Linking linear feasibility and linear programming.

Adrien Chan-Hon-Tong

Linking linear feasibility and linear programming.

Adrien CHAN-HON-TONG

July 23, 2019

Abstract

This paper presents an algorithm based on strongly polynomial linear feasibility queries ($\exists ?x \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$) which solves linear programming ($\min_{x \ / \ Ax \geq b} cx$ and/or $\exists ?x \ / \ Ax \geq b$). Despite that the complexity of the algorithm is not established, discussion shows the potential of the algorithm. In addition, some ideas are presented as potential ways to find if the algorithm is or is not a strongly polynomial time solver for linear programs.

1 Introduction

Linear programming is the very studied task of solving $\min_{x \ / \ Ax \geq b} c^T x$ with A a matrix, b and c some vectors. This problem can be solved in polynomial times since [8, 5, 7] i.e. in a number of binary operations bounded by a L^γ where L is binary size required to write the matrix A and γ a constant. Today, state of the art algorithm to solve linear program are interior point algorithms (e.g. [9]). Yet, [1] shows that major interior point algorithms do not solve linear program in strong polynomial time i.e. in a number of rational operations bounded by $\max(M, N)^\gamma$ where M, N are the sizes of A (independently from the binary size of values of A). Only, some families of linear program can be solved in strongly polynomial times today:

- linear program with 0/1 matrix A [11] (by specific algorithm)
- linear program with at most two variables per inequality [6] (by specific algorithm)
- Markov chain [10] (by simplex)
- system having binary solution [2] (by specific algorithm)

Independently, [3] recently shows that linear feasibility i.e. $\exists ?x \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ can be solved in strongly polynomial time. This is a very interesting result seeing [1]. So, the question of solving linear program as a strong polynomial sequence of linear feasibility queries is relevant.

This paper focus on this last point, and, presents algorithms which solve linear programming by solving a sequence of linear feasibility problems. Despite the maximal length of such sequence is not established, discussion shows why this algorithm could have some good complexity features.

Notations

$\mathbb{N}, \mathbb{Q}$ are the sets of integer and rational numbers. $\setminus$ is the ensemble subtraction. For all integers i, j, $[i, j]$ will symbolize the **integer** range i.e. $\{i, i+1, ..., j\}$ which is empty if $i > j$ (there will be no ambiguity with the interval in $\mathbb{R}$ as there is no real range in this paper).

For all integers i, j, I, J, $\mathbb{Q}^I$ is the set of I dimensional vectors on $\mathbb{Q}$, and, $\mathbb{Q}^{I \times J}$ is the set of matrix with I rows and J columns, with values in $\mathbb{Q}$, and, $._i$ designs the i component: a row for a matrix and a rational for vector or a row. $\mathbb{Q}^I$ would be matched with $\mathbb{Q}^{I \times 1}$ i.e. vectors are seen as columns, and, row of a matrix are matched with $\mathbb{Q}^{1,J}$. For all sets $S \subset \mathbb{N}$, A_S, b_S is the submatrix or subvector obtained when keeping only components indexed by $s \in S$. ^{T} is the transposition operation i.e. $A^T_{j,i} = A_{i,j}$. $\mathbf{0}$ and $\mathbf{1}$ are the 0 and 1 vector i.e. vector contains only 0 or only 1, and $\mathbf{I}$ is the identity matrix.

If $A \in \mathbb{Q}^{I \times J}$, the null vector space of A (i.e. the kernel) is written $Ker(A) = \{v \in \mathbb{Q}^J / Av = \mathbf{0}\}$, with the convention that Ker of empty A is all space.

$\mathbb{U}_I$ is the set of normalized vectors from $\mathbb{Q}^I$ i.e. $\mathbb{U}_I = \{v \in \mathbb{Q}^I, v^T v = 1\}$. $\mathbb{U}_{I,J}$ is the set of matrix from $\mathbb{Q}^{I \times J}$ whose rows are in $\mathbb{U}_J$ (rows not necessarily columns). All notations are quite classical except $\mathbb{U}$ to indicate normalization of vectors and/or matrix rows.

2 The slide and jump algorithm

2.1 Key idea

The starting point of this paper is Chubanov algorithm [3] which offers an algorithm to solve linear feasibility ($\exists ? x \in \mathbb{Q}^N \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ when $A \in \mathbb{Q}^{M \times N}$ has full rank) in strongly polynomial time.

First, with this algorithm, one can solve with the same complexity linear separability problem ($\exists ? y \in \mathbb{Q}^N \mathcal{A} y > \mathbf{0}$ for any $\mathcal{A} \in \mathbb{Q}^{M \times N}$) in strongly polynomial time (see proof just after).

Then, it is easy to get a small improvement from a not optimal admissible solution for linear program by solving a derived linear separability problem. Indeed, if x verifying $Ax \geq b$ has not minimal cx under assumption that $Ac = \frac{3}{5}\mathbf{1}$, then it is possible (see proof in subsection 2.3) to find u such that $\begin{pmatrix} A_D \\ -c^T \end{pmatrix} u > \mathbf{0}$ with D the current saturated constraints (i.e. D such that $A_D w = b_D$).

Now, it could take infinite time to reach the optimal solution by computing such u and updating $x = x + \varepsilon u$. These moves can be see as *jump* because its

always lead to some improvement independently from current saturated constraint. But reaching the solution by jumping is not obvious. So, the algorithm also considers *sliding* moves. It considers moves which aim to keep the structure between x and the constraints. Such moves can be computed by solving the easy optimization problem derived by freezing this structure, for example, solving $\min_{w \ / \ A_D v = \mathbf{0}} c^T v$ keeps D unchanged. This last problem can be trivially solved because it is just a projection on a vectorial space (can be done with Gram Schmidt basis transform).

The point is that *sliding* moves allow to explore completely a particular combinatorial situation (D) while *jumping* moves allow to exit such structure. The termination will be guarantee by the impossibly to see explored D twice, and, the correctness, by capacity to jump to next D until optimal solution is reached.

2.2 linear separability

Before presenting the algorithms, let first consider the lemma linking linear feasibility and linear separability:

Lemma 2.1 (Linking linear feasibility and linear separability)**.** *If it is possible to solve* $\exists ?x \in \mathbb{Q}^I \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ *when* $A \in \mathbb{Q}^{J,I}$ *has rank* J *with less than* $O(\max(I,J)^\gamma)$ *operations on* $\mathbb{Q}$*, then it is possible to solve* $\exists ?y \in \mathbb{N} \ / \ \mathcal{A}y > \mathbf{0}$ *for any* $\mathcal{A} \in \mathbb{Q}^{M \times N}$ *with less than* $O(\max(N,M)^\gamma)$ *operations on* $\mathbb{Q}$*.*

Proof. Let $\mathcal{A} \in \mathbb{Q}^{M \times N}$ a matrix without any assumption. Let consider the matrix $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix} \in \mathbb{Q}^{M \times 2N+M}$ i.e. formed with $\mathcal{A}$ concat with $-\mathcal{A}$ concat with $-\mathbf{I}$ the opposite of identity matrix.

