Friedrich Eisenbrand

Linear Optimization

These are notes of my course Discrete Optimization. They are constantly updated.

February 17, 2017

Contents

1 Algorithms and running time analysis 11
1.1 Analysis of Gaussian elimination 16

2 Linear programming 15
2.1 Softdrink production 15
2.2 Proving optimality 16
2.3 Linear Programs 18
2.4 Fitting a line 19
2.5 Linear Programming solvers and modeling languages 20
2.6 Linear programming for longer OLED-lifetime 21

3 Polyhedra and convex sets 27
3.1 Extreme points and vertices 28
3.2 Linear, affine, conic and convex hulls 30
3.3 Radon's lemma and Carathéodory's theorem 33
3.4 Separation theorem and Farkas' lemma 35
3.5 Decomposition theorem for polyhedra 36
3.5.1 Faces 39

4 The simplex method 45
4.1 Adjacent vertices 45
4.2 Bases, feasible bases and vertices 46
4.3 Moving to an improving vertex 48
4.4 Termination in the degenerate case 49
4.5 Finding an initial basic feasible solution 50
4.6 Removing degeneracy by perturbation 51
4.7 One iteration of the simplex algorithm 54

5 Polyhedra and Integer Programming 57
5.1 Decomposition theorem for polyhedra 58
5.2 Faces 60

5.2.1 Integer Programming 62
5.2.2 Integral Polyhedra 64
5.3 Applications of total unimodularity 65
5.3.1 Further applications of polyhedral theory 67
5.3.2 The matching polytope 67

References 73

6 Duality 75
6.1 Zero sum games 77
6.2 A proof of the duality theorem via Farkas' lemma 80

7 Integer Programming 83
7.1 Integral Polyhedra 85
7.2 Applications of total unimodularity 87
7.2.1 Bipartite matching 87
7.2.2 Bipartite vertex cover 88
7.2.3 Flows 89
7.2.4 Doubly stochastic matrices 89
7.3 The matching polytope 90

8 Paths, cycles and flows in graphs 95
8.1 Graphs 95
8.2 Representing graphs and computing the distance of two nodes 96
8.2.1 Breadth-first search 96
8.3 Shortest Paths 98
8.4 Maximum $s-t$-flows 102
8.5 Minimum cost network flows, MCNFP 106
8.6 Computing a minimum cost-to-profit ratio cycle 116
8.6.1 Parametric search 118

9 The ellipsoid method 121
9.1 The method 124
9.1.1 The separation problem 125
9.2 How to start and when to stop 126
9.3 The boundedness and full-dimensionality condition 128
9.3.1 Boundedness 129
9.3.2 Full-dimensionality 129
9.4 The ellipsoid method for optimization 130
9.5 Numerical issues 130

10 Primal-Dual algorithm 131
10.1 Graphs and Matchings 131
10.2 Matching problems 132
10.2.1 The maximum cardinality matching problem 133
10.2.2 The maximum weight matching problem 134

References 143

Preface

A central topic in linear algebra is the theory around linear equations

$$A\,x = b \tag{0.1}$$

where $A \in \mathbb{R}^{m\times n}$ is a given matrix and $b \in \mathbb{R}^m$ is a given vector. Every student of mathematics learns learns to appreciate *Gaussian elimination* which transforms the system (0.1) into an equivalent system

$$A'x = b'$$

that is in row-echelon form. This means that $A' \in \mathbb{R}^{m\times n}$ is such that for each $i \in \{1, \dots, m-1\}$

$$\min\{j : a'_{ij} \neq 0\} < \min\{j : a'_{i+1j} \neq 0\},$$

see Figure 0.1.

