Algorithms. Lecture Notes 11

Fast Construction of a Topological Order

A possible $O(m)$ time algorithm to construct a topologocial order of a given DAG uses DFS and is based on the following equivalence: G is a DAG if and only if directed DFS, with an arbitrary start node, does not yield any back edges. To see the "only if" part, note that a back edge (u, v) together with the tree edges on the path from v to u form a cycle. The proof of the "if" direction gives a method to construct a topological order: Run DFS again, but with two modifications: Ignore all edges that are not in the DFS tree, and call the children of each node in reverse order (i.e., compared to the first run). Append each node to the result, as soon as it is marked as explored. Since there are no back edges, and all cross edges go "to the left", it is not hard to see that we actually get a topological order. (As you notice, this algorithm is conceptually a bit more complicated than the one proposed above.)

However, there is a simpler algorithm, based on the equivalence we proved in the previous section. Remember that a topological order starts with an arbitrary node without incoming edges. However, for an efficient implementation we have to be careful: How do we find, in every step, a node without incoming edges in the remaining graph? Naive search from scratch is unnecessarily slow. But some good ideas are straightforwardly to obtain: By removing parts of the graph, the in-degree of every node can only decrease. Furthermore, recall that an arbitrary node with in-degree 0 can be chosen as the next node. Together this suggests counting and queuing: In the beginning, count the incoming edges for every node. This is done in $O(m)$ time. Put all nodes with in-degree 0 in a queue. In every step, take the next node u from queue, and subtract 1 from the in-degrees of all v with $(u, v) \in E$. Since this is done only once for every edge (u, v), updating the in-degrees costs $O(m)$ time in total, if G is represented by adjacency lists. Altogether, we can recognize DAGs and construct a topological order in $O(m)$ time.

Problem: Longest Paths

Given: an undirected or directed graph $G = (V, E)$, the lengths $l(u, v)$ of all edges $(u, v) \in E$, and a start ("source") node $s \in V$.

Goal: For all nodes $x \in V$, compute a (directed) path from s to x with maximum length, but such that no node appears repeatedly on the path.

The Shortest Paths problem with a source s is similarly defined.

Motivations:

Finding longest paths is not a silly problem. In particular, it makes much sense on DAGs. For example, if the DAG is the plan of a project with parallelizable tasks modelled by the edges, and the edge lengths are execution times, then the longest path in the graph gives the necessary execution time (makespan) for the whole project. It is sometimes called the critical path.

Shortest and Longest Paths in DAGs

Shortest paths in directed graphs with unit edge lengths can be computed by BFS, as we have seen. An extension of this shortest-path algorithm to directed graphs with arbitrary edge lengths is Dijkstra's algorithm that we do not present here. (It may be known from data structure courses, otherwise we refer to section 4.4 of the textbook.)

The Shortest Paths problem is much easier in DAGs. We can take advantage of a topological order, constructed in $O(m)$ time. Since paths must go strictly from left to right, we may suppose that the source s is the first node in the topological order. Assume that we already know the values $d(s, x)$ for the first $k-1$ nodes x in the topological order. (These are not necessarily the $k-1$ nodes closest to s. Let z denote the kth node. Then we have $d(s, z) = \min d(s, x) + l(x, z)$, where the minimum is taken for all x to the left of z. Correctness is evident, since the predecessor of z must be one of these nodes x. This dynamic programming algorithm needs only $O(m)$ time, as we look at every edge only once.

Next, we want to know the *longest* paths from s to all nodes in a DAG. Amazingly we can apply the same algorithm, replacing min with max. However: Think about the question why this is correct! (For general directed graphs we cannot simply take Dijkstra's algorithm and replace min with max? This would not yield the longest paths.)

We remark that the dynamic programming algorithms for many other problems (e.g. String Editing) can be interpreted as shortest- or longest-paths calculations in DAGs. More formally, we can reduce those problems to Shortest

(or Longest) Paths in DAGs. Nevertheless it is still advantageous to use special-purpose algorithms for those problems, because their DAGs have some regular structures, such that memoizing optimal values in arrays is in practice faster than unnecessarily dealing with data structires for (arbitrary) DAGs.