First, this matrix has rank M due to the identity block. So by assumption, one can apply the solver to know $\exists ?x \in \mathbb{Q}^{2N+M} \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ which will produce an output in less than $O((2N+M)^\gamma) \leq O(3^\gamma \max(N,M)^\gamma) = O(\max(N,M)^\gamma)$ on $\mathbb{Q}$. This output is either a solution or a certificate that no solution exists.

If solver finds a solution $x \in \mathbb{Q}^{2N+M}$. Let split the solution: $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ with $(x_1, x_2, x_3) \in \mathbb{Q}^N \times \mathbb{Q}^N \times \mathbb{Q}^N$. By assumption, $A \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{0}$ and $x_1, x_2, x_3 > \mathbf{0}$. Yet, as $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix}$, it means that $\mathcal{A}x_1 - \mathcal{A}x_2 - \mathbf{I}x_3 = \mathbf{0}$. As $\mathbf{I}x_3 = x_3$ and $x_3 > \mathbf{0}$, it leads that $\mathcal{A}(x_1 - x_2) = x_3 > \mathbf{0}$. So $x_1 - x_2$ is a solution of the desired problem found in less than $O(\max(N,M)^\gamma)$ operation on $\mathbb{Q}$.

Then, let assume that there is y such that $\mathcal{A}y > \mathbf{0}$, and, let consider $x_1 = \max(y, \mathbf{0}) + 1$, $x_2 = \max(-y, \mathbf{0}) + 1$, and, $x_3 = \mathcal{A}y$. Then, $x_1, x_2, x_3 > \mathbf{0}$, and $x_1 - x_2 = y$. So, $A \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathcal{A}y - \mathcal{A}y = 0$. So, there is a solution of the

derived problem, solver will find one. And, so, if solver does not find solution, then, there is not. □

Also, a quite trivial but required lemma:

Lemma 2.2. *Solving* $\exists ?x\mathbb{Q}^N \ / \ Ax > \mathbf{0}$ *or* $\exists ?x\mathbb{Q}^N \ / \ Ax \geq \mathbf{1}$ *is equivalent.*

Proof. Any solution of the second problem are solution of the first, and any solution w of the first can be normalised in $(\frac{1}{\min\limits_m A_m w})w$ to form a solution of the second. □

2.3 Pseudo code

Pseudo code for linear program is presented in algorithm 1: algorithm assumes the input linear program is $c \in \mathbb{Q}^N$, $b \in \mathbb{Q}^M$, $A \in \mathbb{Q}^{M \times N}$ with $Ac > \mathbf{0}$, $Ax \geq b \Rightarrow c^T x \geq 0$, and $Ax_{start} \geq b$ (e.g. $x_{start} = \lambda c$).

These assumptions do not restrict generality because all linear programs can be pushed in this shape using classical primal dual trick (see appendix).

Algorithm 1 Slide and jump algorithm

Require: $c \in \mathbb{Q}^N$, $b \in \mathbb{Q}^M$, $A \in \mathbb{Q}^{M \times N}$ with $Ac > \mathbf{0}$, $Ax \geq b \Rightarrow c^T x \geq 0$, and $Ax_{start} \geq b$ (e.g. $x_{start} = \lambda c$) *this does not restrict generality*
Ensure: return x: $Ax \geq b$, 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**
    call subsolver $\exists ?v \ / \ \begin{pmatrix} A_D \\ -c^T \end{pmatrix} v \geq \mathbf{1}$
    if v not exists **then return** x
    chose δ with $0 < \delta < h = \min\limits_{m \in [1,M] / A_m v < 0} \frac{A_m x - b_m}{-A_m v}$ (e.g. $\delta = \frac{h}{4}$)
    $x = x + \delta v$
  else
    $g = \min\limits_{m \in [1,M] \ / \ A_m u < 0} \frac{A_m x - b_m}{-A_m u}$
    $x = x + gu$

2.4 Correctness and termination

This subsection presents a proof that this slide and jump algorithm is well defined, terminates, and produces an exact optimal solution of linear program.

Lemma 2.3. *Algorithm is well defined*

Proof. Problematic steps are step 3 (non standard operation) and step 10 (set should not be empty). All other steps are standard operations.

step 10:

By assumption cx is bounded by 0 i.e. $\forall x \in \mathbb{Q}^N$, $Ax \geq b \Rightarrow cx \geq 0$. If there was ω such that $c\omega < 0$ and $A\omega \geq \mathbf{0}$, then, one could produce an unbounded admissible point $x + \lambda\omega$ as $A(x + \lambda\omega) \geq Ax \geq b$ and $c(x + \lambda\omega) \underset{\lambda\to\infty}{\to} -\infty$. So, if $c\omega < 0$ then $\exists m \in [1, M]$ / $A_m\omega < 0$.

step 3:

Step 3 is the projection on a vectorial space i.e. it consists to solve $\arg\min_{A_D u=0}(c+u)^T(c+u)$. This step can be done by Gram Schmidt algorithm, and, is more detailed in appendix. This procedure can not fail (returns $-c$ on empty input), and, always returns a vector with a strictly positive scalar product with $-c$ or $\mathbf{0}$.

□

Lemma 2.4. *Algorithm keeps the current point in the admissible space*

Proof. Steps 8 and 11 move the current point.

Step 8:

Seeing the test in step 6, $A_D v > \mathbf{0}$ so for all $m \in D$, $A_m(x + \lambda v) > b_m$. Now, for all $m \notin D$, $A_m x > b_m$, so $\exists \delta > 0$ such that $A(x + \delta v) > b$ (the offered δ works but anyway it could be less).

Step 11:

First, $\forall m$ / $A_m u \geq 0$, $A_m(x + \lambda u) \geq b$.

Then, seeing the definition of u from step 3, if $m \in D$, $A_m u = 0$ and if $m \notin D$, $A_m x > b_m$. So $\forall m \in [1, M]$, $A_m u < 0 \Rightarrow m \notin D \Rightarrow A_m x > b_m$, and so $g > 0$. Now, for m / $A_m u < 0$, g is a minimum, so $\frac{A_m x - b_m}{-A_m u} \geq g$. When multiplying by a negative: $\frac{A_m x - b_m}{-A_m u} A_m u \leq g A_m u$. And, so $\forall m \in [1, M]$, / $A_m u < 0$: $A_m x + g A_m u - b_m \geq A_m x + \frac{A_m x - b_m}{-A_m u} A_m u - b_m = 0$. □

Lemma 2.5. *algorithm outputs optimal exact solution*

Proof. The proof consists to assume the algorithm returns non optimal admissible point x, while optimal solution was x^* ($cx > cx^*$). Then, it is possible to built $\hat{x}$ (by adding εc) which are in the interior of the admissible space and still better than x. But, it means that $\hat{x} - x$ is both better for the objective and for saturated constraints. So, the subsolver should have returned something, and so, there is a contradiction.

