Integer Points in Polyhedra Gennady Shmonin

Some basic algorithms

Preliminary

Almost every complex algorithm relies on the fact that some intermediate quantities (the determinant of a matrix, the greatest common divisor of two integers, a basis of a linear subspace, etc.) can be computed efficiently. We shall discuss these basic "building blocks" on the lectures or during the exercise sessions, whenever it becomes necessary. On the other hand, in order to keep the other lecture notes "clean", these basic algorithms will be collected in this note.

1 Arithmetic operations

Every algorithm that intends to operate with arbitrarily large integers must take care about the very basic operations over such integers: addition, subtraction, multiplication, division, and comparison. Let a and b be two integers, given in binary encoding. Recall that just in order to read these numbers, an algorithm needs $O(\text{size}(a)+\text{size}(b))$ time; thus it is somewhat hopeless to search for the algorithms running faster than $O(\text{size}(a)+\text{size}(b))$. Using the "school method", it is easy to see that this time is actually attained for *addition* and *subtraction* as well as for *comparison*.

On the other hand, the "school method" for *multiplication* gives $O(\text{size}(a)\cdot\text{size}(b))$; of course, this is polynomial, and therefore, is already fine for our purposes. But it is worth mentioning that such a seemingly simple question—an algorithm for multiplication of two integers—is not completely resolved yet. Thus, the Schönhage–Strassen algorithm runs in time $O(n\cdot\log n\cdot\log\log n)$, which was recently improved by Fürer to $O(n\cdot\log n\cdot 2^{\log^* n})$ (where $\log^* n$ stands for the iterated logarithm, i.e., the number of times the logarithm must be taken until the result is less than or equal to 1). However, the conjectured $O(n\cdot\log n)$ running time remains an open problem.

Of particular interest for us is *division with remainder*. It is well-known that, given two integers a and b, with $b>0$, there are unique numbers q and r such that $a = bq + r$ and $0 \leqslant r < b$. These integers q and r can be found by the "school method" for division, which—after some extra care on implementation issues—runs in $O(\text{size}(b)\cdot\text{size}(q))$ time (intuitively, we look $\text{size}(q)$ times at the portions of $\text{size}(b)$ bits in the bit representation of a. Since $\log q = \log a - \log b$, we have a bound $\text{size}(q) \leqslant \text{size}(a) - \text{size}(b) + 1$.

2 Gram–Schmidt orthogonalization

Let $b_1, b_2, \dots, b_k$ be linearly independent vectors in the Euclidean space $\mathbb{R}^n$, equipped with the standard scalar product

$$\langle x, y\rangle := x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

for any two vectors x and y in $\mathbb{R}^n$. These vectors $b_1, b_2, \ldots, b_k$ span the linear subspace of $\mathbb{R}^n$ of dimension k,

$$L = \left\{ \sum_{i=1}^{k} \lambda_i b_i : \lambda_i \in \mathbb{R},\ i = 1, 2, \ldots, k \right\}.$$

The well-known *Gram–Schmidt orthogonalization* procedure computes a basis $b_1^*, b_2^*, \ldots, b_k^*$ of L such that $\langle b_i^*, b_j^* \rangle = 0$ for $i \neq j$. The procedure can be described by the following formulae:

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{ij} b_j^*, \qquad \mu_{ij} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \tag{1}$$

for $i = 1, 2, \ldots, k$. It is easy to see that the vectors $b_1^*, b_2^*, \ldots, b_k^*$ indeed form an orthogonal basis and span L. Moreover, at every step i, the vectors $b_1^*, b_2^*, \ldots, b_i^*$ span the same linear subspace L_i as the vectors $b_1, b_2, \ldots, b_i$, and the vector b_{i+1}^* can be viewed as the orthogonal line from b_{i+1} to L_i. It is also clear that the whole procedure can be carried out in polynomial time. If we denote $B^* = [b_1^*, b_2^*, \ldots, b_k^*]$, then (1) immediately implies that

$$B = B^* T,$$

where T is a square upper-triangular rational matrix with ones on the main diagonal:

$$T = \begin{bmatrix} 1 & \mu_{21} & \mu_{31} & \ldots & \mu_{n1} \\ 0 & 1 & \mu_{32} & \ldots & \mu_{n2} \\ 0 & 0 & 1 & \ldots & \mu_{n3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}.$$

Now, consider a parallelepiped

$$F(B) := \{ Bx : x \in [0,1)^k \}.$$

Such a parallelepiped appears to be an important object when studying lattices. Let us compute the relative volume of this parallelepiped, i.e., the volume of $F(B)$ in the linear subspace spanned by B. It is clear that

$$\operatorname{vol}(F(B)) = \prod_{i=1}^{k} \|b_i^*\|_2 = \sqrt{\det((B^*)^{\mathrm{T}} B^*)} = \sqrt{\det(T^{\mathrm{T}} (B^*)^{\mathrm{T}} B^* T)} = \sqrt{\det(B^{\mathrm{T}} B)}.$$

