Advanced Algorithms Course. Lecture Notes. Part 3

Disjoint Paths and Routing

Given a directed graph with m edges, and k node pairs (s_i, t_i), we wish to find directed paths from s_i to t_i for as many as possible indices i, that do not share any edges. We also call such paths edge-disjoint. This is a fundamental problem in routing in networks. Imagine that we want to send goods, information, etc., from source nodes to destination nodes along available directed paths, without unreasonable congestion. In general we cannot send everything simultaneously, but we may try and maximize the number of served requests.

The problem is NP-complete (which we do not prove here), but we present an algorithm with approximation ratio $O(\sqrt{m})$. The square root is a "small" function, still the quality of the solution deteriorates with growing network size. This result seems to be poor, but it is the best possible guarantee one can achieve in polynomial time, and still better than no guarantee at all.

As often, the idea of a greedy algorithm is simple: Short paths should minimize the chances of conflicts with other paths, and the shortest paths can be computed efficiently. Therefore, the proposed algorithm just chooses a shortest path that connects some yet unconnected pair and adds it to the solution, as long as possible. After every step we delete the edges of the path used, in order to avoid collisions with paths chosen later.

However, the idea is not as powerful as one might hope: In each step there could exist many short paths to choose from, and we may easily miss a good one, since we only take length as selection criterion. But at least we can prove the $O(\sqrt{m})$ factor, as follows. Let I^* and I denote the set of indices i of the pairs (s_i, t_i) connected by the optimal and the greedy solution, respectively. Let P_i^* and P_i denote the selected paths for index i. The analysis works with case a distinction regarding the length: We call a

path with at least $\sqrt{m}$ edges long, and other paths are called short. Let I_s^* and I_s be the set of indices i of the pairs (s_i, t_i) connected by the short paths in I^* and I, respectively.

Since only m edges exist, I^* can have at most $\sqrt{m}$ long paths. Consider any index i where P_i^* is short, but (s_i, t_i) is not even connected in I. (This is the worst that can happen to a pair, hence our worst-case analysis focusses on this case.) The reason why the greedy algorithm has not chosen P_i^* must be that some edge $e \in P_i^*$ is in some P_j chosen earlier. We say that e "blocks" P_i^*. We have $|P_j| \le |P_i^*| \le \sqrt{m}$. Every edge in P_j can block at most one path of I^*. Hence P_j blocks at most $\sqrt{m}$ paths of I^*. The number of such particularly bad indices i is therefore bounded by $|I_s^* \setminus I| \le |I_s|\sqrt{m}$. Finally some simple steps prove the claimed approximation ratio:
$|I^*| \le |I^* \setminus I_s^*| + |I| + |I_s^* \setminus I| \le \sqrt{m} + |I| + |I_s|\sqrt{m} \le (2\sqrt{m} + 1)|I|$.

An Approximation Scheme for Knapsack

So far we have seen some approximation algorithms whose approximation ratio on an instance is fixed, either an absolute constant or depending on the input size. But often we may be willing to spend more computation time to get a better solution, i.e., closer to the optimum. In other words, we may trade time for quality. A **polynomial-time approximation scheme (PTAS)** is an algorithm where the user can freely decide on some accuracy parameter ϵ and gets a solution within a factor $1 + \epsilon$ or $1 - \epsilon$ of optimum, and within a time bound that is polynomial for every fixed ϵ but grows as ϵ decreases. The actual choice of ϵ may then depend on the demands and resources. A nice example is the following Knapsack algorithm.

In the Knapsack problem, a knapsack of capacity W is given, as well as n items with weights w_i and values v_i (all integer). The problem is to find a subset S of items with $\sum_{i \in S} w_i \le W$ (so that S fits in the knapsack) and maximum value $\sum_{i \in S} v_i$. Define $v* := \max v_i$.

