6.856 — Randomized Algorithms

David Karger

Handout #4, September 21, 2002 — Homework 1 Solutions

M. R. refers to this text:
Motwani, Rajeez, and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge: Cambridge University Press, 1995.

Problem 1 MR 1.8.

(a) The min-cut algorithm given in class works because at each step it is very unlikely (probability $\leq 2/n$) that we contract an edge across the min-cut. We will now construct a graph which does not have this property for s-t-min-cuts. Our graph consists of s, t and $n-2$ other nodes. There are edges from s to all nodes except t, and from t to all nodes except s. Moreover, the $n-2$ nodes besides s and t are connected in a single cycle (see figure).

Any s-t-min-cut in this graph must have size at least $n-2$, because there are $n-2$ edge disjoint paths connecting s and t (through each of the $n-2$ vertices in the cycle). Indeed, the are two such min-cuts: one having s alone on one side of the cut, and one having t alone on one side. And these are the only two s-t-min-cuts, since in every cut that splits the $n-2$ cycle vertices, the cycle will cross the cut, increasing the cut size by at least 2 to a total of n or more edges.

Let us now compute the probability that the contraction algorithm will produce the cut where s is alone on one side. Initially, the graph contains $m = 3(n-2)$ edges. We will fail to produce the desired cut if we contract any of the $n-2$ edges incident to s. So the probability p_i that we fail in the i-th iteration, given that we have not failed in any of the previous iterations is:

$$\frac{n-2}{\#\text{edges present in iteration } i} \geq \frac{n-2}{3(n-2)} = \frac{1}{3}$$

We succeed to find the min-cut if we do not contract one of these edges in any of the $n-2$ iterations of the algorithm. This therefore happens with probability $(2/3)^{n-2}$.

By symmetry, this is equal to the probability that we arrive at the s-t-min-cut that has t alone on one side. In summary, the probability that we obtain one of the two s-t-min-cuts is at most $2 \cdot (2/3)^{n-2}$, which is exponentially small.

(This proof crucially used that the Handshaking Lemma is not satisfied anymore: although the minimum s-t-min-cuts have size $n-2$, all nodes besides s and t have only degree 4. We also had to use the fact that there are only 2 min-cuts. Had this number been exponential (see part (b)), then even though it is likely that we will miss any particular min-cut, we will likely end up with *some* min-cut.)

(b) We will give two bounds, one in terms of the number of vertices n, and one in terms of the number of edges m.

For the bound in terms of n, consider the following graph. It contains s, t and $n-2$ other nodes. There are $2(n-2)$ edges in the graph: s is connected to every node besides t, and t is connected to every node besides s.

In this graph, every assignment of the $n-2$ "middle" vertices to the two sides of the cut leads to a different s-t-min-cut of size $n-2$. So the number of s-t-min-cuts is equal to the number of subsets of an $n-2$ element set, which is 2^{n-2}. But clearly every assignment of the $n-2$ vertices besides s and t to the two sides of the cut completely defines a cut, so there also can be at most 2^{n-2} min-cuts (the number of subsets of the $n-2$ vertices).

For the bound in terms of m, consider a graph where $n-2$ is even, and there are $(n-2)/2$ disjoint paths of three edges from s to t. The number of edges in this graph is $m = 3(n-2)/2$. Clearly, the size of the s-t-min-cut is $(n-2)/2$, and every choice of one edge per path corresponds to one min-cut, for a total of $3^{(n-2)/2} = 3^{m/3}$ s-t-min-cuts.