Precisely, let consider $\phi = x^* - x + \frac{c^T(x-x^*)}{2\sqrt{c^T c}} c$, then, $A_D\phi = A_D(x^* - x + \frac{c^T(x-x^*)}{2} c) = A_D(x^* + \frac{c^T(x-x^*)}{2} c) \geq \frac{c^T(x-x^*)}{2} A_D c > \mathbf{0}$ and $c(\phi - x) = -\frac{c^T(x-x^*)}{2} < 0$. □

Lemma 2.6. *algorithm can not loop more then $M+1$ times without calling the subsolver*

Proof. g is exactly build such that the k corresponding to the minimum enters in D. Let consider $k \notin D$ such that $\frac{A_k x - b_k}{-A_k u} = g$. So, $A_k(x + gv) - b_k = A_k x + \frac{A_k x - b_k}{-A_k v} A_k v - b_k = 0$.

So, D strictly increases each time the algorithm reaches step 11, but, D is bounded by $[1, M]$, so, test 4 can not return true more than M consecutive times. □

Lemma 2.7. *All moves strictly decrease the objective function.*

Proof. Steps 8 and 11 move the current point.

step 11: By construction $c^T u < 0$, and, $g > 0$. So, cost decreases when moving along u.

step 8: $c^T v < 0$ so moving a little along v decreases the cost. □

Lemma 2.8. *A value for set D observed in step 5 can not be observed again in step 2.*

Proof. This lemma is proven by contradiction: if D is seen again, one can prove that exploration of D should have been continued (i.e. the test step 3 should have been false) instead of calling the subsolver on step 5.

Let consider $x_2 - x_1$ with x_1 corresponding to the observation of D in step 5 and x_2 to any ulterior observation. As all moves strictly decreases $c^T x$, it means $c^T(x_2 - x_1) < 0$, but, by definition of D, $A_D x_2 = A_D x_1 = b_D$ so $A_D(x_2 - x_1) = \mathbf{0}$. So, projection of $-c$ on $Ker(A_D)$ is not null (at least it could be $x_2 - x_1$), so, algorithm should not have passed the test step 3.

*More precisely, an underlying lemma is that the projection of a vector on a kernel **should** have a strictly positive scalar product with this vector if possible. This is a linear algebra result recalled in appendix.* □

Lemma 2.9 (Slide and jump)**.** *The slide and jump algorithm solves linear programs.*

Let call vertices the points x (with $Ax \geq b$) saturating the set D such that the orthogonal projection of $-c$ on $Ker(A_D)$ is null. Then, the iteration between two vertices is strongly polynomial, and, vertices are never reached twice.

Proof. From all previous lemmas, observing that D is bounded (so looping without call to subsolver is impossible), and that, number of subsets from $[1, M]$ is finite (so number of call to subsolver is bounded because subsolver is never called twice on the same value), algorithm terminates. And, independently, algorithm is well defined, works in admissible space and can not output something else than an optimal solution. So both correction and termination are proven. □

3 Discussion

3.1 Why considering a non polynomial algorithm ?

The slide and jump algorithm solves linear program because D the set of saturated constraint can not be seen again after being explored (which is equivalent

with a call of Chubanov algorithm on it). Yet, there is 2^M possible sets. So, at this point, the worse case complexity of the algorithm has no reason not to be exponential.

However, there are theoretical statements that highlights why this algorithm may be interesting.

3.1.1 Non singular vertices

Let start by the case of non degenerated vertices i.e. the projection of $-c$ on $Ker(A_D)$ is null because A_D is square and non singular (typically $Ker(A_D) = \{\mathbf{0}\}$):

Lemma 3.1 (non singular vertices)**.** *Let H being a non singular square matrix of $\mathbb{Q}^{N\times N}$ and $c \in \mathbb{Q}^N$ such that $Hc = \mathbf{1}$ and $c^Tc = 1$. Let $\phi_m = H^{-1}(-\mathbf{I}_m)$, $R = \{m, c^T\phi_m \leq 0\}$ and $S = \{m, c^T\phi_m > 0\}$. Then:*

- *$c + \sum_m \phi_m = 0$, and so, $\sum_{s\in S} c^T\phi_s + \sum_{r\in R} c^T\phi_r + 1 = 0$*
- *there is no v such that $H_Rv \geq \mathbf{0}$, $H_Sv = \mathbf{0}$ and $cv < 0$*

Proof.

- $H(c + \sum_m \phi_m) = \mathbf{1} - \sum_m \mathbf{I}_m = \mathbf{0}$, as H is non singular $c + \sum_m \phi_m$.
- if there is v such that $H_Rv \geq \mathbf{0}$, $H_Sv = \mathbf{0}$ and $cv < 0$, then, let consider $w = v + \sum_{m\in R} (H_mv)\phi_m$:
  - $\forall m \in S$, $H_mw = 0 + \sum_{m\in R} 0 = 0$ ($H_mv = 0$ by assumption and $H_i\phi_j = -\delta_{i,j} = 0$ by construction)
  - $\forall m \in R$, $H_mw = H_mv - H_mv = 0$

  as H is non singular, $w = 0$ but $c^Tw = c^Tv + \sum_{m\in R} (H_mv)(c^T\phi_m) > 0$ (each H_mv and $c^T\phi_m$ are positive by assumption on v and definition of R), so, there is a contradiction

□

Applied to our context with $H = \frac{5}{3}A_D$, it means that each time the algorithm encounter a vertex x with D such that A_D is square non singular, then, D can be split into R, S such that $\sum_{s\in S} c^TA_D^{-1}(-e_s) + \sum_{r\in R} c^TA_D^{-1}(-e_s) + \frac{5}{3} = 0$ with $c^TA_D^{-1}(-e_s) > 0$ and $c^TA_D^{-1}(-e_r) \leq 0$ and such that $A_Sv = \mathbf{0}$, $A_Rv \geq \mathbf{0}$ and $cv < 0$ is impossible.

Now, this last point is interesting seeing the next lemma:

Lemma 3.2 (Being a subset of the saturated constraints)**.** *Let x, D a vertex, $S \subset D$ and $y, \mathcal{D}$ an ulterior vertex (so $c^Ty < c^Tx$) with $S \subset \mathcal{D}$, then, $\exists v$ such that $A_Sv = \mathbf{0}$, $A_Dv \geq \mathbf{0}$, $c^Tv < 0$.*

Proof. $v = y - x$ validate the stated properties: $A_S x = A_S y = b_S$ so $A_S(y-x) = \mathbf{0}$, $c^T y < c^T x$ so $c^T(y-x) < 0$ and $A_D y \geq b_D$ (as y is admissible) while $A_D x = b_D$, so $A_D(y-x) \geq \mathbf{0}$. □

Indeed, if there exists no v such that $A_S v = \mathbf{0}$, $A_R v \geq \mathbf{0}$ and $cv < 0$, then it means that S will never been included in the set of saturated constraints though the algorithm. And, in particular, if all $c^T A_D^{-1}(-e_m)$ have a similar absolute value, R is somehow expected to be bigger than S, as so, R may be small.

