Graph algorithms: Strongly Connected Components and Single-Source Shortest Path

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

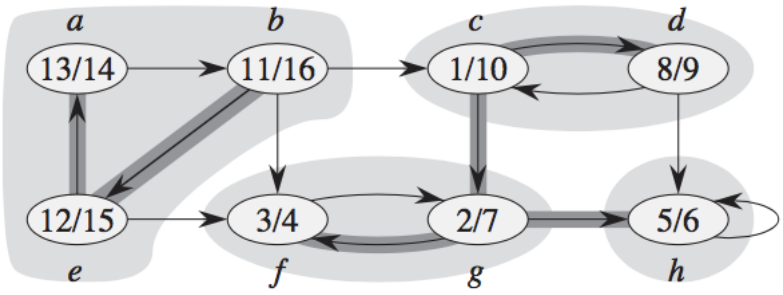
These notes sketch the content of the 10th lecture. The topics of this lecture are treated in any introductory book on Algorithms. You may refer to Chapters 22.5 and 24 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 for a more detailed explanation.

1 Strongly connected components

Decomposing a directed graph into its strongly connected components is a classic application of depth-first search. The problem of finding connected components is at the heart of many graph application. Generally speaking, the connected components of the graph correspond to different classes of objects.

Given digraph or directed graph $G = (V, E)$, a *strongly connected component* (SCC) of G is a maximal set of vertices C subset of V, such that for all u, v in C, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, both u and v are reachable from each other.

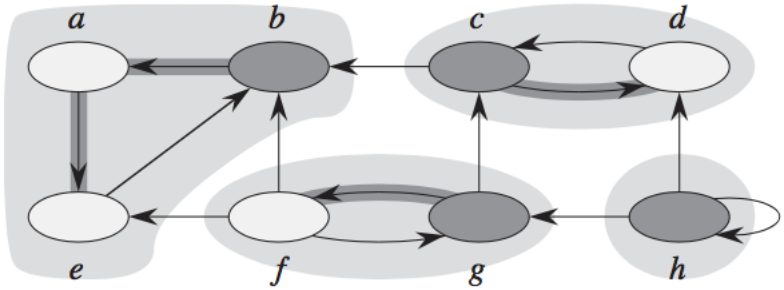
In other words, two vertices of directed graph are in the same component if and only if they are reachable from each other.



The above graph has 4 strongly connected components: $\{a, b, e\}$, $\{c, d\}$, $\{f, g\}$ and $\{h\}$.

The algorithm uses the transpose of graph G. The transposed graph is $G^T = (V, E^T)$, where $E^T = \{(v, u) | (u, v) \in E)\}$.

The figure below shows the transpose of the graph of the previous example.



It is easy to see that G and G^T have the same strongly connected components. If u and v are reacheble from each other in G, the same holds in G^T.

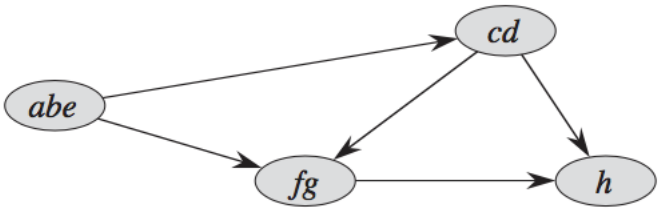
There is an easy (and linear time) algorithm to compute SSCs of G. The algorithm uses two DFSs of G and G^T.

The algorithm is as follows.

1. Call DFS(G) to compute finishing times $f[u]$ for all u.
2. Compute G^T
3. Call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

The idea of the algorithm is inspired by the so-called *Component graph* $G^{SCC} = (V^{SCC}, E^{SCC})$, where V^{SCC} has one vertex for each SCC in G and E^{SCC} has an edge if there is an edge between the corresponding SCC's in G.

Figure below shows the component graph of the previous example.



The key property of is that the component graph is a DAG, which the following lemma implies.

▶ **Lemma 1.** *G^{SCC} is a DAG. More formally, let C and C' be distinct SCCs in G, let u, v in C, u', v' in C', and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.*

Proof. Suppose such a path $v' \rightsquigarrow v$ exists in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ◀

Why the algorithm works?

