A Symmetry-Based Explanation of the Main Idea Behind Chubanov's Linear Programming Algorithm

Olga Kosheleva, Vladik Kreinovich, and Thongchai Dumrongpokaphan

Abstract Many important real-life optimization problems can be described as optimizing a linear objective function under linear constraints – i.e., as a linear programming problem. This problem is known to be not easy to solve. Reasonably natural algorithms – such as iterative constraint satisfaction or simplex method – often require exponential time. There exist efficient polynomial-time algorithms, but these algorithms are complicated and not very intuitive. Also, in contrast to many practical problems which can be computed faster by using parallel computers, linear programming has been proven to be the most difficult to parallelize. Recently, Sergei Chubanov proposed a modification of the iterative constraint satisfaction algorithm: namely, instead of using the original constraints, he proposed to come up with appropriate derivative constraints. Interestingly, this idea leads to a new polynomial-time algorithm for linear programming – and to efficient algorithms for many other constraint satisfaction problems. In this paper, we show that an algebraic approach – namely, the analysis of the corresponding symmetries – can (at least partially) explain the empirical success of Chubanov's idea.

1 Introduction: Why Solve Linear Programming Problems, and How These Problems Are Solved Now

In this paper, we provide an algebraic (symmetry-based) explanation for the empirical success of a new algorithm for solving linear programming problems. Let us

Olga Kosheleva and Vladik Kreinovich
University of Texas at El Paso, El Paso, Texas 79968, USA
e-mail: olgak@utep.edu, vladik@utep.edu

Thongchai Dumrongpokaphan
Department of Mathematics, Faculty of Science, Chiang Mai University, Thailand
e-mail: tcd43@hotmail.com

therefore start with a brief reminder of what are linear programming problems, and how these problems are solved now.

Problems that we solve in real life. In many practical situations, we need to maximize or minimize some objective function:

- When we select a plan for a company, we want to maximize profit.
- When we select a route for a car, we want to minimize travel time.
- When we select medical treatment, we want to minimize side effects, etc.

In all these situations, there are some constraints:

- Pollution generated by a chemical plant cannot exceed the legal limits.
- A car cannot exceed the speed limit – unless it is an emergency vehicle.
- A medical treatment must satisfy a certain rate of cure, etc.

How to describe these problems in precise terms. In general, there are several parameters $x_1,\ldots,x_n$ describing possible alternatives. The objective function $f(x_1,\ldots,x_n)$ depends on all these parameters.

A constraints means that some quantity g cannot exceed the corresponding threshold t. This quantity also depends on the parameters $x_1,\ldots,x_n$:

$$g = g(x_1,\ldots,x_n).$$

Thus, a constraint has the form $g(x_1,\ldots,x_n) \le t$.

In general, we have a *constraint optimization* problem: maximize $f(x_1,\ldots,x_n)$ under constraints

$$g_1(x_1,\ldots,x_n) \le t_1,\ldots,g_m(x_1,\ldots,x_n) \le t_m.$$

Linearization is often possible. In many practical situations, we know a reasonably good solution $x^{(0)} = (x_1^{(0)},\ldots,x_n^{(0)})$. This usually means that the unknown optimal solution $x = (x_1,\ldots,x_n)$ is close to $x^{(0)}$; in other words, the differences $v_i \stackrel{\text{def}}{=} x_i - x_i^{(0)}$ are small.

In physics and engineering, if the quantity is small, we can safely ignore terms which are quadratic in small v_i. For example, if $v \approx 10\%$, then $v^2 \approx 1\% \ll 10\%$. Thus, we can, e.g.:

- take the expression $f(x_1,\ldots,x_n) = f\left(x_1^{(0)}+v_1,\ldots,x_n^{(0)}+v_n\right)$;
- expand it in Taylor series and keep only linear terms in this expansion:

$$f(x_1,\ldots,x_n) \approx y^{(0)} + \sum_{j=1}^{n} c_i \cdot v_i,$$

where $y^{(0)} \stackrel{\text{def}}{=} f\left(x_1^{(0)},\ldots,x_n^{(0)}\right)$ and $c_j \stackrel{\text{def}}{=} \dfrac{\partial f}{\partial x_j}$.

Maximizing this expression for $f(x_1,\ldots,x_n)$ is equivalent to maximizing a linear function $\sum\limits_{j=1}^{n} c_i \cdot v_i$.