We will now see that this is also an upper bound. Suppose we have a graph with a s-t-min-cut size of k. That means that there are k edge-disjoint paths from s to t. Fix one such set of paths $p_1, p_2, \ldots, p_k$. To disconnect s from t, every s-t-min-cut must include one edge from each path p_i. In fact, it must contain *exactly* one edge per path, since all min-cuts have size k. If the number of edges in path p_i is ℓ_i, the upper bound on the number of s-t-min-cuts therefore is $\ell_1 \cdot \ell_2 \cdots \ell_k$. So we can obtain an upper bound by bounding the quantity

$$U := \max_{1 \le k \le m} \max_{1 \le \ell_1 + \ell_2 + \cdots + \ell_k \le m} \ell_1 \cdot \ell_2 \cdots \ell_k.$$

Because a product of k numbers with a constant sum m gets maximized when all numbers have the same value ℓ, we have

$$U \le \max_{1 \le \ell \le m} \ell^{m/\ell}$$

This quantity becomes maximal if $\ell^{1/\ell}$ becomes maximal. Taking the derivative, and setting equal to zero, shows that the quantity becomes maximal for $\ell = e \approx 2.71828$. But since ℓ is constrained to be integral, the maximum is attained for either $\ell = 2$ or $\ell = 3$. A little calculation shows that $2^{1/2} < 3^{1/3}$, and therefore we have as an upper bound for the number of s-t-min-cuts $U \le 3^{m/3}$.

Problem 2 MR 2.3.

(a) We prove by induction on h the following claim. For a height h tree, if the adversary chooses leaf values adaptively as the (deterministic) algorithm queries them, it can keep the value of the tree indeterminate until all leaves have been queried. Rephrasing: until the last leaf is queried, there is a setting of the unqueried bits that makes the tree value 0, and a different setting that makes the tree value 1. Note I am making no assumptions about the deterministic algorithm evaluating one tree before the others.

The base case (height 1 tree, i.e. just one node) is trivial. For the inductive step, we implement the height-$(h+1)$ adversary on top of three height-h adversaries for the three trees T_1,T_2, and T_3. To do so, consider any algorithm leaf query. It is a query to a leaf in the one of the three subtrees T_i. We assume that the sub-adversary for that T_i provide an answer which keeps the value of that subtree indeterminate. This continues until the last unqueried leaf in one of the subtrees is queried. At this point, that subtree's sub-adversary commits to a value, say 0. The remaining adversaries continue to remain uncommitted. Eventually a second subtree will have its last unqueried leaf queried. This sub-adversary returns a value that commits that subtree to value 1. Thus the overall tree value is determined by the final subtree value, which remains indeterminate until that last unqueried leaf in that subtree (which is also the last unqueried leaf in the whole tree) is queried. This completes the inductive step.

Many students had the right idea for how to prove this but had the wrong model in mind. The right model is one in which the algorithm chooses a leaf to read based on the values of the leaves it has already read. Thinking of it top-down and saying that there's a predictable order in which the children of a node are evaluated is correct, but then simply saying that by induction we can make subtrees whose leaves are all read and have the appropriate root values is insufficient. This is because an assignment of the leaves of a certain subtree may influence the execution of the algorithm s.t. this subtree's root is no longer evaluated in the order that we would like. The induction should capture the fact that a subtree's root value can remain indeterminate until we set the last leaf, which then determines the root value as we desire. Many students also unnecessarily assumed that it suffices to consider cases in which the deterministic algorithm completely evaluated one subtree before going to another one. The right induction would not require this assumption.

(b) Consider the three subtrees of a height h tree. If they all have the same value, then I will only need to evaluate two of them regardless of which I pick. This case is easy so I will ignore it.

So assume instead that two subtrees have the same value and the third is different. Since I pick two subtrees at random, there is a 1/3 chance I will choose the two subtrees with the same value, allowing me to stop after two subtree evaluations, and a 2/3 chance that I will not, forcing me to evaluate all 3 subtrees. It follows that

$$\begin{aligned} T(h) &= \frac{1}{3}\cdot 2T(h-1)+\frac{2}{3}\cdot 3T(h-1) \\ &= \frac{8}{3}T(h-1) \\ &= (8/3)^h = n^{\log_3 \frac{8}{3}} \approx n^{0.893} \end{aligned}$$

Some students failed to note that if a tree had 2T's and 1F say, the probability of needing to analyze the third branch is $\frac{1}{3} * 1 + \frac{2}{3} * \frac{1}{2} = \frac{2}{3}$, not $\frac{1}{2}$. Given this mistake, the arguments ends up being $E(h) = 2E(h-1) + \frac{1}{2}E(h-1)$, rather than $\frac{2}{3}E(h-1)$.

