An interesting link between linear feasibility, linear programming, support vector machine and convex hull.

Adrien Chan-Hon-Tong

An interesting link between linear feasibility, linear programming, support vector machine and convex hull.

Adrien CHAN-HON-TONG

April 28, 2019

Abstract

This short paper presents an algorithm based on simple projections and linear feasibility ($Ax = \mathbf{0},\ x > \mathbf{0}$) queries which solves both linear programming ($\min_{x\ /\ Ax \geq b} cx$), and, support vector machine ($\min_{w\ /\ Aw \geq 1} w^T w$).

In addition, this algorithm is strongly polynomial on families of instances of linear program and support vector machine, characterized by similar convex hull properties.

Thus, this algorithm could be interesting as a link between all these four notions: linear feasibility, linear programming, support vector machine and convex hull.

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 [15, 8, 13] i.e. in a number of binary operation bounded by the binary size required to write the matrix A. State of the art algorithm to solve linear program are today interior point algorithms (e.g. [19]). In addition, several family of linear program can be solved in strongly polynomial times i.e. i.e. in a number of rational operation bounded by the size of A:

- combinatorial linear program [22]
- linear program with at most two variables per inequality [11]
- markov chain [21]
- system having binary solution [4]

Support vector machine [6] used to be a central tool of machine learning (before deep learning [16]), and, consist to solve $\min_{w\ /\ Aw \geq 1} w^T w$. Complexity of support vector machine seems to be polynomial as a special case of semi

definite programming complexity [9], although semi definite complexity under degeneracy is not that clear [10] (when *solving* is about producing an exact solution plus a certificate). Approximate solutions for support vector machine exist like [12] (theoretically proven) or [14, 3] (efficient implementations), but, for exact solution, no better algorithm than semi definite ones seems exist.

Independently, [5] recently shows that linear feasibility i.e. $Ax = \mathbf{0},\ x > \mathbf{0}$ can be solved in strongly polynomial time. This result is especially interesting, because [2] shows that major interior point family does not solve linear program in strong polynomial time. And, so, the question of solving linear program as a strong polynomial sequence of linear feasibility queries is relevant.

This short paper focus on this last point, and, presents, in section 2, an algorithm which solves, at least, both linear programming and support vector machine by solving a sequence of linear feasibility. Discussion of section 3 shows that families of linear programming and support vector machine problems can be solved in strong polynomial time under assumptions linked to convex hull properties.

Thus, this short paper provides at least an interesting links between linear feasibility, linear programming, support vector machine and some convex hull properties.

Notations

$\mathbb{N}, \mathbb{Q}$ are the sets of integer and rational numbers. $\setminus$ is the ensemble subtraction. $\forall n \in \mathbb{N}\setminus\{0\}$, $R(n)$ is the range of integer from 1 to n i.e. $R(n) = \{k \in \mathbb{N}, 1 \leq k \leq n\} = \{1, ..., n\}$.

$\forall I, J \in \mathbb{N}\setminus\{0\}$, $\mathbb{Q}^I$ is the set of I dimensional vector of $\mathbb{Q}$, and, $\mathbb{Q}_{I,J}$ is the set of matrix with I rows and J columns, with values in $\mathbb{Q}$. $\mathbb{Q}^I$ would be matched with $\mathbb{Q}_{I,1}$ i.e. vector are seen as a column vector.

$\forall i \in \mathbb{N}\setminus\{0\}$, $._i$ designs the i component: a rational for vector or a row for a matrix. Rows are seen as row vector i.e. if $\forall A \in \mathbb{Q}_{I,J}$, $A_i \in \mathbb{Q}_{1,J}$. Also ^{T} is the transposition operation i.e. $\forall I, J \in \mathbb{N}\setminus\{0\}$, $\forall A \in \mathbb{Q}_{I,J}$, $A^T \in \mathbb{Q}_{J,I}$ with $\forall (i,j) \in R(I) \times R(J)$, $A^T_{j,i} = A_{i,j}$. For all set $S \subset \mathbb{N}$, A_S, b_S is the submatrix or subvector obtained when keeping only rows or components $s \in S$.

$\mathbf{0}$ 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.

$\mathbb{U}_I$ is the set of normalized vector from $\mathbb{Q}^I$ i.e. such that $v^T v = 1$ i.e. $\mathbb{U}_I = \{v \in \mathbb{Q}^I, v^T v = 1\}$. $\mathbb{U}_{I,J}$ is the set of matrix from $\mathcal{M}_{I,J}$ whose rows (after transposition) are in $\mathbb{U}_J$ (only the rows, not necessarily the columns).

Also, if $A \in \mathcal{Q}_{I,J}$, then, the null vector space of A (i.e. the kernel) is written $Ker(A) = \{v \in \mathbb{Q}^J / Av = \mathbf{0}\}$.

All notations are quite classical except $R(n)$ for the range of integer from 1 to n, $\mathbb{Q}_{I,J}$ is the set of matrix with I rows and J columns (more often written $M_{I,J}(\mathbb{Q})$ and $\mathbb{U}$ to indicate normalization of vector and/or matrix.

2 Slide and jump algorithm

2.1 Key idea

Let first consider the following lemma:

Lemma 2.1. *If it is possible to solve* $\exists ?x \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ *when* $A \in \mathbb{Q}_{M,N}$ *has full rank, then it is possible to solve* $\exists ?y, \mathcal{A}y > \mathbf{0}$ *for any* $\mathcal{A} \in \mathbb{Q}_{I,J}$.

