Algorithms. Lecture Notes 12

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. Also remember that this course is primarily about algorithm design and analysis techniques, not about specific algorithms.) Anyway:

The Shortest Paths problem in DAGs is much easier to solve than in general graph. We can take advantage of a topological order, constructed in $O(n+m)$ 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. All nodes to the left of s can be ignored. Let $l(u,v)$ denote the given length of the directed edge (u,v), provided that it exists, and let $d(u,v)$ denote the length of a shortest path from u to v. Assume that we have already computed the values $d(s,x)$ for the first $k-1$ nodes x in the topological order. Let z denote the k-th 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 on the path must be one of the mentioned nodes x. This dynamic programming algorithm needs only $O(m)$ time, because we look at every edge only once.

Remark for those who know Dijkstra's algorithm well (others may skip it): The first $k-1$ nodes in the topological order are, i general, not the $k-1$ nodes closest to s, because topological order and edge lengths are not related. Therefore, the above algorithm is not Dijkstra's algorithm applied to DAGs. It treats the nodes in a different order.

Next, we want to find *longest* paths from s to all nodes in a DAG. Amazingly we can apply the same algorithm, replacing min with max. Why is this correct? Think about this question. (In general directed graphs we cannot simply take Dijkstra's algorithm and replace min with max. This would not yield the longest path. Actually the problem is NP-complete in general graphs.)

We remark that the dynamic programming algorithms for many other problems (e.g. Sequence Comparison) 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 highly regular structures, such that memoizing optimal values in arrays is in practice faster than (unnecessarily) dealing with data structures for arbitrary DAGs.

Union-and-Find

This is an addendum to Kruskal's algorithm for MST. In an endeavor to achieve a good time bound we face two problems: finding the cheapest edge, and checking whether it creates cycles together with previously chosen edges. (In that case, the algorithm skips the edge and goes to the next cheapest edge, and so on.) The first problem is easily solved: In a preprocessing phase we can sort the edges by ascending costs, in $O(m \log n)$ time, and inside Kruskal's algorithm we merely traverse this sorted list.

Checking cycles is more tricky. Remember that the already selected edges build a forest. Every node belongs to exactly one tree in this forest. The key observation is that 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 partitionings of a set (here: of the node set V) into subsets (here: the forests), each denoted by a label, and supports the following operations: Find(i) shall return the label of the subset of the partitioning that contains the 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 assign a label to it. (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 the latter subject here. This is just mentioned to point out that the Union-and-Find data structure is of broader interest and is not a "one-trick pony".

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, by looking at the table entry of i. To make the Union(A, B) operations fast, too, we could store every set A, etc., separately in a list, along with the size $|A|$. (Without that, we have to collect the elements of A which are spread out in the array ...) Now, each element appears twice: in the global array and in the "compact" list of the set it belongs to. These two copies of each element may be joined by pointers.

Now we describe how to perform Union(A, B): Suppose that $|A| \leq |B|$; the other case is symmetric. It is natural to change the labels of all elements in the smaller set from A to B, as this minimizes the work needed 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 to B, and finally we merge the lists of A and B and add their sizes. Note that the union is named "B" again.

The analysis of this data structure is quite interesting. Any 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 $n - 1 = O(n)$ Union operations and $O(m)$ Find operations. The latter ones cost $O(m)$ time altogether. The remaining question is how much time we need *in total* for all Union operations. Intuitively, the aforementioned worst case cannot occur very often, therefore we should not rush and conclude a poor $O(n^2)$ bound.

Instead of staring at the worst case for every single 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. An element is relabeled in a Union operation only if it belongs to the smaller set. Hence, after this Union operation it belongs to a new set of at least double size. It follows immediately that every element is relabeled at most $\log_2 n$ times, because this is the largest possible number of doublings. Thus we get a time bound $O(n \log n)$ for all $n-1$ Union operations together. This 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 "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 within other algorithms. We briefly sketch a famous Union-and-Find data structure that is faster, yet very easy to implement. 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 in the forest within 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 operation in $O(1)$ time and every Find operation 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 just 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 must be omitted here, but the structure itself is rather simple.

Problem: Interval Partitioning

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis.

Goal: Partition the set of intervals into the smallest possible number d of subsets $X_1, X_2, X_3, \ldots, X_d$, each constisting of pairwise disjoint intervals.

Motivations:

They are similar to Interval Scheduling. The difference is that several "copies" of the resource are available, and *all* requests shall be served, using the smallest number of copies.

A Greedy Algorithm for Interval Partitioning

Let the subsets $X_1, X_2, X_3, \ldots$ initially be empty. We sort the intervals such that $s_1 < \ldots < s_n$, and we consider them in this order. (As opposed to the Interval Scheduling algorithm, they are sorted by their start points – no mistake here!) We always put the current interval x into the subset X_i with the smallest possible index i. That is, we choose the smallest i such that x does not intersect any other interval in X_i.

Here we omit the details of an efficient implementation and come to the interesting point: Why is this greedy rule correct? In fact, optimality may be proved again by an exchange argument, but here we illustrate another nice proof technique: We give a simple bound for the value d to be optimized, and then we show that our solution attains this bound, hence it is optimal.

