TD 4: Connectivity Applications

1 Longest Path on a DAG

Give an algorithm which takes as input a DAG G and outputs the longest directed path in G. Consider using a greedy algorithm. Is greediness a good idea?

Solution:

We start with a natural but ill-advised greedy solution:

1: Topological-Sort(G) ▷ Sort into $1, 2, \ldots, n$
2: Path$\leftarrow \{1\}$, cur$\leftarrow 1$
3: **while** $N^+(\text{cur}) \neq \emptyset$ **do**
4: next$\leftarrow \min(N^+(\text{cur}))$ ▷ Pick first outneighbor in ordering
5: Add next to Path
6: cur$\leftarrow$next
7: **end while**
8: Output Path

The informal idea of the algorithm is that we maintain a current vertex and as long as this vertex is not a sink, we move to one of its outneighbors, preferring to use the vertex that is earliest in the topological ordering of the DAG. Although this sounds natural, it is **incorrect** as a criterion, because it is easy to construct a DAG where this algorithm may fail badly to produce the longest path. For example, consider a path $1 \to 2 \to \ldots \to n$ and add a vertex x with arcs $1 \to x$ and $x \to n$. A valid topological sort is $1, x, 2, 3, \ldots, n$, and in case we use this ordering the algorithm will produce a path of length only 2, instead of the obvious answer which is a path of length $n - 1$.

An important note in the above is that, since we are performing worst-case analysis, when we use a subroutine whose output is not fully defined (as is the case here, as a DAG may have many topological orderings), we need to be able to prove that our algorithm works **for any** such output, as we cannot a priori be sure which of the valid outputs will be produced by the subroutine. Concretely, even though the DAG we constructed also admits other topological orderings which make the algorithm work better, the important question is not if the algorithm works correctly for some outputs of line 1, or even for most such outputs, but whether it is guaranteed to work correctly for all such outputs.

Correct algorithm: Let us now use a different strategy. For each vertex u we want to calculate $L[u]$ to be the length of the longest path starting at u and $P[u]$ to be the next vertex in one such path. If we can construct such tables we can solve the problem: find the vertex u with maximum $L[u]$ and output the path $u, P[u], P[P[u]], \ldots$.

The key insight is now that $L[u], P[u]$ only depend on vertices which come **after** u in a topological ordering, since a path that starts at u can never visit a vertex before u. Furthermore, if for some u we already know for all $v \in N^+(u)$ the values $L[v]$, we can compute $L[u]$ by simply picking the maximum.

1: $L[1 \ldots n] \leftarrow 0$, $P[1 \ldots n] \leftarrow -1$
2: Topological-Sort(G) ▷ Sort into $1, 2, \ldots, n$
3: **for** $i = n$ to 1 **do**
4: **if** $N^+(i) = \emptyset$ **then**
5: Skip this i
6: **end if**
7: best$\leftarrow$ First($N^+(i)$)
8: **for** $j \in N^+(i)$ **do**
9: **if** $L[j] > L[\text{best}]$ **then**

```
        best← j
      end if
    end for
    L[i] ← L[best] + 1
    P[i] ←best
end for
```

Correctness: We claim that after every iteration of the main for loop, the values L, P have been correctly computed for vertices $i \dots n$. For the initial iteration, $i = n$, so the current vertex is a sink and the If statement forces us to continue to the next iteration, leaving the initial value $L[n] = 0$, which is correct.

Suppose now that we have a specific value of i but thanks to previous iterations we have correctly computed the values L, P for vertices $i + 1, \dots, n$. The loop will only modify $L[i], P[i]$ and set $L[i]$ to $L[\text{best}] + 1$. Here, best is the index of the outneighbor of i with largest L value, hence which has the property that, among outneighbors of i, best allows the construction of the longest directed path. Therefore, the longest directed path starting at i is the one we can construct by using the arc from i to best.

The running time of the algorithm is $O(n + m)$, first for topological-sort and then because we consider every arc once. The space complexity is $O(n)$.

2 Count Paths on DAGs

Design an algorithm that takes as input a DAG G and two vertices s, t and outputs the number of distinct paths from s to t in G. At most how many such paths may exist in a DAG with n vertices? **NB:** Your algorithm does not need to list the paths, just output their number.