Proof. From $\mathcal{A} \in \mathbb{Q}_{I,J}$, it is sufficient to consider the matrix $A \in \mathbb{Q}_{I,J\times 3}$ formed by $\mathcal{A}$ concatenate with $-\mathcal{A}$ concatenate with $-\mathbf{I}$ (which has full rank due to the identity bloc). Solution can be extracted from $J \times 2$ first variables of derived problem, J last variable ensure positivity. This is more formally proven in appendix. □

With the ability to call a strong polynomial time subsolver [5] dedicated to linear separability ($Ax > \mathbf{0}$), it is easy to get a small improvement from a not optimal admissible solution either for linear program or support vector machine. Indeed, let consider support vector machine as an example, if w verifying $Aw \geq \mathbf{1}$ has not minimal norm, then it is possible to find u such that $\begin{pmatrix} A_D \\ -w^T \end{pmatrix} u > \mathbf{0}$ with D the current support vectors i.e. D such that $A_D w = \mathbf{1}$. Now, it could take infinite time to reach the optimal solution by computing such u and updating $w = w + \varepsilon u$. These moves can be see as *jump* because its always lead to some improvement independently from current saturated constraint. But jumping is not enough.

But, the algorithm also considers *sliding* moves: it considers the current most problematic constraints D, and, solves the easy optimization problem derived by freezing this set i.e. it solves $\min_{w \ / \ A_D = \mathbf{1}} w^T w$. This 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 the algorithm to explore completely a particular combinatorial situation (D) while *jumping* moves allow to exit such structure. The termination will be guarantee by the careful exploration (sliding) of D, and, the capacity to jump to next D until optimal solution is reached.

One key point of this short paper is that this slide and jump idea can solve different problems, at least linear program and support vector machine.

Unfortunately, as there is an exponential number of situations D, no better bound than an exponential bound is proven, here, for this algorithm. Still, this algorithm is strongly polynomial for both linear program and support vector machine under the *same* kind of assumptions linked with convex hull properties.

Finally, in some way, this algorithm is close to [17] but

- it solves linear program and support vector machine
- it is *interior point like* for linear program

- it uses linear feasibility queries to exit hard vertex (jumping move) instead of dealing with combination of constraints: so it replaces a not clearly polynomial sub algorithm from [17] by a strongly polynomial one (thank to Chubanov algorithm [5] for linear feasibility)
- this algorithm comes with a simple characterization of families of instances on which it is strongly polynomial (including the dimension lower than 3)

2.2 Pseudo code

For linear program, the slide and jump algorithm assumes the input linear program is $\min_{x \in \mathbb{Q}^N, \ / \ Ax \geq b} c^T x$ with $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, $Ac = \frac{3}{5}\mathbf{1}$, x_{start} being a trivial admissible point e.g. $(1 + \frac{5}{3}\max_m b_m)c$, and, $c^T x$ being bounded by 0.

These assumptions do not restrict generality because all linear programs can be derived under this form (see appendix). Currently, algorithm can work even with non normalized matrix A and vector c, but, strongly polynomial behaviour is highlighted by normalization.

Pseudo code for linear program is presented in algorithm 1.

Algorithm 1 Slide and jump for linear program

1: $d = \min_{m \in R(M)} A_m x - b_m$
2: $D = \{m \in R(M) \ / \ A_m x - b_m = d\}$
3: compute u the orthogonal projection of $-c$ on $Ker(A_D)$
4: **if** $cu < 0$, $A_D u = \mathbf{0}$ **then**
5: $\quad g = \min_{m \in R(M) \ / \ A_m u < 0} \frac{A_m x - b_m - d}{-A_m u}$
6: $\quad x = x + gu$
7: $\quad$ **GO TO 1**
8: call subsolver $\exists ? v \ / \ \begin{pmatrix} A_D \\ -c^T \end{pmatrix} v > \mathbf{0}$
9: $z = x - \frac{5d}{3}c$
10: **if** v not exists **then**
11: $\quad$ **return** z and the certificate
12: $v = \frac{1}{\min_{m \in D} A_m v} v$
13: $h = \min_{m \in R(M)/A_m v < 0} \frac{A_m z - b_m}{-A_m v}$
14: $x = z + \frac{h}{4} v$
15: **GO TO 1**

For support vector machine, the slide and jump algorithm assumes the input problem is $\min_{w \in \mathbb{Q}^N, \ / \ Aw \geq \mathbf{1}} w^T w$ with $A \in \mathbb{U}_{M,N}$. As, a single call to the subsolver allows to know if $Aw \geq \mathbf{1}$ admits a solution ($\exists w \ / \ Aw > \mathbf{0} \Leftrightarrow \exists w \ / \ Aw \geq \mathbf{1}$ because it is sufficient to scale the vector by the min of $A_m w$), algorithm can assume to start from an admissible w.

Again, algorithm can work with non normalized vectors, but, strongly polynomial behaviour is highlighted by normalization. Independently, normalizing input is relevant from machine learning point of view as this gives equivalent importance to all vectors.

Pseudo code for support vector machine is presented in algorithm 2.

