Definition, problem statement

Given a directed graph Gwhose vertex set $V$and the set of edges - E. Loops and multiple edges are allowed. We denote $n$the number of vertices, through m- the number of edges.

Strongly connected component (strongly connected component) is called (maximum respect to inclusion) subset of vertices Csuch that any two vertices of this subset reachable from each other, ie for $\forall u, v \in C$:

$u \mapsto v, v \mapsto u$

where the symbol $\mapsto$hereafter we denote reachability, ie existence of a path from the first vertex to the second.

It is clear that for strongly connected components of the graph do not intersect, ie in fact it is a partition of all vertices of the graph. Hence the logical definition of**condensation** G^{SCC} as the graph obtained from the compression of the graph of each strongly connected components in a single vertex. Each vertex of the graph corresponds to the condensation of strongly connected component of the graph G, and a directed edge between two vertices $C_i$and $C_j$graph condensation is carried out if a pair of vertices $u \in C_i, v \in C_j$between which there was an edge in the original graph, ie $(u, v) \in E$.

The most important property of the graph condensation is that it **is acyclic** . Indeed, suppose that $C \mapsto C'$, prove that $C' \not\mapsto C$. From the definition of condensation we get that there are two peaks $u \in C$and $v \in C'$that $u \mapsto v$. Will prove the opposite, ie assume that $C' \mapsto C$, if there are two vertices $u' \in C$and $v' \in C'$that $v' \mapsto u'$. But since $u$and $u'$are in the same strongly connected component, then there is a path between them and similarly for $v$and v'. In summary, combining a way we obtain that $v \mapsto u$, at the same time $u \mapsto v$. Consequently, $u$and $v$must belong to the strongly connected components, ie a contradiction, as required.

Algorithm described later in this column identifies all strongly connected components. Count on them to build condensation is not difficult.

Algorithm

Algorithm described here has been proposed independently Kosarayu (Kosaraju) and sarira (Sharir) in 1979 It is very easy to implement algorithm based on two series of searches in depth , and therefore the running time $O(n + m)$.

In the first step of the algorithm, a series of detours in depth, visiting the entire graph. To do this, we go through all the vertices of the graph and each vertex not yet visited call traversal depth. In addition, for each vertex v remember **time to** $\mathrm{tout}[v]$. These retention times play a key role in the algorithm, and this role is expressed in the following theorem.

First, we introduce the notation: time- $\mathrm{tout}[C]$ components of C the strong connectivity is defined as the maximum of the values $\mathrm{tout}[v]$ for all $v \in C$. Moreover, in the proof will be referred to the time of entry, and each vertex $\mathrm{tin}[v]$, and similarly define the time of entry $\mathrm{tin}[C]$ for each strongly connected components of a minimum of values $\mathrm{tin}[v]$ for all $v \in C$.

Theorem . Let C and C' - two different strongly connected components, and let the graph condensation between an edge (C, C'). Then $\mathrm{tout}[C] > \mathrm{tout}[C']$.

The proof there are two fundamentally different cases depending on which of the first component will go crawl in depth, ie depending on the ratio between $\mathrm{tin}[C]$ and $\mathrm{tin}[C']$:

- The first component was achieved C. This means that at some time in depth tour comes to a vertex v components C, and all other vertices component C and C' have not yet visited. However, since by hypothesis, the graph has an edge condensations (C, C'), then from the top v will be achieved not only the whole part C, but the whole part C'. This means that when you start from the top v in depth tour will pass through all the vertices of the component C and C' , and, therefore, they will be in relation to the descendants v in the tree traversal in depth, ie for each vertex $u \in C \cup C', u \neq v$ is satisfied $\mathrm{tout}[v] > \mathrm{tout}[u]$, QED
- The first component was achieved C'. Again, at some time in depth tour comes to a vertex $v \in C'$, and all the other vertices of the component C and C' not visited. Since by hypothesis graph condensations existed edge (C, C'), is due to condensation acyclic graph, there is no way back $C' \not\to C$, ie bypassing the depth from the top v reaches peaks C. This means that they will be visited in the traversal depth later, whence $\mathrm{tout}[C] > \mathrm{tout}[C']$, QED

The above theorem is the **basis for the algorithm** search of strongly connected components. From this it follows that any edge (C, C') in the graph is condensations of components with greater magnitude tout in the component with a lower value.

If we sort all the vertices $v \in V$ in order of exit time $\mathrm{tout}[v]$, the first would be a vertex u belonging to the "root" strongly connected component, ie which excludes none of

the edges in the graph condensations. Now we would like to start a tour of this peak u, which would be visited only this component of the strong connectivity and did not go to any other, learn to do it, we can gradually select all strongly connected components: removing the first vertex of the graph of the selected component, we again find among the remaining vertex with the highest value tout, rerun of it this tour, etc.

To learn how to do this tour, consider **the transpose of the graph** G^T, ie the graph obtained from Geach change in direction of the opposite edge. It is easy to understand that in this graph are the same strongly connected components as in the original graph. Moreover, the condensation graph $(G^T)^{\text{SCC}}$for it is equal transposed graph condensation of the original graph G^{SCC}. This means that from now we are considering the "root" component will not be leaving the edges in other components.

So, to get around the whole "root" strongly connected components containing a vertex v, enough to run from the top of bypass $v$in the graph G^T. This tour will visit all the vertices of the strongly connected components, and only them. As already mentioned, then we can mentally remove these vertices of the graph, find the next vertex with the maximum value $\text{tout}[v]$and run circumvention transposed graph of it, etc.

Thus we have constructed the following **algorithm** selection strongly connected component:

Step 1. Launch a series of detours in depth graph G, which returns the top in order of increasing time output tout, ie some list order.

Step 2. Build transposed graph G^T. Launch a series of detours in depth / width of the graph in the order determined list order(namely, in reverse order, ie, in order to reduce the time of output). Each set of vertices reached as a result of the next crawling run and there will be another component of the strong connectivity.

Asymptotic behavior of the algorithm is obviously $O(n+m)$because it represents only two round trips in depth / width.

Finally, it is appropriate to note the relationship with the notion of **topological sorting** . First, step 1 of the algorithm is not nothing but a topological sort of the graph G(in fact, this means sorting peaks retention time). Secondly, the chart is that the strongly connected components, and it generates in order to reduce their retention times, so it generates components - the vertices of condensation in topological sort order.

Implementation

```
vector < vector<int> > g, gr;
vector<char> 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;
	... чтение n ...
	for (;;) {
		int a, b;
		... чтение очередного ребра (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);
			... вывод очередной component ...
			component.clear();
		}
	}
```

```
}
```

Here in $g$the graph itself is stored, and gr- transposed graph. Function $dfs1$crawls in depth on the graph G, the function $dfs2$- for transposed G^T. Function $dfs1$populates a list of $order$vertices in increasing order of exit time (in fact, makes a topological sort). Function $dfs2$saves all reached the top of the list $component$, which after each run will be another component of the strong connectivity.

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Design and analysis of** [2005]
- M. Sharir. **A strong-Connectivity algorithm and ITS applications in Data-Flow analysis** [1979]