CS 405: Algorithm Analysis II
Homework 4: Depth-first search applications

1. Exercise 22.4-2 in the text.

Given a directed acyclic graph \( G = (V, E) \) and two vertices \( u, v \in V \), construct a linear time algorithm, \( O(V + E) \), that returns the number of distinct simple paths from \( u \) to \( v \). A path is a sequence of edges \( (e_1, e_2, \ldots, e_k) \) with the destination vertex of \( e_i \) equal to the source vertex of \( e_{i+1} \), for \( 1 \leq i < k \). Two paths are distinct if their sequences have different length or if their sequences have the same length but differ in some component. Explain why your algorithm is linear.

2. Problem 22.4 in the text (Reachability).

Let \( G = (V, E) \) be a directed graph in which each vertex \( u \) is assigned a unique integer label \( L(u) \) from the set \( \{1, 2, \ldots, |V|\} \). For each vertex \( u \in V \), let \( R(u) \) be the set of vertices that are reachable from \( u \). Define

\[
F(u) = \arg\min\{L(v) : v \in R(u)\}.
\]

That is, \( F(u) \) is that vertex \( v \) such that \( L(v) = \min\{L(w) : w \in R(u)\} \). Note that \( u \) is always reachable from \( u \) via a path of length zero. Therefore \( u \in R(u) \) and \( L(F(u)) \leq L(u) \). Find a linear algorithm, \( O(V + E) \), that computes \( F(u) \) for all \( u \in V \). Explain why your algorithm is linear.

**Solutions.**

1. Given a directed acyclic graph \( G = (V, E) \) and two vertices \( u, v \in V \), construct a linear time algorithm, \( O(V + E) \), that returns the number of distinct simple paths from \( u \) to \( v \). A path is a sequence of edges \( (e_1, e_2, \ldots, e_k) \) with the destination vertex of \( e_i \) equal to the source vertex of \( e_{i+1} \), for \( 1 \leq i < k \). Two paths are distinct if their sequences have different length or if their sequences have the same length but differ in some component. Explain why your algorithm is linear.

The topological sort algorithm runs DFS and stacks the vertices as they finish. Popping this stack recovers vertices in order of decreasing finish time, and this order constitutes a topological sort in which all edges point forward in the listing. Therefore if \( u.f < v.f \), then \( u \) follows \( v \) in the listing and there can be no paths from \( u \) to \( v \). On the other hand, if \( u.f > v.f \), then \( v \) follows \( u \) in the listing and any nodes that participate in a path from \( u \) to \( v \) must lie between \( u \) and \( v \) in the listing.

The plan for our algorithm is to run DFS as in topological sort, adding each vertex to the beginning of a linked list as it finishes. If \( u \) finishes before \( v \), we report zero for the number of distinct path from \( u \) to \( v \). If \( v \) finishes first, then we place a count attribute in each vertex, say \( \text{cnt} \). We initialize \( v.\text{cnt} \) to 1, acknowledging that there exists exactly one path from \( v \) to \( v \), namely the path with zero links. We initialize \( w.\text{cnt} \) to zero for all other vertices. Now, when the next node finishes, say \( x \), it may have paths to \( v \). If so, nodes on those paths have already finished, because all edges point toward earlier finish times. In particular, the number of distinct paths from \( x \) to \( v \) will be

\[
x.\text{cnt} = \sum_{y \in x.\text{Adj}} y.\text{cnt}.
\]

In the summation, any nodes \( y \in x.\text{Adj} \) that have finish times less than \( v.f \) will contribute zero to the sum. They correspond to neighbors of \( x \) that are downstream from \( v \) and therefore can never continue on to \( v \). By induction, we can assume that nodes \( y \in x.\text{Adj} \) that have nonzero \( \text{cnt} \) attributes are such that the nonzero \( \text{cnt} \) is the number of distinct paths from \( y \) to \( v \). The summation therefore gives the number of distinct paths from \( x \) to \( v \). We can continue the algorithm until it terminates, or we can interrupt it when \( u \) finishes. In either case, \( u.\text{cnt} \) contains the number of distinct paths from \( u \) to \( v \). With some variation for efficiency, the code is as follows. Note that each node visited may have its adjacency list scanned twice instead of once as in the topological sort algorithm. The complexity remains \( \Theta(V + E) \).
PathCount(G, u, v) { // G is directed acyclic, u, v in G.V
    for w in G.V {
        w.cnt = 0;
        w.color = white;
        w.pi = null;
        list = null; // initialize finish-time stack
    }
    time = 0;
    for w in G.V {
        if (w.color == white)
            Visit(G, u, v, w);
    }
    return u.cnt;
}