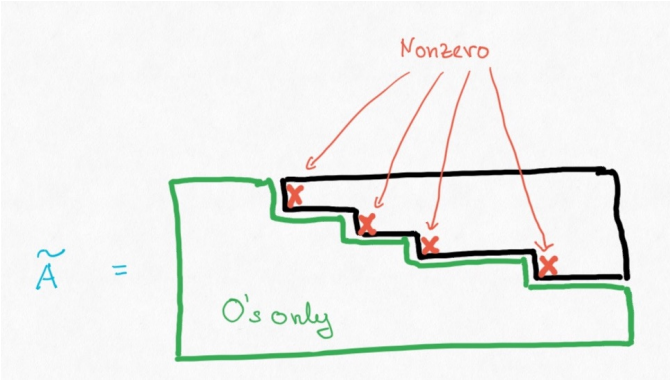

Fig. 0.1: A schematic image of a matrix in row-echelon form

The system (0.1) has a solution if and only if b' has only zero's in those components that correspond to the zero-rows of A' and a solution is readily computed. In fact, Gaussian elimination also provides an invertible matrix $Q \in \mathbb{R}^{m\times m}$ such that $Q \cdot A = A'$ and $Q \cdot b = b'$. If the i-th component of b' is nonzero and the i-th row of A' consists of zeros only, then with q being the column vector corresponding to the i-th row of Q one has

$$q^T A = 0 \text{ and } q^T b \neq 0.$$

Thus there is a convenient *certificate* of the fact that (0.1) is not solvable.

Theorem 0.1. *A linear system* (0.1) *is not solvable if and only if there exists a vector* $q \in \mathbb{R}^m$ *such that*

$$q^T A = 0 \text{ and } q^T b \neq 0.$$

In this course, we will now also deal with systems of *linear inequalities*

$$A\,x \leqslant b, \tag{0.2}$$

where $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. A *solution* to such a system is a vector $x^* \in \mathbb{R}^m$ such that

$$\text{for each } i = 1, \ldots, m \quad : \quad a_{i1}x_1^* + \cdots + a_{in}x_n^* \leqslant b_i. \tag{0.3}$$

If a solution exists, the system (0.2) is called *feasible* otherwise it is called *infeasible*. We will ask analogous questions for systems of linear inequalities as we did for linear equations. Can we efficiently find a solution of (0.2)? Is there a simple certificate to convince somebody of the infeasibility of (0.2)?

To shed a bit of light on this second question, let us consider a vector $\lambda \in \mathbb{R}^m_{\geqslant 0}$. If x^* is a solution, then x^* also satisfies the inequality

$$\lambda^T A\,x \leqslant \lambda^T b. \tag{0.4}$$

If there exists a $\lambda \in \mathbb{R}^m_{\geqslant 0}$ such that

$$\lambda^T A = 0^T \text{ and } \lambda^T b = -1, \tag{0.5}$$

then this λ *certifies* that (0.2) is infeasible. In the case of infeasibility, can such a $\lambda \geqslant 0$ always be found? Among the many results presented in this course, we will show that the answer is "yes".

Theorem 0.2 (Farkas' lemma). *A system of linear inequalities* (0.2) *is infeasible if and only if there exists a* $\lambda \in \mathbb{R}^m_{\geqslant 0}$ *such that*

$$\lambda^T A = 0 \text{ and } \lambda^T b = -1.$$

Central to this course is the *linear programming* or *linear optimization* problem. There, one is given a system of linear inequalities (0.2) together with a linear objective function $f(x) = c^T x$ for some fixed $c \in \mathbb{R}^n$. The task is to solve

$$\max\{c^T x : x \in \mathbb{R}^n,\, Ax \leqslant b\}, \tag{0.6}$$

or in other words, to find an $x^* \in \mathbb{R}^n$ that is a solution of (0.2) such that $c^T x^* \geqslant c^T y^*$ for each solution y^* of (0.2), if such an x^* exists. The linear optimization problem is one of the most important types of mathematical optimization problems. It has numerous applications in science and engineering and, in particular, modern fields like *machine learning*.

