Last update: October 22, 2024 Translated From: e-maxx.ru

Strongly connected components and the condensation graph

Definitions

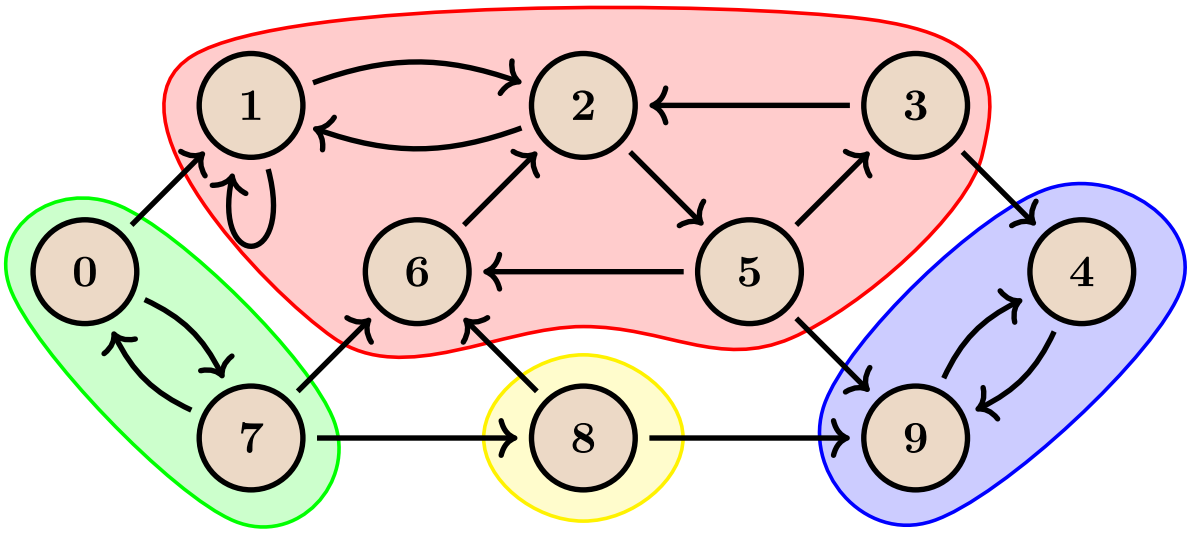
Let $G = (V, E)$ be a directed graph with vertices V and edges $E \subseteq V \times V$. We denote with $n = |V|$ the number of vertices and with $m = |E|$ the number of edges in G. It is easy to extend all definitions in this article to multigraphs, but we will not focus on that.

A subset of vertices $C \subseteq V$ is called a **strongly connected component** if the following conditions hold:

- for all $u, v \in C$, if $u \neq v$ there exists a path from u to v and a path from v to u, and
- C is maximal, in the sense that no vertex can be added without violating the above condition.

We denote with $\mathrm{SCC}(G)$ the set of strongly connected components of G. These strongly connected components do not intersect with each other, and cover all vertices in the graph. Thus, the set $\mathrm{SCC}(G)$ is a partition of V.

Consider this graph G_{example}, in which the strongly connected components are highlighted:

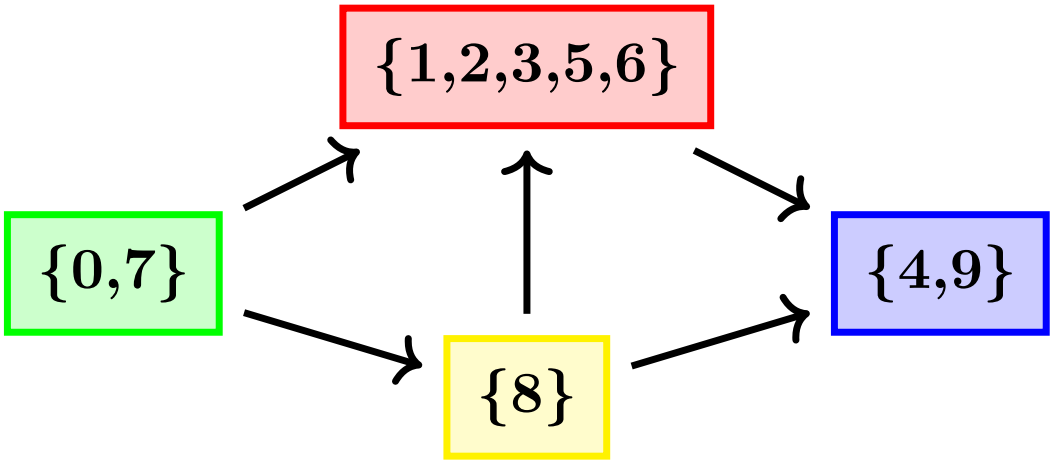


Here we have $\mathrm{SCC}(G_{\text{example}}) = \{\{0, 7\}, \{1, 2, 3, 5, 6\}, \{4, 9\}, \{8\}\}$. We can confirm that within each strongly connected component, all vertices are reachable from each other.