Of course, this property is not sufficient, because, in worse case, R can be reduce to a singleton because all $c^T A_D^{-1}(-e_s)$ are small. In this case, the conclusion has little interest, and, could be proven more easily: let consider y just before hitting x, $y, \mathcal{D}$ leads to slide with u, then, leads to x, D. Obviously, $\mathcal{D}$ can never be included again in the saturated constraints as otherwise moving according to u leads to a contradiction.

But in best case where R is a singleton, the corresponding constraint is never seen again.

Typically, if $N = 3$ (meaning that the underlying problem is a simple 2D problem), there is always a constraint that it rejected (this will be proven bellow). Let consider 3 planes $(\frac{2\sqrt{2}}{5}, \frac{2\sqrt{2}}{5}, \frac{3}{5})$, $(\frac{2\sqrt{2}}{5}, -\frac{2\sqrt{2}}{5}, \frac{3}{5})$, $(\frac{4}{3}, 0, \frac{3}{5})$, there is not point attached to the last one bellow $\mathbf{0}$. But, this is not true in higher dimension, and, if multiple constraints are added in the same time: $(\frac{2\sqrt{2}}{5}, \frac{2\sqrt{2}}{5}, 0, \frac{3}{5})$, $(\frac{2\sqrt{2}}{5}, -\frac{2\sqrt{2}}{5}, 0, \frac{3}{5})$, $(\frac{12}{25}, 0, \frac{16}{25}, \frac{3}{5})$, $(0, 0, \frac{4}{5}, \frac{3}{5})$, $(0, 0, -\frac{4}{5}, \frac{3}{5})$ moving along x is still possible as the 3rd dimension allows the 3rd constraint to have lower ratio x,c, this way, having all constraints saturated is not possible, but, having the 1st and the 3rd or the 2nd and the 3rd is possible. Basically, this seems to be the scheme to build a worse case for the slide and jump: each time the algorithm reaches a vertices m for $\mathcal{D}$, moving along the projection of $-c$ on $A_{\mathcal{D}\setminus\{i\}\cup\{m\}}$ is possible but not moving along $A_{\mathcal{D}\cup\{m\}}$. This way, $R = \{m\}$ So even if all vertices are non singular, the complexity still could be as high as $2^{\frac{N}{2}}$ which is much lower than 2^M but still exponential. Yet, this statement highlights that the case where the set of possible D quickly decreases is no that hypothetical. Last remark: if A_D is square non singular, there is no need to call Chubanov algorithm as $u = \frac{5}{3}c + \sum_{m \in R} A_D^{-1}(-e_m)$ is an admissible vector to continue the descent (descent terminated only if S is empty, as, the point is the optimum). Also, the number of still possible sets could theoretically be actualized on the fly on strongly polynomial time.

3.1.2 Dominated constraints

To complete the previous lemma which link existence of vertices with constraints on the set of saturated constraints (D), let assume that the input of the algorithm to be $\min_{x \in \mathbb{Q}^N, \ / \ Ax \geq b} c^T x$ with the following property: $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, $Ac = \frac{3}{5}\mathbf{1}$, x_{start} being a trivial admissible point e.g. $(\frac{5}{3} \max_{m \in [1,M]} b_m)c$, and, $c^T x$ being bounded by 0.

Again, these assumptions do not restrict generality because all linear programs can be pushed in this shape (see appendix).

The interest of having normalized row is to remove ambiguity due to the scaling, and, so to have a proper geometric view of the constraints. In particular, from any admissible point x, one view x as a ball with radius d the distance to the closest constraints. Then, there is an equivalence between the closest constraints and the constraints that will become saturated is one makes x moving along

$-c$ (precisely $x - \frac{5d}{3}c$). But, most importantly, from any admissible point x saturating D, and any $m \in D$, one can derive a point y such that $A_m y = b_m$ and $A_{[1,M]\setminus\{m\}} y > b_{[1,M]\setminus\{m\}}$.

Lemma 3.3 (Linking linear separability and saturated constraints)**.** *Let x such that $Ax \geq b$ (with A normalized) and D the set of saturated constraints and $m \in D$.*

If there is y such that $Ay \geq b$, $c^T y < c^T x$ and $A_m y = b_m$ then there is w such that $A_m w = 0, A_{D\setminus\{m\}} w > \mathbf{0}, c^T w < 0$.

Proof. If there is y such that $Ay \geq b$, $cy < cx$ and $A_m y = b_m$, then let consider $z = y + \epsilon(c - \frac{3}{5}A_m^T)$ with $\epsilon < \frac{25}{9}c^T(x-y)$. Then, $A_m z = A_m y = b_m$, $cz < cx$ and $\forall i \neq m$, $A_i z = A_i y + \frac{3\epsilon}{5}(1 - A_i A_m^T)$ but $A_i A_m^T < 1$ for $i \neq m$ (and $= 1$ for m) so $A_i z > A_i y \geq b_i$. So, there is z such that $A_i z > b_i$ ($i \neq m$), $cz < cx$ and $A_m z = b_m$. And, so, $A_{D\setminus\{m\}}(z - x) > \mathbf{0}$ and $c^T w < 0$ (as $A_{D\setminus\{m\}} x = b_{D\setminus\{m\}}$ by definition). □

This property can be checked with Chubanov algorithm eventually proving that there is no such w, and, thus that, m will never be saturated again Then, it is easy to check if exists w such that $A_m w = 0$, $A_{D\setminus\{m\}}\omega > \mathbf{0}$ and $c^T w < 0$. Indeed, this can be done by combining linear feasibility and linear separability problem. Let consider the matrix $\begin{pmatrix} A_m & -A_m & \mathbf{0} & 0 \\ A_{D\setminus\{m\}} & -A_{D\setminus\{m\}} & -\mathbf{I} & 0 \\ c^T & -c^T & \mathbf{0} & 1 \end{pmatrix}$, and, let call Chubanov algorithm on it (rank is full as soon as A_m is not null due to the first row + the identity bloc). It will lead to ω verifying the desired property if one exists.

Now, thank to this lemma, one can apply the following statement:

Lemma 3.4 (Dominated constraints)**.** *Let $A \in U_{M\times N}$ be a matrix with normalized rows, and $c \in U_N$ a normalized vector such that $Ac = \frac{3}{5}\mathbf{1}$, and, D a set of index. Assume that $\exists k \in D$ such that $A_k = \sum_{m\in D\setminus\{k\}} \alpha_n A_m - \beta c^T$ with $\alpha \geq \mathbf{0}$ and $\beta \geq 0$. Then, there is no $w \in \mathbb{Q}^N$ such that $A_k w = 0$, $A_{D\setminus\{k\}} w \geq \mathbf{0}, cw < 0$*

Proof. If it exists w, then, $A_k w = 0$ but $A_k w = \sum_{m\in D\setminus\{k\}} \alpha_n A_m w - \beta c^T w < 0$, this is a contradiction. □

So under some assumptions, one can prove that some constraints is never seen twice. Of course, these assumptions are very strong, and thus, the conclusion has little relevance. But, again, this statement shows that this algorithm could have some interest despite this complexity is not bounded.