In this course, we will learn the theory of linear optimization and develop *efficient algorithms* to solve linear programming problems. This means that we do not content ourselves with an algorithm that is correct, but we want this algorithm to be capable to solve large scale problems within a short time.

Efficient algorithms is a subfield of computational mathematics on its own. Before we begin with our considerations on linear programming, we will have to learn the basics of it.

Exercises

1) Provide a certificate as in Theorem 0.1 of the unsolvability of the linear equation

$$\begin{pmatrix} 2 & 1 & 0 \\ 5 & 4 & 1 \\ 7 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

2) Show the "if" direction of the Farkas' lemma (Theorem 0.2).

Chapter 1
Algorithms and running time analysis

An *algorithm* executes a set of *instructions* used in common programming languages like arithmetic operations, comparisons or read/write instructions. The sequence of these instructions is controlled by *loops and conditionals* like `if, while, for` etc.

Each of these instructions requires *time*. The *running time* of an algorithm is the *number of instructions* that the algorithm performs. This number depends on the *input* of the algorithm.

Example 1.1. Consider the following algorithm to compute the product of two $n \times n$ matrices.

for $i = 1, \dots, n$
  for $j = 1, \dots, n$
    $c_{ij} := 0$
    for $k = 1, \dots, n$
      $c_{ij} := c_{ij} + a_{ik} \cdot a_{kj}$

The number of additions is $n^2 \cdot (n-1)$ and the number of multiplications is n^3. The number of store-instructions is $n^2 \cdot (n+1)$. The number of read-instructions is of similar magnitude.

The above example shows that an *exact counting* is sometimes tedious. Looking at the algorithm however, you quickly agree that there exists *some constant* $c \in \mathbb{R}_{>0}$ such that the algorithm performs at most $c \cdot m \cdot n \cdot l$ instructions.

In the analysis of algorithms, one does usually not care so much about the constant c above in the beginning. There are sub-fields of algorithms where this constant however matters. Especially for algorithms on large data sets, where access to external data is costly. However, this is another story that does not concern us here. When we analyze algorithms, we are interested in the *asymptotic running time*.

Definition 1.1 (O-notation).
Let $T, f : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be functions. We say

- $T(n) = O(f(n))$, if there exist positive constants $n_o \in \mathbb{N}$ and $c \in \mathbb{R}_{>0}$ with

$$T(n) \leqslant c \cdot f(n) \text{ for all } n \geqslant n_0.$$

- $T(n) = \Omega(f(n))$, if there exist constants $n_o \in \mathbb{N}$ and $c \in \mathbb{R}_{>0}$ with

$$T(n) \geqslant c \cdot f(n) \text{ for all } n \geqslant n_0.$$

- $T(n) = \Theta(f(n))$ if

$$T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)).$$

Example 1.2. The function $T(n) = 2n^2+3n+1$ is in $O(n^2)$, since for all $n \geqslant 1$ one has $2n^2 + 3n + 1 \leqslant 6n^2$. Here $n_0 = 1$ and $c = 6$. Similarly $T(n) = \Omega(n^2)$, since for each $n \geqslant 1$ one has $2n^2 + 3n + 1 \geqslant n^2$. Thus $T(n)$ is in $\Theta(n^2)$.

We measure the running time of algorithms in terms of the *length of the input*. The matrices A and B that are the input of the matrix-multiplication algorithm of Example 1.1 consist of n^2 numbers each. The algorithm runs in time $O(n^3 = (n^2)^{3/2})$.

What does it mean for an algorithm to be efficient? For us, this will mean that it runs in polynomial time. As a first definition of polynomial time algorithm, we say that an algorithm runs in *polynomial time*, if there exists a constant k such that the algorithm runs in time $O(n^k)$, where n is the length of the input of the algorithm.