Network Flow – Basics

Let $G = (V, E)$ be a directed graph where every edge e has an integer capacity $c_e > 0$. Two special nodes $s, t \in V$ are called **source and sink**, all other nodes are called internal. We suppose that no edge enters s or leaves t. A **flow** is a function f on the edges such that $0 \leq f(e) \leq c_e$ holds for all edges e (capacity constraints), and $f^+(v) = f^-(v)$ holds for all internal nodes v (conservation constraints), where we define $f^-(v) := \sum_{e=(u,v)\in E} f(e)$ and $f^+(v) := \sum_{e=(v,u)\in E} f(e)$. (As a menominic aid: $f^-(v)$ is consumed by node v, and $f^+(v)$ is generated by node v.) The value of the flow f is defined as $val(f) := f^+(s)$. The Maximum Flow problem is to compute a flow with maximum value.

For any flow f in G (not necessarily maximum), we define the **residual graph** G_f as follows. G_f has the same nodes as G. For every edge e of G with $f(e) < c_e$, G_f has the same edge with capacity $c_e - f(e)$, called a **forward edge**. The difference is obviously the remaining capacity available on e. For every edge e of G with $f(e) > 0$, G_f has the opposite edge with capacity $f(e)$, called a **backward edge**. By virtue of backward edges we can "undo" any amount of flow up to $f(e)$ on e by sending it back in the opposite direction. The **residual capacity** is defined as $c_e - f(e)$ on forward edges and $f(e)$ on backward edges

Now let P be any simple directed $s-t$ path in G_f, and let b be the smallest residual capacity of all edges in P. For every forward edge e in P, we may increase $f(e)$ in G by b, and for every backward edge e in P, we may decrease $f(e)$ in G by b. It is not hard to check that the resulting function f' on the edges is still a flow in G. We call f' an **augmented** flow, obtained by these changes. Note that $val(f') = val(f) + b > val(f)$.

Now the basic Ford-Fulkerson algorithm works as follows: Initially let $f := 0$. As long as a directed $s-t$ path in G_f exists, augment the flow f (as described above).

To prove that Ford-Fulkerson outputs a maximum flow, we must show: If no $s-t$ path in G_f exists, then f is a maximum flow.

The proof is done via another concept of independent interest: An $s-t$ **cut** in $G = (V, E)$ is a partition of V into sets A, B with $s \in A, t \in B$. The capacity of a cut is defined as $c(A, B) := \sum_{e=(u,v):u\in A, v\in B} c_e$.

For subsets $S \subset V$ we define $f^+(S) := \sum_{e=(u,v):u\in S,v\notin S} f(e)$ and $f^-(S) := \sum_{e=(u,v):u\notin S,v\in S} f(e)$. Remember that $val(f) = f^+(s) - f^-(s)$ by definition. (Actually we have $f^-(s) = 0$ if no edge goes into s.) We can generalize this equation to any cut: $val(f) = \sum_{u\in A}(f^+(u) - f^-(u))$, which follows easily from the conservation constraints. When we rewrite the last expression for $val(f)$ as a sum of flows on edges, then, for edges e with both nodes in A, terms $+f(e)$ and $-f(e)$ cancel out in the sum. It remains $val(f) = f^+(A) - f^-(A)$. It follows $val(f) \le f^+(A) = \sum_{e=(u,v):u\in A,v\notin A} f(e) \le \sum_{e=(u,v):u\in A,v\notin A} c_e = c(A,B)$. In words: The flow value $val(f)$ is bounded by the capacity of any cut (which is also intuitive).

Next we show that, for the flow f returned by Ford-Fulkerson, there exists a cut with $val(f) = c(A,B)$. This implies that the algorithm in fact computes a maximum flow.

Clearly, when the Ford-Fulkerson algorithm stops, no directed $s-t$ path exists in G_f. Now we specify a cut as desired: Let A be the set of nodes v such that some directed $s-v$ path is in G_f, and $B = V \setminus A$. Since $s \in A$ and $t \in B$, this is actually a cut. For every edge (u,v) with $u \in A$, $v \in B$ we have $f(e) = c_e$ (or v should be in A). For every edge (u,v) with $u \in B$, $v \in A$ we have $f(e) = 0$ (or u should be in A because of the backward edge (v,u) in G_f). Altogether we obtain $val(f) = f^+(A) - f^-(A) = f^+(A) = c(A,B)$. In words: The flow value $val(f)$ equals the capacity of a minimum cut (which is still intuitive).

The last statement is the famous Max-Flow Min-Cut Theorem.

However it should be noticed that the Ford-Fulkerson algorithm in its basic form may need $O(mC)$ time, where C is the sum of capacities of the edges at the source: An augmenting path can be found by DFS, $val(f)$ increases by at least 1 in every iteration, and $val(f) \le C$. This time bound is not polynomial in the input length. By a careful choice of augmenting paths (e.g., taking the shortest path each time) one can make the algorithm polynomial, but we cannot give this analysis here.