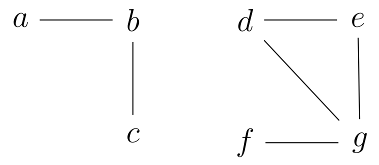
Depth-First Search Forest: Finding Strongly Connected Components

CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2020

Undirected Graphs

We have previously studied depth-first search. Recall that depth-first search creates a search tree for a graph. In undirected graphs, the search tree is a spanning tree if the undirected graph is connected. But, if the graph is not connected, then the search tree will only contain some of the nodes in the graph. We can use this understanding to find all the different connected subgraphs in our undirected graph. If we run depth-first search on all vertices in our graph we will find all of the connected subgraphs. But this would give us duplicate subgraphs. We can improve this by marking vertices found during our search so we don't revisit vertices. Let's make this concrete with pseudo code for the algorithm and an example. Consider the graph G below:



It's easy for us to see there are two components; but we need to use an algorithm for a computer to compute this: this is what depth-first search forest does. Consider the pseudo-code below:

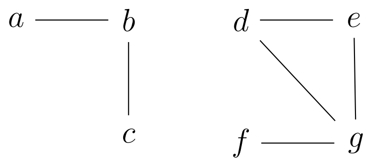
```
Procedure DepthFirstSearchForest(G: Graph)
    forest := empty
    for each vertex v in G
        clear the visit mark for v

    for each vertex v in G
        if v is not marked
            tree := DepthFirstSearch(v)
            add tree to forest
```

Note that it is assumed that all vertices found in a `DepthFirstSearch` are marked. Thus we don't run `DepthFirstSearch` on any vertex already seen (because it will be marked). Recall the pseudo-code for `DepthFirstSearch`:

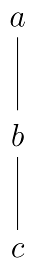
```
Procedure DepthFirstSearch(v: Vertex)
   mark v
   for each vertex w adjacent from v
      if w is not marked
         DepthFirstSearch(w)
```

DFS Forest Example:

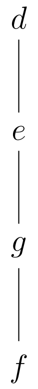
Consider the following graph, G:



Let us assume the vertices are stored in a list in alphabetical order, thus $G_V = a, b, c, d, e, f, g$. When `DepthFirstSearchForest` is run it will start by calling `DepthFirstSearch` on vertex a. That will result in the following search tree:



Now, we have visited and marked a, b, and c and created the corresponding search tree. $G_V =$ ~~a,b,c~~, d, e, f, g, so in the second `for`-loop `DepthFirstSearch` is next called on d. That will result in the following search tree:



Now, we have visited and marked d, e, f, and g, so $G_V =$ ~~a,b,c,d,e,f,g~~. Since there are no more remaining vertices our forest is comprised of two search trees: 1) $\{a, b, c\}$; 2) $\{d, e, f, g\}$. This matches our visual inspection that the graph has two connected components. Therefore, depth-first search forest can be used to find all connected components of an undirected graph.

Directed Graphs

In directed graphs we are interested in *strongly* connected components (SCCs), the cycles of our graphs. There are more steps required to find all strongly connected components of a direct graph than simply running depth-first search forest. We'll go over the full algorithm – this is what you will implement in Project 5.

Algorithm to find Strongly Connected Components

This document will not explain why the algorithm works. If you are interested in those details, see Section 3.4.2 of Algorithms by Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. The algorithm, run on graph G, has three steps and uses depth-first search forest twice:

1. Create the reverse graph of G, denoted G^R
2. Run depth-first search forest on G^R to get the postorder of the vertices of G
3. Run depth-first search forest on G using the reverse posteorder (start at the last vertex in the postorder)

Step 1: Create the Reverse Graph G^R

The first step is to reverse the original graph. This is quite simple. In a directed graph, we just need to reverse all the edges. This can be done in linear time by creating a new graph that has the reverse edge for every edge in the original graph. Consider the graphs, G and G^R, shown below:

$$G = \begin{tikzcd}
a \arrow[drr] & b \arrow[r] & c \arrow[d] \arrow[dr] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l] & g \arrow[u]
\end{tikzcd}
\qquad
G^R = \begin{tikzcd}
a \arrow[dr] & b & c \arrow[l] \arrow[r] & f \arrow[d] \\
 & d \arrow[r] & e \arrow[ull] \arrow[u] & g \arrow[ul]
\end{tikzcd}$$

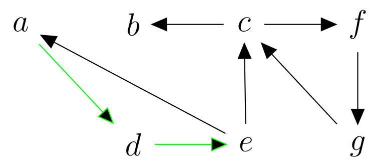
Step 2: Run Depth-First Search Forest on G^R

The second step is to run depth-first search forest on the reverse graph, G^R. This will give us the postorder we need for step three (see the tree traversal notes on how to calculate the postorder during a depth-first search). Let's go through this step for G^R. We'll start with our list of vertices in alphabetical order: a, b, c, d, e, f, g. We'll step through depth-first search forest keeping track of two sets: one for marked vertices, the second, an ordered set, with the postorder of vertices.

