Walking on the vertices of a polytope.

Adrien Chan-Hon-Tong

Walking on the vertices of a polytope.

Adrien CHAN-HON-TONG

December 2019

Abstract

This paper presents an algorithm for linear programming. This algorithm is clearly exponential.

But, this algorithm has the interesting features of being strongly polynomial for a given number of vertices/support sets. Precisely, the number of operation required to solve an instance is linear in the total number of vertices/support sets with a strongly polynomial factor.

1 Introduction

Linear programming is the very studied task of optimizing a linear criterion under linear equality and inequality constraints.

This problem has been first tackled by exponential algorithms like simplex [4] or perceptron [12]. Today, this problem is tackled by polynomial time algorithms like ellipsoid method [7, 5], log barrier method [10], or recently, Chubanov method [3]. Also, some families of linear programs are known to admit strongly polynomial algorithm (question is open for general linear programming, [1] recently shows that log barrier method is not strongly polynomial):

- linear program with -1/0/1 matrix A [13] (by specific algorithm)
- linear program with two variables per inequality [6] (by specific algorithm)
- Markov chain [11] (with simplex)
- system having binary solution [2] (by specific algorithm)

This paper offers a new algorithm for linear programming which is clearly exponential (and, thus, not competitive with [7, 10, 3]), but, strongly polynomial on several (small) families of linear programs. This algorithm is somehow related to simplex, or to [8], but, with the interesting feature of dealing with non singular and singular vertex in the same way.

Precisely, the algorithm walks on the vertices of the polytope. On common instance, the number of vertices is exponential in the binary size of the input, and, thus, this algorithm is exponential too. Yet, when the number of vertices is strongly polynomial, then, the algorithm is too.

Notation

Set of N dimensional vectors is written $\mathbb{Q}^N$, and, set of matrix of size $M \times N$ is written $\mathbb{Q}^{M\times N}$. If A is a matrix, or, x a vector, then, A_i or x_i is the i component of A or x (a N dimensional vectors for A and a scalar for x). If u, v are two N dimensional vectors, uv is the scalar product of the vectors i.e. $uv = \sum_{n\in\{1,\ldots,N\}} u_n v_n$. If $A \in \mathbb{Q}^{M\times N}$ is a matrix and $x \in \mathbb{Q}^N$ is a vector, then Ax is the matrix-vector product of A and x. $\mathbf{0}$, $\mathbf{1}$ are the vectors with all components being 0 or 1.

The binary complexity of an algorithm is a bound on the number of binary operations required for the algorithm to produce the expected solution as function of the binary size of the input classically written L. A $O(f(L))$ binary complexity means that exists a constant $\mathcal{M}$ such that this bound is $\mathcal{M}f(L)$. A $\widetilde{O}(f(L))$ binary complexity means that exists γ such the binary complexity is $O(f(L)\log^\gamma(f(L)))$. One can speak about complexity (not binary) expressed in the number of iterations or rational complexity (for the total number of rational operation). Let stress that binary complexity of addition/multiplication of two numbers of binary size L is already $\widetilde{O}(L)$.

2 The walking-on-vertices algorithm with recurrence

2.1 Pre processing

The native form of linear program is the task of solving $\min_{x\ /\ Ax\geq b} cx$ for given $A \in \mathbb{Q}^{M\times N}$ a matrix, $b \in \mathbb{Q}^M$ and $c \in \mathbb{Q}^N$ some vectors.

Let recall that solving $\min_{x\ /\ Ax\geq b} cx$ is equivalent to solve a system of inequality $Hx \geq h$ due to primal dual theory. As $Hx \geq h$ is equivalent to solve $\min_{x,t\ /\ Hx+t\geq h, t\geq 0} t$ on which it is possible to consider primal dual again, one can always assume that some solution exists, and, that the task is to find it. Then, this last problem $Hx \geq h$ is equivalent to solve a system of strict inequality $Gx > g$: as maximal determinant of a sub matrix is a polynomial in the binary size of the matrix, any solution of $Hx + t \geq h, t \geq 0, -t \geq -\varepsilon$ for a decidable ε can be converted into a solution. Finally, $Gx > g$ is equivalent to $\mathcal{A}x > \mathbf{0}$ because $\begin{pmatrix} G & -g \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ t \end{pmatrix} > \mathbf{0}$.

