Algorithms. Lecture Notes 12 and 13

Bipartite Matching

Here is one of the simplest but also most important examples of a reduction of another graph problem to Maximum Flow.

A **bipartite graph** is a graph $G = (X, Y, E)$ where the node set is split into two sets X, Y, and edges exist only between X and Y. (These are exactly the 2-colorable graphs.) A **matching** is a set of pairwise node-disjoint edges. The Bipartite Matching problem asks to find a matching of maximum size in a bipartite graph. Typical applications are job assignment problems: Nodes in X are jobs to be done, nodes in Y are workers or machines, and an edge means that the worker/machine is able to do the job. A matching is then a set of jobs that can be executed in parallel.

Bipartite Matching is reduced to Maximum Flow as follows: Add a source s and a sink t, insert edges from s to all nodes in X, and from all nodes in Y to t, orient the edges of E from X to Y, and set all edge capacities 1. Then the maximum matchings correspond exactly to the maximum flows with integer values (0 or 1) on the edges. (This equivalence needs a proof, however this is simple enough.) The time to solve the problem is therefore $O(mC) = O(mn)$.

One can also find maximum matchings in general graphs is polynomial time, but this is much more tricky.

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:

Thay 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 consider them in this order. We always put the current interval x into the subset X_i with the smallest possible index i, such that x does not intersect any other interval in X_i.

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 achieves this bound, hence it is optimal. Specifically, let d be the maximum number of intervals that contain the same point. Since d such intervals must be put into d distinct subsets, any solution needs at least d subsets. Our greedy algorithm uses only d subsets: Whenever a new interval x is considered, at most $d-1$ earlier intervals can intersect x, because these intervals must contain the start point of x. Hence we can always put x is some of the first d subsets.

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.

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 needs also $O(nm)$ space, which can be prohibitive for applications in molecular biology where n, m are huge numbers. We may implement the dynamic programming algorithm so that it needs 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 $OPT(i, j)$ 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 $OPT(i, j)$, but then we are back to an $O(nm)$ space algorithm.

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. 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 algorithm in more detail.

Let be $k \approx m/2$. 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 $a_1 \dots a_j$, for some yet unknown j, and the other half of B to the rest of A. Then, 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 j where the sum is minimized. We come from the left and from the right, the edit distance does not change if sequences are reversed. Clearly, the minimum sum is found in $O(n)$ time and space. Finally we divide B at position k, and A at that position j we have just determined, and we make two recursive calls.

We never need more than $O(n)$ space at the same time. The time complexity is given by the recurrence $T(n, m) = 2T(n, m/2) + O(nm)$, since divide-and-conquer is done on sequence B of length m, and time $O(nm)$ is still needed to compute the scores. Note that the 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 $T(n, m) = O(mn \log m)$.

We have sketched an alignment algorithm that needs only a $\log m$ factor more time than the basic dynamic programming algorithm but has the important advantage to work in small space. This is a good deal. Actually, a somewhat more careful analysis yields an $O(nm)$ time *and* $O(n)$ space bound, see section 6.7 of the course book.

Problem: Closest Points

Given: a set of n points in the plane (given as Cartesian coordinates (x_i, y_i)).

Goal: Find a pair of points with minimum distance.

Motivations:

Some approaches to hierarchical clustering of data take the two closest data points and combine them to a cluster by replacing these two points by their midpoint, and this step is repeated until one cluster remains.

Divide-and-Conquer in Geometry: Closest Points

Fast geometric calculations are needed in computer graphics, computer-aided design, robotics, planning (transport optimization, facility location), chemistry (modelling molecules and their dynamics), for extracting information from geographic databases, etc. The amount of data can be huge (e.g., elements of a picture), such that efficient algorithms make a difference.

Divide-and-conquer is suitable for various geometric problems, because instances can be divided in a natural way. (However, the conquer phase is usually

less trivial). To give at least an impression, we discuss another geometric problem example: finding a pair of closest points among n given points in the plane.

An obvious algorithm would compute all pairwise distances and determine the minimum in $O(n^2)$ time. Instead, we aim at a divide-and-conquer algorithm satisfying the recurrence $T(n) = 2T(n/2) + O(n)$, which would have the time complexity $T(n) = O(n \log n)$.

It is natural to divide the set by a straight line. To make the calculation details simple, we first sort the points by their x-coordinates, and then halve the set by a vertical separator line. More formally, we take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively, in the two sets. Recall that sorting takes $O(n \log n)$ time, which does not destroy the desired time bound. Wouldn't it be enough to compute the median in $O(n)$ time, without sorting? Yes, it is enough for the first step, but we will recursively split the point set further, on the lower recursion levels. Sorting the points once in the beginning is simpler and cheaper (in terms of the hidden constant factors) than median computations on every recursion level.

Then, of course, we compute the closest pairs in both subsets recursively. Let d be the minimum of the two minimum distances. The tricky part is to combine the partial solutions. The global solution could be the best of the two closest pairs from the two subsets, but there could also exist a pair of points with distance smaller than d, having one point in each subset. The candidates for such pairs of points are in a stripe of breadth d on both sides of the separating line. Moreover, each point has only constantly many partners (at distance smaller than d) on the other side, hence $O(n)$ such pairs of close points must be considered. These pairs can be identified in $O(n)$ time, if all points are already sorted by their y-coordinates as well. With careful implementation, all steps in the conquer phase run in $O(n)$ time as desired.