Advanced Algorithms Course. Lecture Notes. Part 10

Dynamic Programming on Trees

Problems that are NP-complete in general graphs can become rather easy in special graph classes. In practice, the input to a graph problem often happens to be a tree. (For example, many real networks are hierarchical structures.) Most problems on trees can be solved by bottom-up dynamic programming. We illustrate the principle by the Weighted Vertex Cover problem which is also equivalent to the Weighted Independent Set problem.

In the given tree we distinguish an arbitrary node r as the root. All edges are oriented away from the root. This defines a directed tree T. For every node, let T_v denote the subtree with root v, consisting of v and all nodes reachable from v on directed paths. We denote the weight of a node v by $w(v)$. For every node v we define $OPT(v, 1)$ and $OPT(v, 0)$ as the minimum weight of a vertex cover S in T_v with $v \in S$ and $v \notin S$, respectively. What we want is the minimum of $OPT(r, 1)$ and $OPT(r, 0)$.

These values can be computed as follows. If v is a leaf, we immediately have $OPT(v, 1) = w(v)$ and $OPT(v, 0) = 0$. Now let v be an inner node, and $v_1, \ldots, v_d$ the children of v. If v is not in the vertex cover, we have to take all children, hence

$$OPT(v, 0) = \sum_{i=1}^{d} OPT(v_i, 1).$$

. If v is in the vertex cover, we can independently decide for any child to take it or not, and the minimum value is optimal. Hence we have

$$OPT(v, 1) = w(v) + \sum_{i=1}^{d} \min(OPT(v_i, 1), OPT(v_i, 0)).$$

That's all! The running time is $O(n)$, since every node is involved in only constantly many calculations for its parent node.

An actual solution can be obtained by backtracing in $O(n)$ time. The only difference to usual dynamic programming is that the backtracing procedure does not only follow one computation "path" but works on the whole tree, from the root to all leaves.

It is also recommended to reflect upon the question why our OPT function needed the second (Boolean) argument, and dynamic programming with some "OPT(v)" would not work for this problem.

As a side remark, the unweighted Vertex Cover problem can even be solved by a greedy algorithm on trees. But for the weighted problem we do need dynamic programming.

Randomized Algorithms continued

Median Finding and Selection

The so-called Selection problem asks to find the element of rank k in an unsorted set S of n distinct numbers. The rank is the position that the element would have if S were sorted in ascending oder. A simpler formulation is: Find the k-th smallest element in S. The element with rank $\lfloor n/2 \rfloor$ is called the median.

Median finding and Selection have nice applications in geometry and in the analysis of statistical data. In addition to these motivations, recall the 1.5-approximation algorithn for Load Balancing. We had sorted the jobs by their lengths. A closer look reveals that it is enough to separate the m longest jobs from the shorter jobs, since neither the algorithm nor the analysis uses any sorting within these two subsets of jobs.

To solve the Selection problem we may first sort S in $O(n \log n)$ time, which makes the problem trivial. But instead we can avoid sorting and solve Selection directly in $O(n)$ time. There exists a deterministic divide-and-conquer algorithm for Selection, but it is a bit complicated and, more importantly, the hidden constant in $O(n)$ is rather large. It is much more advisable to apply a simple randomized algorithm like the following.

Choose an element $s \in S$ called the splitter. Compare all elements to s, in $O(n)$ time. Now we know the rank r of s. If $r > k$ then throw out s and all elements larger than s. If $r < k$ then throw out s and all elements smaller than s, and set $k := k - r$. If $r = k$ then return s. Repeat this

procedure recursively. The correctness of this algorithm should be obvious.

The only yet unspecified step is the choice of the splitter. Let us choose a splitter at random. Intuitively, this is a good algorithm because a random element will usually split the set in two reasonably well balanced subsets, hence the number of elements to consider should decrease exponentially.

For a rigorous analysis of the expected time we simply introduce a "cut-off point" that defines whether a splitting is well balanced or not. More precisely, we call an element "central" in a set, if this element is smaller and larger, respectively, than at least $1/4$ of the elements.

We say that the algorithm is "in phase j" if the number of remaining elements is between $n(3/4)^{j+1}$ and $n(3/4)^j$. Clearly, our random splitter is central with probability $1/2$. It follows immediately that the expected number of splitters needed in every phase j is $2 = O(1)$, hence the expected time of every phase is linear in the number of elements. Since $\sum_j n(3/4)^j$ is a geometric series converging to some value $O(n)$, the total expected number of comparisons is still $O(n)$, with some moderate hidden constant whose analysis we omit here.

A Quick but Rigorous Analysis of Quicksort

The basic version of the famous Quicksort algorithm (which we do not repeat here) works with a random splitter in every recursion step. For the sake of a simple analysis we slightly modify the algorithm, however we keep it close to the original Quicksort: We check after comparison to all other elements whether the random splitter is central, and if not, we discard it altogether and pick a new splitter. Of course, this is a certain waste of time. Hence the original Quicksort performs no worse than this "slow Quicksort".

We say that a subproblem is "of type j" if the number of elements is between $n(3/4)^{j+1}$ and $n(3/4)^j$. We find a central splitter after an expected number of only 2 attempts. Thus, the expected time spent on any subproblem of type j is $O(n(3/4)^j)$. Moreover, since we accepted only central splitters, we can easily see that all subproblems of type j are pairwise disjoint, i.e., they deal with disjoint subsets of the entire set. Hence at most $(4/3)^{j+1}$ subproblems of type j can exist during the execution of the algorithm. THerefore, by linearity of expectation, the expected time spent on all subproblems of type j is $O(n)$. Since only $O(\log n)$ types exist, the total expected time is $O(n \log n)$.