Algorithms. Lecture Notes 11

Directed Acyclic Graphs (DAGs) and Topological Ordering

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 it is not hard to 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 the **proof is constructive** in the sense that it also shows how to obtain a topological order, provided that G is a DAG. The proof is done by induction on the number of nodes.

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

The resulting algorithm has a very simple structure: Put some node v without incoming edges at the next position of the topological order, remove v and all incident edges, and so on.

This algorithm is obviously correct if it goes through. But how do we know that there always exists such a node v to continue? 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. But G has no directed cycle by assumption.

A remarkable detail is that we can always take an *arbitrary* node v without incoming edges. We can never miss a topological order by an unlucky choice of v in some step. The proof ensures this. In a sense, we can consider this algorithm a greedy algorithm, even though there is nothing to optimize.

The algorithm is simple enough, but for an efficient implementation we still 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.

An alternative $O(m)$ time algorithm 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.)

Minimum Spanning Trees (MST)

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

Let us do some 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. The first step is natural: If e is a cheapest edge in G, then e belongs to some MST. Why is this true? Let T be an MST not including e. Adding e to T creates a cycle

C consisting of e and some edges of T. From C we can remove any edge other than e and get a new MST which is no more expensive. This was a typical exchange argument and gives hope for a greedy algorithm. However, it is not so obvious how we could generalize this “cheapest edge rule” to later steps. The difficulty is that the problem structure changes when some edges are already chosen. We have to make sure that the selected edges never form a cycle, which is an additional restriction.

At least two different generalizations come to mind, leading to two different greedy algorithms, attributed to their inventors Prim and Kruskal. Both algorithms add, to the already selected edges, a cheapest edge, thereby observing the restriction that the final set must be a spanning tree. The algorithms differ slightly in the condition applied in the greedy rule. In Prim's algorithm, the selected edges must always form a tree. In Kruskal's algorithm, the selected edges always form a cycle-free edge set. This is not necessarily a tree, as it can consist of several connected components of edges. We also speak of a **forest**. The algorithms can be described in a few lines:

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 avoids cycles with already selected edges. In other words, it always adds a cheapest edge that extends the current forest to a larger forest.

As usual, the difficulties are in the implementation details. But first we have to make sure that these greedy algorithms are correct.

Correctness of the MST Algorithms

First we suppose that all edges have different costs, this makes the reasoning somewhat simpler. Basically, an exchange argument will establish the correctness of both greedy algorithms, however we have to formulate it in such a way that it works for every step, not only for the cheapest edge (as above). This turns out to be a bit subtle. There is an elegant exchange argument proving some general property of MST that it interesting for its own, and afterwards we use it to show correctness of both algorithms.

Here it comes: If we partition the node set V arbitrarily in two non-empty sets X and Y, then the cheapest edge e among all edges that connect X and Y must belong to any MST. (As a side remark, from this it follows that the MST is uniquely determined if all edge costs are distinct.) For the proof, assume that T is an MST not containing e. Insertion of e in T yields a cycle C. This C must contain another edge f with one end in X and one end 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.

Now consider the (true) MST T. Let $e_1, \ldots, e_{n-1}$ be the edges chosen by Prim's algorithm. If these edges do not form T, there must be a first edge e_k which is not in T. Then, let X be the set of nodes spanned by edges $e_1, \ldots, e_{k-1}$, and $Y = V \setminus X$. By the rule of Prim's algorithm, e_k is the cheapest edge between X and Y. But the above property says that e_k must belong to T, a contradiction.

Similarly, let $e_1, \ldots, e_{n-1}$ be the edges chosen by Kruskal's algorithm. If these edges do not form T, there must be a first edge $e_k = (u, v)$ which is not in T. Now, let X be the set of nodes reachable from u via edges from $e_1, \ldots, e_{k-1}$, and $Y = V \setminus X$. Note that any one edge between X and Y may be added to $e_1, \ldots, e_{k-1}$ without creating a cycle. By the rule of Kruskals's algorithm, e_k is the cheapest edge avoiding cycles. It follows that e_k is also the cheapest edge between X and Y. But the above property says that e_k must belong to T, a contradiction.