3.2 Implementation

All the previous theoretical statements highlights that not all the 2^M sets are possible as vertices. Yet, slide and jump algorithm obviously needs less linear feasibility queries than the number of possible vertices. So, one can state that: **slide and jump algorithm is strongly polynomial on all instances having a polynomial number of vertices**.

Currently, this statement is both not very useful but not that trivial. This statement is not that trivial because there is not obvious better algorithm in this case:

- interior point algorithm does not explore vertices but space so their complexity may not be lowered by the assumption, and, independently may still depend on L
- naive exploration of all sets is still exponential as enumerating only the vertices is not trivial
- simplex like algorithms explore only the vertices, but, they could cycle exponentially on some vertex while slide and jump never cycle thank to Chubanov algorithm

Yet, it is not very useful because this set of instances does not correspond to an interesting sub problem of linear programming, and, it is not possible to know the number of vertices a priori.

An better way could be to prove that even if there exists, almost all will not be explored. However, no such statement are provided in this paper. Instead, as an ultimate clue of interest, basic numerical experiments are also offered.

Little tricks are embedded in code source offered in appendix. During jump, what is really required is to find v such that $A_D v \geq \mathbf{0}$ and $c^T v < 0$ (or to prove there is not). But, there is not known strongly polynomial algorithm to find such v. This is why data are transformed such that $Ac = \frac{3}{5}\mathbf{1}$. This way, it will be equivalent to find v such that $A_D v > \mathbf{0}$ and $c^T v < 0$ (see proof before) which could be found using Chubanov algorithm which is a strongly polynomial time algorithm.

Yet, this does not change the fact that, from code point of view, subsolver produces v such that $A_D v \geq \mathbf{0}$ and $c^T v < 0$.

Then, as there is no public implementation of Chubanov algorithm, offered source code implements subsolver using projection on $-c$ on kernel of A_S with S a random subset of D in the spirit of a Kaczmarz-Motzkin algorithm [4]. Of course, using Chubanov algorithm should be better: the current implementation does not even guarantee termination. This should be changed as soon as an implementation of Chubanov will be made public.

However, the current implementation seems a *fair simulation of a true Chubanov algorithm*. This way, the experimental experiments performed with this implementation seems almost valid.

Let stress that, in the offered code, either problem are feasible, or, the primal dual of the primal dual is computed. This way, optimal solution always exists and always has 0 cost. This allows a much simpler check of termination (just checking is $Ax \geq b$ and $c^T x = 0$) that using Chubanov algorithm to wait a certificate of impossibility. This is in fact required as the current simulation of Chubanov algorithm only admits feasible instance. Yet, as computing the primal dual of the primal dual is always possible this does not restrict generality or validity of the offered experiments.

Finally, let stress that offered code is mono thread, python base and exact arithmetic base. This way, it is restricted to small instance, but still provides some striking knowledge about the algorithm.

Main point is that, in numerical experiment, slide and jump algorithm solves the Klee Minty cube (up to dimension 50) by exploring a sub linear number of vertices, whereas, simplex would have explored an exponential number of vertices on these instances. Obviously, it is unfair to compare number of explored vertices between slide and jump, and, simplex on the simplex worse case (even if slide and jump has not being designed for Klee Minty cube family). Yet, it still shows that the slide and jump algorithm can explore a low number of vertices even when there is a lot.

3.3 Perspective

The discussion invites to consider several extensions of the slide and jump algorithm. Currently, these extensions do not change the structure of the algorithm. Hence, these extensions probably do no change the complexity of the algorithm. But, they may be a way to prove either if the algorithm is or is not polynomial.

3.3.1 Hypothetical conic programming

Assuming it is possible to solve in strongly polynomial time $\min_{A_D v \geq \mathbf{0}} (c+v)^T(c+v)$, it seems to be a good idea to replace Chubanov query $\exists ? v : A_D v \geq \mathbf{0}$ and $c^T v < 0$ by $\min_{A_D v \geq \mathbf{0}} (c+v)^T(c+v)$. Even, slide moves could also be replaced by $\min_{A_D v \geq \mathbf{0}} (c+v)^T(c+v)$.

But, this is not trivial to prove that the resulting algorithm is strongly polynomial. Currently, this may even not be better to replace sliding move: moving along v such that $\min_{A_D v \geq \mathbf{0}} (c+v)^T(c+v)$ may lead to not strictly increasing D making it harder to even prove a termination.

3.3.2 Sticking constraint

One could wonder if it could be interesting to look for the maximal subset $S \subset D$ (in lexicographic order) for which A_S is full rank, and, $A_S w = 0$, $A_{D \setminus S} w > 0$, $c^T w < 0$ instead of just looking for $A_D v > 0$, $c^T v < 0$.

This can be done by calling subsolver on $\begin{pmatrix} A_S & -A_S & \mathbf{0} & 0 \\ A_{D\setminus S} & -A_{D\setminus m} & -\mathbf{I} & 0 \\ c^T & -c^T & \mathbf{0} & 1 \end{pmatrix}$.

This way, when a constraint leaves D, it is by construction for ever. However, it is not trivial that this just lead to the exact same problem just cut by $A_S x = b_S$ i.e. just leading to the same issue with $M - |S|$ constraints.

More globally, it is difficult to deal with unconstrained jumping moves: $\exists ? v \ / \ A_D v \geq \mathbf{0}, cv < 0$ could lead to different output. Typically, the slide and jump could also work with sliding moves as $\exists ? v \ / \ A_D v = \mathbf{0}, cv < 0$ but using the projection of $-c$ on $Ker(A_D)$ makes the algorithm better specified. Doing

the same for the jumping moves seems required to establish the complexity of the algorithm.

3.3.3 Sliding with maximal set

An other way to extend the algorithm is, instead of just computing the projection of $-c$ on $Ker(A_D)$, to compute the projection of $-c$ on a maximal sets of constraints sorted by the distance to x, and, in particular in order to try to keep the symmetry between constraints.

To unfold this idea in geometry, with normalization, slide and jump consists to make a ball following the gravity path to the lower point, and, this extension is about to do the same with a kind of ellipse.

The idea is first to sort all constraints according to their distance to x. This result in a partition $D = D_1, ..., D_K$ of $[1, M]$ with: $\forall i, j \in D_k, A_i x - b_i = A_j x - b_j$, and, $\forall I < J, i, j \in D_I, D_J, A_i x - b_i < A_j x - b_j$ with D.

