University of California — Berkeley
CS170: Efficient Algorithms and Intractable Problems
Professor Luca Trevisan

Handout MS2
November 19, 2001

Midterm 2 Solutions

Problem 1. Provide the following information: Your name; Your SID number; Your section number (and/or your TA name); Name of the person on your left (if any); Name of the person on your right (if any).

Solutions

The correct spelling of the TA's names are Scott Aaronson, Shyam Lakshmin, Iordanis (Jordan) Kerenidis, Joseph (Joe) Polastre, Beini Zhou.

We gave full credit as long as you could spell your own name. (And we were not too strict in checking that, either.) Common mistakes involved the spelling of Beini's and Jordan's last names.

Problem 2. Consider a undirected graph $G = (V, E)$ with nonnegative weights $w(i, j) \geq 0$ on its edges $(i, j) \in E$. Let s be a node in G. Assume you have computed the shortest paths from s, and minimum spanning tree of the graph. Suppose we change the weights on every edge by adding 1 to each of them. The new weights are $w'(i, j) = w(i, j) + 1$ for every $(i, j) \in E$.

(a) Would the minimum spanning tree change due to the change in weights? Give an example where it changes or prove that it cannot change.

(b) Would the shortest paths change due to the change in weights? Give an example where it changes or prove that it cannot change.

Solutions

The first question was, if T is a minimum spanning tree of a graph G, and if every edge weight of G is incremented by 1, is T still an MST of G? The answer is yes. The simplest proof is that, if G has n vertices, then any spanning tree of G has $n - 1$ edges. Therefore incrementing each edge weight by 1 increases the cost of every spanning tree by a constant, $n - 1$. So any spanning tree with minimal cost in the original graph also has minimal cost in the new graph.

There are alternative proofs that amount to more complicated ways of saying the same thing. For example: assume by way of contradiction that T is not an MST of the new graph. Then there is some other spanning tree of G, call it $\widehat{T} \neq T$, with lower cost on the new graph. Given a cut (R, S) of G, T has exactly one edge e and $\widehat{T}$ has exactly one edge $\widehat{e}$ crossing the cut. Suppose $\widehat{e} \neq e$ and $\widehat{e}$ has lower cost than e. Then by replacing e with

$\widehat{e}$, we obtain a new spanning tree for the original graph with lower cost than T, since the ordering of edge weights is preserved when we add 1 to each edge weight. This contradicts the assumption that T was an MST of the original graph.

Many people gave an argument based on Kruskal's algorithm: that algorithm finds an MST by repeatedly choosing the minimum-weight edge that does not create a cycle. Incrementing each edge weight by 1 leaves the ordering of edge weights unchanged; therefore Kruskal's algorithm returns the same MST that it returned previously. The problem with this argument is that it applies only to the particular MST returned by (some implementation of) Kruskal's algorithm, not to the collection of all MST's. Thus, we only gave partial credit for this argument.

Some people misinterpreted the question—after pointing out, correctly, that G need not have a unique MST, they said that the MST could change if a randomized algorithm chose a different MST for the new graph than for the original graph. But the question was whether the *collection* of MST's can change. Other people said that the MST changes trivially since the weights of its edges change. But the MST is defined by the collection of edges in it, not the weights of those edges.

The second question was, if P is a shortest path from s to t in G, is P still necessarily the shortest path after every edge weight of G is incremented by 1? The answer is no. Suppose, for example, that G consists of an edge from s to t of weight 1, and edges from s to a, a to b, and b to t each of weight 0. Then the shortest path is $s \to a \to b \to t$, with cost 0. But when we increment each edge weight by 1, the shortest path becomes $s \to t$, with cost 2.

Again, some people misinterpreted the question, and said the answer is no because the cost of the shortest path changes trivially, even if the set of edges in the path does not change.

Problem 3. There has been a lot of hype recently about Star Wars Episode II with the release of the newest theatrical trailer. For this problem, suppose you are managing the construction of billboards on the Anakin Skywalker Memorial Highway, a heavily-travelled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1, x_2, ..., x_n$, each in the interval $[0, M]$ (specified by their position along the highway measured in miles from its western end). If you place a billboard at location x_i, you receive a revenue of $r_i > 0$

You want to place billboards at a subset of the sites in $\{x_1, ..., x_n\}$ so as to maximize your total revenue, subject to the following restrictions:

1. *Environmental Constraint.* You cannot build two billboards within less than 5 miles of one another on the highway.
2. *Boundary Constraint.* You cannot build a billboard within less than 5 miles of the western or eastern ends of the highway.

