6.854 **Dan Ports**
Advanced Algorithms `drkp@mit.edu`
2004/11/03 Collaborators: {`sarahl`,`pramook`}

Problem Set 8

Problem 1:

a) We claim that any feasible subset of jobs can be scheduled in order of increasing deadline. Suppose there is a feasible schedule that does not have this property. Then there are two jobs, x and y, with x scheduled before y, and y's deadline $d_y < d_x$. Then we can swap them. y will now be scheduled before x. This is feasible for y, since y's scheduling was feasible before and it is only moved earlier. It is feasible for x because x is now finished at the time y originally did, which was before d_y and hence d_x. We can repeatedly swap jobs until there are no more out-of-order jobs.

b) We give a dynamic program for computing the fastest-completing maximum-weight feasible subset. Define $T(n, W)$ to be the shortest time required to complete a schedule of total weight W using only the first n jobs. Then, since we either include the nth job or not,

$$T(n, W) = \min\left(T(n-1, W), T(n-1, W - w_n) + p_n\right)$$

provided that adding job n is feasible, i.e. $T(n-1, W-w_n)+p_n \leq d_n$. If this inequality does not hold, then $T(n, W) = T(n-1, W)$. This gives a dynamic program, along with $T(0, W) = T(n, 0) = 0$. It has $\sum w_i$ rows and n columns; since we assume the weights are polynomially bounded, their sum is also, and so the complexity of the dynamic program is polynomial.

c) This gives a fully-polynomial approximation scheme for the problem of minimizing lateness penalties. We cannot directly use the dynamic program above, since the weights may not be polynomially bounded. So we must scale the weights. Let $\mathscr{W}$ be the sum of weights, $\mathscr{W} = \sum_i w_i$, and let

$$K = \frac{\mathscr{W}\epsilon}{n}$$

Then we rescale each weight by dividing it by K and rounding:

$$w'_i = \left\lfloor \frac{w_i}{K} \right\rfloor$$

We can then apply the dynamic program described above to find the fastest-completing maximum-weight feasible subset of the scheduling problem with the rescaled weights w'_i. The dynamic program table has $\sum w'_i$ rows and n columns, and constant time is required to fill in each cell. Recall that

$$\sum w'_i = \frac{\sum w_i}{\frac{\mathscr{W}\epsilon}{n}} = \frac{n\mathscr{W}}{\epsilon\mathscr{W}} = \frac{n}{\epsilon}$$

so solving the dynamic program requires $n\frac{n}{\epsilon}$ time. Thus this algorithm has the running time of a fully-polynomial approximation scheme.

We can then schedule all jobs that are in the fastest-completing maximum-weight feasible subset identified by the dynamic program in order of increasing due date (since we have shown that this is an optimal solution), and then scheduling the remaining jobs in any order (since they cannot be completed in the maximum-weight feasible solution, we must incur the lateness penalty associated with them in any case, and so it does not matter when they are scheduled as long as it is after the feasible subset).

The answer to the dynamic program is not the optimal value, however, as we have rounded the weights. Call the weight of the maximum-weight subset the dynamic program found $\mathfrak{W}$. Note that the error introduced by rounding on w'_i is at most $K = \mathscr{W}\epsilon$, and so the error introduced over all weights is at most $n\mathscr{W}\epsilon$. We rounded down, so $\mathfrak{W}$ is less than the optimum $\mathfrak{W}_{opt}$ by at most $n\mathscr{W}\epsilon$. Consider also the maximum weight $\mathfrak{W}'$ obtained if we had rounded *up* instead of down. Then we know that $\mathfrak{W}' - nK < \mathfrak{W} \leq \mathfrak{W}_{\mathtt{opt}}$. So $\mathfrak{W}' - \mathfrak{W} < K\mathscr{W}$. Also $\mathfrak{W}' \leq \mathfrak{W}_{opt}$ since $\mathfrak{W}'$ is obtained using rounding. So $\mathfrak{W}_{opt} - \mathfrak{W} < Kn = \mathscr{W}\epsilon$. This is the amount by which the computed weight of the maximum-weight feasible subset is less than that of the optimum solution. Since the algorithm incurs the lateness penalty for all the jobs not in this subset, if we let X be the total lateness penalty obtained by this approximation algorithm and X_{opt} be the optimum, we have

$$X \leq X_{opt} + \mathscr{W}\epsilon \leq (1+\epsilon)X_{opt}$$

and so our algorithm is a fully-polynomial time approximation scheme.

