1 Lecture 8, December 9, 1999

We shall consider issues such as witnesses for Boolean matrix multiplication and testing the results of matrix multiplication. Then we will start a new subject – that of greedy algorithms.

1.1 Witnesses for Boolean matrix multiplication

Let X and Y be Boolean matrices. Let W be a *witness* matrix for the product XY, with $w_{ij} = 0$ if $\sum_k x_{ik}y_{kj} = 0$, and otherwise $w_{ij} = k$ for some k satisfying $x_{ik}y_{kj} = 1$. (There may be several different witness matrices for the same Boolean product.) A witness matrix can be used to give the parent relation in all pairs shortest paths computations. Naively, a witness matrix can be computed in $O(n^3)$ time. We present here a randomized algorithm that computes a witness matrix in time $O(n^\omega)$, up to logarithmic multiplicative factors.

Assume for simplicity that for every ij, there is at most one k such that $x_{ik}y_{kj} = 1$. Then we can compute W as follows. For each column k in matrix X, multiply its entries by k. Call this new matrix Z. Now perform the integer matrix product ZY. The output is exactly the matrix W. As all entries in X and Y are bounded by n, computations can be done using numbers of $O(\log n)$ bits, and fast matrix multiplication gives a running time of $O(n^\omega)$.

Now consider the more general case in which there may be several k with $x_{ik}y_{kj} = 1$. The entries where ZY is 0 are still exactly those entries where W is 0. However, the nonzero entries of ZY are sums of witnesses rather than individual witnesses. To avoid this problem we use random sampling. We shall have a parameter l that ranges from 1 to $\log n$. for each value of l we shall perform $c \log n$ experiments of the following type, where c is a large enough constant (that controls the error probability). For each column of Z independently, multiply it by 0 with probability $1 - 2^{-l}$. Call the resulting matrix R. Compute the integer matrix product $M = RY$. Now for every ij for which $1 \le m_{ij} \le n$, check if $x_{ik} = 1$ and $y_{kj} = 1$, where $k = m_{ij}$. If both conditions hold, set $w_{ij} = m_{ij}$.

Altogether the algorithm performs $O((\log n)^2)$ integer matrix multiplications.

We now show that with high probability it indeed computes the witness matrix W. Consider an arbitrary ij, and let m denote the number of distinct values of k for which $x_{ik}y_{kj} = 1$. Call these values $k_1, \ldots, k_m$. Consider a value of l for which $2^{l-1} < m \le 2^l$, and consider a single random choice of a matrix R as above, for this value of l. The probability that column k survives (is not multiplied by 0) is 2^{-l} which is between $1/m$ and $1/2m$. Hence the probability that *exactly* one of the m columns $k_1, \ldots, k_m$ survives is some constant. (For $2^l = m$ this constant is at least $1/e$, and for somewhat larger l it is somewhat smaller.) Hence with constant probability we are in the case of a unique witness, in which case the algorithm extracts w_{ij}. Repeating the experiment $c \log n$ times makes the probability of failure polynomially small in n. When the probability of failure for each individual ij is below ϵ/n^2, then the probability of failing to compute W correctly is below ϵ.

We note that the algorithm above can easily be transformed into a Las Vegas algorithm.

1.2 Testing matrix multiplication

Matrix multiplication may involve a complicated algorithm. Given integer matrices A and B, let C denote their product, and let C' be the output of some fast matrix multiplication algorithm on input A and B. We wish to check that C' is indeed the right answer (namely, that $C' = C$) without actually repeating the whole computation.

The following randomized test can be used. Select a 0/1 vector v at random. Now compute $C'v = w'$ and compare it to $Cv = w$. As we do not know what C is, we can use the relation $Cv = (AB)v = A(Bv)$ (which uses associativity of matrix multiplication). Hence computing $C'v$ involves one matrix-vector product, and computing Cv involves two matrix-vector products. The running time is thus $O(n^2)$, which is much faster than the time required to compute AB (using known algorithms).

Comparing w and w', we can test whether $C = C'$. If $C' = C$, then $w' = w$. On the other hand, if $C' \neq C$, then with probability at least $1/2$, $w' \neq w$ (proof left to the reader). Hence if we find that $w' \neq w$, we know that there was an error in the matrix multiplication. If we repeat the experiment with k random and independent vectors v, then we have probabability at least $1 - 2^{-k}$ of detecting that C' is the wrong output.