By applying a similar linearization to $g_i(x_1,\ldots,x_n) = g_i\left(x_1^{(0)}+v_1,\ldots,x_n^{(0)}+v_n\right)$, we conclude that

$$g_i(x_1,\ldots,x_n) \approx g_{i0} + \sum_{j=1}^{m} a_{ij}\cdot v_j,$$

where $g_{i0} \stackrel{\text{def}}{=} g_i\left(x_1^{(0)},\ldots,x_n^{(0)}\right)$ and $a_{ij} \stackrel{\text{def}}{=} \dfrac{\partial g_i}{\partial x_j}$. Thus, each constraint

$$g_i(x_1,\ldots,x_n) \le t_i$$

takes the form $\sum\limits_{j=1}^{n} a_{ij}\cdot v_j \le b_i$, where $b_i \stackrel{\text{def}}{=} t_i - g_{i0}$.

So, we arrive at the problem of maximizing a linear function $\sum\limits_{j=1}^{n} c_i \cdot v_i$ under linear constraints $\sum\limits_{j=1}^{n} a_{ij}\cdot v_j \le b_i$. Such problems are known as *linear programming*; see, e.g., [6].

Comment. Why linear – clear, but why programming? The answer is simple: in the late 1940s, programming was all the range. If you called it programming, your changes of getting a grant drastically increased. So we have dynamic programming, quadratic programming, etc.: all this has nothing to do with programming.

It is somewhat like now, when many folks processing kilobytes of data call it big data :-(

An example of a linear programming problem. One of the first examples of linear programming was developing meals plan for jails.

In this case, $v_1,\ldots,v_n$ are amounts of different products: beef, chicken, beans, bread, milk, etc. The objective is to minimize cost $\sum\limits_{j=1}^{n} c_j\cdot v_j$. The main constraint is that the overall amount of calories should be sufficient: $\sum\limits_{j=1}^{n} a_{1j}\cdot v_j \ge b_1$, where a_{1j} is calories per pound for the j-th product. We must also make sure that the folks get:

- enough proteins b_2,
- enough of different vitamins $b_3,\ldots$,
- enough of different micro-elements b_i, etc.

Comment. The solution, by the way, was indeed cheap. However, we would not advise poor students to use it: this solution does not take taste into account :-(

How can we solve linear programming problems: first idea and the resulting simplex algorithm. Since linear programming problems are ubiquitous, people have been trying to find general efficient algorithms for solving these problems.

The first efficient algorithms started with a simple mathematical analysis. Each constraint $\sum_{j=1}^{n} a_{ij} \cdot v_j \le b_i$ determines a half-space. A half-space H is a convex set: if $h \in H$ and $h' \in H$, then the whole straight line segment is in H:

$$\alpha \cdot h + (1-\alpha) \cdot h' \in H \text{ for all } \alpha \in (0,1).$$

The set of all $v = (v_1, \ldots, v_n)$ that satisfy all the constraints is an intersection of several half-spaces. This intersection is thus also convex: it is a convex polytope.

On each segment, a linear function is linear. The maximum of a linear function of a segment is attained at the endpoints. So, in our problem, the maximum of a linear function is attained at one of the vertices.

A vertex is where n of m constraints are equalities. Once we know which constraints are equalities, to find v, we solve a system of linear equations $\sum a_{ij} \cdot v_j = b_i$. There are efficient algorithms for solving such systems; e.g., Gauss elimination takes time $O(n^3)$; see, e.g., [3].

The problem is that there are exponentially many size-n subsets. So, a natural idea is to start with any vertex, and then replace one of the constraints so as to increase the objective function. This idea – known as *simplex method* – indeed leads to a very efficient algorithm which is still used.

This method's authors, Leonid Kantorovich and Tjalling C. Koopmans, received 1975 Nobel Prize in Economics.

Limitations of the simplex method. The main problem with simplex method is that sometimes, this algorithm requires exponential time.

Interestingly, its average computation time is good. However, this good time assumes that all the coefficients a_{ij}, b_i, and c_j are independent. In contrast, in practice, they are often strongly correlated. As a result, exponential time occurs frequently in practice.

Can we reduce computation time: Khachiyan's algorithm. The authors of the notion of NP-hardness originally thought that linear programming may be NP-hard.