Problem 2:

a) Suppose there are pairs (x_k, y_k) of vertices that must be connected. We define the following integer linear program, which is essentially n unit-cost flows. Define f_{ij}^k to be the indicator random variable indicating whether demand pair k's path uses edge $i \to j$. Then we have the following integer linear program:

$$\begin{aligned} \forall i,j,k \quad & 0 \leq f_{ij}^k \leq 1 \\ \forall i,j \quad & \sum_k f_{ij}^k \leq u_{ij} \\ \forall j,k \quad & \sum_{i \neq j} f_{j,i} - \sum_{i \neq j} f_{i,j} = \begin{cases} 1 & j = x_k \\ -1 & j = y_k \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Any feasible solution to this ILP sends one unit of flow along along a path from each demand pair's source to sink, thanks to the conservation conditions and the requirement that the source-sink flow is 1.

b) We relax this integer LP to be a normal linear program. Then we eliminate the requirement that f_{ij}^k is an integer, and simply enforce that it is between 0 and 1.

The linear program can be solved in polynomial time. The solution will be a fractional flow of total value 1 for each of the demand pairs: each f_{ij}^k will be between 0 and 1, and for fixed k we still have the conservation criterion and the enforcement that flow from the source and into the sink is 1. So for each k we have a (possibly fractional) flow of value 1 from source to sink.

c) We now perform randomized rounding on the relaxed linear program: for each demand pair k, we choose one path from the source x_k to the sink y_k. We choose path i with probability equal to the value of the flow through that path in the solution to the relaxed LP; since the sum of the flows through each path is 1, this defines a probability distribution. We claim that each edge has $O(\log n)$ paths through it. Call the number of paths P_{ij}.

First, note that for any edge $i \to j$, the expected number of paths through it is $\mathrm{E}[P_{ij}] = \sum_k f_{ij}^k$, which is necessarily less than 1. To bound the number of paths, we use the Chernoff bound, which we can do since the number of paths is given by a sum of indicator random variables.

Then, by the Chernoff bound,

$$\Pr[P_{ij} \geq 1 + \log n] \leq 2^{-(1+\log n)}$$

since the mean is at most 1. And, using the union bound, since there are n edges, the probability that there is any edge with more than $1 + \log n$ paths through it is

$$n\left(2^{-(1+\log n)}\right) = \frac{n}{n}2^{-1} = 1/2$$

So the expected number of rounding attempts to be performed with guaranteed at most $1+\log n = O(\log n)$ paths going through every edge is 2, a constant. This is certainly polynomial time, and the remainder of the algorithm is also polynomial, so this proves the claim.

d) Suppose now that each edge has capacity w. Then the expected number of paths through an edge is $\mathrm{E}[P_{ij}] = \sum_k f_{ij}^k \leq w$. We now consider the probability that the number of paths through an edge exceeds w by a factor of ϵ, for some $\epsilon < 1$. By the Chernoff bound,

$$\Pr[P_i \geq (1+\epsilon)w] \leq e^{-\frac{\epsilon^2 w}{4}}$$

and so the probability that any edge has more than $(1 + \epsilon w)$ paths is

$$ne^{-\frac{\epsilon^2 w}{4}}$$

We must make this a constant probability, by letting

$$\epsilon = \left(\frac{\ln n}{w}\right)^{1/2}$$

so that the probability is $ne^{-\frac{\ln n}{4}} = e^{-\frac{1}{4}}$, which is a constant, so we can repeat the rounding a constant number of times to ensure no edge has more than $(1+\epsilon)w$ paths through it. If $w > \log n$, this is $w + O\left(\sqrt{w \log n}\right)$, and if $w \leq \log n$, this is simply $w + O(\log n)$.

