A Chubanov based algorithm with almost linear complexity for linear programs with very large binary size and few number of vertices.

Adrien Chan-Hon-Tong

A Chubanov based algorithm with almost linear complexity for linear programs with very large binary size and few number of vertices.

Adrien CHAN-HON-TONG

September 2, 2019

Abstract

This paper presents an algorithm for linear programming based on Chubanov algorithm for homogeneous linear feasibility.

Despite there is little hope that this algorithm is polynomial for generic linear programming, it is theoretically the current best algorithm to solve linear programs with very large binary size but small number of vertices.

1 Introduction

Linear programming is the very studied task of solving $\min_{x \ / \ Ax \geq b} c^T x$ 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 L be the binary size required to write the given input (i.e. A, b, c), then, this problem can be solved in polynomial times (i.e. in a number of binary operations bounded by a L^γ with a fixed γ) since [9, 6, 8], with interior point method [10] being currently the state of the art.

Yet, there is no known algorithm proven to have a strong polynomial complexity i.e. able to solve the problem in a number of binary operations bounded by $L \times (MN)^\gamma$. And, [1] shows that major interior point algorithms are not strong polynomial. Today, only, some families of linear programs can be solved in strongly polynomial times:

- linear program with 0/1 matrix A [12] (by specific algorithm)
- linear program with at most two variables per inequality [7] (by specific algorithm)
- Markov chain [11] (by simplex algorithm [4])
- system having binary solution [2] (by specific algorithm)
- homogeneous linear feasibility i.e. $\exists ? x \in \mathbb{Q}^N \ / \ Ax = \mathbf{0}, \ x > \mathbf{0}$ given $A \in \mathbb{Q}^{M \times N}$ a full rank matrix [3] by Chubanov algorithm.

As no known algorithm is strongly polynomial, there is an interest to design an algorithm to solve linear programs with very large binary size using prior on the geometry of the problem (this claim is formalized in next section). Then, section 3 presents an algorithm based on Chubanov algorithm [3] for linear programming. Despite there is little hope that this algorithm is polynomial in generic case, it solves theoretically the issue of linear programs with very large binary size when number of vertices is small. This algorithm shares some feature with simplex but exits vertex in strongly polynomial time. These claims are discussed in section 4.

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.

2 Limit of state of the art

There is no known algorithm to solve linear programming in strongly polynomial time, and, major family of interior point are proven unable to do so [1]. Thus, there is a practical and theoretical issue for the state of the art to deal with linear programs with very large integer. Let formalize theoretically this issue.

Let first recall the classical primal dual trick to go from optimization problem into a decision one: Let assume original goal is to solve $\max_{A_{raw}x \leq b_{raw}, x \geq \mathbf{0}} c^T_{raw}x$. It is well known that the dual problem is $\min_{A^T_{raw}y \geq c_{raw}, y \geq \mathbf{0}} b^T_{raw}y$. Now, the primal dual is formed by combining all constraints: $A_{raw}x \leq b_{raw}$, and, $x \geq \mathbf{0}$, and, $A^T_{raw}y \geq c_{raw}$, 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

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

Then, this problem $A_{big}x \geq b_{big}$ can be transformed by adding a variable z into $A_{big}x + z \geq b_{big}$, $z \geq 0$, then, there is an equivalence between feasibility $\exists ?x \ / \ A_{big}x \geq b_{big}$ and minimization of z with constraints $A_{big}x + z \geq b_{big}$, $z \geq 0$.

Now, let introduce the large integer linear programming problem:

Definition 2.1 (large integer linear programming)**.** *The large integer linear programming problem is to decide if* $\exists x \in \mathbb{Q}^N \ / \ Ax \geq b, c^T x = 0$ *or* $NM \geq \log(L)$ *given* $A \in \mathbb{Q}^{M \times N}$, $b \in \mathbb{Q}^M$ *and* $c \in \mathbb{Q}^N$ *with* $Ac > \mathbf{0}$ *and* $A_M = c$ *and* $b_M = 0$.

For all instances for which $NM \geq \log(L)$, the output is trivially true. So, all algorithms could be derived to solve these instances trivially. So, the interesting instances are those with $NM \leq \log(L)$. Any strongly polynomial algorithm will have an almost linear complexity i.e. $\Theta(L) = O(L \times \log^\gamma(L))$ with a constant γ, but, any non strongly polynomial algorithm will be super linear i.e. with a L^γ complexity with $\gamma > 1$.