However, we recall the *binary* representation of natural numbers $n \in \mathbb{N}$. A sequence of bits $a_0, \ldots, a_{k-1}$ with $a_j \in \{0, 1\}$ for $0 \leqslant j \leqslant k-1$ represents the number

$$\sum_{j=0}^{k-1} a_j \cdot 2^j.$$

Conversely, each positive natural number $n \geqslant 1$ has the binary representation that is found recursively by the following process. If $n = 1$, then its representation is $a_0 = 1$. If $n > 1$ and is even, then the sequence representing n is

$$0, b_0, \ldots, b_{k-1},$$

where $b_0, \ldots, b_{k-1}$ is the representation of $n/2$. If $n > 1$ and n is odd, then the sequence representing n is

$$1, b_0, \ldots, b_{k-1},$$

where $b_0, \ldots, b_{k-1}$ is the representation of $\lfloor n/2 \rfloor$. This creates a representation with leading bit one, i.e. $a_{k-1} = 1$. By deleting *leading zeros*, i.e., ensuring $a_{k-1} = 1$, the representation of a natural positive number is unique, (see exercise 1.2).

Example 1.3. This example will make us revise the our first definition of a polynomial-time algorithm. The input of this algorithm is a list of characters. Let us say that the list has n elements.

Input: A list L or characters
$s := 2$
for $c \in L$:
    $s := s \cdot s$
return s

Then clearly, the algorithm carries out a polynomial, even linear, number of operations in the input, if we consider an arithmetic operation also as a basic operation that can be carried out in constant time. However, the algorithm squares 2 repeatedly, n times to be precise. Thus, at the end, the variable s holds the number 2^{2^n}. The number of *bits* in the binary representation that the return value is 2^n which is *exponential* in the input length.

Definition 1.2. The *size* of an integer x is $\text{size}(x) = \lceil \log(|x| + 1) \rceil$ and for $x \in \mathbb{Q}$, $\text{size}(x) = \text{size}(p) + \text{size}(q)$, where $x = p/q$ with $p, q \in \mathbb{Z}$, $q \geqslant 1$ and $\gcd(p, q) = 1$.

Thus the size of a number is asymptotically equal to the number of bits that are needed to store the number. We now provide the definition of what a polynomial-time algorithm is.

Definition 1.3. An is *polynomial time*, if there exists a constant k such that the algorithm performs $O(n^k)$ operations on rational numbers whose size is bounded by $O(n^k)$. Here n is the number of bits that encode the input of the algorithm. We say that the algorithm runs in time $O(n^k)$.

We now use this definition to analyze the famous *Euclidean* algorithm that computes the *greatest common divisor* of two integers.

For $a, b \in \mathbb{Z}$, $b \neq 0$ we say b *divides* a if there exists an $x \in \mathbb{Z}$ such that $a = b \cdot x$. We write $b \mid a$. For $a, b, c \in \mathbb{Z}$, if $c \mid a$ and $c \mid b$, then c is a *common divisor* of a and b. If at least one of the two integers a and b is non-zero, then there exists a *greatest common divisor* of a and b. It is denoted by $\gcd(a, b)$.

How do we compute the greatest common divisor efficiently? The following is called *division with remainder*. For $a, b \in \mathbb{Z}$ with $b > 0$ there exist unique integers $q, r \in \mathbb{Z}$ with

$$a = q \cdot b + r, \text{ and } 0 \leqslant r < b.$$

Now clearly, for $a, b \in \mathbb{Z}$ with $b > 0$ and $q, r \in \mathbb{Z}$ as above one has $\gcd(a, b) = \gcd(b, r)$. This gives rise to the famous *Euclidean algorithm.*

Algorithm 1.1 (Euclidean algorithm).