Algorithm 2 Slide and jump for support vector machine

1: $w = \frac{1}{\min\limits_{m \in R(M)} A_m w} w$
2: $D = \{m \in R(M) \ / \ A_m w = 1\}$
3: compute u be the orthogonal projection of $-w$ on $Ker(A_D)$
4: **if** $wu < 0$, $A_D u = \mathbf{0}$ **then**
5: **if** $A(w + u) \geq \mathbf{1}$ **then**
6: $w = w + u$
7: **else**
8: $g = \min\limits_{m \in R(M)/A_m u < 0} \frac{A_m w - 1}{-A_m u}$
9: $w = w + gu$
10: **GO TO 2**
11: call subsolver $\exists ? v \ / \begin{pmatrix} A_D \\ -w^T \end{pmatrix} v > \mathbf{0}$
12: **if** v not exists **then**
13: **return** w and the certificate
14: **if** $A(w - \frac{wv}{vv} v) \geq \mathbf{1}$ **then**
15: $w = w - \frac{wv}{vv} v$
16: **else**
17: $v = \frac{1}{\min\limits_{m \in D} A_m v} v$
18: $h = \min\limits_{m / A_m w < 0} \frac{A_m w - 1}{-A_m v}$
19: $w = w + \frac{h}{4} v$
20: **GO TO 1**

2.3 Correctness and termination

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

Lemma 2.2. *Algorithms are well defined*

Proof. Global idea: In steps are 5, 13 for linear program and 8, 18 for support vector machine, algorithms consider minimum on conditional subset of $R(M)$. Yet, these sets are non empty by construction for support vector machine, or, because problem is bounded for linear program.

Tricky points: For linear program, 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 $\exists m \in R(M)$ / $A_m\omega < 0$.

For support vector machine, *assuming the current point is admissible*, when reaching step 8 (resp. 18), exists m such that $A_m(w+u) < 1$ (resp. $A_m w + \frac{wv}{vv}v < 1$). Yet, $Aw \geq 1$. So, $A_m u \leq A_m(w+u) - 1 < 0$ (resp. the same for v). □

Lemma 2.3. *Algorithms keep the current point in the admissible space*

Proof. Global idea: $g > 0$ because $\forall m \in R(M),\ A_m u < 0 \Rightarrow m \notin D \Rightarrow A_m w > 1 or A_m x > b_m$. Now, g is a minimum, so $\frac{A_m x - b_m - d}{-A_m u} \geq g$ or $\frac{A_m w - 1}{-A_m u} \geq g$. When multiplying by a negative: $\frac{A_m x - b_m - d}{-A_m u} A_m u \leq g A_m u$ or $\frac{A_m w - 1}{-A_m u} A_m u \leq g A_m u > 0$. And, so $\forall m \in R(M),\ A_m u < 0$: $A_m x + g A_m u - b_m \geq A_m x + \frac{A_m x - b_m - d}{-A_m u} A_m u - b_m = d$ or $A_m(w+gu) \geq A_m w + \frac{A_m w - 1}{-A_m u} A_m u = A_m w + 1 - A_m w = 1$. And, there is no problem for others m. So steps 6 or 9 are safe.

Tricky points: For v, it is necessary to ensure that D is still the set of saturated constraint for z (for linear programming code). Yet, $A_m z > b_m \Leftrightarrow m \notin D$ because $A_m(x + \frac{3d}{5}c) - b_m = A_m x - b_m + d$.

But steps 14 or 19 are obviously safe: for u, g is designed to force D to increase. But, for v, any $0 < \varepsilon \ll 1$ could be used: any $x + \varepsilon v$ allows to go away from $m \in D$ for any $\varepsilon > 0$ while $\varepsilon \ll 1$ allows to be sure not to meet $m \notin D$. h should be seen as en example of such ε because either $m \in D$ and $A_m(z+hv) - b_m = h A_m v > 0$ or $m \notin D$ and $A_m(z+hv) - b_m = \frac{3}{4}(A_m z - b_m) > 0$. But other h could be considered. In support vector machine code, $A_m(w+hv) \geq \frac{3}{4}(A_m w - 1) > 0$. □

Lemma 2.4. *algorithms output optimal exact solution*

Proof. Global idea: Let assume the algorithm returns non optimal admissible point z, w while optimal solution were z^* and w^* ($cz > cz^*$ and $(w^*)^T(w^*) < w^T w$). It is possible to built $\hat{z}$ (by adding εc) and $\hat{w}$ (by scaling by $1+\varepsilon$) which are in the interior of the admissible space and still better than z, w. But, it means that $\hat{z} - z$ or $\hat{w} - w$ are both better for the objective and for saturated constraints. So, the subsolver should have returned something.

Tricky points: For linear program, let consider $\phi = z^* - z + \frac{c^T(z-z^*)}{2}c$ (put $c^T c$ in denominator if not 1), then, $A_D\phi = A_D(z^* - z + \frac{c^T(z-z^*)}{2}c) = A_D(z^* + \frac{c^T(z-z^*)}{2}c) \geq \frac{c^T(z-z^*)}{2} A_D c > \mathbf{0}$ and $c(\phi - z) = -\frac{c^T(z-z^*)}{2} < 0$.