***Erratum**: Two faults plague paper hal-00722920 from version 9 to 22 inclusive. This pre processing was claimed unknown. And, [3] was claimed strongly polynomial, while it is only polynomial !*

This way, the offered algorithm can assume without restricting the generality to have an input $A \in \mathbb{Q}^{M\times N}$, with the task of producing $x \in \mathbb{Q}^N$ such that $Ax > \mathbf{0}$, with the prior that such x exists.

2.2 Vertices

As it is a central notion in this paper, let introduce formally the concept of vertex in the case of homogeneous linear system.
Definition: Vertex

Given $A \in \mathbb{Q}^{M \times N}$, a set of index $I \subset \{1, ..., M\}$ is called a vertex if and only if $\exists x \in \mathbb{Q}^N$ such that $A_I x = -\mathbf{1}$ and $A_{\{1,...,M\}\setminus I} x > -\mathbf{1}$

A non singular maximal vertex is just a point x such that $A_I x = -\mathbf{1}$ with A_I square and non singular, plus, $A_{\{1,...,M\}\setminus I} x > -\mathbf{1}$. But, here, all vertices are considered: size of I can be different from N (lower or higher), and/or, the matrix A_I can be singular. Let stress that this definition specifies that $A_{\{1,...,M\}\setminus I} x > -\mathbf{1}$, so, a point x can not belong to more than one vertex.

Currently, other names like feasible sets or support sets could have been used instead of *vertices*. Yet, there is a link between classical vertices and these vertices: **Geometrical vertices are maximal vertices under this definition.**

The number of vertices is at most bounded by 2^M. Yet, on common instance, the number of vertices **is** 2^M. Only few things could forbid I to be a vertex. Typical example is when i, j, k verify $A_k = (A_i + A_j)$, then, it is impossible that $A_i x = A_j x = A_k x = 1$ because $A_k x = 2 > 1$. Typically, for $N = 2$ there is no more than $M^2 + M$ vertices. So the offered algorithm is not very relevant on general case, but, could still be interesting, on few vertices instances.

2.3 Key ideas and pseudo codes

The key idea of the offered algorithm to find $Ax > \mathbf{0}$ is to reach some vertex $I \neq \{1, ..., M\}$, and then, to call recursively the algorithm on A_I to find some vector v allowing to exit this vertex reaching another one. Yet, this single idea is not efficient as it: alone, there is no bound on the number of times a vertex I is observed.

To avoid this drawback, it is sufficient to store the different steps, this creates a dictionary allowing to exit efficiently already encountered vertices. Currently, such perfect exit which increases equally all constraints strictly increase the working set.

The pseudo code is presented in figure-algorithm 1.

This pseudo code can be improved by sharing history across all recursive calls: currently a perfect exit can only be detected in a single call while it could be detected across different calls. Yet, code is much more easily with this version. Sharing the history across all recursive calls is very hard from coding point of view, as, it requires to maintain at every moment a real point, while here, recurrent call restarts from 0. Also, this sharing requires to deal with constraint outside I: it is possible as I is the set of most negative current constraints so there is a margin to move before any constraints outside of I may parasite the algorithm, but, this requires very careful implementation.

Algorithm 1 walking on vertices with recurrence

// **input:** $A \in \mathbb{Q}^{M \times N}$, such that $\exists x,\ / \ Ax > \mathbf{0}$

// **output:** $x \in \mathbb{Q}^N \ / \ Ax > \mathbf{0}$ when called with $I = \{1, ..., M\}$ and empty dicts $walking\_vertices\_rec(A, I, PerfectExit, Exit) =$

$x = \sum_{i \in I} A_i$
$History = dict()$
while True **do**
  $minAIx = \min_{i \in I} A_i x$
  $J = \{i \in I, \quad A_i x = minAIx\}$
  if $minAIx > 0$ **then**
    if $J == I$ **then**
      $PerfectExit[I] = x$
    else
      $Exit[I] = x$
    returns x
  if $J \in History$ **then**
    $PerfectExit[I] = x - History[J]$
  else
    $History[J] = x$
  if $J \in PerfectExit$ **then**
    $v = PerfectExit[J]$
  else
    if $J \in Exit$ **then**
      $v = Exit[J]$
    else
      $v = walking\_vertices\_rec(A, J, PerfectExit, Exit)$
  if $A_I v > 0$ **then**
    $x = v$
  else
    $l = \max_{\lambda} \min_{m \in I} A_i(x + \lambda v)$
    $x = x + lv$

So, from theoretical point of view, there is not issue in sharing the history, but, the convergence will be proven algorithm 1.