Since $\|b_i^*\|_2 \leqslant \|b_i\|_2$, this implies

$$\sqrt{\det(B^{\mathrm{T}} B)} = \prod_{i=1}^{k} \|b_i^*\|_2 \leqslant \prod_{i=1}^{k} \|b_i\|_2,$$

whence the important *Hadamard inequality* follows: for any square matrix $B = [b_1, b_2, \ldots, b_n]$,

$$|\det(B)| \leqslant \prod_{i=1}^{n} \|b_i\|_2 \tag{2}$$

We demonstrate one application of the Hadamard inequality (2) for algorithms; namely, we show that the determinant of a given rational matrix has a polynomial size. This fact is frequently used to show that in a certain algorithm the sizes of intermediate numbers "do not grow too much". Let $B = [b_1, b_2, \ldots, b_n]$ be a square rational matrix and denote $\Delta = \det(B)$. By definition, $\operatorname{size}(\Delta) = \lceil \log(|\Delta| + 1) \rceil + 1$. Assuming that there are no zero columns in B, the Hadamard inequality implies that

$$\log(|\Delta|) \leqslant \sum_{i=1}^{n} \log(\|b_i\|_2) \leqslant \frac{1}{2} n \log(n\beta),$$

where β denotes the largest (in absolute value) entry of B. This is certainly polynomial in the size of matrix B.

3 Euclidean algorithm

Given two non-negative integers a and b, their greatest common divisor $\gcd(a, b)$ can be computed using the well-known *Euclidean algorithm*. On each step, it applies division with remainder to find numbers q and r such that $a = bq + r$ and $r \leqslant 0 < b$, and reassigns $a := b$ and $b := r$. The algorithm terminates when $r = 0$; in this case, b stores the required greatest common divisor. The sequence of computations can be represented by the following equations:

$$\begin{aligned}
a &= r_0, && \\
b &= r_1, && \\
r_0 &= r_1 q_1 + r_2, && 0 \leqslant r_2 < r_1, \\
&\ldots && \ldots \\
r_{k-2} &= r_{k-1} q_{k-1} + r_k, && 0 \leqslant r_k < r_{k-1}, \\
r_{k-1} &= r_k q_k. &&
\end{aligned}$$

Clearly, the algorithm always terminates, as numbers decrease at each iteration. Correctness of the algorithm easily follows from the fact that on every step, each divisor of r_{i-2} and r_{i-1} is also a divisor of r_i.

Theorem 1. *The running time of the Euclidean algorithm is* $O(\operatorname{size}(a) \cdot \operatorname{size}(b))$.

Proof. First, we obtain an upper bound on the number of iterations k the algorithm performs. Since $q_i \geqslant 1$ for all i, we have $r_{i-2} \geqslant r_{i-1} + r_i$. In other words, on every second iteration, the size of a smaller number decreases by a factor of 2, $r_i \leqslant r_{i-2}/2$. This implies $k \leqslant 2 \cdot \operatorname{size}(b)$.

Now, the running time of the algorithm is bounded by

$$\sum_{i=1}^{k} C \cdot \operatorname{size}(r_i) \cdot \operatorname{size}(q_i),$$

where $C \cdot \operatorname{size}(r_i) \operatorname{size}(q_i)$ is the bound on the time needed to divide r_{i-1} by r_i. Since $r_i \leqslant b$ for

$i \geqslant 1$, we have

$$\begin{aligned}\sum_{i=1}^{k} C\cdot \mathrm{size}(r_i)\cdot \mathrm{size}(q_i) &\leqslant C\cdot \mathrm{size}(b)\cdot \sum_{i=1}^{k}\mathrm{size}(q_i)\\ &\leqslant C\cdot \mathrm{size}(b)\cdot \sum_{i=1}^{k}(\mathrm{size}(r_{i-1})-\mathrm{size}(r_i)+1)\\ &= C\cdot \mathrm{size}(b)\cdot(\mathrm{size}(r_0)-\mathrm{size}(r_k)+k)\\ &\leqslant C\cdot \mathrm{size}(b)\cdot(\mathrm{size}(a)+2\cdot \mathrm{size}(b))\\ &= O(\mathrm{size}(a)\cdot \mathrm{size}(b)).\end{aligned}$$

□

4 Gaussian elimination

The *Gaussian elimination* transforms a given matrix $A \in \mathbb{R}^{m\times n}$ into the form

$$\begin{bmatrix} B & C\\ 0 & 0\end{bmatrix}, \tag{3}$$

where B is a non-singular upper triangular matrix, by applying a sequence of *elementary row operations*:

(1) swap two rows,

(2) swap two columns,

(3) add a multiple of one row to another row.

