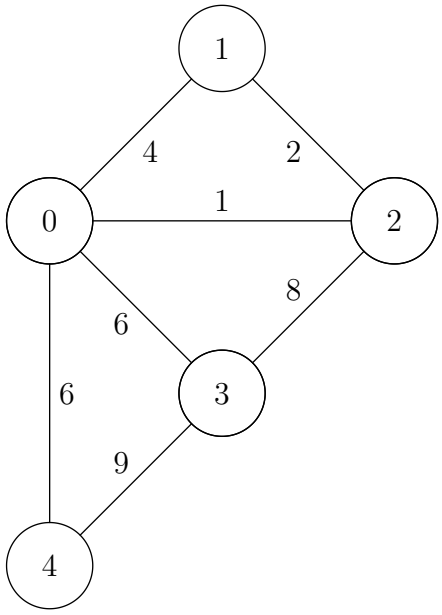
Data Structures – Brett Bernstein

Lecture 19: Tarjan's SCC Algorithm

Review Exercises

1. Suppose we have a directed weighted graph with no negative cycles and we have just run Floyd-Warshall on it. Let d denote the resulting $n \times n$ matrix. Show how to do the following using d.
   (a) Output the size of the component containing vertex v, assuming the graph is undirected.
   (b) Output whether the graph has a cycle.
   (c) Output the strongly connected component containing vertex v.
   (d) Output the number of strongly connected components.
2. Consider the following graph.

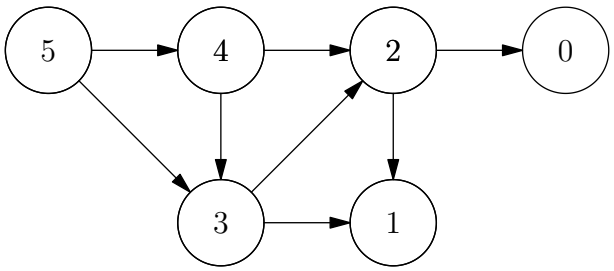


   (a) Show how Prim's algorithm finds the minimum spanning tree starting from vertex 2. In case of a tie, prefer lower-numbered vertices.
   (b) Compute the distance of getting from vertex 4 to all other vertices using Dijkstra's algorithm. In case of a tie, prefer lower-numbered vertices.
3. Consider the following memoized code for computing the kth Fibonacci number:

```
static long[] cache; //Initialized with -1s
static long fib(int k) {
  if (k <= 1) return k;
  if (cache[k] > -1) return cache[k];
  long ret = fib(k-1)+fib(k-2);
  return cache[k] = ret;
}
```

Consider a graph of the values $0, 1, \ldots$. There is a directed edge from vertex v to vertex w if fib(v) calls fib(w). Draw the graph of all nodes reachable from 5.

Review Solutions

1. (a) One plus the number of non-infinite entries in row v that are off-diagonal (i.e., don't consider dist[v][v]).
   (b) Check if any of the diagonal entries aren't ∞ and output true if yes.
   (c) Output all w such that dist[v][w] and dist[w][v] both aren't ∞.
   (d) Make a boolean array of size $|V|$ to mark vertices. Start with vertex 0 and mark all elements in its strongly connected component using the previous part. Then find the next unmarked vertex and repeat.
2. (a) Prim's algorithm will add the vertices in the following order: 2,0,1,3,4.
   (b) Dijkstra's algorithm will visit the vertices in the following order: 4,0,2,1,3 with distances 0,6,7,9,9.
3. 

($\star\star$) Tarjan's Algorithm for Strongly Connected Components

Before we discuss the algorithm we will introduce the concept of a DFS number. As DFS traverses a graph it visits the vertices in some order. Starting with the number 0, we number each vertex according to the order in which DFS visits it. This is called the DFS number of the vertex, which we can write as dfsNum[v].

Tarjan's algorithm works by running DFS on the graph and outputs each strongly connected component when that component's lowest DFS numbered vertex is finished by DFS. To help in outputting the SCCs, Tarjan's algorithm maintains a stack S of vertices. We will also maintain an array, dfsLow, that approximately stores, for each vertex v, the lowest DFS number of any unpopped vertex reachable from v (including v). The algorithm works as follows:

1. Run DFS.
   (a) When DFS visits an unvisited vertex v, update dfsNum[v] accordingly, and push v on the stack S.
   (b) As usual, recursively call DFS on all unpopped neighbors. By taking the min over unpopped neighbors, also compute dfsLow[v].
   (c) When DFS finishes v do the following:
      i. If v can reach an unpopped vertex with lower DFS number than v then do nothing.
      ii. Otherwise, pop and output all vertices from the stack S, stopping when we pop vertex v. Change the state of each popped vertex to "popped". These are the vertices of the strongly connected component containing v.