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

$w^T((\sqrt{\frac{w^Tw}{4(w^*)^T(w^*)}}+\frac{1}{2})w^*) < \sqrt{w^Tw}\sqrt{w^Tw} = w^Tw$ (because $\alpha^T\beta \leq \sqrt{\alpha^T\alpha} \leq \sqrt{\beta^T\beta}$). So $w\psi = (\sqrt{\frac{w^Tw}{4(w^*)^T(w^*)}}+\frac{1}{2})w^Tw^* - w^Tw < w^Tw - w^Tw = 0$. □

Lemma 2.5. *Both algorithms can not loop more then $M+1$ times without calling the subsolver*

Proof. Global idea: g is exactly build such that the k corresponding to the minimum enters in D. Let consider $k \notin D$ such that $\frac{A_kx-b_k-d}{-A_ku} = g$ or $\frac{A_kx-1}{-A_ku} = g$. So, $A_k(x+gv)-b_k = A_kx + \frac{A_kx-b_k-d}{-A_kv}A_kv - b_k = d$ or $A_k(w+gu) = A_kw + \frac{A_kw-1}{-A_ku}A_ku = A_kw + 1 - A_kw = 1$.

So, D strictly increases each time the algorithm reaches step 6 or 9, but, D is bounded by $R(M)$, so, test 4 can not return true more than M consecutive times.

Tricky points: For support vector machine, if program reaches step 6, w will become $w+u$ with u the orthogonal projection of $-w$ on $Ker(A_D)$. But, the projection of u is itself (by definition of a projection). So, the projection of $-(w+u)$ on $Ker(A_D)$ is $u+(-u) = \mathbf{0}$. So, if the program reaches step 6, then during the next loop, test step 4 is false and algorithm directly calls subsolver. □

Lemma 2.6. *A value of set D can not be observed during the algorithm after a call to the subsolver on this value.*

Proof. Global idea: First, all moves strictly decreases c^Tz or w^Tw because by construction $u^Tc < 0$ and $v^Tc < 0$ or $u^Tw < 0$ and $v^Tw < 0$ and $g, h > 0$. So, if D is seen again, it means that exploration of D should have been continued instead of calling the subsolver.

Tricky point: When reaching step 8 in linear program code or step 11 in support vector machine one, it means that projection of $-c$ or $-w$ on $Ker(A_D)$ does not lead to u such that $A_Du = 0$ and $c^Tu < 0$ or $w^Tu < 0$. Now, let assume to observe D again after having called subsolver.

For linear program, let consider $z_2 - z_1$ with z_2 being $z = x - \frac{5}{3}c$ when observing D again, and, z_1 be z when observing D just before calling the subsolver. As all moves strictly decreases c^Tz, it means $c^T(z_2-z_1) < 0$, but, by definition of D, $A_Dz_2 = A_Dz_1 = \mathbf{0}$ so $A_D(z_2-z_1) = \mathbf{0}$. So, algorithm should not have passed the test step 4 when meeting z_1, because $z_2 - z_1$ should have been considered.

For support vector machine code, let consider $w_2 - w_1$. First, $A_D(w_2 - w_1) = 0$. Then, $w_1^T(w_2-w_1) = w_1^Tw_2 - w^1w_1 \leq \sqrt{w_1^Tw_1^T}\sqrt{w_2^Tw_2^T} - w^1w_1 < \sqrt{w_1^Tw_1^T}\sqrt{w_1^Tw_1^T} - w_1^Tw_1^T = 0$. So, again, algorithm should not have passed the test step 4 when meeting w_1, because $w_2 - w_1$ should have been considered.

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.* □

From all previous lemmas, observing that D is bounded (so looping without call to subsolver is impossible), and that, number of subset D from $R(M)$ is finite (so number of call to subsolver is bounded), 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.

Theorem 2.7. *The slide and jump algorithm solves both linear program and support vector machine.*

3 Discussion

3.1 Solving other problems

More generally, slide and jump algorithm can solve any $\min_{x \in \mathbb{Q}^N \ / \ Ax \geq b} f(x)$ with the following requirements:

- f has no local minima
- For any D, let $\rho_D = \min_{x \in \mathbb{Q}^N \ / \ A_D x = b_D} f(x)$ - ρ_D is either a rationnal is the problem is bounded or $-\infty$. Then, For all $x \in \mathbb{Q}^N \ / \ A_D x = b_D$,
  - either $f(x) \neq \rho_D$, and, an oracle gives a continuous function $\sigma_{x,D}$ of $[0, \infty[$ to $\mathbb{R}^N$ such that $\sigma_{x,D}(0) = x$, $f(\sigma_{x,D}(t))$ is decreasing with $f(\sigma_{x,D}(\infty)) = \rho_D$, and, one can compute the t corresponding to the intersection of this trajectory with any hyperplane
  - either $f(x) = \rho_D$, and, an other oracle gives $\tau_x \in \mathbb{Q}^N$ such that $\forall v \in \mathbb{Q}^N$, $v\tau_x > 0 \Leftrightarrow (\exists \delta > 0 \ / \ \forall 0 < \varepsilon \leq \delta, f(x + \varepsilon v) < f(x))$
- $\exists \theta \in \mathbb{Q}^N / \ A\theta > \mathbf{0}$
- $z = \arg\min_x f(x)$ is known and $\neg(Az > b)$ (currently if $Az \geq b$, it is sufficient to return z)

Under these assumptions, the algorithm 3 solves this generic problem.

The algorithm is very similar to linear program or support vector machine version:

- steps 13 to 23 are sliding move: the current point is moved thank to the first oracle into the vectorial space of currently saturated constraints to decrease objective value, eventually meeting an other constraints (D increases) or proving that problem is unbounded or reaching the minimum regarding D.
- when the minimum according to D is reached, the idea is to use the second oracle to exit D (while still being admissible and decreasing objective value).