Problem 3:

a) If the answer is "yes", then there is a set B of size k such that every set in C is the union of some subset of B. There are at most 2^k subsets of B, so there cannot be more than 2^k sets in C without having duplicate sets in C, which is impossible.

b) Suppose that for all S in C, $x \in S$ if and only if $y \in S$. Then we can remove y from all sets in C without changing the answer to the set basis problem. For some S, suppose first that $x \notin S$. Then $y \notin S$, so removing y from C does not affect S. Now suppose $x \in S$. Then $y \in S$ also. In the basis, S is the union of some $\psi_i \in B$. Removing y from C removes it from S. But we can also remove y from all ψ_i that cover S. This does not affect the cover, since the same sets will still cover S without y, and y is no longer needed to cover any set. Thus, y can be removed from all sets in C and B without changing the answer to the set basis problem.

c) Consider an instance of the set basis problem. The set C contains at most 2^k sets if the answer is "yes", so we know that if C contains more than 2^k sets, we can immediately answer "no", and hence we can assume C contains at most 2^k sets.

We now claim that the sets in C can be reduced to at most 2^{2^k} elements. The power set of C contains 2^{2^k} families of the 2^k sets in C. Each element y in a set $S \in C$ is in some collection of sets in S, which is given by one of families in the power set of C. So if there are more than 2^{2^k} elements in the sets of C, then by the pigeonhole principle there must be some x that is in the same family of sets in C. So y can be eliminated. This process can be repeated to reduce C's sets to at most 2^{2^k} elements.

Hence, we can simply test assignments of the 2^{2^k} elements in the sets of C to k sets in B. This can be done in less than $k2^{2^{2^k}}$ time[1], which is a function of k, plus time polynomial in the problem

[1] This bound is presumably not at all tight.

size to identify and remove redundant elements.

Problem 4:

Consider an instance of $MAX - SAT$ with treewidth k. We use essentially the same algorithm as for SAT to show that it is fixed-parameter tractable. We create the graph where vertices represent variables, and two vertices are connected if two variables share a clause; this is the graph that we assume has treewidth k. We then proceed to eliminate variables using the optimal elimination ordering on the graph.

We maintain for each clause a truth table, augmented for each assignment of variables with the number of reduced-clauses (clauses that were previously eliminated and reduced into the current clause, plus the current clause) satisfied by that assignment. So in the initial case, this is simply 1 for every assignment that satisfies the clause, and 0 otherwise.

Let x be the variable being removed. We find all clauses involving x and enumerate all possible assignments of variables for that clause. We can build a truth table for the combination of all clauses involving x essentially by joining the truth tables we have for each clause. There are at most k variables involved, since this is the degree of the x vertex, so the truth table size is 2^k. We combine these into a new clause of at most k variables. Rather than simply identifying settings of the variables such that some x makes all clauses true (as we do in the SAT FPT algorithm), we must consider the reduced-clauses-satisfied. For each combination of the other variables involved (y, z, $\ldots$), we choose the value of x that maximizes the sum of the reduced-clauses-satisfied for that combination of the other variables. We then add that combination to the truth table with the maximized sum of reduced-clauses-satisfied we found, and add a tag indicating which assignment of x we chose. Once we consider each combination of the variables, we have eliminated the variable x and built up the augmented truth table for the new clause.

We apply this procedure for each variable until we have eliminated all variables. Since we choose the maximum sum of reduced-clauses-satisfied at each stage, the final reduction will give the maximum number of satisfiable clauses, and the tags attached will indicate the necessary assignments of each of the variables.

Note that the processing time required to process each clause elimination is proportional to the size of the truth table, which we showed above was 2^k. There are n variables to be eliminated, so the runtime is $O\left(n2^k\right)$, indicating that the problem is fixed-parameter tractable.

Problem 5:

12 hours or so.