A subset of sites satisfying these two restrictions will be called *valid*.

Example Suppose $M = 20, n = 4$

$$\{x_1, x_2, x_3, x_4\} = \{6, 8, 12, 14\}$$

and

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$$

Then the optimal solution would be to place billboards at x_1 and x_3 for a total revenue of 10.

Give an algorithm of running time polynomial in n, that takes an instance of this problem as input, and returns the maximum total revenue that can be obtained from any *valid* subset of sites.

Solutions

Dynamic Programming was the correct way to approach this problem. A key observation is that we want to iterate over the *indices* of the elements in $x_1, x_2, ..., x_n$, not the values of $x_1, x_2, ..., x_n$ in order to develop an algorithm polynomial in n. We were searching for an algorithm that ran in time polynomial in n. The intuition behind the algorithm involved recognizing that at each step, you have two choices: build a billboard at x_i or not build a billboard at x_i. If we build a billboard at x_i and receive revenue r_i, then we can't build at billboard in the interval $[x_i - 4, x_i - 1]$. Assume that the values of x_i are real numbers (fractions, etc...) and $x_1, x_2, ..., x_n$ are in an arbitrary order.

Solution 1: An $O(n^2)$ solution.

Let the subproblems be defined: $OPT[i]$ = the maximum revenue one can receive given the constraints and selecting a subset of the sites between x_1 and x_i. Define a new function $\delta(i, j)$ which returns 1 if it is acceptable to build at x_j knowing that a billboard has been built at x_i. δ is defined:

$$\delta(i,j) = \begin{cases} 1 & \text{if } x_j \geq 5, x_j \leq M - 5, x_j \leq x_i - 5 \\ 0 & \text{otherwise} \end{cases}$$

Notice that if we don't build at x_i, we simply want the best thing that happened so far, or $OPT[i-1]$. If we do build at x_i, we want the best thing that happened up to $x_i - 5$ or $OPT[j] + r_i$ where $x_j \leq x_i - 5$. From these definitions, the recurrence is:

$$OPT[i] = \max_{0 \leq j \leq i} \{OPT[i-1], OPT[j] + \delta(i,j) r_i\}$$

The resulting value of $OPT[n]$ is the maximum revenue achievable by building at a subset of sites between x_1 and x_n. The correctness of this algorithm can be verified by the fact that all of the constraints are accounted for by the $\delta(i, j)$ matrix (which takes $O(n^2)$ time to compute). Instead of using $\delta(i, j)$ in your solution, you could have simply put the constraints underneath the max in the recurrence and still had a completely acceptable correct solution. Why does this algorithm return a maximum? At each step, it is calculating the best thing to do looking at all the options. Assume that the solution does something non-optimal at step i. Then it would have chosen to build or not build at x_i, one of which being optimal, the other being non-optimal. But we check which choice results in a better revenue, and we

check this over all previous values. Since no choices are overlooked, the result is that the best subset for $x_1, ..., x_i$ is always chosen at $OPT[i]$.

The running time of this algorithm is $O(n^2)$. Each iteration requires checks i values, and this is iterated over all n $OPT[i]$ values. The result is $\sum_{i=1}^{n} i = O(n^2)$.

Solution 2: An $O(n)$ solution

The $O(n)$ solution is a heuristic modification to the above algorithm. Instead of storing $\delta(i, j)$, we want to store the index of the greatest position $x_k \leq x_i - 5$. Define a new function $\gamma(i)$ that returns such a k if it exists, and 0 otherwise. We can precompute $\gamma(i)$ for $i \leq i \leq n$ in $O(n)$ time before we start solving the problem with the recurrence. This solution assumes that $x_1, x_2, ..., x_n$ is sorted. If it is, the algorithm described takes $O(n)$ time, otherwise it will take $O(n \log n)$ time dominated by sorting the values of $x_1, x_2, ..., x_n$. Simply change the recurrence above to:

$$OPT[i] = \max\{OPT[i-1], OPT[\gamma(i)] + r_i\}$$

The correctness follows from the correctness of solution 1, and the running time is $O(n)$ for precomputing $\gamma(i)$ plus another $O(n)$ for calculating $OPT[i]$ for $1 \leq i \leq n$.