1.3 Greedy algorithms

When searching for the optimal solution to a problem that has many feasible solutions, search may progress in one of several ways, depending on the structure of the problem. As a rough and incomplete classification, these are three of the major approaches:

1. *Exhaustive search:* Searches through all feasible solutions, or through a large fraction of them (where "large fraction" is a term that may depend on the context). Used for problems that have very little structure, or for which the structure is poorly understood. For example, for finding maximum cliques in graphs.

2. *Dynamic programming:* Searches through a very small fraction of the feasible solution. Typically, dynamic programming is an iterative algorithm. At each irteration, a small (polynomial size) set of partial solutions is maintained, one of which is known to lead to the optimal solution. In the next iteration, members of the set are extended based on information from other members of the set. At the last step, the optimal solution is chosen from the set. To keep the sizes of the sets small, the problem needs to have sufficient structure so that all partial solutions can be grouped into a small number of equivalent classes. For example, subset sum with polynomial weights.

3. *Greedy:* This may be viewed as the ultimate form of dynamic programming, in which only one partial solution is maintained. The problem needs to have much more structure for this approach to work. For example, minimum spanning trees.

Remark: There are other commonly used search strategies, such as *branch and bound*. Also, sometimes the method of finding the optimal solution involves ingredients that do not have intuitive explanations as search procedures. For example, finding triangles in graphs using fast matrix multiplication.

We shall study several problems that can be solved to optimality in polynomial time using the greedy algorithm.

1.4 Independent sets in interval graphs

Let there be a collection of events, where event i is to start at time s_i and end at time t_i. Two events are *conflicting* if their time intervals overlap. We wish to schedule as many mutually nonconflicting events as possible in the same room.

In graph theoretic terms, we can think of each event as a vertex and two conflicting events have an edge between them. This is an *interval graph*, and we wish to find the maximum independent set in such a graph. The interval representation of this graph is given to us.

The following greedy algorithm works:

1. Sort all events in increasing order of their end time t_i.

2. Iteratively, pick the first event in the (remaining list), and remove from the list all events that overlap with it.

Remark: If the list does not contain two events such that the time interval of one completely contains the time interval of the other, then sorting by s_i gives the same order as sorting by t_i.

To see that the algorithm works, let $A = a_1, a_2, \ldots$ be the sequence of events scheduled by the algorithm, and let $O = o_1, o_2, \ldots$ be the sequence of events scheduled by the optimal solution. As there may be more than one optimal solution,we take the optimal solution which has the longest prefix in common with the greedy solution. We claim that necessarily, $A = O$. Otherwise, let i be the first index in which $a_i \neq o_i$. Consider now a schedule O' identical to O, except that o_i is replaced by a_i. As a_i starts after $a_{i-1} = o_{i-1}$ ends, and as a_i ends before o_i does, this is still a legal schedule. As it schedules as many jobs as O but has a longer prefix in common with A, we have a contradiction.

1.5 Scheduling jobs so as to minimize total penalty

There is a collection of jobs. Job i has length l_i and weight w_i. We wish to schedule the exectution of the jobs with no overlaps. Let t_i be the time at which job i finishes according to the schedule. Then the penalty associated with job i is $w_i t_i$, and we wish to minimize $\sum w_i t_i$.

We shall develop a greedy algorithm for this problem by first considering how the proof of optimality may look like, and then deriving the algorithm from the proof.

Consider two schedules, S and S', that are identical except that the order of two jobs i and j that are consecutive in S is flipped in S'. Then simple manipulations show that the total cost of both schedules is the same, except for one term: a term $w_j l_i$ in S is replaced by $w_i l_j$ in S'. If we assume that S is the optimal schedule, then we have $w_j l_i \leq w_i l_j$, which implies $w_i/l_i \geq w_j/l_j$.

This suggests the following algorithm.

1. Sort all jobs in order of decreasing ratios w_i/l_i.

2. Execute jobs in the sorted order.

To see that this algorithm generates an optimal schedule, consider any other schedule. It must have two consecutive jobs i and j with $w_i/l_i < w_j/l_j$. Then by flipping the order of i and j we get a less costly schedule, which is a contradiction.