15-750 Algorithms (Spring 2002) **Prof. Manuel Blum**

Homework #8 **Solutions**

Problem 1: Looking for a hole (10 points)

(this is Problem A from Section 2.1 of "Programming Pearls", vol. 1, by Jon Bentley)

Given a tape containing a list of 1,000,000 20-bit numbers, find a 20 bit number that is not on the list (why is such a number guaranteed to exist?).

(a) given as much RAM as you need and just the 1 tape containing the million words.

(b) given several tapes and only 2 dozen or so words of memory.

(c) *(M. Blum)* given only the 1 tape containing the million words and just 2 dozen or so words of memory.

In all cases, your algorithm should make just $\mathcal{O}(1)$ passes through the tape(s).

(a) Allocate an array A of 2^{20} bits, and initialize it with zeros. Scan the input tape once and upon reading a number k, set $A[k]$ bit to 1. Scan the array and return the first k such that $A[k] = 0$.

(b) (binary search; wlog. assume that all numbers are 20-bits long, possibly with leading zeros) To determine the leftmost bit of a desired number, scan the input tape and copy all the numbers starting with 0 to one auxiliary tape, and the numbers starting with 1 to another, while keeping the count of numbers on each tape. Choose the auxiliary tape with fewer numbers (or arbitrary, if both tapes have equal number of numbers), and set the leftmost bit of a desired number to be equal to leftmost bit of the numbers on the chosen tape.

Now, to determine the second leftmost bit, apply the same procedure using the second leftmost bit as the criterion for split, and the selected auxiliary tape as input. Continue in the same way, until all the bits are determined.

Since in each round we scan only at most half of the numbers scanned in the previous round, the total number of read/written tape-locations is bounded by $\mathcal{O}(N)$, where N is the number of numbers on the input tape.

(c) Initialize the available memory (say, 20 words) with *random* 20-bit numbers. Scan the tape and check whether any of the 20 numbers does not appear on the list.

Since there are at least $2^{20} - 10^6 = 48576$ numbers not on the list, the probability, that at least one of the randomly picked numbers is not on the list is at least

$$1 - (1 - 48574/2^{20})^{20} > 6/10 \ .$$

Therefore after c repetitions of this procedure we'll succeed with probability greater than $(1 - 1/2^c)$.

Note that this approach scales properly, i.e. the 20 words of memory are sufficient also for 10^9 30-bit numbers, 10^{12} 40-bit numbers, and in general for $10^{3\ell}$ 10ℓ-bit numbers (assuming each word of memory can store one number).

Problem 2: Graph orientation (10 points)

(this problem is due to Jim Propp, Electronic J. Combinatorics, 2002)

Give an algorithm for efficiently solving the following problem:

input: a connected d-regular (undirected) graph G with n vertices.

output: a uniformly random orientation of the edges of G such that the resulting graph has no sink.

"Uniformly random" means that every possible acceptable orientation should be exactly equally likely. (As usual, prove that your algorithm does what it is supposed to do, and analyze it, i.e. determine its expected running time).

(a) Do this for the case that $d = \log n$.
(b) Do it for the case that $d = 2$.
(c) *(extra credit)* Do it for the case that $d = 3$.

Be careful: you must prove that every sinkless orientation is exactly equally likely.

(Solution due to Jernej Barbič, Vincent Conitzer, and Vahe Poladian)

(a) $d = \log n$:

The algorithm is the following:
1. For every edge, pick a random orientation
2. If orientation sinkless, accept, otherwise, goto 1.

Clearly this algorithm selects every sinkless orientation with equal probability. We now show that for $n \geq 8$ the probability of success on every round is at least $1/10$.

Let A_i be the random event that node i is a sink for a random orientation of the edges. Clearly $p(A_i) = 1/n$. Events A_i are not independent.

Denote p' the probability that a random orientation has at least one sink. Then, using the principle of exclusion/inclusion, we have

$$p' = p(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} p(A_i) - \sum_{1 \leq i < j \leq n} p(A_i \cap A_j) + \sum_{k=3}^{n} (-1)^{k+1} z_k,$$

where

$$z_k = \sum_{1 \leq i_1 < i_2 < \ldots < i_k \leq n} p(A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}).$$

If no pair of nodes from $i_1, \ldots i_k$ is adjacent in the graph, then the events $A_{i_1}, \ldots, A_{i_k}$ are independent. The reason for this is that in this case, no edge in the graph touches more than one node from $i_1, \ldots i_k$. In this case we have

$$p(A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}) = \frac{1}{n^k}.$$

If there exists a pair of nodes from $i_1, \ldots i_k$ s.t. the two nodes are adjacent in the graph, the events $A_{i_1}, \ldots, A_{i_k}$ cannot happen simultaneously and

$$p(A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}) = 0.$$

So, we have

$$z_k \le \binom{n}{k}\frac{1}{n^k} \le \frac{1}{k!}.$$

Using the above argument for $k = 2$ (and the fact that every node has exactly d neighbors) we obtain

$$\sum_{1\le i<j\le n} p(A_i \cap A_j) = \frac{1}{2}\sum_{i\ne j} p(A_i \cap A_j) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1,j\ne i}^{n} p(A_i \cap A_j) = \frac{1}{2}\sum_{i=1}^{n}(n-d-1)\frac{1}{n^2} = \frac{1}{2}n(n-d-1)\frac{1}{n^2},$$

where $d = \log_2 n$.