Algorithm 3 Generic slide and jump algorithm

1: **if** $Ax > b$ **then**
2: let t the first intersection between $\sigma_{free}(t)$ and one $A_m y = b_m$
3: $x = \sigma_{free}(t)$
4: $D = \{m \in R(M) \ / \ A_m x - b_m\}$
5: **if** $f(x) = \min_{y \ / \ A_D y = b_D} f(y)$ **then**
6: call the subsolver: $\exists ? v \ / \begin{pmatrix} A_D \\ \tau_x \end{pmatrix} v > \mathbf{0}$
7: **if** no such v exists **then**
8: **return** x
9: **else**
10: find ε such that $A(x + \varepsilon v) > b$ and $f(x + \varepsilon v) < f(x)$
11: $x = x + \varepsilon v$
12: **GO TO 1**
13: **else**
14: let t the first intersection between $\sigma_{x,D}(t)$ and one $A_m y = b_m$ $(m \notin D)$
15: **if** no such t exists **then**
16: **if** $\rho_D = \infty$ **then**
17: **return** problem is unbounded
18: **else**
19: $x = \arg\min_{y \ / \ A_D y = b_D} f(y)$
20: **GO TO 6**
21: **else**
22: $x = \sigma_{x,D}(t)$
23: **GO TO 4**

- So, D will never be seen again, and, as exiting D is always possible when D is not the optimal set (thank to the second oracle), it means that the algorithm terminates and produces an optimal solution.

Basically, all these assumptions become simple if f is derivable and convex:

- $\sigma(t)$ can simply be the interval $[x, y_D]$ with $y_D = \underset{y \ / \ A_D y = b_D}{\arg\min} f(y)$
- τ_x is just the opposite of the gradient

So, in simpler words, the slide and jump algorithm solves any $\min_{Ax \geq b} f(x)$ for f convex as soon as solving $\min_{A_D x \geq b_D} f(x)$ is possible, computing gradient of f is possible, and, $\exists x / Ay > b$ i.e. admissible space has an interior.

Yet, naively exploring all possible sets D is also an algorithm to solves convex programming. But, the naively exploration has little interest because it is obviously an exponential algorithm ! So, slide and jump algorithm may be interesting (as it solves a very large set of problem) only if it was better than naive exploration. Unfortunately, at this point, nothing proves that the slide and jump algorithm is better.

Yet, in specific case of linear programming and support vector machine characterized by convex hull properties, the slide and jump algorithm is strongly polynomial.

3.2 Special cases with strongly polynomial time termination

The families of instances on which slide and jump algorithm is proven **strong polynomial** requires normalization of rows of the matrix A (see appendix) contrary to termination/correctness which do not. Also, for linear program, it required a interior point like exploration that may not be completely necessary for simple termination.

This normalization allows to characterize geometrically the fact of being into D: $k \in D$ implies that $A_m(x - dA_k^T) - b_m or A_m(w - A_k)$ is null for $m = k$ but strictly positive otherwise. Indeed, $A_m(w - A_k^T) = A_m w - A_m A_k^T$. If, $m \neq k$, $A_m A_k^T < 1$ (rows of A are normalized), so $A_m(w - A_k^T) > 0$. For linear program, $A_m(x - dA_k^T) - b_m \geq d - dA_m A_k^T > 0$.

Now, let introduce some notations related to convex hull:

$$\forall \mathcal{A} \in \mathbb{Q}_{I,J}, \text{ let write:}$$
$$Conv(\mathcal{A}) = \{\mathcal{A}^T \pi \ / \ \pi \in \mathbb{Q}^I, \pi \geq \mathbf{0} \wedge \mathbf{1}^T \pi = 1\}$$
$$Conv_+(\mathcal{A}) = \{\lambda \chi \ / \ \chi \in Conv(\mathcal{A}), \lambda \in \mathbb{Q}, \lambda \geq 1\}$$

$Conv(\mathcal{A})$ is the convex hull of rows of $\mathcal{A}$ i.e. the set of linear combination with scalar coefficients being in the simplex of $\mathbb{Q}^M$ of rows of $\mathcal{A}$ (after transposition). And $Conv_+(\mathcal{A})$ is the set of vectors x that can be scaled by a number $\frac{1}{\lambda}$ with λ higher than 1 such that $\frac{1}{\lambda} x \in Conv(\mathcal{A})$.

The bridge that links convex hull and the complexity of slide and jump algorithm is to observe that:

Lemma 3.1. $a \in Conv(\mathcal{A}) \Rightarrow \nexists \omega \in \mathbb{Q}^N \ / \ \mathbf{0} \leq a^T\omega\mathbf{1} < A\omega$

Proof. if both ω exists and $a = A^T\pi$, then, $\omega^T a = \omega^T A^T\pi = (A\omega)^T\pi > (\omega^T a\mathbf{1}^T)\pi = (\omega^T a)(\mathbf{1}^T\pi) = \omega^T a$ i.e. $\omega^T a > a^T\omega$. □

And, even,

Lemma 3.2. $a \in Conv_+(\mathcal{A}) \Rightarrow \nexists \omega \in \mathbb{Q}^N \ / \ \mathbf{0} \leq a^T\omega\mathbf{1} < A\omega$

Proof. $a \in Conv_+(\mathcal{A})$ means that $\exists\lambda \geq 1$ such that $\frac{1}{\lambda}a \in Conv(\mathcal{A})$. For previous lemma, there is not ω such that $\mathbf{0} < \frac{1}{\lambda}a^T\omega\mathbf{1} < A\omega$. Now, if there was ω such that $\mathbf{0} < a^T\omega\mathbf{1} < A\omega$, then obviously, $\mathbf{0} < \frac{1}{\lambda}a^T\omega\mathbf{1} < a^T\omega\mathbf{1} < A\omega$ as $\lambda \geq 1$. □