Specifically, let d be the maximum number of given intervals that contain the same point. (More formally: d is the maximum number of given intervals that have a non-empty intersection.) On the one hand, since any d such intervals must be put in d distinct subsets, any solution needs at least d subsets, even an optimal solution. On the other hand, our greedy algorithm uses only d subsets: Whenever a new interval x starts, at most $d-1$ earlier intervals can intersect x, because any such intervals must contain the start point of x. Hence we can always put x in some of the first d subsets.

This proof is an example of **duality**, a principle that plays a central role in optimization. Given a ‘primal" minimization problem, we consider a “dual" maximization problem, which is chosen in such a way that the maximum dual value is a lower bound on the primal values. Then, if some primal solution happens to attain this dual bound, it must be optimal.

Space-Efficient Sequence Comparison

This section deals with an algorithm where dynamic programming and divide-and-conquer work nicely together. We also address the space complexity of a problem (unlike the rest of the course).

Suppose $m \leq n$. We have seen an algorithm that aligns two sequences $A = a_1 \ldots a_n$ and $B = b_1 \ldots b_m$ in $O(nm)$ time. Unfortunately, it also needs $O(nm)$ space, which can be prohibitive for applications in molecular biology where n, m are huge numbers. What can we do about that?

We may implement the dynamic programming algorithm so that it requires only $O(m)$ space: For computing the values $OPT(i, j)$ we need only the previous row or column of the array of OPT values, but we can forget all earlier values.

But this gives us *only the score* $OPT(n, m)$ of a best alignment. If we are supposed to deliver an optimal alignment as well, we need (potentially) all $OPT(i, j)$ values for the backtracing procedure, since we do not know in advance the optimal path through the array. We could maintain the best alignments of prefixes along with the $OPT(i, j)$ values, but then we are back to $O(nm)$ space complexity.

The striking idea to overcome the space problem is to determine one entry (or "node") in the middle of the optimal path. We get it from the scores, which can be computed in small space by dynamic programming (as we have seen above). Once we know one node on the optimal path, we can split our problem instance in two independent instances and solve them recursively, one after another. Thus, everything happens in small memory space, while the divide-and-conquer structure ensures that we do not lose too much time. Below we describe the resulting algorithm in more detail.

Let $k \approx m/2$. We compute the scores (edit distanes) $OPT(j, k)$ for all j by dynamic programming, in $O(nm)$ time and $O(m)$ space. The same is done for the reversed sequences $a_n \dots a_1$ and $b_m \dots b_1$. The half sequence $b_1 \dots b_k$ *must* be aligned to $a_1 \dots a_j$, for some yet unknown j, and the other half of B to the rest of A. After that splitting, the two optimal alignments are completely independent. In order to find the optimal cut-off point j, we can simply add the scores of these two alignments and pick an index j where the sum of scores is minimized. We come from the left and from the right; note that the edit distance does not change if the sequences are reversed. Clearly, the minimum sum is found in $O(n)$ time and space. Finally we divide B at position k, and we divide A at the position j just determined, and we make two recursive calls.

We never need more than $O(n)$ space simultaneuously. The time complexity is given by the recurrence $T(n, m) = 2T(n, m/2) + O(nm)$, since divide-and-conquer is done on a sequence B of length m, and $O(nm)$ time is still needed to compute the scores. Note that this recurrence has two variables. Without the argument n and without factor n in the last termn, we would have the standard recurrence $T(m) = 2T(m/2) + O(m)$ with solution $T(m) = O(m \log m)$. Our n can be treated as a "constant" factor that appears in every recursion level, thus we can immediately conclude that $T(n, m) = O(mn \log m)$. Actually, a slightly more careful analysis yields an $O(nm)$ time *and* $O(n)$ space bound.

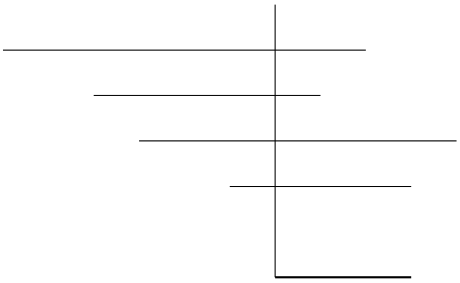

Figure 1: Illustration of the correctness proof for the Interval Partitioning greedy algorithm. Let $d = 5$. The current interval I (the thick line) has conflicts with at most 4 intervals that started earlier, because all these conflicting intervals contain I's start point, and at most 5 intervals can share a point. Hence we can put I in some of the 5 sets.

Further Instructions and Questions for Self-Study

- It is recommended to write down a correctness proof (just for yourself), in order to see why "magically" the longest paths in DAGs are obtained by simply replacing min with max in the Shortest Paths algorithm.

- It was stated that many dynamic programming can be seen as computing shortest paths in DAGs. Is it clear: What would be the mentioned "DAGs with highly regular structure", for example, in the Subset Sum problem, in the Sequence Comparison problem, etc.? What would be their nodes, edges, and edge lengths?

- Remark on the course structure: Now, in the remainder of the course, we go through the main techniques again and present a few examples that are more complex, in one way or the other.

- If you have to prioritize efforts, you may skip the almost linear Union-and-Find data structure (the last variant) and the space-efficient sequence comparison – they are clever and famous but also more "special" than the other examples of algorithms.