Solution:

We start by topologically sorting the DAG (in linear time, as seen in class). Remove from the graph all vertices which are ordered before s or after t. This is safe, as a vertex ordered before s is not reachable from s, and a vertex ordered after t cannot reach t, therefore such vertices do not participate in any of the paths we want to count. Suppose then that without loss of generality, s is vertex 1 in the ordering and t is vertex n.

Consider now the following algorithm:

```
Initialize PathsS[1..n] ← 0 everywhere, except PathsS[1] = 1
for i = 1 to n do
    for j ∈ N+(i) do
        PathsS[j] ←PathsS[i]+PathsS[j]
    end for
end for
```

Correctness: The invariant we want to prove is that after i iterations of the loop we have the following: for all j, the value of PathsS$[j]$ is equal to the number of distinct paths from s to j which only use vertices from $\{1, \dots, i\} \cup \{j\}$.

After 1 iteration the invariant holds: for vertex 1 (s) there is exactly one path from s to s; while for vertices $j > 1$ we either have PathsS$[j] = 1$, if j is an outneighbor of s, so there is a unique path that uses only vertices from $\{1, j\}$; or PathsS$[j] = 0$, if j is not an outneighbor of s, in which case indeed there is no path that restricted to vertices $\{1, j\}$.

Suppose that the invariant holds after i iterations and we want to prove that it still holds after $i+1$ iterations. For $j \leq i + 1$ it is not hard to see that the computed value stays unchanged, as we examine arcs coming out of $(i + 1)$, which must go towards later vertices. We therefore focus on $j > i + 1$.

For vertices such that the arc $(i+1)j$ does not exist, the number of $s \to j$ paths using the set $\{1, \dots, i\} \cup \{j\}$ is the same as the number of paths using the set $\{1, \dots, i + 1\} \cup \{j\}$, as any path that reaches $i + 1$ cannot reach j (without leaving the set of allowed vertices). Hence, we correctly leave PathsS$[j]$ unchanged for such vertices. For vertices j with an incoming arc from $i + 1$, the set of $s \to j$ paths can be partitioned into two sets: those that avoid vertex $i+1$ (which were previously counted in PathsS$[j]$); and those that use vertex $i+1$. In the latter case, the path must use the arc from $i+1$ to j, so the paths of the second case are eactly PathsS$[i+1]$, that is, the number of $s \to i + 1$ paths using only vertices from $\{1, \dots, i + 1\}$, which we had previously computed.

Assuming the invariant holds till the end, after all iterations PathsS$[n]$ will contain the desired value.

Complexity: If we assume that line 4 takes constant time, then this algorithm runs in linear time, via usual arguments (the total number of executions is proportional to the sum of all degrees, giving an $O(m+n)$ time upper bound). Similarly, the array PathsS takes $O(n)$ space, so together with the space complexity of topological sorting we have $O(n)$ space.

For a more careful analysis, we can observe that it is easy to construct DAGs where the total number of paths from s to t is exponential in n. For instance, take a complete graph on n vertices $\{1,\ldots,n\}$ and orient all arcs towards the larger endpoint. The number of paths from 1 to n in this graph is 2^{n-2}. To see this, observe that any subset of $\{1,\ldots,n\}$ that contains 1 and n corresponds to a path from 1 to n if we order its vertices in increasing order. Therefore, the numbers we are manipulating in line 4 have values exponential in n, meaning that additions have cost $\Theta(n)$ (and not $O(1)$), if we want to make sure we avoid overflows. The precise time complexity of this algorithm is therefore $O(nm+n^2)$ (and the space complexity is $O(n^2)$).

We also note that 2^{n-2} is an upper bound on the number of paths from s to t in any DAG. Indeed, if $T(n)$ is the maximum number of such paths in an n-vertex DAG, we have $T(n) \leq \sum_{i=1}^{n-1} T(n-i)$, because the first step of any path will advance i vertices in a topological sorting of the DAG, leaving $n-i$ vertices left to use. However, if we take into account that $T(1) = T(2) = 1$, then $\sum_{i=1}^{n-1} T(n-i) \leq \sum_{i=1}^{n-1} T(i) \leq 1 + \sum_{i=2}^{n-1} 2^{i-2} = 1 + 2^{n-2} - 1 = 2^{n-2}$ as required, where we used as inductive hypothesis that $T(i) \leq 2^{i-2}$ for $i \geq 2$.

