4.2.10 Finding Strongly Connected Components (Directed Graph)

Yet another application of DFS is to find *Strongly* Connected Components (SCCs) in a *directed* graph, e.g., UVa 11838 - Come and Go. This is a different problem to finding Connected Components (CCs) in an undirected graph. In Figure 4.7, we have a directed graph. Although this graph looks like it has one CC (running `dfs(0)` does reach all vertices in the graph), it is actually not an SCC (for example, vertex 1 cannot go to vertex 0). In directed graphs, we are more interested with the notion of SCC instead of the more basic CC. An SCC is defined as such: if we pick any pair of vertices u and v in the SCC, we can find a path from u to v and vice versa. There are actually three SCCs in Figure 4.7, as highlighted with the three boxes: {0}, {1, 2, 3}, and {4, 5, 6, 7}. Note that if these SCCs are contracted (replaced by larger vertices), they form a DAG (see Book 2).

There are at least two known algorithms to find SCCs: Kosaraju's—explained in [5] and Tarjan's algorithm [55]. In this section, we explore both versions. Kosaraju's algorithm is easier to understand but Tarjan's version extends naturally from our previous discussion of finding Articulation Points and Bridges—also due to Tarjan.

Kosaraju's Algorithm

To understand how Kosaraju's algorithm works, we need to do two observations.

First, running `dfs(u)` on a directed graph where u is part of its "smallest SCC" (SCC where all outgoing edges of the vertices in the SCC only point to another member of the SCC itself) will only visit vertices in that smallest SCC. For example in Figure 4.7—left, if we run `dfs(4)` (or `dfs(5)`, `dfs(6)`, or `dfs(7)`), we can only visit vertices $\{4, 5, 6, 7\}$. Notice that if we run `dfs(3)` for example, we will be able to reach vertices $\{1, 2, 3\}$ *as well as* vertices $\{4, 5, 6, 7\}$ due to presence of edge $3 \rightarrow 4$ that can cause 'leakage'. The question is how to find the "smallest SCC"?

Second, the SCCs of the original directed graph and the SCCs of the transposed graph are identical.

Kosaraju's algorithm combine the two ideas. Running DFS on the *original directed graph*, we can record the explored vertices in *decreasing finishing order* (or post-order, similar as in finding topological sort[10] in Section 4.2.6). For example in Figure 4.7—left, the decreasing finishing order of the 8 vertices is $\{0, 1, 3, 4, 5, 7, 6, 2\}$. In turns out that on the transposed graph, these ordering can help us identify the "smallest SCC" (read [5] for the details).

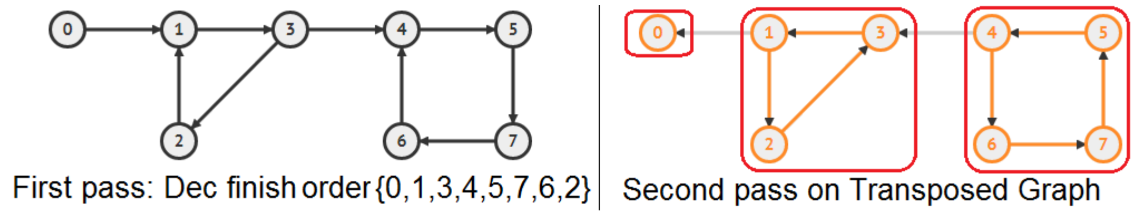
First pass: Dec finish order{0,1,3,4,5,7,6,2}
Second pass on Transposed Graph

Figure 4.7: Execution of Two Passes Kosaraju's Algorithm

Running `dfs(0)` on the *transposed graph* (see Figure 4.7—right), we immediately get stuck as there is no outgoing edge of vertex 0. Hence we find our first (and smallest) SCC. If we then proceed with `dfs(1)`, we have the next smallest SCC $\{1, 2, 3\}$ (as now DFS will not go via edge $1 \rightarrow 0$ as vertex 0 has been visited, i.e., we have "virtually removed" the first SCC). We skip `dfs(3)` as it will not do anything. Finally, if we then proceed with `dfs(4)`, we have the next (and final) smallest SCC $\{4, 5, 6, 7\}$ (as now DFS will not go via edge $4 \rightarrow 3$ as vertex 3 has been visited, i.e., we again have "virtually removed" the second SCC).

[10]But this may not be a valid topological sort as the original directed graph will very likely be cyclic.

