void dfs(int x)
{
   c++;
   ob[x]++;
   for (int i=0;i<v[x].size();i++)
    if (!ob[v[x][i]])
     dfs(v[x][i]);

    return ;
}

void ocisti()
{
    for (int i=1;i<=n;i++)
        {
            v[i].clear();
            ob[i]=0;
        }
    ans=0;

    return;
}

int main()
{
     cin>>t;

      while(t--)
       {

        ocisti();
        scanf("%d%d%lld%lld",&n,&m,&x,&y);

         for (int i=1;i<=m;i++)
         {
             int x1,y1;
             scanf("%d%d",&x1,&y1);
             v[x1].push_back(y1);
             v[y1].push_back(x1);
         }

         for (int i=1;i<=n;i++)
            if (!ob[i])
         {
              c=0;
              dfs(i);
              ans+=min(x+(c-1)*y,c*x);
         }

         printf("%lld\n",ans);
     }
 return 0;
}
```

Journey to the Moon

locked

by amititkgp

This problem can be thought of as a graph problem. The very first step is to compute how many different countries are there. For this, we apply Depth First Search to calculate how many different connected components are present in the graph where the vertices are represented by the people and the people from the same country form one connected component. After we get how many connected components are present, say M, we just need to calculate the number of ways of selecting two persons from two different connected component. Let us assume that component i contains Mi people. So, for the number of ways selecting two persons from different components, we subtract the number of ways of selecting two persons from the same component from the total numbers of ways of selecting two persons i.e.

Ways = N choose 2 - (∑(Mi Choose 2) for i = 1 to M)

Problem Setter's code :

C++

```
    //Animesh Sinha

#include <iostream>
#include <list>
#include <vector>
#include <stdio.h>
#include <iterator>
#include <cmath>

#define MAX 100000

using namespace std;

list<int> *ad;
int *visited;
int vertices;

void DFS(int u)
{
    visited[u] = 1;

    vertices++;

    list<int>::iterator it;

    for(it=ad[u].begin();it!=ad[u].end();it++)
    {
        if(visited[*it] == 0)
        {
            visited[*it] = 1;
```

Statistics

Difficulty: Medium
Success Rate: 49.95%
Time Complexity: O(N+I) Required
Knowledge: Depth First Search, Combinatorics
Publish Date: Aug 13 2013

Of the 3656 contest participants, 1079(29.51%)submitted code for this challenge.

```
            DFS(*it);
        }
    }
}

int main()
{
    int i,m,u,v,numComponents=0,allv=0,temp=2,count=0;
    long long int n;
    int eachC[MAX];

    cin >> n >> m;

    if(n == 1)
    {
        cout <<"0\n";
        return 0;
    }

    ad = new list<int>[n];

    list<int>::iterator it;

    for(i=0;i<m;i++)
    {
        cin >> u >> v;

        ad[u].push_back(v);
        ad[v].push_back(u);
    }

    visited = new int[n];

    for(i=0;i<n;i++)
    {
        visited[i] = 0;
    }

    for(i=0;i<n;i++)
    {
        if(visited[i] == 0)
        {
            vertices = 0;
            DFS(i);
            eachC[numComponents] = vertices;
            numComponents++;
        }
    }

    long long int totalWays = n*(n-1) / 2;
    long long int sameWays = 0;

    for(i=0;i<numComponents;i++)
    {
        sameWays = sameWays + (eachC[i]*(eachC[i]-1) / 2);
    }
    cout << (totalWays - sameWays) << endl;
    return 0;
}
```

Problem Tester's code :

Python 2

```
#!/usr/bin/python

N, I = map(int, raw_input().strip().split())
assert 1 <= N <= 10**5
assert 1 <= I <= 10**6
lis_of_sets = []
```

```
for i in range(I):
    a,b = map(int, raw_input().strip().split())
    assert 0 <= a < N and 0 <= b < N
    indices = []
    new_set = set()
    set_len = len(lis_of_sets)
    s = 0
    while s < set_len:
        if a in lis_of_sets[s] or b in lis_of_sets[s]:
            indices.append(s)
            new_set = new_set.union(lis_of_sets[s])
            del lis_of_sets[s]
            set_len -= 1
        else:
            s += 1

    new_set = new_set.union([a, b])

    lis_of_sets.append(new_set)

answer = N*(N-1)/2
for i in lis_of_sets:
    answer -= len(i)*(len(i)-1)/2

print answer
```

Even Tree

locked

by HackerRank

Editorial by Kamlesh Meher

Statistics

Difficulty: Medium
Time Complexity: $O(N)$ Required
Knowledge: DFS
Publish Date: Sep 05 2014

Problem Overview

Given a tree print the maximum number of edges that can be removed to form a forest(a disjoint union of trees) in which each connected component of the forest should contain an even number of vertices. Also, note that the input graph will be such that it can always be decomposed into components containing an even number of nodes **which indirectly means that N is even**.

Approach

What we need is a subtree that has even number of vertices and as we get it we can remove the edge that connects it to the remaining of the tree so that we are left with a subtree of even vertices and the remaining tree with vertices N -(No. of vertices in the subtree removed). Now we are left with the same problem again with remaining tree having even number of vertices because remember that N is even so even-even=even, so we have to repeat this algorithm until the remaining tree cannot be decomposed further in the above manner. To maximize the number of subtree remove(which is equal to the number of edges removed) we have to remove subtree which cannot be decomposed in the above manner.

To do this we can traverse the tree using dfs and the dfs function should return the number of vertices in the subtree of which the current node is the root. If a node gets in return an even number of vertices from one of it child then the edge from that node to its child should be removed and if the number is odd add it to the number of vertices that the subtree will have if the current node is the root of it.

To make it clear below is the code in c++ for the dfs function:

```
// ans is total number of edges removed and al is adjacency list of the tre
e.
int dfs(int node)
{
    visit[node]=true;
    int num_vertex=0;
    for(int i=0;i<al[node].size();i++)
    {
        if(!visit[al[node][i]])
        {
            int num_nodes=dfs(al[node][i]);
```

```
            if(num_nodes%2==0)
                ans++;
            else
                num_vertex+=num_nodes;
        }
    }
    return num_vertex+1;
}
```

Note: You can start the dfs from any node assuming that it's the root because it will have no effect on the answer.

Tested by Kamlesh Meher

Problem Tester's code :

C++

```
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

vector<int> al[105];
bool visit[105];
int n,m;
int ans;

int dfs(int node)
{
    visit[node]=true;
    int num_vertex=0;
    for(int i=0;i<al[node].size();i++)
    {
        if(!visit[al[node][i]])
        {
            int num_nodes=dfs(al[node][i]);
            if(num_nodes%2==0)
                ans++;
            else
                num_vertex+=num_nodes;
        }
    }
    return num_vertex+1;
}

int main() {
    int x,y;
    cin>>n>>m;
    for(int i=0;i<m;i++)
    {
        cin>>x>>y;
        al[x].push_back(y);
        al[y].push_back(x);
    }
    dfs(1);
    cout<<ans;
    return 0;
}
```