3 Alternative Topological Sort

In class we saw a linear-time algorithm for topologically sorting a DAG based on DFS. For this exercise we will design an algorithm from first principles. Show that a DAG can be sorted in linear time by computing the indegree of each vertex and then repeatedly removing sources and updating the indegrees as needed.

Solution:

```
1:  Initialize indeg[1...n] ← 0
2:  for i = 1 to n do                                  ▷ Calculate indegrees
3:      for j ∈ N⁺(i) do
4:          indeg[j] + +
5:      end for
6:  end for
7:  Initialize Queue Q ← ∅
8:  for i = 1 to n do
9:      if indeg[i] == 0 then
10:         Enqueue(Q,i)
11:     end if
12: end for
13: while Q not empty do
14:     x ← Dequeue(Q)
15:     Append x to output
16:     for y ∈ N⁺(x) do
17:         indeg[y] − −
18:         if indeg[y] == 0 then
19:             Enqueue(Q,y)
20:         end if
21:     end for
22: end while
```

Correctness: The first part of the algorithm computes the indegree of every vertex and enqueues all sources. For the while loop we maintain the following invariant: we will call the "current graph", the graph obtained from G if we delete all vertices which have been dequeued and we claim that indeg maintains the indegree of every vertex of the current graph (in that graph). Furthermore, the vertices in the queue are exactly the vertices

of indegree 0 in the current graph.

The invariant is true before we enter the loop (we have not dequeued or deleted anything, we have the correct indegrees, and we have enqueued all sources). After one iteration of the loop, we observe that indegrees are correctly updated, as we are preparing to remove x from the graph. We will only enqueue a vertex y if it becomes a source by the removal of x. Furthermore, since indegrees are correctly computed so far and the queue already contains the sources of the current graph, only outneighbors of x can become new sources and such vertices are handled in lines 18–19.

By the above we conclude that at each iteration we dequeue a source of the current graph, hence the output is a topological ordering.

Complexity: Assuming $O(1)$ complexity for queue operations, we spend $O(1)$ time per arc, hence $O(m+n)$ time complexity. The space complexity is $O(n)$, to store the indegrees and the queue.

4 A bridge too far

In class we saw that an articulation point is a vertex whose removal disconnects the graph (more precisely, increases the number of components). The equivalent notion for edges is that of a **bridge**: an edge e is a bridge if removing it from a graph increases the number of connected components.

Give an algorithm which takes as input a connected undirected graph and identifies all bridges.

Solution:

Rather than reinventing an algorithm, we can try a **reduction**: given a graph $G = (V, E)$ we construct a graph $G' = (V', E')$ as follows:

- $V' = \{v_1, v_2 \mid v \in V\} \cup \{e \mid e \in E\}$. In other words, we keep every vertex of V and also make a second copy of that vertex; and we turn every edge of G into a vertex.
- $E' = \{v_i e \mid i \in \{1,2\},\ e \in E\ v \in e\}$. In other words, if the vertex representing $e \in E$ we had $e = uv$, then in the new graph e is adjacent to u_1, u_2, v_1, v_2.

In graph theory terminology, what we did was **sub-divide** every edge of the graph (introducing a new vertex in the middle) and then we added a **twin** for every original vertex.

We now claim the following: all bridges of G are articulation points of G' and vice-versa. Therefore, since the number of vertices and edges of $G' = O(n+m)$, if we can produce G' in linear time (see below), we can run the articulation algorithm we saw in class on G' to find the bridges of G.

To see that a bridge e of G is an articulation point of G' is easy: removing e in G' has the same effect for connectivity as removing e in G. For the converse direction, we need to argue that all articulation points of G' correspond to edges of G. Indeed, for $v \in V$ we see that v_1 cannot be an articulation point, as v_2 has the same neighbors as v_1 in G', hence deleting v_1 does not destroy connectivity. Therefore, an articulation point of G' is in fact an edge of G and again connectivity is preserved by sub-divisions of edges.

What is perhaps not entirely obvious is that G' can be produced in linear time. For this, we first observe that if we ignore the twin vertices G' will have $n+m$ vertices and $2m$ edges. We therefore initialize an array of $n+m$ adjacency lists which are initially empty. We then do the following: for each edge $e = ij$, if e is the k-th edge considered, we add $n+k$ to the adjacency lists of i, j and i, j to the adjacency list of $n+k$.

Adding twins to the graph is now easy: we simply have to copy the adjacency list of v to a new vertex to obtain a twin for v.