Problem 3 MR 2.6.

Yao's minimax principle says that it suffices to lower-bound the time taken by any deterministic sorting algorithm on a specific input distribution. Let us consider the input distribution that chooses a permutation uniformly at random, and lower bound the running time of any deterministic algorithm. Any deterministic comparison-based sorting algorithm corresponds to a decision tree in which each leaf corresponds to a different input order. Since the decision tree must have a different leaf for each order, it must have $n!$ leaves. The running time on an input is the depth of the decision tree leaf that input takes us to, so on a (uniform) random input the expected running time is the expected depth of a leaf. So it suffices to prove the following: any binary tree with $n!$ leaves has average leaf-depth $\Omega(n \log n)$.

To do so, note that at most $(n!/2)$ leaves can be at depth less than $\log(n!/2)$. It follows that at least half the leaves have depth at least $\log(n!/2) = \Omega(n \log n)$. Thus is we pick a random leaf, at least half the time its depth will be $\Omega(n \log n)$. This in turn implies that the average leaf depth is $\Omega(n \log n)$.

Problem 4 Optional Problem.

If NP $\subseteq$ BPP, then there is a BPP algorithm to solve any given NP-complete problem. So we can use this BPP algorithm to create an RP solution for the NP-complete problem. Since we already know that RP $\subseteq$ NP, all we need to show is that NP $\subseteq$ RP, and then we know that NP = RP.

If we take any specific NP-complete algorithm and show that we have an RP solution to it, then we know that NP $\subseteq$ RP, because all other NP problems reduce to the NP-complete problem for which we showed RP membership.

Let us take the NP-complete problem of showing the existence of a Hamiltonian circuit in a graph. We assume that there is a BPP algorithm that solves this problem. We will use this algorithm as a guide to find a Hamiltonian circuit if it exists (there is a small probability that we will miss it). If we do find such a circuit, we accept, otherwise we reject. We exploit the fact that the Hamiltonian circuit problem is 'self-reducible', i.e. can be solved by solving a small number of 'smaller' instances of the same problem.

Our algorithm is the following.

1. Take a vertex and remove all but two if its edges.
2. Run the BPP algorithm a number of times polynomial in n, the number of vertices in the graph, to determine with high probability whether or not the removal of all but the two edges around the vertex still allow for the existence of a Hamiltonian circuit.
3. Repeat the previous step with a different set of two edges around the vertex until a pair is found. We know this pair exists, because only two edges touch a vertex in a

Hamiltonian circuit. The number of edges on a vertex is polynomial in n. If, after a polynomial number of tries of each of the edge pairs, there is still no accept, return Reject.

4. Repeat this process from step one with another vertex, working all the way through the graph. When this process has been run on each vertex, only two edges exists for each vertex, and the graph is a potential Hamiltonian circuit.
5. Now verify if this graph is in fact a Hamiltonian circuit. This can be done easily in polynomial time.
6. If the verifier accepts, then return Accept. Done.
7. If the verifier rejects, repeat from step one until the entire algorithm has been run a number of times polynomial in n.
8. Return Reject.

This algorithm runs in polynomial time, and has at least a 50% chance of returning Accept if there exists a Hamiltonian circuit. It also never returns Accept if there isn't a Hamiltonian circuit.

So this is an RP algorithm to solve the NP-complete problem of deciding if a Hamiltonian circuit exists or not. Since all all problems reduce to Hamiltonian circuit, NP $\subseteq$ RP. So NP = RP.

References

[Elias72] Peter Elias, The efficient construction of an unbiased random sequence, *The Annals of Mathematical Statistics*, 43(3):865–870, 1972.

[Shannon48] Claude E. Shannon, A Mathematical Theory of Communication, 1948.