Strongly Connected Components

Kevin Geng and Lawrence Wang

27 January 2017

1 Introduction

Let's start by considering connectivity in undirected graphs. In such a graph, two nodes u and v are connected if there is some path from u to v. Correspondingly, a connected component in such a graph is a subgraph in which every node is reachable from every other node. Finding such components is straightforward: we can simply perform a DFS. If we want to be able to connect components dynamically, we can use the Union-Find data structure.

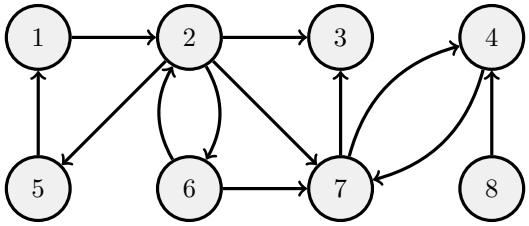

Figure 1: A directed graph. *Credit: Crash Course Coding Companion.*

If our graph is directed, however, our previous definition of connectivity poses a problem. If v is reachable from u, this doesn't imply that u is reachable from v. So instead, we'll say that u and v are *strongly connected* if they are both reachable from each other.[1] Extending this to the idea of connected components, a *strongly connected component* is a subgraph where any two nodes in this subgraph are strongly connected.

2 Kernel graph

We can travel between any two nodes in the same strongly connected component. So what if we replaced each strongly connected component with a single node? This would give us what we call the *kernel graph*, which describes the edges between strongly connected components. Note that the kernel graph is a directed acyclic graph (why?). This means that there are orderings of the nodes, where if node A comes before node B in our ordering, then it is impossible to get to A from B by traversing along edges.

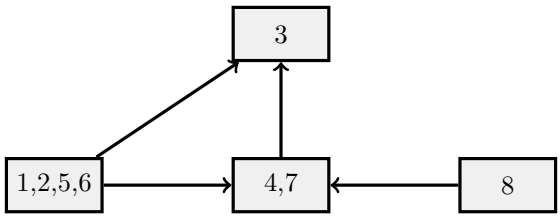

Figure 2: Kernal graph of the above directed graph.

[1] Two nodes in a directed graph are *weakly connected* if at least one is reachable from the other — as if you just replaced every directed edge with an undirected edge.

3 Kosaraju-Sharir algorithm

First, let's reverse all of the edges of the original graph to get the reverse graph. Perform a postorder traversal on this graph using DFS, and store the reverse of the postorder in a list. This is similar to a topological sort of the reverse graph.

Then, we simply perform an ordinary flood-fill DFS on the graph, but looping through the nodes in the order given above. All we need to do is assign components in the order in which we visit them, and this gives us the strongly connected components! To understand why, consider what we've done in terms of the kernel graph. Each component that we flood-fill is unreachable from any other unvisited component.

Algorithm 1 Kosaraju-Sharir

```
function VISIT(vertex u)
    if u has not been visited then
        mark u as visited
        for all in-neighbors v of u do                ▷ such that v → u
            VISIT(v)
        add v to front of L
function ASSIGN(vertex u, num)                        ▷ flood-fill an entire component
    id(u) ← num
    for all out-neighbors v of u do                   ▷ such that u → v
        ASSIGN(v, num)
function KOSARAJUSHARIR(G(V, E))
    initialize new empty list L
    for all vertices v ∈ V do
        VISIT(v)
    num ← 0
    for all vertices v ∈ L do
        if id(v) is undefined then                    ▷ v has not been visited
            num ← num + 1
            ASSIGN(v, num)
```

4 Tarjan's Algorithm

Kosaraju-Sharir requires two traversals through the entire graph. We can find strongly connected components with only one traversal using Tarjan's algorithm, though it is more difficult to implement. Essentially, the algorithm performs a DFS traversal while adding visited nodes to a stack. A node remains on the stack *iff* it can reach a higher node on the stack; otherwise, it is the root node in the search tree of its component.

5 Problems

Mowing the Field (USACO January 2016, Platinum)

In an effort to better manage the grazing patterns of his cows, Farmer John has installed one-way cow paths all over his farm. The farm consists of N fields ($1 \leq N \leq 100000$), conveniently numbered $1..N$, with each one-way cow path connecting a pair of fields. For example, if a path connects from field X to field Y, then cows are allowed to travel from X to Y but not from Y to X.

Bessie wonders how much grass she will be able to eat if she breaks the rules and follows up to one path in the wrong direction. Please compute the maximum number of distinct fields she can visit along a route starting and ending at field 1, where she can follow up to one path along the route in the wrong direction. Bessie can only travel backwards at most once in her journey. In particular, she cannot even take the same path backwards twice.