Using all the above results, we have

$$p' = 1 - \frac{1}{2}n(n-d-1)\frac{1}{n^2} + \sum_{k=3}^{n}(-1)^k z_k.$$

We can estimate

$$|\sum_{k=3}^{n}(-1)^k z_k| \le \sum_{k\ge 3}\frac{1}{k!} = e - \frac{5}{2}.$$

Hence,

$$p' \le \frac{1}{2} + (e - \frac{5}{2}) + \frac{1}{2n}(1 + \log_2 n) < 0.9$$

for $n \ge 8$. As we let n grow this bound improves up to the limit $1/2 + e - 5/2 \approx 0.7182$.

This means that on every iteration, algorithm succeeds with probability at least $1/10$, hence the expected number of rounds is at most 10. Every round takes $O(m) = O(n \log n)$ time (to assign orientations and to check for sinks), hence the expected running time is $O(n \log n)$.

(b) $d = 2$:

A connected 2-regular graph is a cycle. Cycle has exactly two orientations which are sinkless, namely the two directions of the cycle. So, flip a fair coin to decide orientation of an edge (any edge will do), and then just walk around the cycle, assigning to every edge the orientation in which it was traveled.

Problem 3: Graph coloring (10 points)

(this problem is due to Abie Flaxman)

Graph 3-colorability is an NP-hard problem. Nevertheless ... give an efficient algorithm to properly 3-color the vertices of a 3-colorable graph $G(V, E)$ with n vertices, given that the vertices are all of degree $\ge d$, in the case that:

(a) $d = 2n/3$

(b) $d = n/(\log n)$

Recall: a 3-coloring is a map $f : V \to \{1, 2, 3\}$; a 3-coloring is *proper* if for every edge $(u, v) \in E$, $f(u) \ne f(v)$. (As usual, prove that your algorithm does what it is supposed to do, and ...)

(a) $d = 2n/3$:
Pick any edge (u, v) and color u with color 1 and v with color 2. Since u and v have each $2n/3$ neighbors, there at least $n/3$ vertices connected to both u and v (i.e. in the intersection of neighbors of u and v). Clearly, all these vertices have to be colored with color 3. Also, there cannot be more than $n/3$ in the intersection because the graph is 3-colorable. It follows, that the intersection contains all the vertices which will be colored with color 3, and that each vertex in the intersection is connected to each vertex outside of the intersection. Now, consider an edge (u, w), where w is colored with 3 (and u is colored with 1). By an analogous argument, the intersection of neighborhoods of u and w has size $n/3$ and has to be colored with 2, and similarly the remaining vertices have to be colored with 1.

The correctness of the algorithm follows from the fact, that we are coloring a vertex only when its color is uniquely determined by the constraints in the graph (up to a permutation of the colors). The running time is $\mathcal{O}(n)$.

(b) $d = n/(\log n)$
Note: This part showed to be harder than we expected — we don't know whether it this problem can be solved in time polynomial in n. The solution presented below runs in time $\mathcal{O}(n^{\log n})$. However, a similar technique gives an algorithm (how?), which runs polynomial time in the case when $d = n/c$ for any constant c.

Consider the following algorithm:

1. Let $p = 2 \log e (\log n)^2/n$, where $e = 2.71\ldots$ and log is to base 2. For each vertex $v \in V$ with probability p "accept" it, and with probability $1 - p$ "reject" it. Let W be a set of all "accepted" vertices.
2. If size of W is smaller than $2c \log e (\log n)^2$, for some constant $c > 1$, and if the neighborhood of W is equal V, i.e. if every vertex has at least one neighbor in W, go to step 3, otherwise fail.
3. Iterate through all possible proper 3-colorings of W, until a proper 3-coloring of G is found:

   (a) Extend the current coloring of W by propagating the constraints imposed by the current coloring of W and extensions made so far, i.e. whenever a not-yet-colored vertex v has two neighbors colored with two different colors, color v with the remaining third color. If propagation is not possible, because some vertex u has already 3 neighbors colored with different colors, start over at step 3, with the next proper 3-coloring of W.
   (b) If previous step colored all vertices in G, we are done. Otherwise, we are left with a set of vertices, which have neighbors of one color only, i.e. each has two candidate colors left. But completing the coloring in this case corresponds to finding a satisfying assignment to a 2-CNF formula (how does this formula look like?), i.e. to solving a 2-SAT problem, which can be done efficiently. If a satisfying assignment is found, we are done, otherwise start over at step 3, with the next proper 3-coloring of W.

Note that once the algorithm reaches step 3, it will find a proper 3-coloring with high probability. The running time then will be clearly dominated by the iteration through all possible proper 3-colorings of W, and can be bounded by $\mathcal{O}(2^{|W|}) = \mathcal{O}(2^{(\log n)^2}) = \mathcal{O}(n^{\log n})$.