The interesting point is that no algorithm are known almost linear to solve large integer linear programming and major state of the art algorithm are known not to be (counter examples from [1] have unbounded L, eventually, $NM \leq \log(L)$).

Now, one could consider simplex with classical pivoting rule [4] as a candidate for large integer linear programming as simplex complexity depend on the number of pivots which could be bounded depending on N and M and not L. However, it is not almost linear because it could need an exponential (in N, M) number of integer operations to terminate. Precisely, let introduce the concept of vertex:

Definition 2.2 (vertex)**.** *Given* $A \in \mathbb{Q}^{M \times N}$, $b \in \mathbb{Q}^M$ *and* $c \in \mathbb{Q}^N$ *with* $Ac > \mathbf{0}$ *and* $A_M = c$ *and* $b_M = 0$*, a set of index* $D \subset [1, M]$ *is called a* ***vertex*** *if and only if* $\exists x \in \mathbb{Q}^N$ *such that* $A_D x = b_D$ *and* $A_{[1,M]\setminus D} x > b_{[1,M]\setminus D}$*, and, there is no* $v \in \mathbb{Q}^N$ *such that* $A_D v = 0$, $c^T v < 0$.

In non singular case, a vertex is just a point x *such that* $A_D x = b_D$ *with* A_D *square and non singular, plus,* $A_{[1,M]\setminus D} x > b_{[1,M]\setminus D}$ *(so, obviously, there is no* $v \in \mathbb{Q}^N$ *such that* $A_D v = 0$, $c^T v < 0$*). But, in degenerated case,* $A_D x = b_D$ *may define a sub vectorial space included in a iso-c hyperplane.*

Let write $\Gamma(A, b, c)$ ***the number of vertices given*** A, b, c ***(at most bounded by*** 2^M***).***

Now, the idea is that vertex based algorithm are efficient if number of vertices is small.

Definition 2.3 (few vertex large integer linear programming)**.** *The few vertex large integer linear programming problem is to decide if* $\exists x \in \mathbb{Q}^N \ / \ Ax \geq b, c^T x = 0$ *or* $NM \geq \log(L)$ *or* $\Gamma(A,b,c) \geq \log(L)$ *given* $A \in \mathbb{Q}^{M \times N}$, $b \in \mathbb{Q}^M$ *and* $c \in \mathbb{Q}^N$ *with* $Ac > \mathbf{0}$ *and* $A_M = c$ *and* $b_M = 0$.

Obviously, this problem is unrealistic and not interesting in practice. But from theoretical point of view, this problem is well defined.

Now, simplex algorithm (for example) may become interesting for this problem compared to interior point because during the exploration either there is less than $\log(L)$ vertices and the simplex will output a solution after having explored these only $\log(L)$ vertices, or, there is more vertices but the algorithm could stop anyway after the $\log(L)$ first explored vertices. So complexity will be $\Theta(L \times \log(L) \times \Upsilon)$ with Υ being the complexity of the algorithm to go from a vertex to an other. Yet, the problem with the simplex (with all known pivoting rules) is that it can require exponential time to exit a vertex, so, Υ is not just bounded by $(NM)^\gamma$.

However, an algorithm similar to the simplex, but, with the feature of exiting a vertex with strongly polynomial complexity will be especially interesting from the few vertex large integer linear programming (obviously this problem is designed in this purpose). Fortunately, such algorithm, build from Chubanov algorithm, is presented on this paper in next section.

3 The slide and jump algorithm

3.1 Key idea

The starting point of this algorithm is Chubanov algorithm [3] which solves homogeneous 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) and/or it dual $\exists ? x \in \mathbb{Q}^N \ / \ Ax > \mathbf{0}$ in strongly polynomial time.

Then, it is easy to get a small improvement from a not optimal admissible solution for linear program by solving a dual homogeneous linear feasibility problem. Indeed, if x verifying $Ax \geq b$ has not minimal cx under assumption that $Ac > \mathbf{0}$, then it is possible (see proof in subsection 3.3) to find v such that $\begin{pmatrix} A_D \\ -c^T \end{pmatrix} v > \mathbf{0}$ with D the current saturated constraints (i.e. D such that $A_D x = b_D$).