Input: Integers $a \geqslant b \geqslant 0$ not both equal to zero

Output: The greatest common divisor $\gcd(a, b)$
if $(b = 0)$ **return** a
else
Compute $q, r \in \mathbb{N}$ with $b > r \geqslant 0$ and $a = q \cdot b + r$
(division with remainder)
return $\gcd(b, r)$

Theorem 1.1. *The Euclidean algorithm runs in time $O(n)$.*

Proof. Suppose that a and b have at most n bits each. Clearly, the numbers in the course of the algorithm have at most n bits. Furthermore, if $a \geqslant b$, then $r \leqslant a/2$, where r is the remainder of the division of a by b. Thus each second iteration, the first parameter of the input has one bit less. Thus the number of operations is bounded by $O(n)$.

Example 1.4. The *determinant* of a matrix $A \in \mathbb{R}^{n \times n}$ can be computed by the recursive formula

$$\det(A) = \sum_{i=1}^{n} (-1)^{1+j} a_{1j} \det(A_{1j}),$$

where A_{1j} is the $(n-1) \times (n-1)$ matrix that is obtained from A by deleting its first row and j-th column. This yields the following algorithm.

Input: $A \in \mathbb{Q}^{n \times n}$
Output: $\det(A)$
if $(n = 1)$
return a_{11}
else
$d := 0$
for $j = 1, \ldots, n$
$d := (-1)^{1+j} \cdot \det(A_{1j}) + d$
return d

This algorithm is *recursive*. This basically means that a tree is constructed, where nodes of the tree correspond to recursive calls to the algorithm $\det(\cdot)$ including the *root* call. Any node that corresponds to a matrix with $k > 1$ rows and columns is expanded by adding k *child nodes* corresponding to the recursive calls that are executed. Once the lower-level nodes of the tree cannot expanded anymore, one can evaluate the tree, in this case the determinant, in a bottom-up fashion. See Let $T(n)$ denote the number of basic operations that the algorithm performs. Then $T(1) \geqslant 1$ and

$$T(n) \geqslant n \cdot T(n-1),$$

which shows that $T(n) >= n! = 2^{\Omega(n \log n)}$ which is *exponential* in n.

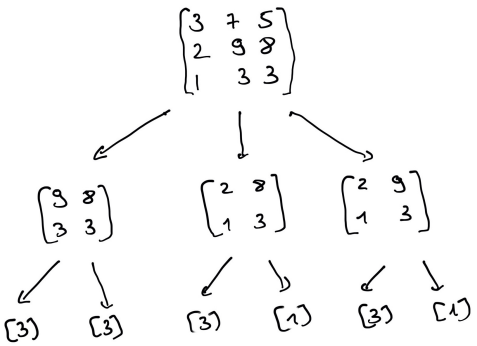

Fig. 1.1: An example of the recursion tree of the algorithm from Example 1.4. The tree corresponds to the run of the algorithm on input $\left(\begin{smallmatrix}3&7&5\\2&9&8\\1&3&3\end{smallmatrix}\right)$.

Exercises

1. Find the binary representation of 134.
2. Show that the binary representation with leading bit one of a positive natural number is unique.
3. Show that there are n-bit numbers $a, b \in \mathbb{N}$ such that the Euclidean algorithm on input a and b performs $\Omega(n)$ arithmetic operations. *Hint: Fibonacci numbers*
4. Show $n! = 2^{\Omega(n \log n)}$.
5. Let $A \in \mathbb{R}^{n \times n}$ and suppose that the n^2 components of A are pairwise different. Suppose that B is a matrix that can be obtained from A by deleting the first k rows and some of the k columns of A. How many (recursive) calls of the form $\det(B)$ does the algorithm of Example 1.4 create?
6. Let $A \in \mathbb{R}^{n \times n}$ and suppose that the n^2 components of A are pairwise different. How many different submatrices can be obtained from A by deleting the first k rows and some set of k columns? Conclude that the algorithm of Example 1.4 remains exponential, even if it does not expands repeated subcalls.
7. Complete the algorithm below such that it adds two natural numbers in binary representation $a_0, \ldots, a_{l-1}$, $b_0, \ldots, b_{l-1}$. What is the asymptotic running time (number of basic operations) of your algorithm? Can there be an asymptotically faster algorithm?