To complete the analysis, we have to show that with non-negligible probability step 1 yields W with $|W| < c(\log n)^2$ and whose neighborhood is equal to V (then a few repetitions of step 1 will with high probability pick W which will reach step 3).

Since the expected size of W is $2\log e(\log n)^2$, then by Markov Inequality

$$\mathbf{Pr}[|W| \geq 2c\log e(\log n)^2] \leq \frac{1}{c} \,,$$

hence if we take $c = 4$, then with probability at least $1/2$ W will be of the right size[1]. Now, given that $|W|$ is about $2\log e(\log n)^2$, the probability that any $v \in V$ W has no neighbors in W is at most $(1 - \log n/n)^{2\log e(\log n)^2} \leq 1/n^2$. Hence the probability that neighborhood of W is not equal to V is at most $1/n$. □

Problem 4: Size of a maximum matching (10 points)

(this is Problem 7.7 from M&R; you should solve it without using the reference given in the book)

Let $G(U, V, E)$ be a bipartite graph, and let A be the corresponding matrix of indeterminates as defined in Section 7.3 (Edmonds' matrix). Show that the size of a maximum matching in G is exactly equal to the rank of the matrix A.

(Solution due to Ting Liu, Takayuki Osogami, and Hua Zhong)

We already know that if a bipartite graph has a perfect matching, its Edmonds' matrix's determinant is non-zero. A maximum matching of bipartite graph can be seen as the maximum sub-graph that has a perfect matching. We say this maximum sub-graph is G' and its Edmonds' matrix is A'. If we delete the rows and columns corresponding to the vertices that are not in G' from the A (Edmonds' matrix of G), we get A'. And since G' has a perfect matching, we have det(A')≠0. A and A' are all square matrices and their numbers of rows are the size of G and G'. And we know: the rank of a matrix is equal to the rank of the maximum sub-matrix that has non-zero determinant. Also we know A' is this maximum sub-matrix. If not, it means there is another sub-graph that is larger than G' which has a perfect match. This contradicts with the assumption that G' is the maximum sub-graph with a perfect matching. So rank(A) = rank(A'), and rank(A') is the size of the maximum matching. So we get: the size of the maximum matching equal to the rank of A.

Problem 5: Randomized Rank (10 points)

Give a randomizing algorithm to compute the rank of the matrix of indeterminates from Problem 4. (As usual, ...)

(Solution due to Ting Liu, Takayuki Osogami, and Hua Zhong)

Algorithm Initialize $k = 0$. Let p be a prime such that $2n < p < 4n$. For each variable x_{ij} in the Edmonds' matrix, assign a number r_{ij} independently, uniformly, and randomly chosen from $\mathbf{F}_p$. Calculate the rank r of the matrix using the Gaussian elimination. If $r > k$, let $k := r$. Do this procedure (i.e., choose random numbers, calculate the rank, and update k) n times. Return k as the rank of the matrix.

[1] To be precise, for the rest of the analysis we should bound also the probability that $|W|$ is significantly smaller than the expectation. This can be done using Chebyshev's Inequality.

Correctness of the Algorithm By problem 4, the rank of the Edmonds' matrix corresponds to the size of the maximum matching for the corresponding graph. Let M be a maximum matching. If we deleted the columns and rows corresponding to the nodes that are not matched in M, we would get a matrix M' of the size of the maximum matching. Since the corresponding graph has the perfect matching, the determinant of the matrix is nonzero. For the calculated rank to be less than the actual rank the matrix M' has to be non-singular for all the maximum matchings M. Thus, the probability that the calculated rank is less than the actual rank is less than or equal to the probability that a matrix M' is singular.

The determinant of the matrix M' is a polynomial of degree k, which is the rank of M. So, the probability P that a matrix M' is singular is

$$P \leq \frac{k}{|\mathbf{F}_p|} < \frac{n}{2n} = \frac{1}{2}.$$

Thus, the probability that the calculated rank is less than the actual rank is less than $\frac{1}{2}$. This procedure is repeated n times. Therefore, the probability that the returned rank is less than the actual rank is less than $\frac{1}{2^n}$.

Running Time We find a prime p such that $2n < p < 4n$ by randomly choosing a odd number from $(2n, 4n)$ and test whether it is a prime. The process is repeated until we find a prime. Since the number of primes in $(2n, 4n)$ is roughly $\frac{2n}{\log 2n}$, the probability of choosing a prime is $\frac{1}{\log 2n}$. So, the expected time of the repetition is $O(\log n)$. Each primality test, using the algorithm given in the lecture, takes $O(n^3 k)$ steps to achieve the error probability less than $\frac{1}{2^k}$. By choosing $k = n$, the overall running time is $O(n^4 \log n)$

The expected time to find a prime p such that $2n < p < 4n$ is

Each elementary operation such as addition and multiplication in $\mathbf{F}_p$ takes $O(\log^2 n)$ (addition is $O(\log n)$). The Gaussian elimination takes $O(n^3)$ elementary operations. Since the Gaussian elimination is performed n times, the overall running time is $O(n^4 \log^2 n)$.