Solution 3: An $O(Mn)$ solution — did not receive full credit

An algorithm that runs in time $O(Mn)$ is considered to be pseudo-polynomial and is not polynomial in n. Such a solution is detailed below and did not receive full credit.

Assume that $x_1, x_2, ..., x_n$ has been sorted in increasing order. Let the subproblems be defined: $OPT[i, m]$ = the maximum revenue one can receive given the constraints, selecting a subset of the sites between x_1 and x_i, and placing the billboards upto mile m from the left end.

$$OPT[i,m] = \begin{cases} OPT[i-1,m] & \text{if } x_i > m \\ OPT[i,m-1] & \text{if } x_i < m \\ \max\{OPT[i-1,m], OPT[i-1,m-5] + r_i\} & \text{if } x_i = m \text{ and } x_i \geq 5 \text{ and } x_i \leq M-5 \end{cases}$$

We initialized the matrix OPT to be zero everywhere. The interesting case is when $x_i = m$. We then have two choices: $OPT[i-1, m]$ corresponds to the case where we don't pick site x_i and the second quantity $OPT[i-1, m-5] + r_i$ is the case where we pick site x_i. Since we have to obey the boundary constraint, the second index in the second quantity becomes $m-5$, i.e., we cannot place any billboard within five miles of x_i. The final answer is stored at $OPT[n, M]$.

Note A lot of students have come up with solutions along the lines of the above ones. For each solution, there's more than one way to write down the recurrence, thus your recurrence does not have to look exactly like the ones above to receive credit.

Common Mistakes The common mistake is the $O(Mn)$ or $O(M)$ solution. Neither of these is a polynomial function in n. If you recall the knapsack problem from lecture 14, the dynamic-programming solution to this problem runs in $O(nB)$. Therefore, this is not a polynomial-time solution to the knapsack problem. One should also notice that when placing billboards in a universe far far away, the solution to this problem is not identical to the knapsack problem. Placing billboards has no upper bound on the number of billboards that may be built—rather the constraints are on which elements we can pick. This is inherently opposite of knapsack—which has constraints on the size of the knapsack but not on the items that are chosen. We'll see later in this class that solving the knapsack problem in polynomial time is conjectured to be a hard problem. However, for this Star Wars problem on the midterm, there's a polynomial time solution as detailed above.

Problem 4. In a country with no antitrust laws, m software companies with values $W_1, ..., W_m$ are merged as follows. The two least valuable companies are merged, thus forming a new list of $m-1$ companies. The value of the merged company is the sum of the values of the two companies that merged (we call this value the *volume of the merge*). this continues until only one company remains.

Let V be the total reported *volume of the merges*. For example if initially we had 4 companies of value (3,3,2,2), the merges yield

$$(3, 3, 2, 2) \rightarrow (4, 3, 3) \rightarrow (6, 4) \rightarrow (10)$$

and $V = 4 + 6 + 10 = 20$

Let us consider a situation with initial values $W_1, \ldots, W_m$, and let V be the total volume of the merges in the sequence of merges described above. Prove that merging the smallest pair at each step results in the minimum possible total volume after all companies are merged (i.e., $V \leq V'$ where V is the result of our "algorithm" and V' is the result of any other algorithm for merging companies).

Solution

The process of merging the companies is similar to the construction of a Huffman encoding. In fact we are creating a tree, as prescribed by Huffman encoding, where:

- the original companies are the encoded symbols, and
- the frequencies, f_i, used to calculate the code are the original values, w_i, of these companies.

We wish to minimize the total sum of all of the mergers, V. In relation to our tree, V is the sum of the weights of all of the non-leaf nodes (created companies). However, we can sum these values in a different way. The weight of one of these created companies is the sum of the weights of the original companies it encompasses. So each original weight, W_i, is present in the number of mergers that company i was involved in. Let us call the number of mergers company i was involved in n_i.

So $V = \Sigma_i W_i \cdot n_i$.

But notice that the number of mergers, n_i, that company i was involved in is exactly the depth of a company i, in the tree (i.e. the number of bits in the encoding of that symbol).

Huffman minimizes the sum $\Sigma_i f_i \cdot d_i$. (**No need to reprove this**!) However, this is precisely V. So our algorithm will minimize V.