The algorithm constructs a sequence of matrices $A_1, A_2, \ldots$, of the form

$$A_k = \begin{bmatrix} B_k & C_k\\ 0 & D_k\end{bmatrix}, \tag{4}$$

where B_k is a non-singular upper-triangular matrix of order k. The matrix A_{k+1} is then computed as follows. After permuting rows and columns, we may assume (unless we are done) that the element at position $(1,1)$, say d_{11}, in matrix D is non-zero (this element will be a *pivot element*). Add appropriate multiples of the first row in D to the other rows in D such that d_{11} becomes the only non-zero element in the first column of D. This process continues until the matrix is brought into the form (3).

A similar process can be applied further to bring the matrix (3) into the form

$$\begin{bmatrix} B' & C\\ 0 & 0\end{bmatrix}, \tag{5}$$

where B' is a non-singular diagonal matrix. Also, this is done by adding appropriate multiples of the k-th row to the rows $1, 2, \ldots, k-1$ for $k = r, r-1, \ldots, 2$, where r denotes the order of B'.

The number of arithmetic operations needed to transform a rational matrix $A \in \mathbb{Q}^{m\times n}$ into the form (3) is $O(m^2 n)$. However, this is not yet sufficient to claim that the algorithm is polynomial! We still need to check that all numbers appearing during the execution of the algorithm have polynomial size. In fact, this is true and we shall prove it very soon. But in order to illustrate that the contrary can actually happen, consider a modification of the Gaussian elimination algorithm, in which we wish to keep all intermediate matrices $A_1, A_2, \dots$ integral. This can be done as follows. Let d_{ij} denote the element at position (i, j) in matrix D_k. Then we can eliminate d_{i1} *by cross-multiplication*, i.e., replacing each d_{ij} by $d_{ij}d_{11} - d_{i1}d_{1j}$. Now, run the algorithm on the following matrix

$$\begin{bmatrix} 2 & 0 & 0 & \dots & 0 & 0 \\ 1 & 2 & 0 & \dots & 0 & 0 \\ 1 & 1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 1 & \dots & 1 & 2 \end{bmatrix}$$

(2's on the main diagonal, 1's below the main diagonal and 0's above the main diagonal). If we always choose elements on the main diagonal to be pivoting elements, then on the k-th step, matrix D_k has the numbers 2^{2^k} on the main diagonal, and the size of these numbers, $\Theta(2^k)$, grows exponentially in the size of the input. However, this does not happen if we operate on rational data, as was described initially.

Before we proceed with the proof, observe that elementary row operations (1) and (2) change the determinant of a matrix to its inverse. On the other hand, operation (3) does not change neither the determinant of the matrix, nor the determinants of its submatrices.

Theorem 2. *Gaussian elimination runs in polynomial time.*

Proof. It only remains to show that all entries in the intermediate matrices (4) have polynomial size. Without loss of generality, we assume that during the execution of the algorithm, we do not permute rows and columns (we apply all those permutations beforehand). First, observe that we do not need to consider the entries of C_k, as they appear at some earlier step k' as the entries of $D_{k'}$. Now, let d_{ij} be the element at position (i, j) in D_k. It is easy to see that

$$d_{ij} = \frac{\det(B'_k)}{\det(B_k)}, \tag{6}$$

where B'_k is a submatrix of A_k composed of rows $1, 2, \dots, k, i$ and columns $1, 2, \dots, k, j$ of A. Furthermore, since we did not permute rows and columns, i.e., applied only the third elementary operation (adding a multiple of one row to another), which does not change the determinant, the determinants of submatrices B_k and B'_k are exactly the same as they were initially in matrix A. Thus,

$$d_{ij} = \frac{\det\left(A^{\{1,2,\dots,k,j\}}_{\{1,2,\dots,k,i\}}\right)}{\det\left(A^{\{1,2,\dots,k\}}_{\{1,2,\dots,k\}}\right)},$$

where A^J_I denotes the submatrix of A induced by rows in I and columns in J. But the size of both numerator and denominator is bounded by a polynomial in the size of matrix A. Thus, if we keep all intermediate fractions in irreducible form (which we can do by exploiting the Euclidean algorithm), these fractions will have polynomial size. □

This proves that the Gaussian elimination algorithm is weakly polynomial, as the running time of the Euclidean algorithm depends on the sizes of the entries in A. It seems that the Euclidean algorithm is rather necessary to run Gaussian elimination in polynomial time: if we never reduce the occurring fractions, we may end up with essentially the same problem as for cross-multiplication Gaussian elimination algorithm. However, a slight modification allows to make the algorithm strongly polynomial.

Suppose that in the beginning A is an integral matrix. From (6) we see that at iteration k, we may write all entries in the matrix C_k with the same denominator, which is the product of (the numerators of) the diagonal entries in B_k. It is easy to see that we can keep the numbers in this form throughout all iterations, and therefore, avoid the Euclidean algorithm.

The Gaussian elimination method helps to solve many important problems in polynomial time.

Corollary 2a. *The following problems are solvable in polynomial time:*

(a) computing the determinant of a rational matrix;

(b) computing the rank of a rational matrix;

(c) computing the inverse of a square non-singular rational matrix;

(d) testing if vectors are linearly independent;

(e) solving the system of rational equations.