The key to this algorithm is understanding how DFS traverses a graph. When a vertex v is finished by DFS, all reachable vertices are finished (descendents or have no ancestral relationship) or are visited and unfinished (ancestors). The finished vertices, if unpopped, must all be able to reach v or vertices that have not been finished yet. In other words, all finished vertices in the stack must be in the same SCC as v. Also note that the finished vertices in the stack must have been pushed after v since the stack is ordered by dfsNum. These ideas can be used to prove that our popping code exactly removes the SCC containing v.

Source code follows:

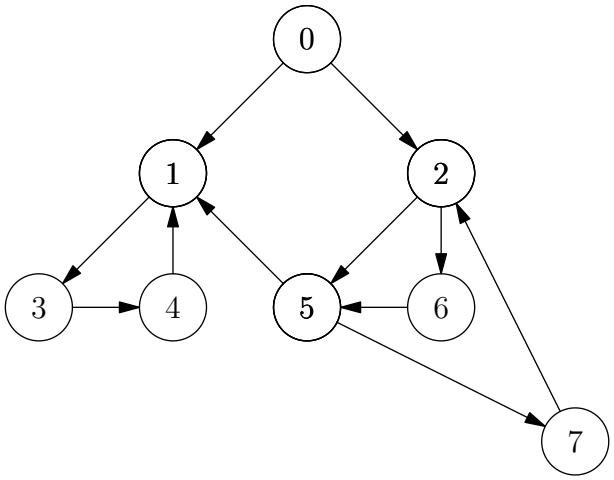
TarjanSCC.java

```
import java.util.ArrayList;

public class TarjanSCC
{
	static final int UNVISITED = 0;
	static final int VISITED = 1;
	static final int FINISHED = 2;
	static final int POPPED = 3;

	static int num;
	public static void tarjanSCC(ArrayList<ArrayList<Integer>> adj)
	{
		int N = adj.size();
```

```
        num = 0;
        int[] state = new int[N];
        int[] dfsNum = new int[N];
        int[] dfsLow = new int[N];
        ArrayList<Integer> stack = new ArrayList<>();
        for (int v = 0; v < N; ++v)
            tarjanSCC(adj, v, state, dfsNum, dfsLow, stack);
    }
    //Returns dfsLow[v]
    public static int tarjanSCC(ArrayList<ArrayList<Integer>> adj,
            int v, int[] state, int[] dfsNum,
            int[] dfsLow, ArrayList<Integer> stack)
    {
        if (state[v] != UNVISITED) return dfsLow[v];
        state[v] = VISITED;
        dfsNum[v] = num++;
        dfsLow[v] = dfsNum[v];
        stack.add(v);
        for (int w : adj.get(v))
        {
            if (state[w] == POPPED) continue;
            int wNum = tarjanSCC(adj, w, state, dfsNum, dfsLow, stack);
            dfsLow[v] = Math.min(dfsLow[v], wNum);
        }
        state[v] = FINISHED;
        if (dfsLow[v] >= dfsNum[v])
        {
            System.out.print("SCC:");
            while (true)
            {
                int w = stack.remove(stack.size()-1);
                state[w] = POPPED;
                System.out.print(" "+w);
                if (w == v) break;
            }
            System.out.println();
        }
        else {}  //dfsLow[v] < dfsNum[v]
        return dfsLow[v];
    }
}
```

Tarjan SCC Exercises

1. Run Tarjan's SCC algorithm on the following graph.



(a) What are the DFS numbers for each vertex?

(b) Give the strongly connected components in the order that Tarjan's algorithm outputs them.

2. ($\star\star$) Given any directed graph G we can create a new graph H where each vertex of H is an SCC of G. There is a directed edge from SCC C_1 to SCC C_2 in H if there is an edge from some vertex of C_1 to some vertex of C_2 in G. What type of graph must H be?

Tarjan SCC Solutions

1. (a) The order DFS visits the vertices is 0,1,3,4,2,5,7,6 so the DFS numbers for $0, 1, \ldots, 7$ are $0, 1, 4, 2, 3, 5, 7, 6$.

   (b) 4,3,1 then 6,7,5,2 and then 0.

2. The resulting graph H is a DAG. If there was a cycle in H, then all of the components in the cycle could be merged into a strongly connected component, and this is a contradiction.