2.4 Consistency of the algorithm

To summarize the proof, this algorithm 1 converges because there can not have more than M step 17 successively (because step 17 leads to add something in J), and, because step 20/22 can not be done more than one per vertex (in a call as sharing is not optimal - but globally with sharing).

2.4.1 lemmas

Let first present some basic result which are currently at the heart of the perceptron algorithm [12].

Definition: Positive combination of vectors from A

Let $\Omega(A)$ be the set of vectors $v \in \mathbb{Q}^N$ such that there exists $\alpha \in \mathbb{Q}^M$ with $v = \sum_{m \in \{1,...,M\}} \alpha_m A_m$, $\alpha \geq \mathbf{0}, and, \alpha \neq \mathbf{0}$

Ω will be used instead of $\Omega(A)$ is it creates no ambiguity. Let note that $v, w \in \Omega \Rightarrow v + w \in \Omega$. Then, the following lemma holds:

Lemma: $\forall v \in \Omega \backslash \{0\}$, there is $i \in \{1, ..., M\}$, such as $A_i v > 0$.

If $\forall m \in \{1, ..., M\}$, $A_m v \leq 0$, then, $vv \leq 0$ but $v \neq \mathbf{0}$, so, $vv > 0$.

Corollary: $\forall v \in \Omega \backslash \{0\}$, if $Av = \lambda \mathbf{1}$, then, $\lambda > 0$.

Lemma: if $\exists x, Ax > 0$ then $\mathbf{0} \notin \Omega$.

$xv \geq (\sum_{m \in \{1,...,M\}} A_m x \alpha_m) > 0$. So, $v \neq 0$.

2.4.2 Termination and correctness of walking-on-vertices

Lemma: all vectors in the algorithm belongs to Ω and $minAIx$ strictly increases.

This can be proven by inducting. Step 1 creates a vector in Ω. By induction, recurrent calls $walking\_vertices\_rec$ return $v \in \Omega$, and, for step 13, as $PerfectExit[J]$ is before x, it means that $x = PerfectExit[J] + v$ with $v \in \Omega$. So, $v \in \Omega$.

Then, the induction hypothesis is extended in step 27 as soon as $l > 0$. As, $A_J v > 0$ and $A_J x = minAIx\mathbf{1}$ and $A_{I \backslash J} x > minAIx\mathbf{1}$, then, there is some $l > 0$ which makes $minAIx$ increases.

Lemma: step 17 leads J to strictly increase.

By construction when using a perfect exit all vertors from J are maintained in J while performing step 27. But, one will enter otherwise setp Now, while moving along $x - y$, one constraint is going less and less validated. For l being the maximal possible move, one constraint enter in J. And, J is bounded by $\{1, ..., M\}$ so

Corollary: there can not be more then M successive steps 17.

Lemma: J can not be I (in particular recursion is not infinite).

This is linked with the lemma that $\forall v \in \Omega$, if $Av = \lambda \mathbf{1}$, then, $\lambda > 0$. So, J can **not** be I.

Lemma: algorithm never call twice step 20/22 on the same set J.

By construction, if J is seen twice, there is a perfect exit (in particular as $minAIx$ strictly increases it can be a pure loop). This allows to state one of the main claim of this paper:

Corollary: walking-on-vertices terminates after at most 2^M steps 20/22, each separated by at most M successive steps 17.

2.5 Why not using bisection ?

Importantly, one could wonder why not looking for v such that $A_J v = 1$ directly when reaching step 12. Because, applying the algorithm only when there is no v such that $A_J v = 1$ **seems** a good idea. Algorithm will only call himself on maximal vertex.

But, such v may **not** be in Ω. Thus, moving along this v may lead to reach 0 and loop infinitely !

Currently, this is even worse: $-x$ is always such bisector. But, moving along $-x$ is never a good idea.