We define the **condensation graph** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ as follows:

- the vertices of G^{SCC} are the strongly connected components of G; i.e., $V^{\mathrm{SCC}} = \mathrm{SCC}(G)$, and
- for all vertices C_i, C_j of the condensation graph, there is an edge from C_i to C_j if and only if $C_i \neq C_j$ and there exist $a \in C_i$ and $b \in C_j$ such that there is an edge from a to b in G.

The condensation graph of G_{example} looks as follows:



The most important property of the condensation graph is that it is **acyclic**. Indeed, there are no 'self-loops' in the condensation graph by definition, and if there were a cycle going through two or more vertices (strongly connected components) in the condensation graph, then due to reachability, the union of these strongly connected components would have to be one strongly connected component itself: contradiction.

The algorithm described in the next section finds all strongly connected components in a given graph. After that, the condensation graph can be constructed.

Description of the algorithm

The described algorithm was independently suggested by Kosaraju and Sharir around 1980. It is based on two series of depth first search, with a runtime of $O(n + m)$.

In the first step of the algorithm, we perform a sequence of depth first searches (`dfs`), visiting the entire graph. That is, as long as there are still unvisited vertices, we take one of them, and initiate a depth first search from that vertex. For each vertex, we keep track of the *exit time* $t_{\text{out}}[v]$. This is the 'timestamp' at which the execution of `dfs` on vertex v finishes, i.e., the moment at which all vertices reachable from v have been visited and the algorithm is back at v. The timestamp counter should *not* be reset between consecutive calls to `dfs` . The exit times play a key role in the algorithm, which will become clear when we discuss the following theorem.

First, we define the exit time $t_{\text{out}}[C]$ of a strongly connected component C as the maximum of the values $t_{\text{out}}[v]$ for all $v \in C$. Furthermore, in the proof of the theorem, we will mention the *entry time* $t_{\text{in}}[v]$ for each vertex $v \in G$. The number $t_{\text{in}}[v]$ represents the 'timestamp' at which the recursive function `dfs` is called on vertex v in the first step of the algorithm. For a strongly connected component C, we define $t_{\text{in}}[C]$ to be the minimum of the values $t_{\text{in}}[v]$ for all $v \in C$.

Theorem. Let C and C' be two different strongly connected components, and let there be an edge from C to C' in the condensation graph. Then, $t_{\text{out}}[C] > t_{\text{out}}[C']$.

Proof. There are two different cases, depending on which component will first be reached by depth first search:

- Case 1: the component C was reached first (i.e., $t_{\text{in}}[C] < t_{\text{in}}[C']$). In this case, depth first search visits some vertex $v \in C$ at some moment at which all other vertices of the components C and C' are not visited yet. Since there is an edge from C to C' in the condensation graph, not only are all other vertices in C reachable from v in G, but all vertices in C' are reachable as well. This means that this `dfs` execution, which is running from vertex v, will also visit all other vertices of the components C and C' in the future, so these vertices will be descendants of v in the depth first search tree. This implies that for each vertex $u \in (C \cup C') \setminus \{v\}$, we have that $t_{\text{out}}[v] > t_{\text{out}}[u]$. Therefore, $t_{\text{out}}[C] > t_{\text{out}}[C']$, which completes this case of the proof.
- Case 2: the component C' was reached first (i.e., $t_{\text{in}}[C] > t_{\text{in}}[C']$). In this case, depth first search visits some vertex $v \in C'$ at some moment at which all other vertices of the components C and C' are not visited yet. Since there is an edge from C to C' in the condensation graph, C is not reachable from C', by the acyclicity property. Hence, the `dfs` execution that is running from vertex v will not reach any vertices of C, but it will visit all vertices of C'. The vertices of C will be visited by some `dfs` execution later during this step of the algorithm, so indeed we have $t_{\text{out}}[C] > t_{\text{out}}[C']$. This completes the proof.

The proved theorem is very important for finding strongly connected components. It means that any edge in the condensation graph goes from a component with a larger value of t_{out} to a component with a smaller value.

