Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 25
Wednesday, November 23, 2005

Problem Set 11 Solutions

Problem 1.

We will remove vertices from a given graph G of treewidth k in a good elimination ordering, and we can assume w.l.o.g. that G is connected. Some cliques in a current graph will be *active cliques*. For each active clique C and for each subset S of vertices in C, we store information if there exists a vertex cover of the original graph G that chooses exactly vertices in S out of the vertices in C. This is possible if and only if for each edge (in G) between a pair of vertices in C, at least one of them is chosen. Moreover, if this is possible, we store as well a specific subset of some removed vertices (details follow). Note that the size of any clique during the elimination is bounded by $k+1$ (actually active cliques are always of size at most k).

The algorithm is following. In the beginning there is no active clique. When we remove a vertex v from the graph, we connect all its neighbors, and make them an active clique C. Note that if v is a member of some active clique K, K disappears from the graph, and all the other members of K become members of C. For each subset S of vertices in C, we make sure that this subset is "possible" in the aforementioned way, and if so, we consider two possibilities, either we choose v or not, and the second case is considered only if under this choice all edges between v and C that are present in G are covered. For each case, we collect subsets in all active cliques K_i (where K_i is the i-th active clique to which v belongs right before the removal) that are consistent with our choice, possibly adding vertex v, if necessary, and we store for S in active clique C the smaller of them. Note that the sets of which subsets active cliques store at a given time have empty intersection. This is a consequence of the fact that a vertex that has been already removed can be present in subsets of exactly one active clique at any time.

Eventually, we have only one vertex left, which constitutes an active clique on its own, and we simply choose a better of two possibilities.

To show that the runtime of the algorithm is polynomial, it is enough to say that a single removal of a vertex takes polynomial time. This is true, since the number of active cliques is less than the number of vertices, and we consider only a constant (for a given k) number of subsets.

The algorithm returns an optimal solution, since it can be proven by induction, that for any choice of vertices of an active clique C, we compute the smallest vertex cover of the part separated from the rest of the graph by vertices in C. Eventually, we have two possibilities for the last variable, and one of them must be the optimal vertex cover.

Problem 2.

(a) The proof goes almost like the one that we have seen in class for the k-server problem on a line. We use the same potential function

$$\Phi = kM + \sum_{i<j} d(s_i, s_j).$$

First, when OPT's server moves d, it increases the cost of the optimum matching between OPT's and DC's by at most d, and therefore, the potential increases by at most kd.

On the other hand, we can show that when DC moves d, the total potential decreases by at least d. DC may move a few servers simultaneously, but some of them may be blocked and stop during the procedure. Consider a phase when q servers move, each of them l. If $q = 1$, only one server S moves towards a request, and OPT's server T is already at the request. There is an optimum matching in which S is matched to T, so the cost of the matching decreases by l, and since the sum of distanced increases by $(k-1)l$, the total change Δ of the potential is

$$\Delta = -kl + (k-1)l = -l.$$

When $q > 1$ servers move, then in the optimal matching, at least one of them must me matched to either OPT's server at the request or OPT's server behind the server at the request. This implies that the cost of matching may grow by at most

$$-l + (q-1)l = (q-2)l.$$

The sum of distances between moving servers will decrease by

$$\binom{q}{2} 2l = q(q-1)l,$$

and the sum of distances between them and any other DC's server changes by

$$-(q-1)l + l = (q-2)l,$$

because exactly $q-1$ of them move closer and one farther. Eventually, the total change Δ can be bounded in the following way:

$$\Delta \leq k(q-2)l - q(q-1)l - (k-q)(q-2)l = -ql.$$

We conclude, as for k-server on a line, that the algorithm is k-competitive.

(b) We create a star tree of n leafs, where n is the total number of pages, and set the length of each edge to $1/2$. Each leaf corresponds to a one page, and each server corresponds to a single memory slot. When a server reaches a leaf, it evicts a page it holds, and loads the page represented by the leaf. In the beginning, each server is at the leaf representing the page it holds. If we model paging in this way, an exchange of a single page in a slot costs at least 1, and taking an optimal solution we can interpret a page replacement as a move of length 1 of a server from one leaf to another. Therefore, the cost of the optimal solution m to this k-server problem is exactly the same as the cost of the optimal solution to the paging problem.

