Last update: June 9, 2024 Translated From: e-maxx.ru

Search for connected components in a graph

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

An algorithm for solving the problem

- To solve the problem, we can use Depth First Search or Breadth First Search.
- In fact, we will be doing a series of rounds of DFS: The first round will start from first node and all the nodes in the first connected component will be traversed (found). Then we find the first unvisited node of the remaining nodes, and run Depth First Search on it, thus finding a second connected component. And so on, until all the nodes are visited.
- The total asymptotic running time of this algorithm is $O(n + m)$: In fact, this algorithm will not run on the same vertex twice, which means that each edge will be seen exactly two times (at one end and at the other end).

Implementation

```
int n;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;

void dfs(int v) {
    used[v] = true ;
    comp.push_back(v);
    for (int u : adj[v]) {
        if (!used[u])
            dfs(u);
    }
}

void find_comps() {
    fill(used.begin(), used.end(), 0);
    for (int v = 0; v < n; ++v) {
        if (!used[v]) {
            comp.clear();
            dfs(v);
            cout << "Component:" ;
```

```
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

- The most important function that is used is `find_comps()` which finds and displays connected components of the graph.
- The graph is stored in adjacency list representation, i.e `adj[v]` contains a list of vertices that have edges from the vertex `v`.
- Vector `comp` contains a list of nodes in the current connected component.

Iterative implementation of the code

Deeply recursive functions are in general bad. Every single recursive call will require a little bit of memory in the stack, and per default programs only have a limited amount of stack space. So when you do a recursive DFS over a connected graph with millions of nodes, you might run into stack overflows.

It is always possible to translate a recursive program into an iterative program, by manually maintaining a stack data structure. Since this data structure is allocated on the heap, no stack overflow will occur.

```
int n;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;

void dfs(int v) {
    stack<int> st;
    st.push(v);

    while (!st.empty()) {
        int curr = st.top();
        st.pop();
        if (!used[curr]) {
            used[curr] = true;
            comp.push_back(curr);
            for (int i = adj[curr].size() - 1; i >= 0; i--) {
                st.push(adj[curr][i]);
            }
        }
    }
}

void find_comps() {
    fill(used.begin(), used.end(), 0);
    for (int v = 0; v < n ; ++v) {
```

```
        if (!used[v]) {
            comp.clear();
            dfs(v);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

Practice Problems

- SPOJ: CT23E
- CODECHEF: GERALD07
- CSES : Building Roads