Finding a way to compute bisector from Ω may be a very good improvement. Allowing to consider only maximal vertices and not all vertices (this way this algorithm may theoretically outperform simplex in every cases). Yet, this seems as hard as the original problem.

3 Discussion

3.1 Complexity

The algorithm is exponential in common case.

Now, let assume that K the number of vertices is small.

By correctly sharing the history, each vertex is only explored one, or, will be linked with a perfect exit. As, the number of move without exploration is bounded by M, then the complexity is bounded by $O(KM)$. But complexity of a step is only $O(MN + \log(K)M)$: MN to compute step 4,5 and 26 (26 can be done just by a for loop on all constraint), $\log(K)M$ for step 12,16,19 (assuming a correct implementation of the dictionary - maybe less but query has size M so probably not much less).

finally, numerical computation have a L binary complexity (currently this statement is not that clear because number can increase during the loop. Yet, there is no more than M multiplications so the final binary size should no be more than $\widetilde{O}(LM)$). This allows to state the main claim of this paper:

Rational complexity of walking-on-vertices is only $\widetilde{O}(KM^2N)$, and, binary complexity no more than $\widetilde{O}(KLM^3N)$.

Let not that the offered pseudo code does not optimize sharing. Thus, in this pseudo code, the complexity may be $\widetilde{O}(K^M LM^2 N)$ which is clearly bad. Yet, there is no theoretical issue by sharing a common history (not just perfect exit and exit), only very hard implementation effort.

3.2 comparison to the state of the art

This algorithm is in some way similar to simplex [4]. Yet, main difference with the simplex is that simplex can be trapped in a vertex I such that size of I is very large seeing N. In such case, simplex typically loops over all the subset of I with size N to find one allowing to increase the solution. Thus, exiting one single vertex is exponential in N and M with simplex (and there can be an exponential number of vertices).

Inversely, the offered algorithm will just basically compute the average of A on I. This may **not** allows to exit the current vertex (increasing the minimal constraint satisfaction). **But**, anyway, this will lead to reach another vertex (which is necessarily different from I as $A_I v \neq \mathbf{0}$ for $v \in \Omega$). This way, the algorithm goes from vertex to vertex. When, the algorithm has visited all vertices, then, it finishes in M steps.

Of course, the number of vertices K can be larger than the number of square sub matrices, in this case, just exploring all sub matrices is better. Also, if the number of vertices is small, but, that vertices are not singular, then, simplex is better because simplex will just explore maximal vertices. In other words, simplex handles common vertex more efficiently, but, may require exponential number of steps (in both N and M) to exit a singular one. While, the walking-on-vertices algorithm handles all vertices poorly, but, without being sensible to the singularity.

Very briefly, this algorithm is also close to [8]. However, it is not very clear in which way [8] should be better than simplex. Indeed, [8] claims to find a way to exit any vertex by projecting gradient. Yet, exiting a vertex is as hard as solving the problem itself - this is known since [9], and, is very simple considering that $Ax > b \Leftrightarrow \begin{pmatrix} A & -b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ t \end{pmatrix} > \mathbf{0}$. The key idea of this paper is not to focus on exiting vertex, but, only to collect vertices. When, all vertices are collected, then the algorithm has converged.

Finally, when, this number is small, this algorithm is theoretically better than current polynomial algorithms. They requires today at least $\widetilde{O}(L)$ steps (generally there is even N or M factors), so, their binary complexity is $\widetilde{O}(NML^2)$ which is higher than $\widetilde{O}(KLM^2N)$ (when K is negligible). So, the following claim holds:

In the (restricted and unrealistic) use case of small number of vertices with maximal vertices being heavily degenerated, the walking-on-vertices algorithm is theoretically better than brute exploration, simplex, and, current polynomial methods.

In practice, there is not many case where the number of vertices is small. The only real interest of this algorithm seems to be matrix A with very large L, M, low N, and, almost all sub matrices of A having a very low rank (but A having full rank). For example, if A is the concatenation of a $N \times N$ identity matrix with $\mathcal{A}$ a $M \times N$ 2 rank matrix (L, M large). Then, current polynomial methods will be bad (due to L), exploring directly all square sub matrices or exiting a vertices is exponential in N and M, while, the number of possible vertices is only $2^N M^2$ which is not polynomial both in N and M.