The global idea linking convexity and complexity is that being into an ulterior D is not compatible with being in $Conv_+$ of current D. So let states the theorem that makes slide and jump algorithm potentially interesting:

Theorem 3.3. *Let consider a support vector machine problem where D^* the set of optimal support vectors verifies that for all $m \in R(M)$, $A_m \in Conv_+(A_{D^*})$, then, the slide and jump algorithm will solve this instance in strongly polynomial time with almost only sliding move (at most M sliding move + one call to subsolver to initialize w, and, one to check that the solution is optimal).*

Proof. Let $k \notin D^*$ such that $A_k \in Conv_+(A_{D^*})$, and, let assume that at some point k enters into $D \neq D^*$.

Then let consider $\omega = w - A_k^T$. $A_k\omega = 1 - 1 = 0$ ($A_k w = 1$ because $k \in D$, and, $A_k^T A_k = 1$ by assumption). But, $\forall m \in D^*$, $A_m\omega \geq 1 - A_m A_k^T > 1 - 1 = 0$ except if $A_m A_k^T = 1$ which is impossible as $k \notin D^*$. ω is a direct contradiction with previous lemma.

So none of the $k \notin D^*$ can be in D, so the algorithm will start with a subset of D^* and performs at most $M + 1$ sliding moves to collect all the set D^*, and, terminates after calling the subsolver to get a certificate. □

In other words, if the convex hull of all vectors is in the cone formed by the convex hull of all support vectors. Then, slide and jump algorithm will solve this instance very efficiently. Let stress that the convex hull of all vectors or the admissible space be complex, only count the fact of being inside the cone formed by the support vectors. So this convex hull assumption is not trivial.

A similar theorem exists for linear program.

Theorem 3.4. *If for any, $A_D z = b_D$ on which the subsolver is called, there is $k \in D$ such that $A_k \in Conv_+(\begin{smallmatrix} A_{D\setminus\{k\}} \\ -c^T \end{smallmatrix})$, then k is never observed in any ulterior D, and so, under the assumption that at least one index is rejected at each call of the subsolver, then, the algorithm for linear program terminates after at most M calls to the subsolver.*

Proof. Let consider $A_D z = b$ on which the subsolver is called. Let assume $\exists k \in D$ such that $A_k \in Conv_+(\begin{smallmatrix} A_{D\setminus\{k\}} \\ -c^T \end{smallmatrix})$. Let suppose k is seen on an other set D after having called the subsolver.

Let consider $\mu = x - dA_k$. $A_k\mu = 0$ by definition. And, $-c^T\mu < 0$ because x is strictly bellow z (bellow from c point of view) and $c^T A_k > 0$. And, $A_m\mu \geq d - dA_m^T A_k > 0$. This is a direct contradiction with the previous lemma. □

One could wonder if such theorem holds for generic convex programming.

Yet, even if such theorem may exist, it may miss the real idea lying behind: the theorem about strong polynomial cases is just about geometrical properties of vectors (of course, if $a = 2b$, then a is never in D because b should have enter first), but, the real idea is that the hypothesis leading to strong polynomial complexity matches the algorithms behaviour.

For example, in support vector machine, in average case, support vectors tend to be far from each other. This is always the case for $N \leq 2$, because in 2D a set of vectors in a half space are *between* the two extreme ones. Of course, this is not true in higher dimension: for example, in 3D, let consider the 4 vectors $(\pm\frac{\sqrt{(3)}}{3}, \pm\frac{\sqrt{(3)}}{3}, \frac{\sqrt{(3)}}{3})$, then $(0, \frac{\sqrt{(2)}}{2}, \frac{\sqrt{(2)}}{2})$ is not between the others. Yet, the behaviour of the algorithm may be to increase the surface of the convex hull of support vectors until all vectors are covered. So dynamic of exploration matches strong polynomial required hypothesis.

Again, for linear program, on a vertex D on which $A_D y = \mathbf{1}$ is impossible, the underlying hope is that one constraint is problematic to decrease the objective, and, that jumping may allow to quit this constraint for ever (this is true under convex hull hypothesis but it could be true even without).

For this reason, there is little interest to extend these strongly polynomial cases to generic convex programming.

3.3 Numerical experiments

Algorithm has been implemented with a toy implementation of the linear feasibility queries (see source code in appendix - this implementation is python based, mono thread and uses exact arithmetic operation) to perform empirical evaluation of previous statements.

Algorithm has been tested on both random problems, and, classical Klee Minty cube.

On Klee Minty cube, it behaves more efficiently than basic simplex. Yet, cube are especially hard for simplex based algorithm because simplex explores adjacent vertices which are designed to be confusing in Klee Minty cube. The slide and jump algorithm also jumps from vertex to vertex. But these vertices may not be adjacent. So, it is not that surprising that algorithm is better than simplex on simplex worst cases. Currently, the convergence is only *medium* on this classical example, highlighting possible limitation of the algorithm.

Contrariwise, algorithm behaves surprisingly efficiently on random problem (for a python based mono thread, exact arithmetic implementation). Indeed, on

random case, algorithm explores a very low number of D. This last observation may be linked with the situation in which the algorithm is strongly polynomial.