   Input: Two natural numbers a and b in their binary representation $a_0, \ldots, a_{l-1}$, $b_0, \ldots, b_{l-1}$.
   Output: The binary representation $c_0, \ldots, c_l$ of $a + b$

carry $:= 0$
for $i = 0, \ldots, l-1$
  $c_i = \text{carry} + a_i + b_i \pmod 2$
  carry $:=$
$c_l :=$
return $c_0, \ldots, c_l$

1.1 Analysis of Gaussian elimination

We recall Gaussian elimination.

Algorithm 1.2 (Gaussian elimination).

Input: $A \in \mathbb{Q}^{m \times n}$
Output: A' in row echelon form such that there exists an invertible $Q \in \mathbb{Q}^{m \times m}$ such that $Q \cdot A = A'$.

$A' := A$
$i := 1$
while $(i \leqslant m)$
  find *minimal* $1 \leqslant j \leqslant n$ such that there exists $k \geqslant i$ such that $a'_{kj} \neq 0$
  If no such element exists, then **stop**
  swap rows i and k in A'
  for $k = i+1, \ldots, m$
    subtract (a'_{kj}/a'_{ij}) times row i from row k in A'
  $i := i + 1$

We can easily prove correctness of the algorithm. First of all, the algorithm does only perform elementary row-operations of the form

i) swap two rows
ii) subtract a multiple of one row from *another row*.

This means that the resulting matrix A' can be obtained via

$$A' = Q \cdot A$$

with a non-singular $Q \in \mathbb{Q}^{m \times m}$. On the other hand we have the following invariant.

> After each iteration of the while-loop, the matrix H obtained from the first first j columns of A' is in row-echelon form and rows $i, i+1, \ldots, m$ of H are entirely zero, see Figure**??**.

How many arithmetic operations does Gaussian elimination perform? Subtracting a multiple of one row from another row in A' can be done in time

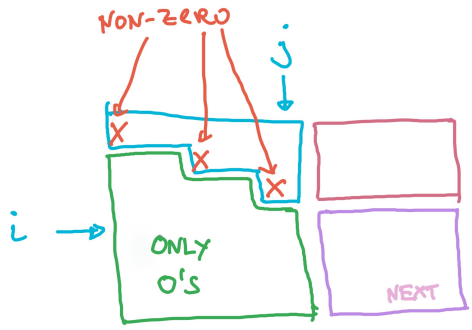

Fig. 1.2: Gaussian elimination: The matrix A' before the i-th iteration of the while loop.

$O(n)$ Thus the number of operations that are performed within the **for**-loop are $O(m \cdot n)$ in total. There are $O(m)$ iterations through the while-loop. All-together this shows that Gaussian elimination performs $O(m^2 \cdot n)$ iterations.But how large can the numbers grow in the course of the Gaussian algorithm Could it be that numbers have to be manipulated, whose binary encoding length is *exponential* in the total encoding length of the matrix A? Luckily the answer to this question is "No". To provide this answer, we have to show the following.

Theorem 1.2 (Hadamard bound). *Let $A \in \mathbb{R}^{n \times n}$ be non-singular. Then*

$$|\det(A)| \leqslant \prod_{i=1}^{n} \|a_i\|_2 \leqslant n^{n/2} \cdot B^n,$$

where B is upper bound on absolute values of entries of A.

Proof. The Gram-Schmidt orthogonalization of A yields a factorization

$$A = Q \cdot R,$$

where R is an upper triangular matrix with ones on the diagonal. The matrix Q has orthogonal columns, where the length of the i-th column $q^{(i)}$ is upper bounded by the length of the i-th column of A. The assertion follows from

$$\det(A)^2 = \det(Q)^2 = \det(Q^T)\det(Q) = \prod_i \|q^{(i)}\|^2.$$

Corollary 1.1. *If $A \in \mathbb{Z}^{n \times n}$ is integral and each entry in absolute value is bounded by B, then* $\text{size}(\det(A)) = O(n \log n + n \cdot \text{size}(B))$.