Now, instead of considering u to be the projection of $-c$ on $Ker(A_D)$, the idea is to consider the projection of $-c$ on matrix ϕ_k and ψ_k with:

$$\phi_k = \begin{pmatrix} \psi_{k-1} \\ A^T_{i_{k,2}} - A^T_{i_{k,1}} \\ ... \\ A^T_{i_{k,r}} - A^T_{i_{k,1}} \end{pmatrix} \qquad \psi_k = \begin{pmatrix} \phi_k \\ A^T_{i_{k,1}} \end{pmatrix}$$

So first priority is to keep distance within $D = D_1$ equal (ϕ_1), and, then unchanged (ψ_1). Then, priority is to keep distance within D_2 equal (ϕ_2) and then unchanged (ψ_2). Then, the priority is to keep distance within D_2 equal... When it is not possible to handle more constraint, u is returned and used to do a sliding move, or, if $\neg A_D u \geq \mathbf{0}$ a jumping move is done instead.

Currently, the move along u can be done either until some plan enter in D or until the partition change. Computing u each time partition change seems a more straightforward implementation of the idea. Yet, just changing the computation of u, and, keeping all other instructions unchanged allows to apply exactly the same proof that for the original algorithm, and, thus, this updated version also solves linear program.

The main difference with the original version is that if i, j, k were in D, then only k then i but not j, and, then j but not i, it raises the question about why algorithm does not keep i, j as a block.

Yet, currently no better bound has been built on this observation.

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 strongly polynomial algorithm for linear systems having a binary solution. *Mathematical programming*, 134(2):533–570, 2012.

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

[4] Jesus A De Loera, Jamie Haddock, and Deanna Needell. A sampling kaczmarz–motzkin algorithm for linear feasibility. *SIAM Journal on Scientific Computing*, 39(5):S66–S87, 2017.

[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] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

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

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

[10] 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.

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

Appendix

Normalizing linear program

If the linear program given as input is $\min_{Ax \geq b} cx$ and verifies $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$ and $Ac = \gamma \mathbf{1}$ with $\gamma > 0$, and, cx being bounded by 0, then the offered algorithm of section 2 can be directly used.

Otherwise, the linear program has to be normalized with the following scheme:

1. If the linear program is as an optimisation problem (e.g. $\max_{Ax \leq b, x \geq \mathbf{0}} cx$), it should first be converted into a inequality system $A'x \geq b'$. This could be done by combining primal and dual.

2. After that (or directly is input was an inequality system), an other normalisation is performed to reach required property (here with $\gamma = \frac{3}{5}$)

3. the important point is that from any linear program, pre processing can form an equivalent linear program meeting these requirements

Primal dual

The conversion of an linear program to be optimized into a linear inequality system is quite classical. A brief recall is provided bellow.

Let assume original goal is to solve $\max_{A_{raw}x \leq b_{raw}, x \geq \mathbf{0}} c_{raw}x$. It is well known that the dual problem is $\min_{A_{raw}^T y \geq c_{raw}, y \geq \mathbf{0}} b_{raw}y$. Now, the primal dual is formed by combining all constraints: $A_{raw}x \leq b_{raw}$, and, $x \geq \mathbf{0}$, and, $A_{raw}^T y \geq c_{raw}$, and $c_{raw}x = b_{raw}y$, and finally, $y \geq \mathbf{0}$.

So, the problem $\max_{A_{raw}x \leq b_{raw}, x \geq \mathbf{0}} c_{raw}x$ can be folded into $A_{big}x_{big} \geq b_{big}$ with

$$A_{big} = \begin{pmatrix} -A_{raw} & 0 \\ I & 0 \\ 0 & A_{raw}^T \\ 0 & I \\ c_{raw} & -b_{raw} \\ -c_{raw} & b_{raw} \end{pmatrix} \text{ and } b_{big} = \begin{pmatrix} -b_{raw} \\ 0 \\ c_{raw} \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Normalized primal dual error minimization

This normalisation step takes a linear program $\Gamma\chi \geq \beta$ as input, and, produces an equivalent linear program $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, and, $Ac = \frac{3}{5}\mathbf{1}$, and, cx being bounded by 0.

It is sufficient to consider:

$$A = \begin{pmatrix} \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\Gamma_1 & \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\frac{\Gamma_1\Gamma_1}{2} & \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)} & \frac{3}{5} \\ \dots & \dots & \dots & \\ \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\Gamma_M & \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\frac{\Gamma_M\Gamma_M}{2} & \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)} & \frac{3}{5} \\ \mathbf{0} & \frac{4}{5} & 0 & \frac{3}{5} \\ \mathbf{0} & -\frac{4}{5} & 0 & \frac{3}{5} \\ \mathbf{0} & 0 & \frac{4}{5} & \frac{3}{5} \\ \mathbf{0} & 0 & -\frac{4}{5} & \frac{3}{5} \end{pmatrix}$$

and

$$b = \begin{pmatrix} \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\beta_1 \\ \dots \\ \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\beta_M \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad , \quad c = \begin{pmatrix} 0 \\ \dots \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

First, the produced linear program is in the desired form: $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}_{J,I}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, and, $Ac = \frac{3}{5}\mathbf{1}$.

Trivially, $Ac = \frac{3}{5}\mathbf{1}$ by construction, and, all rows of A are normalized either directly because $(\frac{4}{5})^2 + (\frac{3}{5})^2 = 1$, or, because of that, and the fact that, $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2\Gamma_m\Gamma_m + (\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2\frac{(\Gamma_m\Gamma_m)^2}{4} + (\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2$ is $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2 \times (\Gamma_m\Gamma_m + \frac{(\Gamma_m\Gamma_m)^2}{4} + 1)$ which is $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2 \times (\frac{\Gamma_m\Gamma_m}{2} + 1)^2$ which is 1 !

Then, the 4 last constraint prevent x_{N+3} to be negative so cx is well bounded by 0. Indeed, if $x_{N+2} + x_{N+3} \geq 0$ and $-x_{N+2} + x_{N+3} \geq 0$, then $x_{N+3} \geq 0$), and, when $x_{N+3} = 0$ these constraints force $x_{N+1} = x_{N+2} = 0$ because the three constraints $x_{N+2} + x_{N+3} \geq 0$, $-x_{N+2} + x_{N+3} \geq 0$, and $x_{N+3} = 0$ can be reduced to $x_{N+2} \geq 0$ and $-x_{N+2} \geq 0$ which force $x_{N+2} = 0$.