You may already know that Knapsack is NP-complete but can be solved by some dynamic programming algorithm. Its time bound $O(nW)$ is polynomial in the numerical value W, but not in the input size n, therefore we call it pseudopolynomial. (A truly polynomial algorithm for an NP-complete problem cannot exist, unless P=NP.) However, for our approximation scheme we need another dynamic programming algorithm that differs from the most natural one, because we need a time bound in terms of values rather than weights. (This point will become more apparent later on.) Here it comes:

Define $OPT(i, V)$ to be the minimum (necessary) capacity of a knapsack that contains a subset of the first i items, of total value at least V. We can compute $OPT(i, V)$ using the OPT values for smaller arguments, as follows. If $V > \sum_{j=1}^{i-1} v_j$ then, obviously, we *must* add item i to reach V. Thus we have $OPT(i, V) = w_i + OPT(i-1, V-v_i)$ in this case. If $V \leq \sum_{j=1}^{i-1} v_j$ then item i may be added or not, leading to

$$OPT(i, V) = \min(OPT(i-1, V), w_i + OPT(i-1, \max(V - v_i, 0))).$$

Since $i \leq n$ and $V \leq nv^*$, the time is bounded by $O(n^2 v^*)$. As usual in dynamic programming, backtracing can reconstruct an actual solution from the OPT values.

Now the idea of the approximation scheme is: If v^* is small, we can afford an optimal solution, as the time bound is small. If v^* is large, we round the values to multiples of some number and solve approximately the given problem. The point is that we can divide all the rounded values by the common factor without changing the solution sets, which gives us again a small problem. In the following we work out this idea precisely. We do not specify what "small" and large" means in the above sketch, instead, some free parameter $b > 1$ controls the problem size.

First compute new values v_i' as follows: Divide v_i by some fixed b and round up to the next integer: $v_i' = \lceil v_i/b \rceil$. Then run the dynamic programming algorithm for the new values v_i' rather than v_i.

Let us compare the solution S found by this algorithm, and the optimal solution S^*. Since we have not changed the weights of elements, S^* still fits in the knapsack despite the new values. Since S is optimal for the new values, clearly $\sum_{i \in S} v_i' \geq \sum_{i \in S^*} v_i'$. Now one can easily see: $\sum_{i \in S^*} v_i/b \leq \sum_{i \in S^*} v_i' \leq \sum_{i \in S} v_i' \leq \sum_{i \in S}(v_i/b + 1) \leq n + \sum_{i \in S} v_i/b$. This shows $\sum_{i \in S^*} v_i \leq nb + \sum_{i \in S} v_i$, in words, the optimal total value is larger than the achieved value by at most an additional amount nb.

Depending on the maximum value v^* we choose a suitable b. By chosing $b := \epsilon v^*/n$, the above inequality becomes $\sum_{i \in S^*} v_i \leq \epsilon v^* + \sum_{i \in S} v_i$. Since trivially $\sum_{i \in S^*} v_i \geq v^*$, this becomes $\sum_{i \in S^*} v_i \leq \epsilon \sum_{i \in S^*} v_i + \sum_{i \in S} v_i$, hence $(1-\epsilon) \sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i$. In words: We achieve at least a $1-\epsilon$ fraction of the optimal value. The time is $O(n^2 v^*/b) = O(n^3/\epsilon)$. Thus we can compute a solution with at least $1-\epsilon$ times the optimum value in $O(n^3/\epsilon)$ time.

For any fixed accuracy ϵ this time bound is polynomial in n (not only pseudopolynomial as the exact dynamic programming algorithm). However, the smaller ϵ we want, the more time we have to invest.

The presented approximation scheme is even an FPTAS, which is stronger than a PTAS. Here is the definition: A **fully polynomial-time approximation scheme (FPTAS)** is an algorithm that takes an additional input parameter ϵ and computes a solution that has at least $1 - \epsilon$ times the optimum value (for a maximization problem), or at most $1+\epsilon$ times the optimum value (for a minimization problem), and runs in a time that is polynomial in n and $1/\epsilon$.