3.3 Detecting impossibility

The offered algorithm allows to detect case where there is no $Ax > \mathbf{0}$. In these cases, J can be I. But, by checking this point, one can detection impossibility.

Currently, this is not trivial that $(\neg\exists x \ / \ Ax > 0) \Leftrightarrow (\exists v \in \Omega \ / \ Av = \mathbf{0})$. However, the algorithm relies on the assumption that $\exists x \ / \ Ax > 0$ only to state that $I \neq J$. So, either the algorithm terminates because at no moment $I = J$ and there is some solution, or, the algorithm observes $I = J$ but it means that $\exists v \in \Omega \ / \ Av = \mathbf{0}$ which implies that there is no solution (it can be lower than 0 otherwise $(Av)(Av) < 0$).

So, there is an equivalence between observing $I = J$, and, impossibility.

3.4 Toy numerical experiments

Implementation with low sharing is offered in appendix. This implementation outperforms simple perceptron for small N as soon as M becames large.

3.5 Application to support vector machine

Support vector margin is a very studied problem consisting in solving $\min_{Ax \geq \mathbf{1}} xx$. This problem is a particular class of semi definite programming which can be solved efficiently using log barrier algorithm [10].

However, it can also be solved by enumerating vertices of the polytope of $Ax \geq \mathbf{1}$, and, to compute xx for each. Obviously, this last way is exponential in common cases where there is an exponential number of vertices. But, this way could be interesting when there is a few vertices i.e. in the same case where the walking-on-vertices algorithm is efficient. Indeed, as the offered algorithm already works by collecting vertices, it can be extended to solve support vector machine using this idea.

Pseudo code of the algorithm is presented in figure algorithm 2. The idea is to walk on vertices using either projection or call to the walking-on-vertices algorithm to find v such that $A_I v > 0$.

Let prove quickly that this algorithm is correct. Globally, the proof is close to the one of walking-on-vertices: step 10 can only happen M successive times as E strictly increases. But, E is never seen again after a step 4, as, the algorithm has found something better than the minimum (step 3) - it will be a contradiction with the definition of the minimum

Algorithm 2 walking on vertices for support vector machine

// **input:** $A \in \mathbb{Q}_{M,N}$ and w with $Aw \geq \mathbf{1}$
// **returns:** w such that $Aw \geq \mathbf{1}$ and $w^T w$ is minimal
$walking\_vertice\_svm(A, w) =$

1: **while** True **do**
2: $E = \{m \in \{1, ..., M\} \ / \ A_m w = 1\}$
3: compute $w_E = \underset{v \in \mathbb{Q}^N \ / A_E v = \mathbf{1}}{\arg\min} v^T v$ and $u = w_E - w$
4: **if** $A_{\{1,...,M\} \setminus E} w_E > \mathbf{1}$ **then**
5: $v = walking\_vertices\_rec \begin{pmatrix} A_E \\ -w_E^T \end{pmatrix}$
6: **if** v is None **then return** w_E
7: chose δ: $0 < \delta < h = \min \left(\{ \frac{w_E^T v}{v^T v} \} \cup \{ \frac{A_m w_E - 1}{-A_m v} | A_m v < 0 \} \right)$
8: $w = w_E + \delta v$
9: **else**
10: $g = \underset{m \in \{1,...,M\} / A_m u < 0}{min} \frac{A_m w - 1}{-A_m u}$
11: $w = w + gu$

The main question is *is there necessarily a vector v in step 5, if, w_E is not optimal ?*. This statement is true see proof:

Let $\psi = (\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) w^* - w$ (currently, ψ is in $\mathbb{R}^N$ and not in $\mathbb{Q}^N$ but it still exists). Then, $A_E \psi = (\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) A_E w^* - A_E w = (\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) A_E w^* - \mathbf{1} \geq (\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) \mathbf{1} - \mathbf{1} \geq \frac{1}{2} (\sqrt{\frac{w^T w}{(w^*)^T (w^*)}} - 1) \mathbf{1} > \mathbf{0}$ (because w^* is admissible and $(w^*)^T (w^*) < w^T w$). Independently, norm of $(\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) w^*$ is strictly less than $\sqrt{\frac{w^T w}{(w^*)^T (w^*)}} (w^*)^T (w^*) = \sqrt{w^T w}$. So, $w^T ((\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) w^*) < \sqrt{w^T w} \sqrt{w^T w} = w^T w$ (because $\alpha^T \beta \leq \sqrt{\alpha^T \alpha} \sqrt{\beta^T \beta}$). So $w\psi = (\sqrt{\frac{w^T w}{4(w^*)^T (w^*)}} + \frac{1}{2}) w^T w^* - w^T w < w^T w - w^T w = 0$.