Now, the goal is to minimize $cx = x_{N+3}$. So, either the minimum is $x_{N+3} = 0$ or either there is no such solution. In the case $Ax \geq b, x_{N+3} = 0$, it holds $x_{N+1} = x_{N+2} = x_{N+3} = 0$, and, $x_1, ..., x_N = \chi$ with $\frac{4}{5(\frac{\Gamma_m\Gamma_m}{2}+1)}\Gamma_m\chi \geq \frac{4}{5(\frac{\Gamma_m\Gamma_m}{2}+1)}\beta_m$. But this last inequality can be reduced to $\Gamma_m\chi \geq \beta_m$. So, if the solution of the derived linear program is x with $Ax \geq b, x_{N+3} = 0$, then $x_1, ..., x_N = \chi$ is a solution of the original set of inequality.

And, inversely, if there is a solution χ, then, $x = \chi, 0, 0, 0$ is a solution of the optimisation problem (because x_{N+3} is bounded by 0).

So, this derived linear program is equivalent to the inequality set.

For any linear program, it is possible to create a derived form meeting the requirement of the offered algorithm.
All this normalization is entirely done in $\mathbb{Q}$ i.e. no square root are needed.

Projection on vectorial space

Let $c \in \mathbb{Q}^N$ and $A \in \mathbb{Q}^{M \times N}$, let consider the problem: $\min_{p \in \mathbb{Q}^N, Ap = \mathbf{0}} (c-p)^T(c-p)$.

Let $\nu_1, ...\nu_K$ be a basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$, completed by $\nu_{K+1}, ..., \nu_N$ into a basis $\mathbb{R}^N$. By applying Gram Schimd, one forms $e_1, ..., e_K, ..., e_N$ a orthonormal basis of $\mathbb{R}^N$ such that $e_1, ..., e_K$ is a basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$.

Now, as $e_1, ..., e_N$ is an orthonormal basis of $\mathbb{R}^N$, $c = \sum_{k \in [1,N]} e_k^T c e_k$. And, as $e_1, ..., e_N$ is an orthonormal basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$, it means that $\forall p, Ap = \mathbf{0} \Rightarrow p = \sum_{k \in [1,K]} e_k^T p e_k$. In particular, $\forall p, Ap = \mathbf{0} \Rightarrow (c-p)^T(c-p) \geq \sum_{k \in [K+1,N]} (e_k^T c)^2$. Independently, $q = \sum_{k \in [1,K]} e_k^T c e_k$ verifies both that $Aq = \mathbf{0}$ and that $(c-q)^T(c-q) = \sum_{k \in [1,N] \setminus [1,K]} (e_k^T c)^2$.

So q is the solution of $\min_{p \in \mathbb{Q}^N, Ap = \mathbf{0}} (c-p)^T(c-p)$. Thus, solving $\min_{p \in \mathbb{Q}^N, Ap = \mathbf{0}} (c-p)^T(c-p)$ is easy, and, can be used in algorithm 1.