The theoretical breakthrough was achieved in 1979 by Leonid Khachiyan's polynomial-time algorithm; see, e.g., [5]. His main idea was to enclose the convex polytope P by an ellipsoid.

Why ellipsoids? The class of problems remains the same if we have a linear change of variables: $v_j \to v'_j = \sum_{j'=1}^{n} d_{jj'} \cdot v_j$. The simplest domain is a sphere. If we apply different linear transformations to a sphere, we get ellipsoids.

Crudely speaking, in this algorithm, we divide the ellipsoid in two. In the upper half-ellipsoid – if it has any points satisfying all the constraints – the values of the objective function are higher. So, we enclosed this half-ellipsoid a (smaller) ellipsoid, etc.

Karmarkar's algorithm. While Khachiyan's algorithm was theoretically good, in practice, it was very inefficient. In 1984, Narendra Karmarkar proposed a practically efficient polynomial-time algorithm; see, e.g., [4].

Karmarkar used the fact that the class of ellipsoids is also invariant with respect to projective transformations – examples of which are projections producing a 2-D map of a 3-D Earth.

So, if we know a point in P, we first perform a projective transformation that makes P the ellipsoid's center, and only then we bisect.

Karmarkar's algorithm – and its improvements – are still widely used in practice, but it still sometimes takes too long.

Why cannot we decrease computation time by parallelization? When it takes too long for a person to perform a task, this person asks for help. When several people work on different parts of the task, the task gets done faster.

Similarly, many computations become faster if we use several processors working in parallel. Unfortunately, this idea does not work for linear programming. It has been proven that linear programming is the worst possible problem for parallelization; such problems are known as *P-hard*; see, e.g., [7]. So, we cannot just parallelize the existing algorithms: we need new algorithms to speed up computations.

2 Chubanov's Algorithm: Main Idea

Let us go back to constraint satisfaction. To find out what to do let us go back and consider constraint satisfaction in general.

In real life, we often have many constraints that we want to be satisfied. For example, in economics, we want:

- inflation not larger than some reasonably small threshold,
- unemployment not larger than some small number,
- growth larger than some minimal amount, etc.

In practice, several of these constraints are usually not satisfied. So, what do we do?

We select a constraint that is the farther from satisfaction, and concentrate on this particular constraint. For example, if inflation is high, we decrease the money supply. Then, inflation goes down, but unemployment goes up and growth stagnates. If stagnation becomes the main issue, we concentrate on growth and stimulate economy, etc.

Iterative constraint satisfaction: a brief reminder. The same strategy is often used in general:

- We start with some alternative $v^{(0)}$ – which, in general, does not satisfy all the constraints.
- Then, we pick a constraint C.
- We find an alternative $v^{(1)}$ which is the closest to $v^{(0)}$ among those that satisfy this constraint:
$$d(v^{(1)}, v^{(0)}) = \min_{x \in C} d(v, v^{(0)}).$$
- After that, we pick another constraint C'.

- We find an alternative $v^{(2)}$ which is the closest to $v^{(1)}$ among those that satisfy this constraint:
$$d(v^{(2)}, v^{(1)}) = \min_{x \in C'} d(v, v^{(1)}), \text{ etc.}$$

In many cases, this process converges either in finitely many steps or in the limit. As a result, we get an alternative v that satisfies all the constraints.

Limitations of the iterative constraint satisfaction. The main problem with the above method is that convergence is often slow. For example, for linear programming, this method often requires exponential time.

Sergei Chubanov's idea. We want to have $g_i(x_1, \ldots, x_n) \le t_i$ for all i.

If all these inequalities hold, then, for any $\alpha_i \ge 0$, we have $g(x_1, \ldots, x_n) \le t$, where
$$g(x_1, \ldots, x_n) = \sum_{i=1}^{m} \alpha_i \cdot g_i(x_1, \ldots, x_n) \text{ and } t = \sum_{i=1}^{m} \alpha_i \cdot t_i.$$

These new constraints are known as *derivative constraints*.

Sergei Chubanov suggested [1, 2] to use iterative constraint satisfaction, but:

- instead of cycling through *original* constraints,
- to generate *new* derivative constraints every time, selecting α_i so as to speed up convergence.