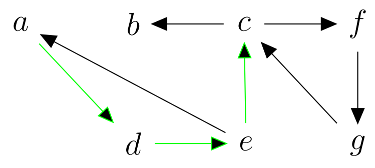
Marked $= a$
postorder $=$

$$\begin{tikzcd}
a \arrow[dr] & b & c \arrow[l] \arrow[r] & f \arrow[d] \\
 & d \arrow[r] & e \arrow[ull] \arrow[u] & g \arrow[ul]
\end{tikzcd}$$

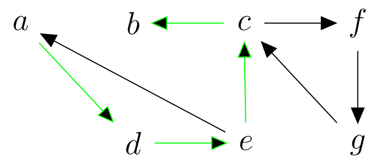
Marked = a, d
postorder =



Marked = a, d, e
postorder =

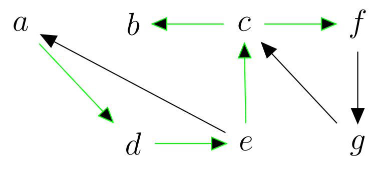


Marked = a, d, e, c
postorder =

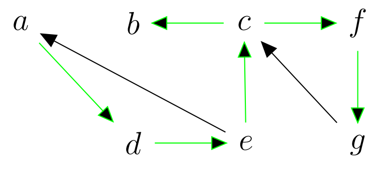


At this point we choose to visit b and since it has no outgoing edges it returns, which gives us the first vertex in our postorder. The postorder number for b is one, so we'll put it first in our ordered set representing the postorder.

Marked = a, d, e, c, b
postorder = b



Marked = a, d, e, c, b, f
postorder = b



Marked = a, d, e, c, b, f, g
postorder = b, g

$$\begin{tikzcd}
a \arrow[dr, green] & b & c \arrow[l, green] \arrow[r, green] & f \arrow[d, green] \\
 & d \arrow[r, green] & e \arrow[ull] \arrow[u, green] & g \arrow[ul]
\end{tikzcd}$$

After visiting g, all vertices are visited, so our search just backtracks (note that for this graph depth-first search would have been adequate, but that is not always the case so we must run depth-first search forest). This backtracking will give us the rest of the postorder: b, g, f, c, e, d, a (note how this ordering relates to the order we visited vertices). This postorder is now our input to step 3 of the algorithm.

Step 3: Run depth-first search forest on G

The third and final step is to run depth-first search forest on the original graph, G, using the *reverse* of the postorder determined by step 2. Let's be very clear about this. The postorder from step 2 is: b, g, f, c, e, d, a. That means we run depth-first search forest on G with the vertices ordered as follows: a, d, e, c, f, g, b. The different search trees found by depth-first search forest will be the desired strongly connected components. Let's run through this. The first call to depth-first search will be with vertex a (because it is the first vertex in our reverse postorder):

vertices = ~~a~~, d, e, c, f, g, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l] & g \arrow[u]
\end{tikzcd}$$

vertices = ~~a~~, d,~~e~~, c, f, g, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u]
\end{tikzcd}$$

vertices = ~~a~~,~~d~~,~~e~~, c, f, g, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u]
\end{tikzcd}$$

At vertex d we stop the depth-first search started from a, and return the search tree that contains a, e, d. This is our first tree in the forest and thus our first strongly connected component. Now depth-first search forest will call depth first search on c, since d and e are already marked.

vertices = ~~a~~,~~d~~,~~e~~,~~e~~, f, g, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr, green] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u]
\end{tikzcd}$$

Note that since e is already marked, depth-first search goes to g when called from c.

vertices = ~~a~~,~~d~~,~~e~~,~~e~~, f,~~g~~, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr, green] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u, green]
\end{tikzcd}$$

vertices = ~~a~~,~~d~~,~~e~~,~~e~~,~~f~~,~~g~~, b

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr, green] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u, green]
\end{tikzcd}$$

At vertex f we stop the depth-first search started from c, and return the search tree that contains c, g, f. This is our second tree in the forest and thus our second strongly connected component. Now depth-first search forest will call depth first search on b, since f and g are already marked.

vertices = ~~a~~,~~d~~,~~e~~,~~e~~,~~f~~,~~g~~,~~b~~

$$\begin{tikzcd}
a \arrow[drr, green] & b \arrow[r] & c \arrow[d] \arrow[dr, green] & f \arrow[l] \\
 & d \arrow[ul] & e \arrow[l, green] & g \arrow[u, green]
\end{tikzcd}$$

At vertex b we stop immediately, because c is already marked, and return the search tree that contains only b. This is our third tree in the forest and thus our third strongly connected component. Now depth-first search forest is complete. We now have all strongly connected components of the graph G. Note: the order of the strongly connected components is *very* important for Project 5. See details in the next section.

Project 5

Make sure you understand both the depth-first search forest algorithm and the algorithm for finding strongly connected components in a directed graph. You will have to create code to implement and integrate these algorithms into your Datalog interpreter to optimize Datalog rule evaluation. You will need to create vertices for all of your Datalog rules. Then you will run the three-step algorithm above to find the strongly connected components (the rules that are cyclic). The order in which you find the strongly connected components is important. It tells you which set of rules to evaluate first. You will evaluate all rules in the first strongly connected component using the fixed-point algorithm for Project 4. Then you will proceed to the second strongly connected component and so on until you have evaluated all rules. Note how this is an optimization from iterating over all rules using our fixed-point algorithm without considering dependencies.

Conclusion

There are many applications for strongly connected component. We've discussed one, which is representing items as a graph and then discovering cycles through finding strongly connected components. Discovering strongly connected components is also useful for applications where the entire graph is unknown (think about the internet).