Finding strongly connected components Building condensation graph

Table of Contents

Definitions

You are given a directed graph G with vertices V and edges E. It is possible that there are loops and multiple edges. Let's denote n as number of vertices and m as number of edges in G.

Strongly connected component is subset of vertices C such that any two vertices of this subset are reachable from each other, i.e. for any $u, v \in C$:

$$u \mapsto v, v \mapsto u$$

where $\mapsto$ means reachability, i.e. existence of the path from first vertex to the second.

It is obvious, that strongly connected components do not intersect each other, i.e. this is a partition of all graph vertices. Thus we can give a definition of condensation graph G^{SCC} as a graph containing every strongly connected component as one vertex. Each vertex of the condensation graph corresponds to the strongly connected component of graph G. There is an oriented edge between two vertices C_i and C_j of the condensation graph if and only if there are two vertices $u \in C_i, v \in C_j$ such that there is an edge in initial graph, i.e. $(u, v) \in E$.

The most important property of the condensation graph is that it is **acyclic**. Indeed, suppose that there is an edge between C and C', let's prove that there is no edge from C' to C. Suppose that $C' \mapsto C$. Then there are two vertices $u' \in C$ and $v' \in C'$ such that $v' \mapsto u'$. But since u and u' are in the same strongly connected component then there is a path between them; the same for v and v'. As a result, if we join these paths we have that $v \mapsto u$ and at the same time $u \mapsto v$. Therefore u and v should be at the same strongly connected component, so this is contradiction. This completes the proof.

The algorithm described in the next section extracts all strongly connected components in a given graph. It is quite easy to build a condensation graph then.

Description of the algorithm

Described algorithm was independently suggested by Kosaraju and Sharir at 1979. This is an easy-to-implement algorithm based on two series of depth first search, and working for $O(n+m)$ time.

On the first step of the algorithm we are doing sequence of depth first searches, visiting the entire graph. We start at each vertex of the graph and run a depth first search from every non-visited vertex. For each vertex we are keeping track of **exit time** $tout[v]$. These exit times have a key role in an algorithm and this role is expressed in next theorem.

First, let's make notations: let's define exit time $tout[C]$ from the strongly connected component C as maximum of values $tout[v]$ by all $v \in C$. Besides, during the proof of the theorem we will mention entry times $tin[v]$ in each vertex and in the same way consider $tin[C]$ for each strongly connected component C as minimum of values $tin[v]$ by all $v \in C$.

Theorem. Let C and C' are two different strongly connected components and there is an edge (C, C') in a condensation graph between these two vertices. Then $tout[C] > tout[C']$.

There are two main different cases at the proof depending on which component will be visited by depth first search first, i.e. depending on difference between $tin[C]$ and $tin[C']$: ** The component C was reached first. It means that depth first search comes at some vertex v of component C at some moment, but all other vertices of components C and C' were not visited yet. By condition there is an edge (C, C') in a condensation graph, so not only the entire component C is reachable from v but the whole component C' is reachable as well. It means that depth first search that is running from vertex v will visit all vertices of components C and C', so they will be descendants for v in a depth first search tree, i.e. for each vertex $u \in C \cup C', u \neq v$ we have that $tout[v] > tout[u]$, as we claimed. ** Assume that component C' was visited first. Similarly, depth first search comes at some vertex v of component C' at some moment, but all other vertices of components C and C' were not visited yet. But by condition there is an edge (C, C') in the condensation graph, so, because of acyclic property of condensation graph, there is no back path from C' to C, i.e. depth first search from vertex v will not reach vertices of C. It means that vertices of C will be visited by depth first search later, so $tout[C] > tout[C']$. This completes the proof.

Proved theorem is **the base of algorithm** for finding strongly connected components. It follows that any edge (C, C') in condensation graph comes from a component with a larger value of $tout$ to component with a smaller value.