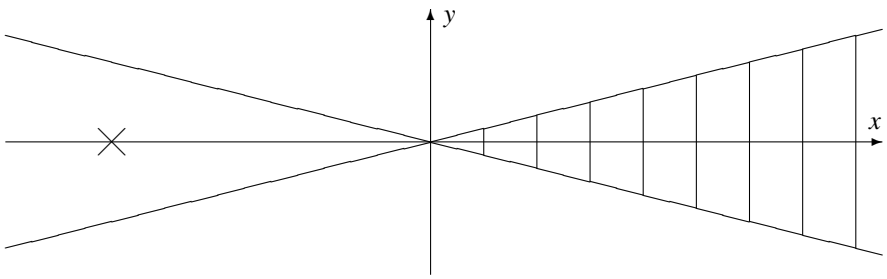
Chubanov has shown that by appropriately selecting derivative constraints, we can get a polynomial-time algorithm [1].

To find α_i, we – approximately – solve an optimization problem on each step. This is rather technical, not easy to explain. But what is easy to explain is why this often drastically speed up convergence.

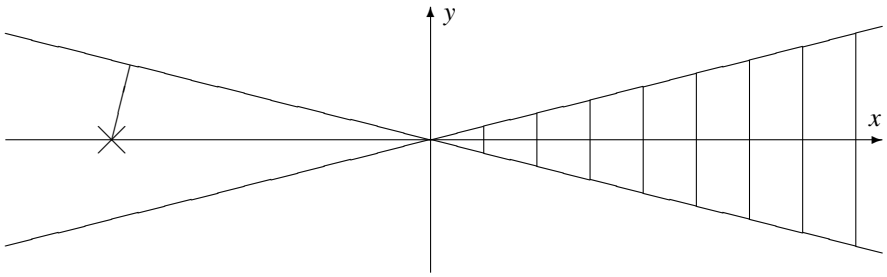
Example of how derivative constraints speed up the algorithm. Suppose that we want to satisfy two constraints

$$y \le \varepsilon \cdot x \text{ and } -y \le \varepsilon \cdot x \text{ for some small } \varepsilon > 0.$$

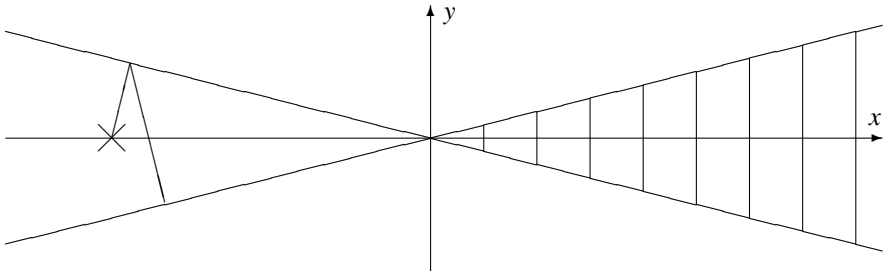
Let us start with a point $(-1, 0)$.



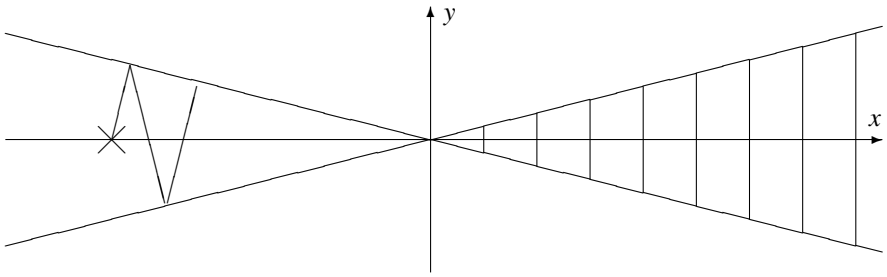
In the traditional constraint satisfaction algorithm, we first "project" onto one of the constraints:



Then we project onto another constraint:

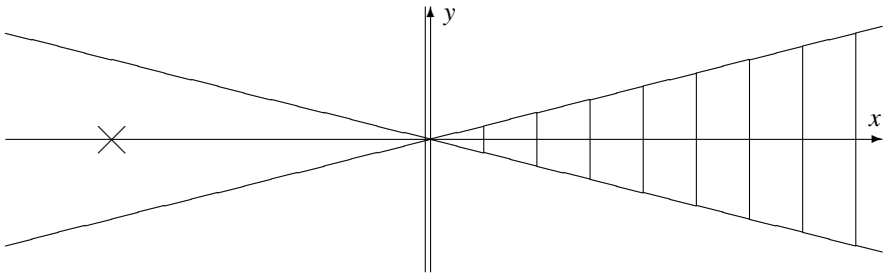


Then onto another one, etc.:

