Advanced Algorithms Course.
Lecture Notes. Part 10

Finding a Path by Color Coding and Dynamic Programming on Subsets

This is a beautiful combination of ideas from randomized and FPT algorithms and a variant of dynmaic programming..

We consider the following k-path problem. Given a graph G and an integer k, find a path of exactly k nodes that does not cross itself, i.e., each of the k nodes shall be visited only once.

What is the motivation? Why should one be interested in finding such a k-path? Here is one real application from computational biology: Molecules like proteins, DNA, RNA are long sequences. Under some experimental conditions one cannot observe these sequences directly but obtain only information about pairs of short molecules that could possibly be neighbors in a sequence. The reconstruction of sequences from such local information leads to the problem of finding simple paths, of at least some prescribed length, in the graph of possible neighborhood relationships.

At first glance the k-path problem looks simple. We may start from some node and try all possible paths of length k. But if we do that naively by breadth-first-search, we need $O(n^k)$ time, showing that the problem is in XP. In fact, finding a k-path is easy if the graph has diameter at least k. Then it suffices to compute shortest paths between all pairs of nodes, which is possible in polynomial time. Since some of these paths has the desired length, and no nodes appear repeatedly on the path, we get a solution. But ironically, a smaller diameter makes the problem hard. Now we devise an algorithm that works also in this case.

The idea of **color coding** is to use k colors and assign one color to each node, randomly and independently. (Do not confuse it with the graph coloring problem where adjacent nodes must get different colors. This restriction is not applied here.) Then, we only search for a k-path where all k colors appear. The point is that this subproblem can be solved in $O(2^k n^2)$ time! Now we show how this is done, and how this result is used to solve the original problem. (A simpler but also slower way is to try all permutations of the colors, which takes $O(k!n^2)$ time.)

Let $c(v)$ denote the color of node v. We do **dynamic programming on subsets**, specifically, on all 2^k subsets C of the colors. For every set C and every node v we compute a value $p(C,v)$ which is 1 if there exists a path of $|C|$ nodes that ends in v and contains exactly the colors from C, otherwise we define $p(C,v) := 0$. For $|C| = 1$, we obviously have $p(C,v) = 1$ if and only if $C = \{c(v)\}$. Next suppose that we know all $p(C,v)$ with $|C| = i$. Then all $p(C',v)$ with $|C'| = i+1$ are obtained as follows. We have $p(C',v) = 1$ if and only if $p(C' \setminus \{c(v)\}, u) = 1$ for some node u adjacent to v. The time bound is easy to see.

The algorithm succeeds if some path with k nodes (if it exists) actually carries all k colors. Let P be a fixed path of k nodes. It can be colored in k^k different ways, and $k!$ colorings are good. Hence the success probability in every attempt is $k!/k^k$, which is essentially $1/e^k$ due to Stirling's formula. It follows that we need $O(e^k)$ iterations to find a k-path with high probability, and if we do not detect some, we can report that no k-path exists, with an arbitrarily small error probability. The time is $O((2e)^k n^2) = O^*((2e)^k)$ in total.

Dynamic Programming on Trees

Problems that are NP-complete in general graphs can become rather easy in special graph classes. Often it happens in practice that the input to a graph problem is a tree. (For example, many networks are hierarchically structured.) 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 via directed edges. We denote the weight of a node v by $w(v)$. For every v we define $OPT(v,1)$ and $OPT(v,0)$ as the minimum weight of a vertex cover in T_v with v and without v, respectively. What we want is the minimum of $OPT(r,1)$ and $OPT(r,0)$.

These values are 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. It is recommended to reflect upon the question why our OPT function needed the second (Boolean) argument.

As a side remark, the unweighted Vertex Cover problem can even be solved by a greedy algorithm on trees.