Let stress that careful implementations should share information about vertices of A between different call of walking-on-vertices algorithm. Bisection computed at step 3 should not be merged with bisection computed into the walking-on-vertices, as, the bisection of step 3 are not positive combination of vectors from A.

3.6 Application to linear program

For linear programs, classical conversions show that solving $Ax > \mathbf{0}$ is equivalent to solve $Ax \geq b$. Yet, this implies to consider $\min_{Ax + t \geq b, t \geq 0} t$, then, it primal dual $\Gamma \chi \geq \gamma$, then, $\Gamma \chi + \mu \geq \gamma, \mu \geq 0, -\mu \geq -\varepsilon$, then, $\Gamma \chi + \mu - \gamma \rho > 0, \mu > 0, -\mu + \varepsilon \rho > 0$. For example if $1 \ll N \ll M$, considering the primal dual may not be very efficient.

So, there is an interest to solve linear program $\min_{Ax+t\geq b, t\geq 0} t$ without conversion. This could be done using the walking-on-vertices algorithm see algorithm 3. This will be efficient under the same condition on which the walking-on-vertices algorithm is efficient on matrix A.

Algorithm 3 walking on vertices for linear program without conversion

Require: $c \in \mathbb{Q}^N$, $b \in \mathbb{Q}^M$, $A \in \mathbb{Q}^{M\times N}$ with $Ac > \mathbf{0}$, $Ax \geq b \Rightarrow c^T x \geq 0$, $Ax \geq b$

Ensure: return x: $Ax \geq b$ and cx is minimal

while True **do**
  $D = \{m \in [1, M] \ / \ A_m x = b_m\}$
  compute u the orthogonal projection of $-c$ on $Ker(A_D)$
  if $u = \mathbf{0}$ **then**
    $v = walking\_vertices\_rec \begin{pmatrix} A_E \\ -w_E^T \end{pmatrix}$
    if v is None **then return** w_E
    chose δ with $0 < \delta < h = \min_{m\in[1,M]/A_m v<0} \frac{A_m x - b_m}{-A_m v}$ (e.g. $\delta = \frac{h}{4}$)
    $x = x + \delta v$
  else
    $g = \min_{m\in[1,M] \ / \ A_m u<0} \frac{A_m x - b_m}{-A_m u}$
    $x = x + gu$

4 Conclusion

This paper offers a new algorithm for linear programming which is not challenging polynomial methods (ellipsoid, log barrier or Chubanov) on general output, but, may be interesting on families of instances where the number of vertices is small (despite it is exponential in common instance).

Typically, this algorithm will be very efficient when number of constraints and binary size are large, number of variables is moderate, and, almost all sub matrices of the input matrix have low rank. In such case, polynomial methods will be limited by binary size. Considering all square sub matrices will be intractable due to the large number of constraints. Finally, simplex may be locked by heavily degenerated vertices.

References

[1] Xavier Allamigeon, Pascal Benchimol, Stéphane Gaubert, and Michael Joswig. Log-barrier interior point methods are not strongly polynomial. *SIAM Journal on Applied Algebra and Geometry*, 2(1):140–178, 2018.

[2] Sergei Chubanov. A polynomial algorithm for linear optimization which is strongly polynomial under certain conditions on optimal solutions, 2015.

[3] Sergei Chubanov. A polynomial projection algorithm for linear feasibility problems. *Mathematical Programming*, 153(2):687–713, 2015.

[4] George B et. al. Dantzig. The generalized simplex method for minimizing a linear form under linear inequality restraints. In *Pacific Journal of MathematicsAmerican Journal of Operations Research*, 1955.

[5] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[6] Dorit S Hochbaum and Joseph Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.