Common Mistakes. First off, some errors came from misunderstanding the problem to begin with. Yes, the volume of the final company is always constant, but that's not what we're asking about. We're concerned with the sum of all intermediate companies. Other misinterpretations led people astray as well.

The most common method of attack people tried (after they understood the problem) was a merge by merge analysis (or in some cases, induction on the number of merges). This actually tries to prove something stronger, that the total sum of the reported volume of the mergers is minimized after every merger. We only wish to show that it is minimized at the end. Not surprisingly, this stronger statement ends up requiring some effort.

At first it seems rather straight-forward. Using our method, the volume of the merger we are producing at a particular step is minimal since the two companies we chose were minimal. This implies that another algorithm which chooses any other pair would increase the value reported here. Since V is the sum of the reported volumes and this other algorithm reports a larger volume at this step, the other algorithm will produce a larger V. This is not a logical implication! The problem is that once this other algorithm performs a merge that differs from our algorithm it will proceed with a different set of values. Our argument that our algorithm produces the smallest reported volume no longer holds after this step.

Example: Consider the example $(3, 3, 2, 2)$ as in the exam. Our algorithm first reports 4, then 6, and then 10. Consider an algorithm that first merges the 3's (reporting 6), then merges the 2's (reporting 4), and finally reports 10. This algorithm reports a higher value after the first merger, but that allowed it beat our algorithm on the second merger.

This doesn't mean that the stronger claim doesn't hold, but it's just more of a pain. The key to the problem was to use the already established optimality of Huffman codes.

Problem 5. Given a satisfiable system of linear inequalities

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\leq b_1 \\ &\cdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

we say that an inequality is *forced-equal* in the system if for every $\mathbf{x}$ that satisfies the system, the inequality is satisfied as an equality. (Equivalently, an inequality $\sum_i a_{ij}x_i \leq b_j$ is *not* forced-equal in the system if there is an $\mathbf{x}$ that satisfies the whole system and such that in that inequality the left-hand side is strictly smaller than the right-hand side, that is $\sum_i a_{ij}x_i < b_j$.)

For example in

$$\begin{aligned} x_1 + x_2 &\leq 2 \\ -x_1 - x_2 &\leq -2 \\ x_1 &\leq 1 \\ -x_2 &\leq 0 \end{aligned}$$

the first two inequalities are forced-equal, while the third and fourth are not forced-equal.

Observe that, in any satisfiable system, there is always a solution where all inequalities that are not forced-equal have a left-hand side strictly smaller than the right-hand side. In the above example, we can set $x_1 = -1$ and $x_2 = 3$, so that we have $x_1 < 1$ and $-x_2 < 0$, as well as $x_1 + x_2 = 2$ and $-x_1 - x_2 = -2$.

Given a satisfiable system of linear inequalities, show how to use linear programming to determine which inequalities are forced-equal, and to find a solution where all inequalities that are not forced-equal have a left-hand side strictly smaller than the right-hand side.

Solution

We want to find out which of the constraints are forced equal and then find a solution where the rest of the constraints have a LHS strictly less than the RHS. The easiest way to do this is to introduce a new "slack" variable for each constraint so that the i-th constraint will become:

$$\sum_{j=1}^{n} a_{ij}x_j + t_i \leq b_i$$

We also need some constraints on these "slack" variables so we add the constraints

$$t_i \geq 0, t_i \leq 1, \text{for all } i$$

We need t_i's to be positive but we also need an upper bound on them so that the problem will remain bounded! Then, we use linear programming trying to maximize $\sum_i t_i$, the summation of the "slack" variables. Every constraint i for which $t_i = 0$ is forced-equal, and for every constraint such that $t_i > 0$ we have that $\sum_j a_{ij}x_j + \leq b_i - t_i < b_i$, so the solution found by the linear program is the solution that the problem was asking for.

Many people tried to make a case considering the two dimensional case, arguing that forced equal inequalities will give you a feasible region which will actually be a line. However, plotting the constraints and checking what the feasible region looks like cannot solve the problem, since there is no algorithm that can actually "look" at a picture and decide what it looks like.

There were also a few mistakes by people misinterpreting the meaning of *forced equal*, so please read the problems carefully before trying to answer them. Points were taken off if you used the same "slack" variable for every constraint (if there is even one forced-equal constraint, the single variable will be zero, and you will be unable to distinguish the forced-equal constraint from the others) or if you didn't specify an upper bound (which could make the linear program unbounded).