Algorithms. Lecture Notes 11

Algorithms for Minimum Spanning Trees (MST)

The MST problem is one of the most prominent algorithmic problems on graphs. It is not only important in its own right. Here we also use it as yet another illustration of some general issues in algorithm design.

Assume in the following that no two edge costs are equal.

Let us do some rough problem analysis first. Remember that a greedy approach is preferable *if* some greedy rule works for the problem at hand. We would like to specify edges that we can safely put in an MST. It would be natural to choose "somehow" the cheapest edges that do not form cycles. However, there are several ways to turn this idea into an algorithm, and we must prove correctness, probably by some exchange argument.

In fact, there is an elegant lemma about MST, proved by an exchange argument. The exact form of this lemma is not so obvious. (But here we skip the trial-and-error development steps.) The idea is that a spanning tree is still a connected graph, hence it must contain some edge between any two subsets of nodes that are complements. What if we always take the cheapest such edge?

Here comes the lemma: If we partition the node set V arbitrarily in two non-empty sets X and Y, then the cheapest edge $e = (x, y)$ with $x \in X$ and $y \in Y$ must belong to any MST.

For the proof, assume that T is an MST not containing e. Insertion of e in T yields a cycle C. This cycle C must contain another edge $f = (u, v)$ with $u \in X$ and $v \in Y$. By removing f from the cycle we get another spanning tree. Since e was cheaper than f, this spanning tree is cheaper than T, a contradiction.

There exist two different greedy algorithms for MST following the above idea. They are attributed to their inventors Prim and Kruskal, and both are easy to describe in one sentence:

Prim's algorithm starts from an arbitrary node (a "tree" with empty edge set), and always adds a cheapest edge that extends the current tree to a larger tree.

Kruskal's algorithm starts from an empty edge set and always inserts a cheapest edge that does not create cycles with already selected edges.

Equivalently, Kruskal's algorithm always adds a cheapest edge that connects two distinct connected components of already selected edges. In other words, it maintains a **forest** (a partitioning of the node set into a union of trees) and adds a cheapest edge to the forest in every step. For clarity, note that every isolated node without edges is also considered a tree in the forest. The initial forest has no edges.

To prove that these algorithms never choose an erroneous edge, we apply the above lemma. Let T denote the MST.

In every step of Prim's algorithm, we define X to be the set of nodes spanned by the already chosen edges. (Before the first step, X contains only the start node.) Since Prim's algorithm chooses the cheapest edge e between X and Y, we conclude that e belongs to T.

In every step of Kruskal's algorithm, we denote by F the set of the already chosen edges. Let $e = (x, y)$ to be the edge that the algorithm would choose next. We put the connected component of F containing x in X, and we put the connected component of F containing y in Y. All other connected components of F are put in X or Y arbitrarily. Now we have partitioned the node set in two disjoint non-empty sets X and Y, We claim that e is the cheapest edge between X and Y (and thus e belongs to T). If there were a cheaper edge f between X and Y, then f would also satisfy the condition to connect two distinct components of F. But then Kruskal's algorithm would prefer f to e, a contradiction.

Thanks to the lemma, these proofs are elegant, aren't they?

Finally, we can et rid of the restriction that all edge costs be distinct: If edges with equal costs appear in G, we add sufficiently small distinct costs (so caleld perturbations) to them. The above result shows that the algorithms are correct when applied to these distinct costs. But the resulting edge set is an MST also for G with the original edge costs, if the sum of perturbations is small enough, compared to the edge costs. For both algorithms this simply means that we can replace the phrase "*the* cheapest edge" with "*some* (arbitrary) cheapest edge".

A third, less prominent greedy algorithm for MST starts from the given edge set E and always *deletes* the most expensive edge, keeping the graph connected. Correctness is proved along the same lines. We do not further

study this third algorithm. For dense graphs it is slower than the others, as it has to delete most edges.

Implementing Prim's Algorithm

In every step we must find the cheapest edge between the tree X and the remaining set node set Y. A naive implementation that determines every such edge from scratch needs $O(nm)$ time; n steps, each considering m edges. One feels that this is very redundant. X and Y change only by a single node in every step, so why should we compare all edge costs again and again?

A better idea is to maintain a small set of "candidate" edges: Specifically, for every node $y \in Y$ we may store, in some data structure, the cheapest edge (x, y) connecting y with X. Clearly, the cheapest edge between X and Y is one of these candidate edges. Updating the data structure requires $|Y| < n$ comparisons in each step: One node v moves from Y to X. Hence we only have to check for every node $y \in Y$ whether the edge (v, y) is cheaper than the currently stored edge (x, y). Altogether this improves the total running time to $O(n^2)$.

An alternative implementation works with a data structure that explicitly provides what we need in every step, namely the cheapest element in the set H of edges between X and Y. It uses a standard data structure that supports fast insertion and deletion of items, and fast delivery of the currently smallest item in H. It is known as **priority queue**. Now the key observation is: Every edge enters H only once and leaves H only once. (Why?) That is, we have to perform $O(m)$ insert and delete operations. The minimum element in F must be determined $n - 1$ times. It is possible to create a priority queue that executes every operation in $O(\log m)$ time, where m is the maximum size of the set to maintain. Thus, Prim's algorithm can run in $O(m \log m)$ time, which can also be written as $O(m \log n)$.