Typically, for support vector machine, algorithm considers $\min_{w \ / \ Aw \geq \mathbf{1}} w^T w$. The convex hull of support vectors (i.e. A_m^T such that $A_m w^* = 1$) is in the hyper plane $H = \{u \ / \ u^T w^* = 1\}$, and more precisely in $H \cap C$ with $C = \{u \ / \ u^T u = 1\}$. Now, all $\frac{1}{A_k v} A_k$ are in $H \cap B$ with $B = \{u \ / \ u^T u \leq 1\}$. Loosely speaking, $H \cap C$ is the excircle of the convex hull of support vectors. And, the algorithm is strongly polynomial if all other vectors being in $H \cap B$ ater scaling are in this convex hull. So, the situation is: a set of vectors D^*, from which one can consider the excircle of the corresponding convex hull. And, good cases are when all other vectors (after scaling) are in the convex hull knowing that there are in the excircle. Inversely, annoying vectors must be strictly between the convex hull and the corresponding excircle.

So, basically, the probability of being strongly polynomial in random case, is somehow linked to the ratio of the volume of a convex hull by the volume of the corresponding excircle. This is a well studied question e.g. [7, 1, 18, 20].

And, even, if all the vectors are not directly in the cone formed by support vectors, the more the D covers a large sets of vectors, the more the convergence of the algorithm toward D^* will be fast. The same considerations are true for linear program. This can be an explanation of the efficiently of the slide and jump algorithm on random problem.

4 Perspectives

This short paper presents an algorithm based on the idea of greedy optimisation with frozen combinatorial structure, and, linear feasibility queries which allows to refine the combinatorial structure. To give a metaphor, greedy moves slides on linear constraints, and linear feasibility allows to jump on new ones.

The first interest of this *slide and jump* idea is it ability to solve different type of problem linear program and support vector machine (but possibly a large set of convex problem and potentially non convex one).

And, the second is that algorithm is strongly polynomial for both linear program and support vector machine under similar convex hull assumptions.

Thus, this short paper presents at least a link between linear feasibility, linear program, support vector machine and convex hull.

In future work, average complexity or smoothed complexity should be considered for slide and jump algorithm for linear programming.

References

[1] Fernando Affentranger. The expected volume of a random polytope in a ball. *Journal of Microscopy*, 151(3):277–287, 1988.

[2] 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.

[3] Olivier Chapelle. Training a support vector machine in the primal. *Neural computation*, 19(5):1155–1178, 2007.

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

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

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] A Giannopoulos and A Tsolomitis. Volume radius of a random polytope in a convex body. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 134, pages 13–21. Cambridge University Press, 2003.

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

[9] Christoph Helmberg, Franz Rendl, Robert J Vanderbei, and Henry Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2):342–361, 1996.

[10] Didier Henrion, Simone Naldi, and Mohab Safey El Din. Exact algorithms for semidefinite programs with degenerate feasible set. *arXiv preprint arXiv:1802.02834*, 2018.

[11] 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.

[12] Don Hush, Patrick Kelly, Clint Scovel, and Ingo Steinwart. Qp algorithms with guaranteed accuracy and run time for support vector machines. *Journal of Machine Learning Research*, 7(May):733–769, 2006.

[13] 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.

[14] S Sathiya Keerthi and Dennis DeCoste. A modified finite newton method for fast solution of large scale linear svms. *Journal of Machine Learning Research*, 6(Mar):341–361, 2005.

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

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[18] Josef S Müller. Approximation of a ball by random polytopes. *Journal of Approximation Theory*, 63(2):198–209, 1990.

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

[20] Pablo Pérez-Lantero. Area and perimeter of the convex hull of stochastic points. *The Computer Journal*, 59(8):1144–1154, 2016.

[21] 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.

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

Appendix

Linear feasibility and linear separability (lemma 2.1)

[5] presents an algorithm to solve in strongly polynomial time the following problem: $\exists ?x \in \mathbb{Q}^N / Ax = \mathbf{0}, x > \mathbf{0}$ under the assumption that $A \in \mathbb{Q}_{M,N}$ has a rank of M.

Let $\mathcal{A}$ a matrix without any assumption. Let consider the matrix $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix}$ formed with $\mathcal{A}$ concat with $-\mathcal{A}$ concat with $-\mathbf{I}$ the opposite of identity matrix.

First, this matrix has full rank M due to the identity block.

Then, applying the Chubanov algorithm (or any linear feasibility solver) to this matrix A will lead (if a solution exists) to x_1, x_2, x_3 such that

$$\begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{0}$$

and $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} > \mathbf{0}$ So, let $x = x_1 - x_2$, it holds that $\mathcal{A}x = \mathbf{I}x_3 = x_3 > \mathbf{0}$. And, if no solution exists, then Chubanov algorithm will provide a certificate.

Chubanov algorithm can be applied on a derived problem to solve linear separability $\exists ?x \in \mathbb{Q}^N / \mathcal{A}x > \mathbf{0}$ with $\mathcal{A} \in \mathbb{Q}_{M,N}$
(under no assumption on $\mathcal{A}$, especially on the rank).

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

So q is the solution of $\min_{p \in \mathbb{Q}^N, Ap=\mathbf{0}} (c-p)^T(c-p)$.