Some source code

```
from __future__ import print_function

##############################################################################
############################## MAIN ALGORITHM ################################
##############################################################################

from fractions import Fraction

import random

import datetime

################ BASIC LINEAR ALGEBRA FUNCTIONS ################

def combinaisonlineaire(u,l,v):
    return [u[n]+l*v[n] for n in range(len(u))]

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

def opposedvector(v):
    return [-e for e in v]

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

def saturatedconstraints(A,b,x):
    return [m for m in range(len(A)) if produitscalaire(A[m],x)==b[m]]

def projection(u, BOG):
    pu = [Fraction()]*len(u)
    for v in BOG:
        l = produitscalaire(u,v)/produitscalaire(v,v)
        pu=combinaisonlineaire(pu,l,v)
    return pu

def resteprojection(u,BOG):
    return combinaisonlineaire(u,Fraction(-1),projection(u,BOG))

def gramschimdBOG(H,BOG):
    BOG = BOG.copy()
    while True :
        H = [resteprojection(h,BOG) for h in H]
        H = [h for h in H if produitscalaire(h,h) != Fraction()]
        if H!=[]:
            BOG.append(H.pop())
        else:
            return BOG

#returns v such that Av=0 and (v+a)(v+a) is minimal
def projection_on_ker(A,a,BOG):
    BOG = gramschimdBOG(A,BOG)
    if len(BOG)==0:
        return opposedvector(a),BOG
    else:
        return resteprojection(opposedvector(a),BOG),BOG

################ ALGORITHM FOR LP ################


#ASSUME exist v: Av>0, av<0
#return v such that Av>=0, av<0
## COULD BE IMPLEMENTED AS CHUBANOV ALGORITHM ##
## BUT CURRENTLY TERMINATION IS NOT GUARANTEE ##
def no_guarantee_termination_subsolver(A,a):
    counter=0
    while True:
        index = [i for i in range(len(A))]
        random.shuffle(index)
        subsetI = index[0:random.randint(0,len(index))]

        subset = [A[i] for i in subsetI]
        v,BOG = projection_on_ker(subset,a,[])

        AV = [produitscalaire(A[i],v) for i in range(len(A))]
        FORBID = [A[i] for i in range(len(A)) if AV[i]<Fraction()]
        while FORBID!=[]:
            subset += FORBID
            v,BOG = projection_on_ker(subset,c,BOG)
            AV = [produitscalaire(A[i],v) for i in range(len(A))]
            FORBID = [A[i] for i in range(len(A)) if AV[i]<Fraction()]

        counter+=1
        value = produitscalaire(v,a)
        if value<Fraction():
            print("nb trial in false Chubanov",counter)
            return v
```

```
#ASSUME Ax>=b, xc>0, Ac>0, Ay>=b with cy=0
#let d in D <=> A_dx=b_d
#return v such that A_Dv>=0 and cv<0
def jump(A,b,c,x):
    D = [i for i in range(len(A)) if produitscalaire(A[i],x)==b[i]]
    block = [A[i] for i in D]
    return no_guarantee_termination_subsolver(block,c)

#returns l such that A (x+lv) >= b, l maximal
## FAIL ON UNBOUNDED SITUATION ##
## FAIL IF NO POSITIVE L EXIST ##
def maximalmoves(A,b,x,v):
    AV = [produitscalaire(A[m],v) for m in range(len(A))]
    AXb =  [produitscalaire(A[m],x)-b[m] for m in range(len(A))]

    S = [m for m in range(len(A)) if AV[m]<Fraction()]
    if S==[]:
        print("maximalmoves: S==[]")
        quit()

    R = [-AXb[m]/AV[m] for m in S]
    l = min(R)
    if l==Fraction():
        print("maximalmoves: l==0")
        quit()

    return l

#ASSUME Ax>=b
#ASSUME Ax>=b => cx>=0
#returns y such that cy<=cx, Ay>=b and projection_on_ker(A_D,c)=0
# with d in D <=> A_Dy=b_D
def slide(A,b,c,x):
    print("entering slide",datetime.datetime.now())
    BOG = []
    while True:
        print("$", end="", flush=True)
        D = saturatedconstraints(A,b,x)

        v,BOG = projection_on_ker([A[m] for m in D],c,BOG)
        if produitscalaire(v,v)==Fraction():
            print("exiting slide",datetime.datetime.now())
            return x
        l = maximalmoves(A,b,x,v)
        x = combinaisonlineaire(x,l,v)


#ASSUME Ac >0, Ax>=b => cx>=0
#return x such that Ax>=b and cx minimal
def slideandjump(A,b,c):
    x = [Fraction()]*len(A[0])
    x[-1] = Fraction(max(b)+1) * Fraction(5,3)
    counter = 0

    while True:
        print("sliding move")
        x = slide(A,b,c,x)
        D = saturatedconstraints(A,b,x)
        print(D)

        if produitscalaire(x,c)==Fraction():
            print("found optimal")
            return x,counter

        print("chubanov jump")
        v = jump(A,b,c,x)
        l = maximalmoves(A,b,x,v)/4
        x = combinaisonlineaire(x,l,v)
        counter+=1


##############################################################################
########################## PRE PROCESSING ####################################
##############################################################################

def normalize(rawA, rawb, rawxoptimal):
#input rawA rawx >= rawb
#return A,b,c such that min{cx / Ax>=b} if equivalent
#+ A is normalized, c is normalized, cx is 0 bounded, Ac = 3/4 vector(1)
    M = len(rawA)
    N = len(rawA[0])
    A = [[Fraction() for n in range(N+3)] for m in range(M+4)]
    b = [Fraction()]*(M+4)
    c = [Fraction()]*(N+3)
    c[-1] = Fraction(1)

    normRawA = [Fraction()]*M
    normRawAtrick = [Fraction()]*M
    for m in range(M):
```

```
            normRawA[m] = produitscalaire(rawA[m],rawA[m])
            normRawAtrick[m] = normRawA[m]/Fraction(2)+Fraction(1)

        for m in range(M):
            for n in range(N):
                A[m][n] = rawA[m][n]*Fraction(4,5)/normRawAtrick[m]
            A[m][-3] = normRawA[m]*Fraction(4,5*2)/normRawAtrick[m]
            A[m][-2] = Fraction(4,5*2)/normRawAtrick[m]
            A[m][-1] = Fraction(3,5)
            b[m] = rawb[m]*Fraction(4,5)/normRawAtrick[m]

        A[-4][-3] = Fraction(4,5)
        A[-4][-1] = Fraction(3,5)
        A[-3][-3] = -Fraction(4,5)
        A[-3][-1] = Fraction(3,5)
        A[-2][-2] = Fraction(4,5)
        A[-2][-1] = Fraction(3,5)
        A[-1][-2] = -Fraction(4,5)
        A[-1][-1] = Fraction(3,5)

        xoptimal = [Fraction()]*(N+3)
        for n in range(N):
            xoptimal[n] = rawxoptimal[n]

        return A,b,c,xoptimal

    def primaldual(rawA,rawb,rawc):
    #primal: max {rawc rawx / rawA rawx<= rawb, rawx>=0}
    #dual: min {rawb rawy / transpose(rawA) rawy>= rawc, rawy>=0}
    #primal dual: {rawx / rawA rawx<=rawb, rawx>=0,
    #     transpose(rawA) rawy >=rawc, rawy>=0, rawc rawx=rawb rawy}
    #unfolded into A x >= b
        M = len(rawA)
        N = len(rawA[0])
        A = [[Fraction() for n in range(N+M)] for m in range(M+N+N+M+2)]
        b = [Fraction()]*(M+N+N+M+2)

        offsetY = N
        offset = 0
        for m in range(M):
            for n in range(N):
                A[m+offset][n] = -rawA[m][n]
            b[m+offset] = -rawb[m]

        offset += M
        for n in range(N):
            A[n+offset][n] = Fraction(1)

        offset += N
        for n in range(N):
            for m in range(M):
                A[n+offset][m+offsetY] = rawA[m][n]
            b[n+offset] = rawc[n]

        offset += N
        for m in range(M):
            A[m+offset][m+offsetY] = Fraction(1)

        for n in range(N):
            A[-2][n] = rawc[n]
        for m in range(M):
            A[-2][m+offsetY] = -rawb[m]

        for n in range(N):
            A[-1][n] = -rawc[n]
        for m in range(M):
            A[-1][m+offsetY] = rawb[m]

        return A,b

################################################################################
############################## TOY EXPERIMENT  #################################
################################################################################

    def cubeproblemPrimal(N):
        twopower =[Fraction()]*N
        twopower[0]=Fraction(2)
        for n in range(1,N):
            twopower[n] = Fraction(2)*twopower[n-1]

        b = [Fraction()]*N
        b[0] = Fraction(5)
        for n in range(1,N):
            b[n] = Fraction(5) * b[n-1]

        c = twopower[::-1]

        A = [[Fraction() for n in range(N)] for m in range(N)]
        for n in range(N):
            for k in range(n):
                A[n][k] = twopower[n-k]
```

```
            A[n][n] = Fraction(1)

        return A,b,c

    def cubeproblem(N):
        Araw,braw,craw = cubeproblemPrimal(N)
        A,b = primaldual(Araw,braw,craw)

        xoptimal = [Fraction()]*(2*N)
        xoptimal[-1]=Fraction(2)
        xoptimal[N-1] = braw[-1]

        return A, b, xoptimal

    def randomVector(N):
        return [Fraction(random.randint(-100,100)) for n in range(N)]
    def randomNegVector(N):
        return [Fraction(random.randint(-100,-1)) for n in range(N)]

    def randomMatrix(M,N):
        return [randomVector(N) for m in range(M)]

    def randomproblem(N,M):
        xoptimal = randomVector(N)

        Aequal = randomMatrix(M,N)
        bequal = [produitscalaire(Aequal[m],xoptimal) for m in range(M)]

        Agreater = randomMatrix(M,N)
        left = randomNegVector(M)
        bgretmp = [produitscalaire(Agreater[m],xoptimal) for m in range(M)]
        bgreater = [bgretmp[m] + left[m] for m in range(M)]

        return Aequal+Agreater,bequal+bgreater,xoptimal

##########################################################################
############################## MAIN ######################################
##########################################################################


    print("########### random ###########")
    rawA,rawb,rawxoptimal = randomproblem(10,30)
    A,b,c,xoptimal = normalize(rawA,rawb,rawxoptimal)

    x,_ = slideandjump(A,b,c)
    if any([produitscalaire(A[m],x)<b[m] for m in range(len(A))]):
        print("????????")
    if produitscalaire(c,x)!=Fraction():
        print("????????")



    print("########### cube ###########")

    for N in [15,20,25,30,40,50,60]:
        rawA,rawb,rawxoptimal = cubeproblem(N)
        A,b,c,xoptimal = normalize(rawA,rawb,rawxoptimal)

        x,counter = slideandjump(A,b,c)
        if any([produitscalaire(A[m],x)<b[m] for m in range(len(A))]):
            print("????????")
        if produitscalaire(c,x)!=Fraction():
            print("????????")

        print("==============> cube:",N,"leads to",counter,"jumps")
```