MATH 409 LECTURE 4
RUNNING TIMES OF ALGORITHMS (CONTINUED)

REKHA THOMAS

Theorem 1. *The simplex method for linear programming is not a polynomial time algorithm.*

Proof. This theorem was proved by Klee and Minty in the seventies by exhibiting an infinite family of linear programs (one in each dimension $n \geq 3$) on which the simplex method takes $2^n - 1$ iterations. In dimension three the Klee-Minty example is:

$$\begin{array}{lll} \max & 100x_1 + 10x_2 + x_3 & \\ \text{s.t.} & x_1 & \leq 1 \\ & 20x_1 + x_2 & \leq 100 \\ & 200x_1 + 20x_2 + x_3 & \leq 10,000 \\ & x_1, x_2, x_3 & \geq 0 \end{array}$$

The feasible region is a squashed cube with optimal vertex $(0, 0, 10000)$. The eight vertices of the feasible region are: $(0, 0, 0), (0, 0, 10000), (0, 100, 0), (0, 100, 8000), (1, 0, 0), (1, 0, 9800), (1, 80, 0), (1, 80, 8200)$. Check that if we start at $(0, 0, 0)$ and run the simplex method with the largest coefficient rule as pivot rule, then the simplex path will visit every vertex of the cube before reaching the optimum. Therefore, in dimension 3 the simplex method takes $2^3 - 1 = 7$ iterations. Since the size of the instance is a polynomial function of n, the running time function is exponential in the size of the instance.

The Klee-Minty linear program in dimension n is:

$$\begin{array}{lll} \max & \sum_{j=1}^{n} 10^{n-j} x_j & \\ \text{s.t.} & \left(2 \sum_{j=1}^{i-1} 10^{i-j} x_j\right) + x_i & \leq 100^{i-1}, i = 1, \ldots, n \\ & x_j \geq 0 & j = 1, \ldots, n. \end{array}$$

The simplex method takes $2^n - 1$ iterations on the n-dimensional member of the family. □

Despite the above result, the simplex method works extremely well in practice. Thus the measure of complexity we are studying has its limitations and may not give a good feel for how an algorithm behaves

Date: April 6, 2010.

on the average (on most problems). It is a **worst case** model that computes the complexity of an algorithm by considering its performance on all instances of a fixed size. A few pathological examples can skew the running time function considerably. Yet, we still get a pretty good indication of the efficiency of an algorithm using this idea.

Interestingly, the simplex method with almost any pivot rule has a Klee-Minty type family of linear programs on which it behaves badly. It is a major open question in discrete optimization whether there is some pivot rule for which the simplex method runs in time polynomial in the input size. On the other hand, linear programs can be solved in polynomial time using algorithms such as *interior point methods* and the *ellipsoid method*. Both these algorithms were discovered around 1980 and use non-linear mathematics. They are quite complicated to explain compared to the simplex method.

Now we look at a more sophisticated model of computational complexity.

(2) **The bit complexity model.** This model for measuring complexity takes into account the sizes of the numbers that need to be dealt with during the algorithm. This makes sense as large numbers require more storage space and we don't get a very good measure of complexity if we assume that all numbers are equal. So we begin by learning how to measure the complexity of an integer $x \in \mathbb{Z}$.

Recall that in the usual decimal notation of a number $x \in \mathbb{Z}$, we record the number as sums of powers of 10 starting with the highest power of 10 in x and working our way down to 10^0. For instance, $101 = 1{\cdot}10^2{+}0{\cdot}10{+}1{\cdot}10^0$. The digits we see in 101 are the coefficients of the powers of 10 present in the above decomposition gotten by greedily recording the largest power of 10 in 101, then the largest power of 10 in the rest etc. Similarly we can compute the *binary representation* of $x \in \mathbb{Z}$ by doing the same as above but with powers of 2. The largest power of 2 in 101 is $2^6 = 64$. Thus $101 = 2^6 + 37$. The largest power of 2 in 37 is 2^5 and so we have $101 = 2^6 + 2^5 + 5 = 2^6 + 2^5 + 2^2 + 2^0$. If we include all powers of 2 in this decomposition starting at 2^6, then

$$101 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 \cdot 2^0.$$

The binary representation of 101 is the vector that records just the coefficients: 1100101 starting with 2^6 and working down to 2^0. The number of **bits** needed in this binary representation is $7 = \lceil log_2(101 + 1) \rceil$. We also use one bit to record the sign of 101 and thus need $8 = 1 + \lceil log_2(101 + 1) \rceil$ bits in the **binary encoding** of 101.

Lemma 2. *The number of bits needed to encode the binary representation of $x \in \mathbb{Z}$ is $1 + \lceil log_2(|x| + 1) \rceil$.*

If $x = \frac{p}{q}$, $p, q \in \mathbb{Z}$, is a rational number then its bit complexity is just the sum of the bit complexity of p and q. Real numbers are not dealt with in this model. This is no huge loss since if we are inputting a problem into a computer we always assume that the data is rational since the computer has only a finite amount of precision.

