Algorithms. Lecture Notes 13

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 in such a way that it requires only $O(m)$ space: For computing the values $OPT(i, j)$ we need only the previous row 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 of 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 $a_n \ldots a_1$ and $b_m \ldots b_1$. The half sequence $b_1 \ldots b_k$ *must* be aligned to $a_1 \ldots a_j$, 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. 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 optimal position j just determined, and we make two recursive calls to solve the sub-instances.

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.

Problem: Clustering with Maximum Spacing

A **clustering** of a set of (data) points is simply a partitioning into disjoint subsets of points, called **clusters**. Some distance function is defined between the points. The distance of two point sets A and B is the minimum distance of two points $a \in A$ and $b \in B$. The **spacing** of a clustering is the minimum distance of two clusters (or equivalently, the minimum distance of any two points from different clusters).

Given: a set of n points in some geometric space, and an integer $k < n$. The pairwise distances of points are known, or they can be easily computed from their coordinates.

Goal: Construct a clustering with k clusters and maximum spacing.

Motivations:

Clustering in general has many applications in data reduction, pattern recognition, classification, data mining, and related fields. Coordinates of points are often numerical features of objects. Every cluster shall consist of "similar" objects, whereas objects in different clusters shall be "dissimilar". However, we have to make these intuitive notions precise. There exist myriads of meaningful quality measures for clusterings, and each one gives rise to an algorithmic problem: to find a clustering that optimizes this quality measure.

Many clustering problems can be formulated as graph problems, where the data objects are nodes. For instance, Graph Coloring can be seen as a clustering problem: The desired number k of clusters is given, and every cluster must fulfill some "internal" criterion, namely, not to contain any pair of dissimilar nodes. Spacing is an "external" quality measure. It demands that any two clusters be far away from each other, while nothing is explicitly said about the inner structure of clusters.

Clustering with Maximum Spacing via MST

Kruskal's MST algorithm has a nice application and interpretation in the field of clustering problems. Suppose that the nodes of our graph are data points, and the 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 $T_1, \dots, T_k$ as clusters.

We prove that the obtained spacing d is in fact optimal: Consider any partitioning into k clusters $U_1, \dots, U_k$. 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 two different "U clusters", hence the spacing of the other clustering can never exceed d.