Corollary 1.2. *If $A \in \mathbb{Q}^{n\times n}$ is a rational matrix and ϕ is an upper bound on the size of each component of A, then* $\text{size}(\det(A)) = O(n^3 \cdot \phi)$.

Proof. Suppose that $a_{ij} = p_{ij}/q_{ij}$ for each ij, where p and q are integers with $gcd(p, q) = 1$. Then $(\prod_{ij} q_{ij})A$ is an integer matrix and

$$\det(A) = (\prod_{ij} q_{ij})^n \det((\prod_{ij} q_{ij})A).$$

The size of $(\prod_{ij} q_{ij})^n$ is $O(n^2\phi)$ and the size of $\det((\prod_{ij} q_{ij})A$ is $O(n^3\phi)$ with Corollary 1.1 .

Now that we have shown that the determinant of a rational matrix is a number of polynomial encoding length, we can prove that Gaussian elimination is indeed a polynomial time algorithm.

Theorem 1.3. *The Gaussian algorithm runs in polynomial time.*

Proof. We need to show that each entry of the part of A' that still has to be transformed (the magenta-colored part of the matrix in Figure **??**) is of polynomial size in the input encoding.

Consider the absolute value of the product of the pivot elements (red x's in the Figure). Clearly, this product is the absolute value of the determinant of the matrix that is obtained from choosing the rows and columns of A. By Corollary 1.2 this rational number is of polynomial size in the input encoding of this sub-matrix and thus of A.

Now, consider a nonzero element z in the part of A' that still needs to be transformed (the magenta part in Figure **??**). The absolute value of the product of this element with the pivot elements is the absolute value of the submatrix that is obtained by choosing the corresponding rows and columns. Therefore, the rational number z is of polynomial encoding length in the input.

Exercises

1. Show that the matrix $Q \in \mathbb{Q}^{m\times m}$ that transforms A into A' in the Gaussian algorithm via $Q \cdot A = A'$ has entries that are od polynomial size in the binary encoding length of A. Prove that this holds after each iteration of the while loop.

References

1. J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69:125–130, 1965.
2. J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.
3. F. Eisenbrand, A. Karrenbauer, and C. Xu. Algorithms for longer oled lifetime. In *6th International Workshop on Experimental Algorithms, (WEA 07)*, pages 338–351, 2007.
4. M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.
5. L. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1097, 1979.
6. V. Klee and G. J. Minty. How good is the simplex algorithm? In *Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin)*, pages 159–175. Academic Press, New York, 1972.
7. T. Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004. ZIB-Report 04-58.
8. B. Korte and J. Vygen. *Combinatorial optimization*, volume 21 of *Algorithms and Combinatorics*. Springer-Verlag, Berlin, second edition, 2002. Theory and algorithms.
9. E. L. Lawler. *Combinatorial optimization: networks and matroids*. Holt, Rinehart and Winston, New York, 1976.
10. L. Lovász. Graph theory and integer programming. *Annals of Discrete Mathematics*, 4:141–158, 1979.
11. J. E. Marsden and M. J. Hoffman. *Elementary Classical Analysis*. Freeman, 2 edition, 1993.
12. J. Matoušek and B. Gärtner. *Understanding and Using Linear Programming (Universitext)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
13. N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4(4):414–424, 1979.
14. A. S. Nemirovskiy and D. B. Yudin. Informational complexity of mathematical programming. *Izvestiya Akademii Nauk SSSR. Tekhnicheskaya Kibernetika*, (1):88–117, 1983.
15. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley, 1986.
16. N. Z. Shor. Cut-off method with space extension in convex programming problems. *Cybernetics and systems analysis*, 13(1):94–96, 1977.