The sum of lengths of all moves of servers in our solution will be therefore at most

$$km + \binom{k}{2},$$

where $\binom{k}{2}$ is the initial potential, and this is also an upper-bound on the number of page faults.

(c) Note first that each server can only be either in a leaf or in the internal vertex. If having a choice (i.e. a few servers in the internal vertex), we choose the one that already holds a page that we request, or if there is no such server, a random server, then the algorithm that we get is the marking algorithm. A page is marked if and only if a server is at the node that corresponds to this page. It is easy to see that a state of pages, and eviction works exactly like in the marking algorithm.

(d) As before we create a star tree, and this time we set $w_i/2$ as the length of the edge between the internal node and the node corresponding to a page i, where w_i is the cost of a miss of this page. For every solution to the paging problem we can create a scheme of moves of servers which cost will differ from the paging cost only in this that we may have to pay a half of price for pages that we have in the beginning, and possibly only a half of price for pages that we have in the end. On the other hand, for every solution to k-server in this star tree, converting it to page evictions, we may have to pay additionally a half of prices of pages that we have in the end, and not have to pay a half of price of the initial page. Overall, these differences for a single server can be bounded by the greatest cost $\hat{w}$ of a page miss, and therefore, using k-server DC algorithm we get as well a k-competitive algorithm.

Problem 3.

(a) If the algorithm makes a mistake, faculty of at least a half of the total weight gave a wrong answer. And since exactly one half of their weight is removed, which is at least one fourth of the total weight, the total weight decreases by a factor of 4/3.

As the initial total weight of faculty is n, and with each wrong answer chosen it decreases by a factor of 4/3, the final total weight W can be bounded

$$W \leq n\left(\frac{3}{4}\right)^k,$$

where k is the number of incorrect answers of the algorithm.

(b) The final total weight W is not less than the final weight of the wisest faculty member, hence

$$W \geq \left(\frac{1}{2}\right)^m .$$

(c) We combine the above two inequalities:

$$\begin{aligned}
\left(\frac{1}{2}\right)^m &\leq n\left(\frac{3}{4}\right)^k, \\
-m &\leq \lg n - (2 - \lg 3)k, \\
k &\leq \frac{1}{2 - \lg 3}(m + \lg n), \\
k &\leq 2.41(m + \lg n).
\end{aligned}$$

(d) The multiplicative change in the total weight is

$$\beta F + (1 - F) = 1 - (1 - \beta)F,$$

and can be upper-bounded by

$$e^{-(1-\beta)F}.$$

Note that F is the probability of an incorrect answer.

(e) The final total weight W can be lower bounded by the final weight of the wisest faculty member:

$$W \geq \beta^m.$$

Let F_i be the fraction of the faculty weight with a wrong answer in our algorithm when we want to determine the answer in the i-th question. The final total weight is exactly

$$W = n \prod_i (1 - (1 - \beta)F_i),$$

and therefore, can be upper bounded by

$$W \leq n e^{-(1-\beta)\sum_i F_i}.$$

Note that $\sum_i F_i$ is the expected number of wrong answers in our algorithm. Denote it by k. Combining the two bounds we get

$$\begin{aligned}
\beta^m &\leq n e^{-(1-\beta)k}, \\
-m \ln(1/\beta) &\leq \ln n - (1 - \beta)k, \\
k &\leq \frac{m \ln(1/\beta) + \ln n}{1 - \beta}.
\end{aligned}$$

(f) We will use (without notice) the three following inequalities. First, for $x > 1$,

$$\ln(1+x) \le x,$$

second, for $a, b \ge 0$,

$$\sqrt{a} + \sqrt{b} \ge \sqrt{a+b},$$

and third, for $0 \le \epsilon \le 3/4$,

$$\frac{1}{1-\epsilon} \le 1 + 4\epsilon.$$

We can easily get rid of the logarithm in the upper bound on the expected number of wrong answers.