Finally we get rid of the restriction that all edge costs be distinct. We use a perturbation argument: If edges with equal costs appear in G, we add sufficiently small distinct costs (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, e.g., smaller than the difference of costs of any two spanning trees. 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 a 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.

Clustering with Maximum Spacing

Kruskal's algorithm has a nice application and interpretation in the field of clustering problems. Suppose that the nodes of our graph are data points, and edge costs are the distances. (The graph is complete, that is, all possible edges exist.) A clustering with maximum spacing (i.e., maximized minimum distance between any two clusters) can be found as follows: Do $n-k$ steps of Kruskal's algorithm and take the node sets of the so obtained k trees $T_1, \ldots, T_k$ as clusters. We prove that the obtained spacing d is in fact optimal: Consider any partitioning into k clusters $U_1, \ldots, U_k$. There must exist two nodes p, q in some T_r that belong to different clusters there, say $p \in U_s, q \in U_t$. Due to the rule of Kruskal's algorithm, all edges on the path in T_r from p to q have cost at most d. But one of these edges joins U_s with U_t, hence the spacing of the other clustering can never exceed d.

Implementing MST Algorithms

Which algorithm is faster: Prim's or Kruskal's? Before we can decide this, we need to study the implementation details.

First we review Prim's algorithm. For each step $k = 1, \ldots n-1$, the problem is to find the cheapest edge e_k between X and Y, that the algorithm will insert next. (Recall that X is always the set of nodes already included in the tree, and $Y = V - X$.) A naive implementation that determines each e_k from scratch needs $O(nm)$ time. A better idea is to maintain a small set of promising candidates for e_k. For every node $y \in Y$ we may store the cheapest edge to some node $x \in X$. Clearly, e_k must be one of these n edges. Updating this "data structure" requires $|Y| < n$ comparisons in each step: One node v is moved from Y to X, hence we have to check for every node in $y \in Y$ whether edge (v, y) is cheaper than edge (x, y), where $x \in X$ is the current "partner" of y. Altogether we have improved 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 F of edges between X and Y. We use a data structure that supports fast insertion and deletion of items, and fast delivery of the currently smallest item in F. It is known as **priority queue**. Every edge enters F only once and leaves F only once. 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 is $O(m \log n)$. One possibility to implement a priority queue is a **heap** (the data structure used in Heapsort), see section 2.5 of the textbook. A balanced search tree may be used as well, but then the constant factor in the time complexity becomes worse. 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.

Union-and-Find

As for the implementation of Kruskal's Algorithm, we face two problems: finding the cheapest edge, and checking whether it creates cycles together with previously chosen edges. (In that case we skip the edge and go to the next cheapest edge, and so on.) The first problem is easily solved: In a preprocessing phase we sort the edges by ascending costs, in $O(m \log n)$ time, and inside Kruskal's algorithm we traverse this sorted list.

Checking cycles is more tricky. The key to fast cycle testing is the following observation. Remember that the already selected edges build a forest. Every node belongs to exactly one tree in this forest. (A node which is not yet incident to a selected edge forms a tree on its own.) Now, a newly inserted edge does not create cycles if and only if it connects two nodes from different trees in this forest.

Thus, we would like to have a data structure that maintains partitions of a set (here: V) into subsets, each denoted by a label, and supports the following operations: FIND(i) shall return the label of the subset of the partitioning that contains element i. UNION(A, B) shall merge the subsets with labels A and B, that is, replace these sets with their union $A \cup B$ and give it a label. (In the following we will not clearly distinguish between a set and its label, just for convenience.) Such a data structure is not only needed in Kruskal's algorithm. It also appears in, e.g., the minimization of the set of internal states of finite automata with specified input-output behaviour. We cannot treat this subject here and just point out that the data structure is of broader interest.