Visit(G, u, v, w) {
    time = time + 1;
    w.d = time;
    w.color = gray;
    for x in w.Adj
        if (x.color == white)
            x.pi = w;
            Visit(G, u, v, x);
    w.color = black;
    time = time + 1;
    w.f = time;
    if (w == v)
        v.cnt = 1;
    else if (v.cnt == 1)
        for x in w.Adj
            w.cnt = w.cnt + x.cnt;
}

2. Let \( G = (V, E) \) be a directed graph in which each vertex \( u \) is assigned a unique integer label \( L(u) \) from the set \{1, 2, ..., |V|\}. For each vertex \( u \in V \), let \( R(u) \) be the set of vertices that are reachable from \( u \). Define

\[
F(u) = \arg\min\{L(v) : v \in R(u)\}.
\]

That is, \( F(u) \) is that vertex \( v \) such that \( L(v) = \min\{L(w) : w \in R(u)\} \). Note that \( u \) is always reachable from \( u \) via a path of length zero. Therefore \( u \in R(u) \) and \( L(F(u)) \leq L(u) \). Find a linear algorithm, \( O(V + E) \), that computes \( F(u) \) for all \( u \in V \). Explain why your algorithm is linear.

The Strongly Connected Components (SCC) algorithm identifies the components as the \( G_\pi \) trees obtained by scanning the vertices of \( G^T \) in the decreasing finish order that was established by an initial depth-first search. Since each vertex in the same strongly connected component must be able to reach the same minimally labeled vertex, all vertices \( u \) in a component will have the same \( F(u) \) value. We establish a vector \( A[1..n] \), where \( n = |G.V| \). We will not necessarily use the entire vector, just an initial section corresponding the number of strongly connected components. The following code establishes \( A[1..c] \), where \( c \) is the number of strongly connected components. In particular, at its conclusion, we have \( A[i] \) equal to the vertex with the smallest label (L-value) in strongly connected component \( i \). Moreover, each vertex \( u \) contains \( u.scc \), which identifies its strongly-connected component. But, we are not quite finished because a component may have an edge pointing downstream where a vertex with a smaller label may be reachable. Therefore, the depth-first algorithm must be run once more, updating the values in the \( A \)-array to reflect downstream accessible nodes. The final code is MinLabel. The complexity remains linear, \( \Theta(V + E) \), because it involves three passes through minimally modified depth-first search followed by a vertex scan to install in vertex \( u \) the proper \( u.minReachableVertex \) attribute.
MinLabel(G) { // G is directed, u in G.V => u.L is distinct label in range 1..|G.V|
    for v in G.V {
        v.F = 0;
        v.color = white;
        v.TransAdj = null; // will hold adjacency list representation of G-transpose
        finList = null; // initialize finish-time list
    }
    time = 0;
    for v in G.V // start first DFS
        if (v.color == white)
            Visit(G, v);
    Compute G-transpose; // puts transpose links for u in u.TransAdj;
    compCnt = 0;
    for v in finList // start 2nd DFS, taking vertices in decreasing finish order from 1st DFS
        v.color = white;
        for v in finList
            if (v.color == white) {
                compCnt = compCnt + 1;
                A[compCnt] = v;
                TransVisit(G, v);
            }
    for v in finList // start 3rd DFS to update A[1..c]
        v.color = white;
        for v in finList
            if (v.color == white)
                FinalVisit(G, v);
        for v in finList // in each node, identify the reachable vertex with the minimal label
            v.minReachableVertex = A[v.scc];
}

Visit(G, w) {
    time = time + 1;
    w.d = time;
    w.color = gray;
    for x in w.Adj
        if (x.color == white)
            Visit(G, x);
    w.color = black;
    time = time + 1;
    w.f = time;
    finList = new cell(w, finList);
}

TransVisit(G, w) {
    w.color = gray;
    w.scc = compCnt;
    for x in w.TransAdj
        if (x.color == white) {
            if (x.L < A[compCnt].L)
                A[compCnt] = x;
            TransVisit(G, x);
        }
    w.color = black;
}
FinalVisit(G, w) {
    w.color = gray;
    for x in w.Adj {
        FinalVisit(G, x);
        lowVertex = A[x.scc];
        if (lowVertex.L < A[w.scc].L)
            A[w.scc] = lowVertex;
    }
    w.color = black;
}