$$k \le \frac{m\ln(1/\beta) + \ln n}{1-\beta} \le \frac{m(1/\beta - 1) + \ln n}{1-\beta} = \frac{m}{\beta} + \frac{\ln n}{1-\beta}.$$

We will consider two cases. The first of them occurs when $m \ge \ln n$. Then we take

$$\beta = 1 - \sqrt{\frac{\ln n}{m + \ln n}},$$

and get

$$\sqrt{\frac{\ln n}{m + \ln}} \le \sqrt{\frac{1}{2}} < \frac{3}{4},$$

$$\frac{m}{\beta} = \frac{m}{1 - \sqrt{\frac{\ln n}{m+\ln n}}} \le m\left(1 + 4\sqrt{\frac{\ln n}{m + \ln n}}\right) \le m + 4\sqrt{m\ln n},$$

and

$$\frac{\ln n}{1-\beta} = \frac{\ln n}{\sqrt{\frac{\ln n}{m+\ln n}}} \le \sqrt{m + \ln n} \cdot \sqrt{\ln n} \le \sqrt{m \ln n} + \ln n.$$

The second case is $m \le \ln n$, and taking

$$\beta = \sqrt{\frac{m}{m + \ln n}} \le \sqrt{\frac{1}{2}} < \frac{3}{4},$$

we get

$$\frac{m}{\beta} = \sqrt{m + \ln n} \cdot \sqrt{m} \le m + \sqrt{m \ln n},$$

and

$$\frac{\ln n}{1-\beta} \le (1 + 4\beta)\ln n = \ln n + 4\sqrt{m\ln n} \cdot \sqrt{\frac{\ln n}{m + \ln n}} \le \ln n + 4\sqrt{m \ln n}.$$

Eventually, in both cases

$$k \le \frac{m}{\beta} + \frac{\ln n}{1-\beta} \le m + \ln n + 5\sqrt{m \ln n}.$$

Problem 4.

(a) Let (x_1, y_1) and (x_2, y_2) be the coordinates of the opposite vertices of a rectangle R. Note that it belongs to a query rectangle $[x : x'] \times [y : y']$ if and only if the point (x_1, x_2, y_1, y_2) in the 4-dimensional space belongs to the box $[x : x'] \times [x : x'] \times [y : y'] \times [y : y']$.

We have seen in class how to solve the range query problem for 2 dimensions. This result is easily generalizable to d dimensions. Namely, a data structure of size $O(n \log^{d-1} n)$ can answer queries in time $O(\log^d n)$. It suffices to show this to solve the task.

For one dimension, we can construct a balanced binary search tree, and in each node store the greatest and smallest element (i.e. the range of elements) in the subtree rooted at this node. The size of the tree is $O(n)$, and in time $O(\log n)$ we can determine $O(\log n)$ intervals corresponding to nodes of the tree, and output solutions in time linear in their number.

Suppose we have the data structure for $d-1$ dimensions, where $d \geq 2$. We create a balanced binary tree for the d-th coordinate of points, and in each node for all points that belong to the interval that is represented by it, we create a $(d-1)$-dimensional structure for the other dimensions. There are $O(\log n)$ levels in the tree (i.e. the depth of the tree is $O(\log n)$), and on each level a given point appears at most once. The total size of $(d-1)$-dimensional structures on a one level can be bounded by

$$\sum_i O(k_i \log^{d-2} k_i) \leq \sum_i O(k_i \log^{d-2} n) \leq n \log^{d-2} n,$$

where k_i is the number of points in the i-th node on the level, and

$$\sum_i k_i \leq n.$$

The total size of the structure is therefore $O(n \log^{d-1} n)$. To answer a query, we determine in time $O(\log n)$ at most $O(\log n)$ nodes that contain points in the required interval, and we continue searching in the $d-1$-dimensional structures in these nodes. The total query times is therefore $O(\log^d n)$.

(b) We reduce this case to the previous one. For each polygon P we determine the smallest axis-parallel rectangle R containing it, and it is obvious that for every query rectangle Q it holds that $R \in Q$ iff $P \in Q$.