Source code of numerical experiments

```
from __future__ import print_function

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

from fractions import Fraction

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

def combinaisonlineaire(u,l,v):
    w = []
    for i in range(len(u)):
        w.append(u[i]+l*v[i])
    return w

def allocateVector(N):
    output = []
    for n in range(N):
        output.append(Fraction())
    return output

def produitscalairevecteur(l,v):
    return combinaisonlineaire(allocateVector(len(v)),l,v)

def produitscalaire(u,v):
    w = Fraction()
    for i in range(len(u)):
        w+=u[i]*v[i]
    return w

def allocateVector(N):
    output = []
    for n in range(N):
        output.append(Fraction())
    return output

def projection(u, BOG):
    pu = allocateVector(len(BOG[0]))
    for v in BOG:
        pu=combinaisonlineaire(pu,produitscalaire(u,v)/produitscalaire(v,v),v)
    return pu

def gramschimdBOG(H_):
    H = [h.copy() for h in H_ if produitscalaire(h,h) != Fraction()]
    BOG = []
    while len(H)>0:
        BOG.append(H.pop())
        for i in range(len(H)):
            H[i] = combinaisonlineaire(H[i],Fraction(-1),projection(H[i],BOG))
        H = [h for h in H if produitscalaire(h,h) != Fraction()]
    return BOG
```

```
################ ALGORITHM ################

def solvenormalized(A,b,c,xoptimal,x):
    M = len(A)
    N = len(A[0])

    while True:
        print(x)

        d = produitscalaire(A[0],x)-b[0]
        for m in range(M):
            if produitscalaire(A[m],x)-b[m]<d:
                d = produitscalaire(A[m],x)-b[m]
        print(d)

        D = []
        for m in range(len(A)):
            if produitscalaire(A[m],x)-b[m]==d:
                D.append(m)
        print(D)

        if produitscalaire(x,c)==produitscalaire(xoptimal,c):
            return None

        cm = produitscalairevecteur(Fraction(-1),c)
        projOnVect = projection(cm, gramschimdBOG([A[m].copy() for m in D]))
        projOnKer = combinaisonlineaire(cm,Fraction(-1),projOnVect)
        v = projOnKer
        print(v)

        if produitscalaire(v,v)!=Fraction():
            candidate = []
            for m in range(M):
                if produitscalaire(v,A[m])<Fraction():
                    currentdir = produitscalaire(A[m],x)-b[m]
                    desireddir = d
                    move = -(currentdir-desireddir)/produitscalaire(v,A[m])
                    candidate.append(move)

            move = min(candidate)
            x = combinaisonlineaire(x,move,v)
            print("### bisector move ###")
            continue

        y =combinaisonlineaire(x, -d/produitscalaire(A[0],c) ,c)

        if produitscalaire(y,c)==Fraction():
            x = y
            print("done")
            continue

        equic = produitscalaire(y,c)*Fraction(999,1000)
        optANDc = combinaisonlineaire(xoptimal,equic,c)
        y09 = produitscalairevecteur(Fraction(999,1000),y)
        simulateChubanov = combinaisonlineaire(y09,Fraction(1,1000),optANDc)
        x = simulateChubanov
        print("### chubanov jump ###")
        continue

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

def allocateMatrix(M,N):
    mymatrix = []
    for m in range(M):
        mymatrix.append(allocateVector(N))
    return mymatrix

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 = allocateMatrix(M+4,N+3)
    b = allocateVector(M+4)
    c = allocateVector(N+3)
    c[-1] = Fraction(1)

    normRawA = allocateVector(M)
    normRawAtrick = allocateVector(M)
    for m in range(M):
        normRawA[m] = produitscalaire(A[m],A[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 = allocateVector(N+3)
    for n in range(N):
        xoptimal[n] = rawxoptimal[n]

    #check x solution
    if produitscalaire(c,xoptimal)!=Fraction():
        print("produitscalaire(c,xoptimal)!=Fraction()")
        quit()
    for m in range(M):
        if produitscalaire(A[m],xoptimal)<b[m]:
            print("produitscalaire(A[m],xoptimal)<b[m]")
            quit()

    x = allocateVector(N+3)
    x[-1] = max(b)+1
    x[-1] *= Fraction(5,3)
    return A,b,c,xoptimal,x

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

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

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

    c = twopower[::-1]

    A = allocateMatrix(N,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 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 = allocateMatrix(M+N+N+M+2,N+M)
    b = allocateVector(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

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

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

    #check xoptimal is really optimal:
    for m in range(len(A)):
        if produitscalaire(A[m],xoptimal)<b[m]:
            print("not optimal",m,A[m],xoptimal,produitscalaire(A[m],xoptimal),b[m])
            quit()

    return A, b, xoptimal

import random

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

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


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

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

    Agreater = randomMatrix(M,N)
    bgreater = allocateVector(M)
    for m in range(M):
        ifequal = produitscalaire(Agreater[m],xoptimal)
        bgreater[m] = ifequal-Fraction(random.randint(1,100))

    return Aequal+Agreater,bequal+bgreater,xoptimal

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

rawA,rawb,rawxoptimal = randomproblem(10,30)

print(rawA,rawb,rawxoptimal)

A,b,c,xoptimal,x = normalize(rawA,rawb,rawxoptimal)
print(A,b,c,xoptimal,x)

solvenormalized(A,b,c,xoptimal,x)

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

rawA,rawb,rawxoptimal = cubeproblem(10)

print(rawA,rawb,rawxoptimal)

A,b,c,xoptimal,x = normalize(rawA,rawb,rawxoptimal)
print(A,b,c,xoptimal,x)

solvenormalized(A,b,c,xoptimal,x)
```