The problem of making an efficient data structure for Union-and-Find is nontrivial. A natural approach is to store all elements, together with the labels of sets they belong to, in an array. Then, FIND(i) is obviously performed in $O(1)$ time. To make the UNION(A, B) operations fast, we need to store every set A, etc., separately in a list, along with information about the size of A. Now, each element appears twice (in the global array and in the list of its set). Copies of the same element may be joined by pointers. We describe how to perform UNION(A, B). Suppose $|A| \leq |B|$. It is natural to change the labels of all elements in the smaller set from A to B, as this minimizes the work to update the partitioning. That is, we traverse the list of A, use the pointers to find these elements also in the array, change their labels, and finally we merge the lists of A and B and add the sizes.

The analysis of this data structure is quite interesting. Every single UNION(A, B) operation can require $O(n)$ steps, namely if the smaller set A is already a considerable fraction of the entire set. However, we are not so much interested in the worst-case complexity of every single UNION operation. In Kruskal's algorithm we need $n - 1$ UNION operations and $O(m)$ FIND operations. The latter ones cost $O(m)$ time altogether. The relevant question is how much time we need *in total* for all UNION operations. Intuitively, the aforementioned worst case cannot occur very often.

Let us study the total time for the first k UNION operations (for any number $k < n$), if the initial partitioning consists of n singleton sets. Instead of staring at the worst case for each UNION operation we change the viewpoint and ask how often every element is relabed and moved. That is, we sum up the elementary operations in a different way.

> "No matter how I sweep a courtyard, the amount of dirt is the same in the end." (An anonymous caretaker.)

An element is relabeled in a UNION operation if it belongs to the smaller set. Hence, after this operation it belongs to a new set of at least double size. Since the largest set after k UNION operations can have at most $k + 1$ elements, it follows that every element is relabeled at most $\log_2(k+1)$ times. Thus we get a time bound $O(k \log n)$ for the first k UNION operations. Applied to Kruskal's algorithm this yields $O(n \log n)$ time for all UNION operations, which is within the $O(m \log n)$ bound that we already needed to sort the edges.

Thus, the above Union-and-Find data structure is "just good enough" for Kruskal's algorithm. However, a faster Union-and-Find structure would further improve the physical running time and might also be useful for other algorithms. We briefly mention a famous Union-and-Find data structure that is faster, yet very easy to implement. (This paragraph may be skipped, without missing anything important for later topics.) We represent the sets of the partitioning as trees whose nodes are the elements. Every tree node except the root has a pointer to its parent, and the root stores the label of the set. Beware: These trees should not be confused with the trees of edges from graph $G = (V, E)$ in a snapshot of Kruskal's algorithm. Instead, they are formed and processed as follows. When FIND(i) is called, we start in node i and walk to the root (where we find the label), following the pointers. When UNION(A, B) is called, then the root of the smaller tree is "adopted" as a new child by the root of the bigger tree. This works in $O(1)$ time, since only one new pointer must be created. By the same doubling argument as before, the depth of any node can increase at most $\log_2(k+1)$ times during the first k UNION operations. As a consequence, every FIND operation needs at most $O(\log k)$ time. Now we can perform every UNION in $O(1)$ time and every FIND in $O(\log k)$ time, where k is the total number of these data structure operations. In the really good implementation, however, trees are also modified upon FIND(i) operations: Root r adopts all nodes on the path from i to r as new children. This "path compression" is not much more expensive than walking the path, but it makes the paths for future FIND operations much shorter. It can be shown that, with path compression, k UNION and FIND operations need together only $O(k)$ time (rather than $O(k \log k)$), subject to an extra factor that grows so extremely slowly that we can ignore it in practice. The time analysis is intricate and is skipped here, but the structure itself is simple.