15-451 Algorithms, Fall 2003

Homework # 5 **due: Tuesday November 4, 2003**

Please hand in each problem on a separate sheet and put your **name** and **recitation** (time or letter) at the top of each sheet. You will be handing each problem into a separate box, and we will then give homeworks back in recitation.

Remember: written homeworks are to be done individually. Group work is only for the oral-presentation assignments.

Problems:

(30 pts) 1. [Fair carpooling] The n employees of Algs-R-Us (run by former VP Al Gore-ithm) sometimes carpool to work together. Say there are m days, and S_i is the set of people that carpool together on day i. For each set, one of the people in the set must be chosen to be the driver that day. Driving is not desirable, so the people want the work of driving to be divided as equitably as possible. Your task in this problem is to give an algorithm to do this efficiently and fairly.

The fairness criterion is the following: A person p is in some of the sets. Say the sizes of the sets that p is in are $a_1, a_2, \ldots, a_k$. Person p should really have to drive $\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_k}$ times, because this is the amount of resource that this person effectively uses. Of course this number may not be an integer, so let's round it up to an integer. The fairness criterion is simply that she should drive no more than this many times.

For example, say that on day 1, Alice and Bob carpool together, and on day 2, Alice and Carl carpool together. Alice's fair cost would be $1/2 + 1/2 = 1$. So Alice driving both days would not be fair. Any solution except that one is fair.

(a) Prove that there always exists a fair solution.

(b) Give a polynomial-time algorithm for computing a fair solution.

Hint: Try to model the problem using network flow in such a way that part (a) falls out directly from the integrality theorem for network flow, and part (b) just follows from the fact that we can solve max flow in polynomial time. So, it all boils down to coming up with the right flow graph to model the problem.

(Note: this is not an online problem. We are assuming we know all the sets $S_1, \ldots, S_m$ up front).

(30 pts) 2. [Options pricing] An *option* (specifically, an "American call option") gives the holder the right to purchase some underlying asset (e.g., one share of IBM) at some specified exercise price (e.g., \$100) within some specified time period (e.g., 1 month). The value of an option depends on the current price of the asset, the exercise price of the option, the length of the time period, and one's beliefs about how the asset's price is likely to behave in the future.

For example, to take an easy case, suppose we have an option to buy one share of IBM at \$100 that expires right now. If IBM is currently going for \$105, then the value of this option is \$5. If IBM is currently going for \$95, then the value of this option is \$0 (we wouldn't want to exercise it). However, suppose the option expires tomorrow, and suppose we have some simple model of how IBM shares behave. E.g., perhaps our model says that each day, with probability 1/4 the share price goes up by \$10, and with probability 3/4 the share price goes down by \$5. In that case, if IBM is currently worth \$95, then the value of this option is \$1.25 because that is our expected gain if we use the strategy "wait until tomorrow, and then exercise the option if IBM went up"; if IBM is currently worth \$105, then the value is \$5 (and our optimal strategy is to exercise the option right away).

(a) Suppose we are guaranteed the stock price will be an integer between 0 and b. Suppose we also have have a probabilistic model p_{ij} for how the stock behaves: specifically, if the stock has value i on day t, then p_{ij} is the probability that the stock will have value j on day $t+1$. So, for each i, $\sum_j p_{ij} = 1$.

Describe a dynamic-programming algorithm to calculate the value of an option of exercise price X that expires T days in the future, given that the current price of the stock is S. The running time of your algorithm should be $O(b^2 T)$.

(b) Suppose our option can only be exercised at exactly time T, rather than any time $\leq T$ (this is called a "European option"). Describe an algorithm to solve for the option's value that runs in time $O(b^3 \log T)$.

Note: this problem is different from the "competitive analysis" setting because we are assuming we have a probabilistic model of how the world will behave. Often, if you have this kind of probabilistic model of the world, you can use dynamic programming to solve for the strategy that does best in expectation (like you are doing in this problem).

(40 pts) 3. [The List-Update Problem] This is a competitive-analysis problem.

Suppose we have n data items $x_1, x_2, \ldots, x_n$ that we wish to store in a linked list. Let's say the cost for performing a *lookup*(x) operation is \$1 if x is in the head of the list, \$2 if x is the second element in the list, and so on. Let's also say that when some item x is accessed, we can move x up in the list (as far as we want) for free.

The *Move-to-Front* (MTF) strategy is the following algorithm for organizing the list if we don't know in advance how many times we will access each element. We begin with the elements in their initial order $(x_1, x_2, \ldots, x_n)$. Then, whenever we perform a *lookup*(x) operation, we move the item accessed all the way up to the front of the list. For instance, if the first operation was *lookup*(x_3), then we pay \$3, and afterwards the list will look like $(x_3, x_1, x_2, x_4 \ldots)$. If we then access x_1 we pay \$2, and afterwards the list will look like $(x_1, x_3, x_2, x_4 \ldots)$.

It turns out the MTF has a very good competitive ratio for this problem. In particular, for any sequence of lookup requests, the total cost of MTF is at most twice the cost of the best strategy we could have followed in hindsight (i.e., the best we could have done if each time we accessed an element, a smart birdie told us how far forward to move it). In contrast, other natural algorithms like moving the accessed element x up by just 1, or ordering the elements by their access frequencies so far, have pretty bad competitive ratios.

(a) Give an example of a sequence of requests that shows that the strategy "move the accessed element up by 1" (i.e., swapping it with the element that was directly in front of it in the list) has a competitive ratio $\Omega(n)$. In other words, your request sequence should cause that strategy to pay $\Omega(n)$ times more than the optimal strategy in hindsight.

Your job is now to prove that the competitive ratio of the MTF algorithm is at most 2. We are going to do this in two steps. The first step is to show that MTF is never more than twice as bad as doing nothing. This doesn't sound so exciting, but this step will contain most of the ideas of the final proof. The second step will be to extend your argument to show that MTF never has more than twice the cost of OPT, where OPT is the best strategy you could have used in hindsight.

(b) Prove that the cost of Move-to-Front is at most $2C_{nomove}$ where C_{nomove} is the cost of the algorithm that just keeps the list in the original ordering $(x_1, x_2, \ldots, x_n)$ and never moves elements around.

Hint: If $i < j$ but x_j is in front of x_i in the MTF list, let's say that x_j has "cut in line" in front of x_i. Now, imagine that each element x_i has a piggy bank with \$1 for everyone that is currently cutting in line in front of it.

(c) Now prove that the cost of MTF is at most $2C_{OPT}$, where C_{OPT} is the cost of the optimal strategy in hindsight.

Hint: Formally, what you were doing in the first part is defining a potential function $\Phi = \#\text{pairs } (i, j)$ such that j is in front of i in the MTF list, but i is in front of j in nomove's list. Try replacing "nomove" with "OPT" in this potential function. Notice that initially $\Phi = 0$ and Φ can never become negative. So, all you need to prove is that on each lookup, cost(MTF) + (change in Φ) < 2*cost(OPT). Also, remember that when some item x is accessed, the only thing OPT can do is move x up by some number of positions in the list.