Next we need to analyze the complexity of an algorithm which requires us to account for the complexity of all elementary operations such as addition, multiplication etc. Adding two numbers with $O(M)$ bits takes $O(M)$ additions and $O(M)$ carry overs and is thus an $O(M)$ operation. Subtraction is the same. How about multiplying two numbers with $O(M)$ digits? Just as in decimal notation, this takes $O(M^2)$ digit-to-digit multiplications, $O(M)$ additions and $O(M)$ carry overs. So this is an $O(M^2)$ operation. Comparing two numbers of bit complexity M is an $O(M)$ calculation. How about taking the square root of a number of bit complexity M ?

Exercise 3. (i) Compute the binary representation and then bit complexity of 1025, 1024 and -1023.
(ii) Add the binary numbers 11100101 and 11001. Check your answer by converting everything into decimal notation.
(iii) Multiply the numbers 213 and 425 in binary form and give the answer in binary form.

Example 4. Example from Lecture 3 continued.
Recall the problem from Lecture 3 for computing the Euclidean distance between two points P and Q. Let us try to analyze the complexity of this calculation in the bit complexity model with the assumption that $P, Q \in \mathbb{Z}^n$. Let M be the maximum bit complexity of the coordinates of P and Q. Then the size of a problem instance is $2nM$ since there are $2n$ numbers $p_1, \ldots, p_n, q_1, \ldots, q_n$, each of size at most M, as input. The algorithm takes the following steps:

- Compute $(p_i - q_i)$ which is an $O(M)$ calculation. The resulting number has complexity $O(M)$.
- Compute $(p_i - q_i)^2$ which is an $O(M^2)$ calculation and the resulting number is $O(2M) = O(M)$.
- The above calculations need to be done n times which take $O(n(M^2 + M)) = O(nM^2)$ work.
- Now we add the n numbers $(p_i - q_i)^2$ which is n additions of $O(M)$ numbers. This takes M additions each of which requires

adding n digits together with at most $O(M)$ carry overs. This takes $O(nM + M) = O(nM)$ work. How big is the resulting number? Let r be a number of bit complexity M. Then $log_2(r) \sim M$. This implies that $log_2(nr) \sim log_2(n) + M$. The sum of n numbers, each of bit complexity M is therefore roughly $O(log_2(n) + M)$ in complexity.

- Finally we need a squareroot of this final sum. This is not an easy operation to calculate the complexity of. In particular, the square root of an integer could be an irrational number which cannot even be defined in this model. So we abandon the analysis at this point and hope that everywhere we need a distance calculation between two points P and Q (as in the next example) we will be satisfied with computing $\sum_{i=1}^{n}(p_i - q_i)^2$. This is surely enough if we only need to compare distances.

Example 5. The nearest neighbor algorithm for the TSP. The data for this problem is the coordinates (x_i, y_i) for cities $v_1, \ldots, v_n$. Thus problem size in the bit complexity model is $2nM$ if we assume that x_i, y_i are integers with maximum bit complexity M.

The algorithm can be executed by a for-loop.

The initialization step: The algorithm orders the n cities in a list, records the n coordinates and marks city v_1 with 1 and all the others with 0. This takes $O(2nM + n)$ steps: $O(2nM)$ to list the coordinates and $O(n)$ steps to mark the cities.

The first step: Set $c_{min} := c_{12}$, the distance between cities v_1 and v_2. Then for each $j = 3, \ldots, n$ compute the distance c_{1j} from city 1 to city j and replace c_{min} by c_{ij} and record this j if $c_{ij} < c_{min}$. Mark the city at minimum distance from v_1 with mark 1.

The kth step: Let v be the city marked 1 in the previous step of the algorithm and c_{min} be the distance between v and the first city in the list marked 0. Run through all remaining cities marked 0, calculating their distance from v and replacing c_{min} by this distance and recording its index j if it is the least distance encountered thus far. Mark the city that contibutes to c_{min} with 1 and go to the next step in the loop.

This kth step takes n calculations to check the city's mark (whether it's 0 or 1). Then at most $n - 1$ distance calculations of the form $(x_i - x_j)^2 + (y_i - y_j)^2$ each of which takes 3 additions, 2 multiplications and 1 comparison. The additions are $O(M)$ and the multiplications $O(M^2)$. (You need to convince yourself that adding two numbers of bit complexity M results in a number of bit complexity $O(M)$ and similarly, multiplying two such numbers results in a number of complexity $O(M)$.) Thus the kth step takes $n + (n - 1)(2M^2 + 4M)$

calculations. Putting it all together the algorithm has complexity $O(2nM+n)+(n-1)O(n+(n-1)M^2) = O(n^2M^2)$. Thus this algorithm is polynomial time in the bit complexity model. Check that it is also polynomial time in the arithmetic model.

Exercise 6. (i) Calculate the running time function for Gaussian elimination on a $m \times n$ integer matrix A in both the arithmetic model and the bit complexity model.
(ii) Does this differ from the complexity of Gauss-Jordan elimination? (Gauss-Jordan elimination produces an identity matrix of rank equal to rank(A) in the top left while Gaussian elimination produces an upper triangular matrix of rank rank(A) in the top left.)
(iii) Use the above to calculate the complexity of computing the inverse of a non-singular matrix.
(iv) Write down the most efficient algorithm you know for computing the rank of a matrix.