If we sort all vertices $v \in V$ by decreasing of their exit moment $tout[v]$ then the first vertex u is going to be a vertex from "root" strongly connected component, i.e. a vertex that no edges in a condensation graph come into. Now we want to run such search from this vertex u so that it will visit all vertices in this strongly connected component, but not others; doing so, we can gradually select all strongly connected components: let's remove all vertices corresponding to the first selected component, and then let's find a vertex with the largest value of $tout$, and run this search from it, and so on.

Let's consider transposed graph G^T, i.e. graph received from G by changing direction of each edge on the opposite. Obviously this graph will have the same strongly connected components, as an initial graph. More over, condensation graph G^{SCC}. It means that there will be no edges from our "root" component to other components.

Thus, for visiting the whole "root" strongly connected component, containing vertex v, is enough to run search from vertex v in graph G^T. This search will visit all vertices of this strongly connected component and only them. As was mentioned before, we can remove these vertices from the graph then, and find the next vertex with a maximal value of $tout[v]$ and run search in transposed graph from it, and so on.

Thus, we built next **algorithm** for selecting strongly connected components:

1st step. Run sequence of depth first search of graph G which will return vertices with increasing exit time $tout$, i.e. some list $order$.

2nd step. Build transposed graph G^T. Run a series of depth (breadth) first searches in the order determined by list $order$ (to be exact in reverse order, i.e. in decreasing order of exit times). Every set of vertices, reached after the next search, will be the next strongly connected component.

Algorithm asymptotic is $O(n+m)$, because it is just two depth (breadth) first searches.

Finally, it is appropriate to mention topological sort here. First of all, step 1 of the algorithm represents reversed topological sort of graph G (actually this is exactly what vertices' sort by exit time means). Secondly, the algorithm's scheme generates strongly connected components by decreasing order of their exit times, thus it generates components - vertices of condensation graph - in topological sort order.

Implementation

```
vector < vector<int> > g, gr;
vector<bool> used;
vector<int> order, component;

void dfs1 (int v) {
	used[v] = true;
	for (size_t i=0; i<g[v].size(); ++i)
		if (!used[ g[v][i] ])
			dfs1 (g[v][i]);
	order.push_back (v);
```

```
}

void dfs2 (int v) {
	used[v] = true;
	component.push_back (v);
	for (size_t i=0; i<gr[v].size(); ++i)
		if (!used[ gr[v][i] ])
			dfs2 (gr[v][i]);
}

int main() {
	int n;
	... reading n ...
	for (;;) {
		int a, b;
		... reading next edge (a,b) ...
		g[a].push_back (b);
		gr[b].push_back (a);
	}

	used.assign (n, false);
	for (int i=0; i<n; ++i)
		if (!used[i])
			dfs1 (i);
	used.assign (n, false);
	for (int i=0; i<n; ++i) {
		int v = order[n-1-i];
		if (!used[v]) {
			dfs2 (v);
			... printing next component ..
			component.clear();
		}
```

```
        }
    }
```

Here, g is graph, gr is transposed graph. Function $dfs1$ implements depth first search on graph G, function $dfs2$ - on transposed graph G^T. Function $dfs1$ fills the list $order$ with vertices in increasing order of their exit times (actually, it is making a topological sort). Function $dfs2$ stores all reached vertices in list $component$, that is going to store next strongly connected component after each run.

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. Introduction to Algorithms [2005].
- M. Sharir. A strong-connectivity algorithm and its applications in data-flow analysis [1979].

Practice Problems

- SPOJ - Submerging Islands
- SPOJ - Good Travels
- SPOJ - Lego
- Codechef - Chef and Round Run
- Dev Skills - A Song of Fire and Ice
- UVA - 11838 - Come and Go
- UVA 247 - Calling Circles
- UVA 13057 - Prove Them All
- UVA 12645 - Water Supply

- UVA 11770 - Lighting Away
- UVA 12926 - Trouble in Terrorist Town
- UVA 11324 - The Largest Clique
- UVA 11709 - Trust groups
- UVA 12745 - Wishmaster
- SPOJ - True Friends
- SPOJ - Capital City
- Codeforces - Scheme
- SPOJ - Ada and Panels