Revisiting simplex method in the light of Chubanov algorithm.

Adrien Chan-Hon-Tong

Revisiting simplex method in the light of Chubanov algorithm.

Adrien CHAN-HON-TONG

August 16, 2019

Abstract

Simplex core is to move from vertices to vertices following edges.

This short paper argues that it could be relevant to modify simplex with a specific preprocessing, and, the strongly polynomial time Chubanov algorithm for homogeneous linear feasibility.

The preprocessing links moving far away from constraints and minimizing the objective value. This way, one could move from vertices to vertices following faces of undertermined dimension. Then, Chubanov algorithm can be used to solve the homogeneous linear feasibility problem corresponding to the computation of such move.

The main feature of this algorithm is to handle both singular and non singular vertices in a strongly polynomial number of operations contrary to raw simplex method which may takes exponential time to exit a degenerated vertex.

Despite, there is little hope that complexity is strongly polynomial in worse case (as core of the algorithm is close to simplex core), the offered algorithm may outperforms interior point state of the art if binary size is very large compared to number of vertices, and, marginally, opens the question about if moving along faces may lead to a faster convergence than moving along edges.

1 Introduction

Linear programming is the very studied task of solving $\min_{x\in\mathbb{Q}^N \ / \ Ax\geq b} c^T x$ with $A \in \mathbb{Q}^{M\times N}$ a matrix, $b \in \mathbb{Q}^M$ and $c \in \mathbb{Q}^N$ some vectors. This problem is equivalent to solve $\exists ? x \in \mathbb{Q}^N \ / \ Ax \geq b$ with A and b some bigger matrix (classical primal dual trick).

This problem can be solved in polynomial times since [9, 6, 8] 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. [10]). 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 [12] (by specific algorithm)
- linear program with at most two variables per inequality [7] (by specific algorithm)
- Markov chain [11] (by simplex algorithm)
- system having binary solution [2] (by specific algorithm)
- homogeneous linear feasibility i.e. $\exists ?x \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ (by Chubanov algorithm) [3]

Interestingly, despite having exponential worse case complexity, the simplex algorithm [4] is not affected by the binary size as it handles set of contraint indexes, and, not raw values. This is why, revisiting simplex algorithm in the light of Chubanov algorithm could be relevant.

Precisely, this short paper offers two increments to form a simplex like algorithm with the good features of handling singular vertices efficiently (i.e. in a strongly polynomial number of operations) contrary to simplex, and, of moving along faces instead of edges.

Even if, complexity of the new algorithm is not established, there is little hope it is not exponential. Yet, good feature of the offered algorithm could be interesting. Currently, this algorithm seems the current best shot to solve linear programming instances with very large binary size, but, with a number of vertices bounded by a strongly polynomial number (especially when all vertices are singular).

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.

1.1 Core idea

Let c be the objective vector (our goal is to minimize $c^T x$ while ensuring $Ax \geq b$), and, let consider a vertices: a sub matrix of A (let call it A) $A \in \mathbb{Q}^{M \times N}$ such that there is no $\phi \in \mathbb{Q}^N$, $A\phi = \mathbf{0}$ and $c^T \phi < 0$. The main step of the algorithm is to find v such that $Av \geq \mathbf{0}$ and $c^T v < 0$. Then, the algorithm will move the current point along v either finding an unbounded situation, or, hitting some new constraints typically reaching a new vertices.

If A is square and 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 (steppest descent) is to move following ϕ_k with $c^T \phi_k = \min_{n \in \Upsilon} c^T \phi_n$.
- The greedy optimal vector $v = \min_{A\nu \geq \mathbf{0}} (c + \nu)^T (c + \nu)$ is an optimal combination of vectors from both Ω and Υ
- Solving $v = \min_{\mathbf{0} \leq A\nu \leq \mathbf{1}} c^T \nu$ correspond to move along $\sum_{n \in \Upsilon} \phi_n$
- Other ideas could be to use optimal combination from only vectors of Υ, or, other.

Unfortunately, none of these ideas seems to lead to a strong polynomial time algorithm (even assuming that these v can be efficiently combinated). Even, the idea of using all vectors from Υ because it split symetrically the vertices into one set kept and one set never see again.

The idea of the slide and jump (which is probably not polynomial either) is basically to use all vectors. This can be done using Chubanov algorithm: requiring $Aw \geq 1$ (this is the dual of $Ax = 0, x \geq 1$ from Chubanov algorithm).

Of course, at this point nothink prevent that some of the constraint will no be included in the next vertex (after leaving a vertex by a face, one need to collect other constraints - this is direct with simplex as moving along edge allows to move directly from vertex to vertex) - even if the collecting of contraints could probably be updated to try to forbid old contraint to enter in the next vertex.

Yet, the main point is that like simplex, the algorithm moves from vertex to vertex but vertex are not adjacent - move is done in the interior of the polytop.

2 The slide and jump algorithm

2.1 Pseudo code

Algorithm 1 presents the pseudo code of the offered algorithm.

Let stress that the 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: $x = x + gu$

2.2 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 2.1. *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 $\arg\min_{A_D u=0}(c+u)^T(c+u)$. 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 2.2. *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 2.3. *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. 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 2.4. *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 2.5. *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 2.6. *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.

*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.* □

Lemma 2.7 (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 Hypothetical use case

Let state one of the main feature of this algorithm: **despite being possibly exponential (like simplex), the offered algorithm is currently the best theoretical way to solve linear programs with very large binary size, but with strongly polynomially bounded number of vertices if large ratio of these vertices being singular.**

Indeed, simplex algorithm is exponential for two reasons: it may take exponential time to exit a vertex, and, it may meet an exponential number of vertices. As opposite, the slide and jump is possibly exponential only because it may meet an exponential number of vertices. But, it handles each vertices (singular or not) in a strongly polynomial number of operations.

So, simplex algorithm may take exponential time to deal with an instance of linear program with polynomially bounded number of vertices if most of them are singular, while, slide and jump will be strongly polynomial in this case.

Now, if the problem has very large binary size, current interior point could be inefficient even if number of vertices is bounded. In this case, using slide and jump is relevant. To joke a little, if one consider a problem where the input is a matrix $\mathcal{A}$ and a vector b, and, the expected output is x such that $Ax \geq b$ with A the matrix $A_{i,j} = 3^{\mathcal{A}_{i,j}}$. Then, current interior point methods are not polynomial algorithm to solve this problem (neither simplex or slide and jump - except for slide and jump if number of vertices is polynomially bounded).

3.2 Numerical experiment

Now, main limitations of this algorithm is the need to handle infinite precision number to keep a connection between set of constraint indexes and point in the polytope, and, necessity to have a subsolver implementing Chubanov algorithm.

These limitations are important pratical issues. So, this short paper currently offers only limited numerical experiments with infinite precision arithmetic, and, *fake* implementation of Chubanov algorithm - in the spirit of a Kaczmarz-Motzkin algorithm [5] (code source offered in appendix).

In these numerical experiments, this slide and jump implementation solves the Klee Minty cube (up to dimension 50) and random instances by exploring very low number of vertices (typically less than 20 for cube in dimension 50 on 10 runs). Let stress that a large set of solvers are not able to deal with Klee Minty cube in dimension 50 with some number reaching 2^{50}.

Currently, these experiments invite to improve this algorithm to make it efficient, and so, to allow a real comparition with the state of the art: it is currently not interesting, but, it passes minimal requirements to not be dumped.

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.

Source code of numerical experiment

```
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")
```