Now, it could take infinite time to reach the optimal solution by computing such v and updating $x = x + \varepsilon v$. Yet, if x is not on a vertex, it is always possible to decrease $c^T x$ just by moving in the kernel of current saturated constraints. So, the algorithm does sliding moves to reach a vertex from a non vertex point, and, then, a jumping move to go from a vertex to an interior point. The good feature 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 a vertex D twice, and, the correctness, by capacity to jump to next D until optimal solution is reached.

3.2 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.

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

1: **while** True **do**
2: $D = \{m \in [1, M] \ / \ A_m x = b_m\}$
3: compute u the orthogonal projection of $-c$ on $Ker(A_D)$
4: **if** $u = \mathbf{0}$ **then**
5: call subsolver $\exists ? v \ / \ \begin{pmatrix} A_D \\ -c^T \end{pmatrix} v \geq \mathbf{1}$
6: **if** v not exists **then return** x
7: chose δ with $0 < \delta < h = \underset{m \in [1,M] / A_m v < 0}{min} \frac{A_m x - b_m}{-A_m v}$ (e.g. $\delta = \frac{h}{4}$)
8: $x = x + \delta v$
9: **else**
10: $g = \underset{m \in [1,M] \ / \ A_m u < 0}{min} \frac{A_m x - b_m}{-A_m u}$
11: $x = x + gu$

3.3 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 3.1. *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 $\underset{A_D u = 0}{\arg\min}(c+u)^T(c+u)$. This step can be done by Gram Schmidt algorithm. 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 3.2. *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 3.3. *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 for c. 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 3.4. *algorithm can not loop more then $M+1$ times without calling the subsolver (equivalently reaching a vertex)*

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 (for inclusion, as set) 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 3.5. *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 3.6. *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. □

Lemma 3.7 (Slide and jump)**.** *The slide and jump algorithm solves linear programs because 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 reaching a vertex is impossible), and that, number of subsets from $[1, M]$ is finite an never see twice (i.e number vertices is bounded and vertex are never explored twice), 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. □

4 Discussion

4.1 Theoretical view

The main claim of this paper is the following:

Theorem 4.1. *The slide and jump algorithm solves in almost linear time the few vertex large integer linear programming.*

Proof. Complexity of the slide and jump algorithm to decide a few vertex large integer linear programming instance is bounded by $L \times \log^{\mu}(L) \times \log(L)(NM)^{\gamma} \leq L \times \log^{1+\mu+\gamma}(L)$ where $(NM)^{\gamma}$ is the number of integer operation to go from a vertex to an other, and, μ the exponent such that integer operation can be done in $O(L \log^{\mu}(L))$ binary operations. Indeed, such step consists in calling Chubanov algorithm (strongly polynomial) plus projecting at most M times the vector $-c$ on a sub space defined by at most M vectors. So such step is strongly polynomial, and, hence solved in a number of integer operation bounded by some $(NM)^{\gamma}$. And, such step is done at most $\log(L)$ times. □

This result may have almost none practical implication as the few vertex large integer linear programming is a very unrealistic problem. **Yet, from theoretic point of view, the offered algorithm is almost linear on this problem while all known algorithms are super linear on it**:

- interior point are naturally super linear (in L)

- simplex may require exponential (in NM) number of operations to exit a vertex
- direct enumeration of all sets D may be exponential (in NM): enumerating set is easy but enumerating vertices is hard, as set can be unsatisfiable, or, incomplete (i.e. moving into D is possible)
- only best algorithm would be un algorithm able to enumerate only vertices, and, not just sets of index - no such algorithm is known today

4.2 Perspective

4.2.1 Complexity

There is little hope that slide and jump algorithm is strongly polynomial for standard linear program. Indeed, depending on the implementation of inner subsolver (to solve $A_D v \geq \mathbf{0}, c^T v < 0$), slide and jump can behave more or less like a simplex. The most distant behaviour is if $A_D v > \mathbf{0}$, as, it is even possible that two consecutive vertices have no constraint in common (but it depends not just on how x exit the vertex, but also, how x collects constraints to reach a vertex). But, the closest behaviour is when the produced vector v verifies $A_{D\setminus\{d\}} v = \mathbf{0}$ i.e. moves follow an edge to directly go from a vertex to an adjacent vertex - like a simplex with the corresponding pivoting rule.

So, slide and jump could be just a particular simplex with the important feature that exiting a vertex is strongly polynomial. So, without additional implementation detail, there is no reason for slide and jump to have a better complexity than simplex (except for exiting vertex) which is exponential for all known pivoting rules.