These two passes of DFS is enough to find the SCCs of the original directed graph. The simple C++ implementation of Kosaraju's algorithm is shown below.

```
void Kosaraju(int u, int pass) { // pass = 1 (original), 2 (transpose)
  dfs_num[u] = 1;
  vii &neighbor = (pass == 1) ? AL[u] : AL_T[u]; // by ref to avoid copying
  for (auto &[v, w] : neighbor)                   // C++17 style, w ignored
    if (dfs_num[v] == UNVISITED)
      Kosaraju(v, pass);
  S.push_back(u);                                 // similar to toposort
}
```

```
// inside int main()
  S.clear();                                      // first pass
  dfs_num.assign(N, UNVISITED);                   // record the post-order
  for (int u = 0; u < N; ++u)                     // of the original graph
    if (dfs_num[u] == UNVISITED)
      Kosaraju(u, 1);
  numSCC = 0;                                     // second pass
  dfs_num.assign(N, UNVISITED);                   // explore the SCCs
  for (int i = N-1; i >= 0; --i)                  // based on the
    if (dfs_num[S[i]] == UNVISITED)               // first pass result
      ++numSCC, Kosaraju(S[i], 2);                // on transposed graph
  printf("There are %d SCCs\n", numSCC);
```

Tarjan's Algorithm

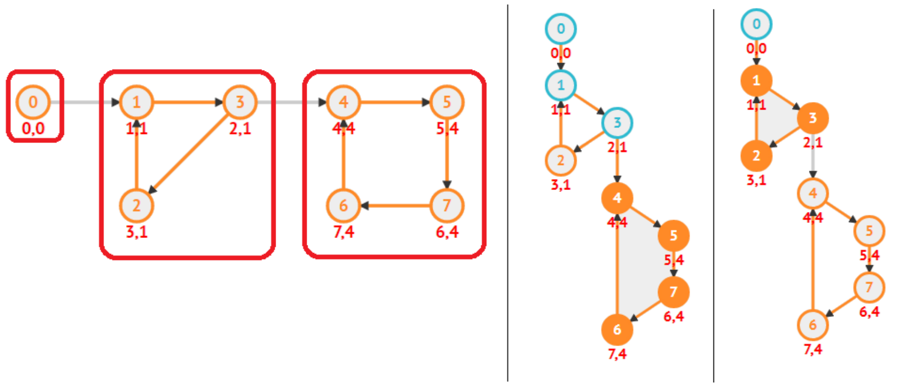

Figure 4.8: Left: Directed Graph; Middle+Right: DFS Spanning Tree Snapshots

The basic idea of Tarjan's algorithm is that SCCs form subtrees in the DFS spanning tree (compare the original directed graph and the two snapshots of its DFS spanning trees in Figure 4.8). On top of computing `dfs_num(u)` and `dfs_low(u)` for each vertex, we also append vertex u to the back of a stack S (here the stack is implemented with a vector) and keep track of the vertices that are currently explored via `vi visited`. The condition to update `dfs_low(u)` is slightly different from the previous DFS algorithm for finding articulation points and bridges. Here, only vertices that currently have `visited` flag turned on (part of the current SCC) that can update `dfs_low(u)`. Now, if we have vertex u in this DFS spanning tree with `dfs_low(u) = dfs_num(u)`, we can conclude that u is the root (start) of an SCC (observe vertex 0, 1, and 4) in Figure 4.8 and the members of those SCCs are identified by popping the current content of stack S until we reach vertex u again.

In Figure 4.8—middle, the content of S is $\{0, 1, 3, 2, \underline{4, 5, 7, 6}\}$ when vertex 4 is found as root of an SCC (`dfs_low(4) = dfs_num(4) = 4`), so we pop elements in S one by one until we reach vertex 4 and we have this SCC: {4, 5, 6, 7}. Next, in Figure 4.8—right, the content of S is $\{0, \underline{1, 3, 2}\}$ when vertex 1 is identified as the root of another SCC (`dfs_low(1) = dfs_num(1) = 1`), so we pop elements in S one by one until we reach vertex 1 and we have SCC: {1, 2, 3}. Finally, we have the last SCC with one member only: {0}.

The C++ implementation of Tarjan's algorithm is shown below. This code is basically a tweak of the standard DFS code. The recursive part is similar to standard DFS and the SCC reporting part will run in amortized $O(V)$ times, as each vertex will only belong to one SCC and thus reported only once. In overall, this algorithm still runs in $O(V + E)$.

```
int dfsNumberCounter, numSCC;                   // global variables
vi dfs_num, dfs_low, visited;
stack<int> St;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter;    // dfs_low[u]<=dfs_num[u]
  dfsNumberCounter++;                            // increase counter
  St.push(u);                                    // remember the order
  visited[u] = 1;
  for (auto &[v, w] : AL[u]) {
    if (dfs_num[v] == UNVISITED)
      tarjanSCC(v);
    if (visited[v])                              // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v]);
  }
  if (dfs_low[u] == dfs_num[u]) {                // a root/start of an SCC
    ++numSCC;                                    // when recursion unwinds
    while (1) {
      int v = St.top(); St.pop();
      visited[v] = 0;
      if (u == v) break;
    }
  }
}
```

```
// inside int main()
  dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
  while (!St.empty()) St.pop();
  dfsNumberCounter = numSCC = 0;
  for (int u = 0; u < V; ++u)
    if (dfs_num[u] == UNVISITED)
      tarjanSCC(u);
```

Source code: `ch4/traversal/UVa11838.cpp|java|py|ml`

Exercise 4.2.10.1: Prove (or disprove) this statement: "If two vertices are in the same SCC, then there is no path between them that ever leaves the SCC"!