Both implementations of Prim's algorithm are justified: $O(n^2)$ is somewhat faster if the graph is dense (has a quadratic number of edges), but otherwise $O(m \log n)$ is considerably faster.

Implementing Kruskal's algorithm in a good way is more tricky and therefore not considered in this section.

Problem: Detecting Directed Cycles

A **directed cycle** in a directed graph is a cycle that can be traversed respecting the orientation of the edges. In other words, it is a sequence of nodes $v_1, v_2, v_3, \ldots, v_n, v_1$ where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge. A *directed acyclic graph* (DAG) is a directed graph without directed cycles. DAGs should not be confused with trees which are connected graphs without *any* cycles (which are in general undirected).

Given: a directed graph $G = (V, E)$.

Goal: Find a directed cycle in G, or report that G is a DAG.

Motivations:

Directed cycles are undesirable in plans of tasks where directed edges (u, v) model pairwise precedence relations (task u must be done before task v). These tasks can be calculations in a program or logic circuit, jobs in a project, steps in a manufacturing process, etc. In such models, directed cycles indicate errors in the design.

Some systems in Artificial Intelligence, so-called partial order planners, automatically create plans to achieve some goal, given a formal description of the goal and of available actions. Part of the construction algorithms are tests for directed cycles. If such cycles are detected in a plan, some actions must be removed, and the corresponding partial goals must be realized in a different way, avoiding new cycles.

Problem: Topological Order

A **topological order** of a directed graph $G = (V, E)$ is an order of all nodes of V so that all directed edges go to the right. In other words, for every directed edge (u, v), node u appears earlier than node v in the order.

Given: a directed graph $G = (V, E)$.

Goal: Construct a topological order of G, or report that G does not admit a topological order.

Motivations:

The nodes are jobs with pairwise dependency constraints, as above. Any topological order is a possible order of executing these jobs without violating the precedence constraints.

Directed Acyclic Graphs (DAGs) and Topological Order

We continue with fast algorithms that detect directed cycles or construct a topological order (if existing) in a directed graph G.

Looking at the specifications of these problems, perhaps one can easily guess that G is a DAG if and only if G allows a topological order. The "if" direction is obvious: If all edges go in the same direction, one can never close a directed cycle. The "only if" direction is more interesting, and it has a **constructive proof** in the sense that the proof also shows how to obtain a topological order, provided that G is a DAG. The proof can be done by induction on the number of nodes, as shown below.

Observe that the first node v in a topological order must not have any incoming edges (u, v). Conversely, an arbitrary node v without incoming edges can be put at the first position of a topological order. Here comes the inductive argument why this is true: Remove v and all incident edges from G. The remaining graph is still a DAG. (No directed cycles can be created by removing parts of a graph.) Hence, G without v has a topological order (by the induction hypothesis), and by setting v in front of this topological order, we get a topological order of the entire graph G.

The resulting algorithm is very simple: For $k = 1, \ldots, n$, put an arbitrary node v without incoming edges at the k-th position of the topological order, and remove v and all incident edges from the graph.

This algorithm is obviously correct if it goes through. But how do we know that there always exists such a node v to continue with? Assume by way of contradiction that every node has an incoming edge. Then we can traverse a path of such edges in opposite direction. But since G is finite, we must sometimes meet a node again, contradicting the assumption that G has no directed cycle.

A remarkable detail is that we can always take an *arbitrary* node v without incoming edges. As the proof shows, we can never get stuck and miss a topological order by an unlucky choice of v in some step. In a sense, we can consider this algorithm a greedy algorithm (taking local decisions without looking ahead), although it is not concerned with an optimization problem.

Fast Construction of a Topological Order

But how much time is needed? We first consider an alternative way, also in order to mention some application of DFS.

A possible $O(n+m)$ time algorithm to construct a topologocial order of a given DAG uses DFS and is based on another 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 may have noticed, this DFS-based algorithm is conceptually more complicated than the one based on the equivalence we proved in the previous section. We mention it only for comparison. (Therefore, do not worry if the details are unclear.) Instead let us aim for an efficient implementation of the indegree-based algorithm instead. The in-degree of a node v in a directed graph is the number of incomig directed edges (u, v).

Remember that, in every step, an arbitrary node with in-degree 0 (within the remainder of the graph) can be chosen as the next node in a topological order. But how do we find such a node in every step? Naive search from scratch is unnecessarily slow. But some good ideas can be obtained straightforwardly: Removal of parts of a graph can only decrease the in-degrees of nodes. This suggests counting and queuing: In the beginning, count all incoming edges at every node. This is done in $O(n + m)$ time. Put all nodes with in-degree 0 in a queue. In every step, take any node u from queue, and subtract 1 from the in-degrees of all nodes 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, provided that G is represented by adjacency lists. Altogether, we can recognize DAGs and construct a topological order in $O(n+m)$ time, also without DFS.

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 also known as the critical path.