[7] Leonid Khachiyan. A polynomial algorithm for linear programming. *Doklady Akademii Nauk SSSR*, 1979.

[8] H.C. Lui and P.Z. Wang. The sliding gradient algorithm for linear programming. In *American Journal of Operations Research*, 2018.

[9] Nimrod Megiddo. A note on degeneracy in linear programming. *Mathematical programming*, 35(3):365–367, 1986.

[10] Yurii Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13. Siam, 1994.

[11] Ian Post and Yinyu Ye. The simplex method is strongly polynomial for deterministic markov decision processes. *Mathematics of Operations Research*, 40(4):859–868, 2015.

[12] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[13] Eva Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34(2):250–256, 1986.

Appendix

```
def add(u,v):
    return [u[n]+v[n] for n in range(len(u))]

def mul(l,v):
    return [l*e for e in v]

def dot(u,v):
    return sum([u[n]*v[n] for n in range(len(u))])

def perceptron(A):
    x = A[0].copy()
    while True:
        ax,a = min([(dot(a,x),a) for a in A])
```

```
        if ax>Fraction(0):
            return x
        else:
            x = add(x,a)

def argmaxmin(A,I,J,x,minAIx,v):
    assert(all([dot(A[j],x)==minAIx for j in J]))
    sJ = set(J)
    assert(all([dot(A[i],x)>minAIx for i in I if i not in sJ]))
    assert(all([dot(A[j],v)>0 for j in J]))

    minAJv = min([dot(A[j],v) for j in J])
    K = [i for i in I if dot(A[i],v)<minAJv]
    candidate = [(dot(A[k],x)-minAIx)/(minAJv-dot(A[k],v)) for k in K]
    return min(candidate)

def walking_vertices_rec(A,I,Bisector,Exit):
    if I is None:
        I = list(range(len(A)))
    if Bisector is None:
        Bisector = dict()
    if Exit is None:
        Exit = dict()

    History = dict()
    x = [0]*len(A[0])
    for i in I:
        x = add(x,A[i])

    while True:
        minAIx = min([dot(A[i],x) for i in I])
        J = sorted([i for i in I if dot(A[i],x)==minAIx])

        if minAIx>0 and J==I:
            Bisector[tuple(I)] = x
            Exit[tuple(I)] = x
            return x
        if minAIx>0:
            Exit[tuple(I)] = x
            return x

        if minAIx==0 and J==I:
            return None

        if tuple(J) in History:
            if tuple(J) not in Bisector:
                y = History[tuple(J)]
                Bisector[tuple(J)] = add(x,mul(-1,y))
        else:
            History[tuple(J)] = x

        if tuple(J) in Bisector:
            v = Bisector[tuple(J)]
        else:
            if tuple(J) in Exit:
                v = Exit[tuple(J)]
            else:
                v = walking_vertices_rec(A,J,Bisector,Exit)
                assert(tuple(J) in Exit)
                if v is None:
                    return None


        if min([dot(A[i],v) for i in I])>0:
            x = v
            continue
        else:
            l = argmaxmin(A,I,J,x,minAIx,v)
            x = add(x,mul(l,v))
            continue

from fractions import Fraction
import random

def randintfraction():
    return Fraction(random.randint(-pow(10,5),pow(10,5)))

def randomVector(N):
    while True:
        x = [randintfraction() for n in range(N)]
        if dot(x,x)!=0:
            return x

def randomMatrix(M,x):
    A = []
    while len(A)!=M:
        a = randomVector(len(x))
        if dot(a,x)!=0:
            if dot(a,x)>0:
                A.append(a)
            else:
```

```
                A.append(mul(-1,a))
    return A


print("########### MAIN ###########")


solution_for_debug = randomVector(5)
A = randomMatrix(1000,solution_for_debug)

print(solution_for_debug)
print(min([dot(a,solution_for_debug) for a in A]))

import time

print("perceptron")
print(time.time())
x=perceptron(A)
print(all([dot(a,x)>0 for a in A]))
print(time.time())

print("walking_vertices")
print(time.time())
x=walking_vertices_rec(A,None,None,None)
if x is None:
    print("no solution")
else:
    print(all([dot(a,x)>0 for a in A]))
print(time.time())
```