Yet, complexity of the simplex (polynomial Hirsch conjecture) is still opened. So, not behaving like simplex is not especially interesting as it. What will be interesting is a polynomial termination (even if most moves follow edges), but, this question is out of the scope of this paper. Harder point is that link between consecutive vertices in slide and jump does not depend only on the exiting condition (jumping moves), but, also on how constraint are collected (sliding moves).

4.2.2 Non singular case

To advance on this complexity question, one can consider a non singular vertex: A_D (written A here) is square and non singular. The goal is to find $Av \geq \mathbf{0}, c^T v < 0$ (with $Ac > \mathbf{0}$).

As the vertex is non singular, one can consider A^{-1} and $\phi_1, ..., \phi_N$ the vector such that $(A\phi_n)_i = \begin{cases} 1 & if\ i = n \\ 0 & otherwise \end{cases}$ (i.e. the transposed of the row of A^{-1}).

Let $\Omega = \{n \in [1, N] \ / \ c^T \phi_n >= 0\}$ and $\Upsilon = \{n \in [1, N] \ / \ c^T \phi_n < 0\}$.

Υ is important because it will never again be included in any vertices. Indeed, there is no ψ such that $A_\Upsilon \psi = \mathbf{0}$, $A_\Omega \psi \geq \mathbf{0}$, and, $c^T \psi < 0$. If it was, by

removing the corresponding vector from Ω, one could form a vector such that $A\omega = 0$ and $c^T\omega < 0$ (this is impossible both because it is a vertex and because A is non singular).

The basic idea of the simplex (with greedy descent pivot rule) is to move following ϕ_k with $c^T\phi_k = \min_{n\in\Upsilon} c^T\phi_n$.

Now, if one is interested to push slide and jump in the simplex framework, it seems that the underlying pivoting rule should be to move along $\sum_{n\in\Upsilon} \phi_n$ (corresponding to solve $v = \min_{\mathbf{0}\leq A\nu\leq\mathbf{1}} c^T\nu$ in singular case - but it is not relevant to solve such linear problem just to exit a vertex...). Typically, if Υ is just a pair i, j, greedy descent pivot rule will select either i either j while it seems interesting to move on the bisector of both. These way, when hitting a constraint k, k will give a direction which may lead to hit i xor j. But the good point is that if i is the one meet after k, then, at least the vertex j, k has been avoided.

Now, this exiting rule probably already be studied (even if depending on the sliding move), and, is probably also exponential. The only real new is that the slide and jump is able to deal with singular vertex.

4.2.3 Implementation

The question about if this algorithm is relevant in practice is harder. First, it requires exact arithmetic (because relevancy is when L is large). Then, mainly, it depends on the efficiency of the Chubanov implementation, and, on the average number of encountered vertices.

Small numerical experiments are provided to evaluate this last point (see source code in appendix), but, as there is no public implementation of Chubanov algorithm, these experiments are somehow limited. To bypass this issue 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 [5] (This is equivalent as adding ϵc provides v such that $A_D v > \mathbf{0}$ - currently the algorithm does not require functionally that $A_D v > \mathbf{0}$ and $c^T v < 0$, but, only that $A_D v \geq \mathbf{0}$ and $c^T v < 0$ - or to prove there is not). Of course, using Chubanov algorithm should be better: the current implementation does not even guarantee termination. This would be changed as soon as an implementation of Chubanov will be made public.

Also, direct running time is not very relevant as the offered code is mono thread, python base and exact arithmetic base. This way, it is somewhat slow, and thus, restricted to small instances. Yet, it still provides some interesting hint about the algorithm.

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.

Main point is that, in numerical experiments, slide and jump algorithm solves the basic tests:

- it solves the Klee Minty cube (up to dimension 50) by exploring very low number of vertices
- it solves random instances by exploring low number of vertices

Obviously, there is no big challenge in solving either Klee Minty cubes (even if the slide and jump is not designed for, these instances are easy except for naive simplex) or random instances. But, these are the minimal tests to be candidate of being relevant, and, the point is that the slide and jump at least solves them.

This result highlights that this algorithm should not immediately be dumped, and, invites to perform more complete experiments on slide and jump numerical behaviour.

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] 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] 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.

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

[7] 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.

[8] 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.

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

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

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")
```