If we sort all vertices $v \in V$ in decreasing order of their exit time $t_{\text{out}}[v]$, then the first vertex u will belong to the "root" strongly connected component, which has no incoming edges in the condensation graph. Now we want to run some type of search from this vertex u so that it will visit all vertices in its strongly connected component, but not other vertices. By repeatedly doing so, we can gradually find all strongly connected components: we remove all vertices belonging to the first found component, then we find the next remaining vertex with the largest value of t_{out}, and run this search from it, and so on. In the end, we will have found all strongly connected components. In order to find a search method that behaves like we want, we consider the following theorem:

Theorem. Let G^T denote the *transpose graph* of G, obtained by reversing the edge directions in G. Then, $\text{SCC}(G) = \text{SCC}(G^T)$. Furthermore, the condensation graph of G^T is the transpose of the condensation graph of G.

The proof is omitted (but straightforward). As a consequence of this theorem, there will be no edges from the "root" component to the other components in the condensation graph of G^T. Thus, in order to visit the whole "root" strongly connected component, containing vertex v, we can just run a depth first search from vertex v in the transpose graph G^T! This will visit precisely all vertices of this strongly connected component. As was mentioned before, we can then remove these vertices from the graph. Then, we find the next vertex with a maximal value of $t_{\text{out}}[v]$, and run the search in the transpose graph starting from that vertex to find the next strongly connected component. Repeating this, we find all strongly connected components.

Thus, in summary, we discussed the following algorithm to find strongly connected components:

- Step 1. Run a sequence of depth first searches on G, which will yield some list (e.g. `order`) of vertices, sorted on increasing exit time t_{out}.
- Step 2. Build the transpose graph G^T, and run a series of depth first searches on the vertices in reverse order (i.e., in decreasing order of exit times). Each depth first search will yield one strongly connected component.
- Step 3 (optional). Build the condensation graph.

The runtime complexity of the algorithm is $O(n+m)$, because depth first search is performed twice. Building the condensation graph is also $O(n+m)$.

Finally, it is appropriate to mention topological sort here. In step 1, we find the vertices in the order of increasing exit time. If G is acyclic, this corresponds to a (reversed) topological sort of G. In step 2, the algorithm finds strongly connected components in decreasing order of their exit times. Thus, it finds components - vertices of the condensation graph - in an order corresponding to a topological sort of the condensation graph.

Implementation

```
vector<bool> visited; // keeps track of which vertices are already visited

// runs depth first search starting at vertex v.
// each visited vertex is appended to the output vector when dfs leaves it.
void dfs(int v, vector<vector<int>> const& adj, vector<int> &output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected components in G
// output: adj_cond -- adjacency list of G^SCC (by root vertices)
void strongly_connected_components(vector<vector<int>> const& adj,
                                   vector<vector<int>> &components,
                                   vector<vector<int>> &adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be a sorted list of G's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0); // gives the root vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(component), end(component));
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
```

```
            adj_cond[roots[v]].push_back(roots[u]);
}
```

The function `dfs` implements depth first search. It takes as input an adjacency list and a starting vertex. It also takes a reference to the vector `output`: each visited vertex will be appended to `output` when `dfs` leaves that vertex.

Note that we use the function `dfs` both in the first and second step of the algorithm. In the first step, we pass in the adjacency list of G, and during consecutive calls to `dfs`, we keep passing in the same 'output vector' `order`, so that eventually we obtain a list of vertices in increasing order of exit times. In the second step, we pass in the adjacency list of G^T, and in each call, we pass in an empty 'output vector' `component`, which will give us one strongly connected component at a time.

When building the adjacency list of the condensation graph, we select the *root* of each component as the first vertex in its list of vertices (this is an arbitrary choice). This root vertex represents its entire SCC. For each vertex `v`, the value `roots[v]` indicates the root vertex of the SCC which `v` belongs to.

Our condensation graph is now given by the vertices `components` (one strongly connected component corresponds to one vertex in the condensation graph), and the adjacency list is given by `adj_cond`, using only the root vertices of the strongly connected components. Notice that we generate one edge from C to C' in G^{SCC} for each edge from some $a \in C$ to some $b \in C'$ in G (if $C \neq C'$). This implies that in our implementation, we can have multiple edges between two components in the condensation graph.

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 - Good Travels
- SPOJ - Lego
- Codechef - Chef and Round Run
- 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
- CSES - Flight Routes Check
- CSES - Planets and Kingdoms
- CSES - Coin Collector
- Codeforces - Checkposts