By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topological sort order.

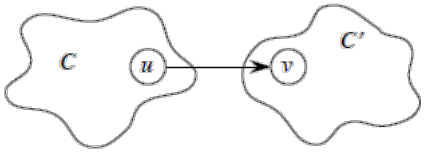
To prove that it really works, first we deal with two notational issues:

We will be discussing $d[u]$ and $f[u]$. These always refer to the first DFS in the above algorithm.

We extend notation for d and f to sets of vertices $U \subseteq V$:

1. $d(U) = \min\{d(u) | u \in U\}$, which is the earliest discovery time of any vertex in U;
2. $f(U) = \max\{f(u) | u \in U\}$, which is the latest finishing time of any vertex in U.

▶ **Lemma 2.** *Let C and C' be distinct SCCs in $G = (V, E)$. Suppose there is an edge (u, v) in E such that u in C and v in C'. Then $f(C) > f(C')$.*



Proof. There are two cases, depending on which SCC had the first discovered vertex during the first DFS.

Case 1. If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

By the parenthesis theorem, we have $f[x] = f(C) > f(C')$.

Case 2. If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in C'. This implies that all vertices in C' become descendants of y. Again, $f[y] = f(C')$.

At time $d[y]$, all vertices in C are white.

By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C. So, no vertex in C is reachable from y. Therefore, at time $f[y]$, all vertices in C are still white. Therefore, for all w in C, $f[w] > f[y]$, which implies that $f(C) > f(C')$. ◀

▶ **Corollary 3.** *Let C and C' be distinct SCCs in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^T.*

Can you see now why it works?

When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum. The second DFS starts from some x in C, and it visits all vertices in C. Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^T. Therefore, DFS will visit only vertices in C.

Which means that the depth-first tree rooted at x contains exactly the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCCs other than C. DFS visits all vertices in C', but the only edges out of C' go to C, which we've already visited.

Therefore, the only tree edges will be to vertices in C'.

▶ **Problem 4** (Cormen et al exercise 22.5-3). *Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?*

The answer is NO. Consider a graph with two SCCs: A and B with an egde from A to B. Assume DFS starts from a vertex of A then it visits all vertecies of B and, then, the remaining vertecies of A. This way, the second visit will start from one vertex of A and will find just one SCC instead of two.

2 Single-source shortest path

I invite you to study Chapter 24 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 by yourself.

3 Problems

▶ **Problem 5** (Learning Languages). *The "BerCorp" company has got n employees. These employees can use m approved official languages for the formal correspondence. The languages are numbered with integers from 1 to m. For each employee we have the list of languages, which he knows. This list could be empty, i.e., an employee may know no official languages. But the employees are willing to learn any number of official languages, as long as the company pays their lessons. A study course in one language for one employee costs 1 berdollar.*

Find the minimum sum of money the company needs to spend so as any employee could correspond to any other one (their correspondence can be indirect, i.e., other employees can help out translating).

We build an undirected graph G as follows. We have a vertex for each enployee and an edge connecting employees i and j iff i and j have a language in common. The minimum number of languages to teach equals the number of connected components of the graph minus 1. Indeed, employees within the same connected component can correspond to each other and it suffices to join with two connected components with an edge to allows communications between employees of the two components.

Actually, there is a special case. If all the employees speek no language, then the result is n.

▶ **Problem 6** (Checkposts). *Your city has n junctions. There are m one-way roads between the junctions. As a mayor of the city, you have to ensure the security of all the junctions.*

To ensure the security, you have to build some police checkposts. Checkposts can only be built in a junction. A checkpost at junction i can protect junction j if either $i = j$ or the police patrol car can go to j from i and then come back to i.

Building checkposts costs some money. As some areas of the city are more expensive than others, building checkpost at some junctions might cost more money than other junctions.

You have to determine the minimum possible money needed to ensure the security of all the junctions. Also you have to find the number of ways to ensure the security in minimum price and in addition in minimum number of checkposts. Two ways are different if any of the junctions contains a checkpost in one of them and do not contain in the other.

Find the SCCs of the underlying graph. From each component choose a node with the lowest cost.