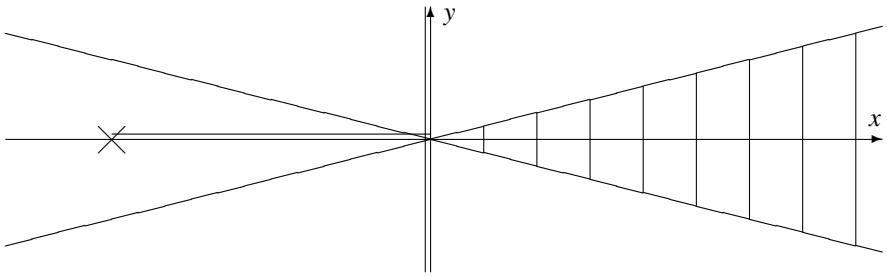


For small ε, in the traditional approach, we get a very slow convergence to the desired area.

In contrast, in Chubanov's approach, we come up with a derivative constraint $0 \leq x$:



The corresponding projection bring us immediately into a point $(0,0)$ satisfying both constraints:



3 Why Chubanov's Algorithm Works? Why Other Algorithms Work?

Why symmetries. For linear programming, as we have mentioned, there are symmetries behind efficient algorithms. Let us explain why this makes sense.

Indeed, let us assume that there are natural symmetries T on the set of alternatives A. In our case:

- alternatives are algorithms, and
- symmetries are, e.g., linear transformations that keep the problem unchanged.

On the set A, we have a preference relation $\preceq$. This relation should be reflexive and transitive – i.e., in mathematical terms, it should be a (partial) *pre-order*.

Since T is a reasonable transformation that does not change the problem, the relation $\preceq$ should be T-invariant: if $a \preceq a'$, then $T(a) \preceq T(a')$.

How many "best" alternatives should we have? If several alternatives are the best, this means that we can use this non-uniqueness to optimize something else. For example:

- if several algorithms have the same worst-case complexity w,
- we can select the one with the best average-time t.

In other words, we will use a new preference relation:

$$a \preceq_{\text{new}} a' \Leftrightarrow (w(a') < w(a) \vee (w(a') = w(a) \,\&\, t(a') < t(a))).$$

If we still have several best alternatives, we can optimize something else, etc. At the end, we get a *final* preference relation for which only one optimal alternative is the best.

Let us show that this optimal alternative a_{opt} is itself T-invariant. Indeed, a_{opt} is better than any other alternative: $a \preceq a_{\text{opt}}$. In particular, for each a, we have

$$T^{-1}(a) \preceq a_{\text{opt}}.$$

Since $\preceq$ is T-invariant, we conclude that

$$T(T^{-1}(a)) = a \preceq T(a_{\text{opt}}) \text{ for all } a.$$

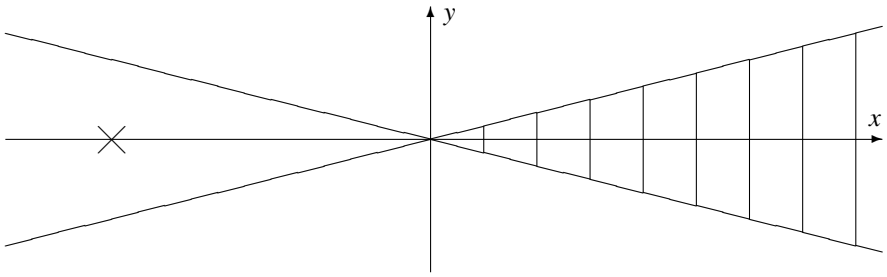
Thus, $T(a_{\text{opt}})$ is also optimal. However, since the preference relation is final, there is only one optimal alternative; thus $T(a_{\text{opt}}) = a_{\text{opt}}$.

Back to Chubanov's algorithm. From this viewpoint:

- if it turned out that Chubanov's algorithm is invariant relative to some natural symmetries,
- this will be a good indication that it is indeed optimal in some sense.

From this angle, let us look at the above example:

- constraints $y \le \varepsilon \cdot x$ and $-y \le \varepsilon \cdot x$ with
- initial approximation $x^{(0)} = -1$ and $y^{(0)} = 0$.



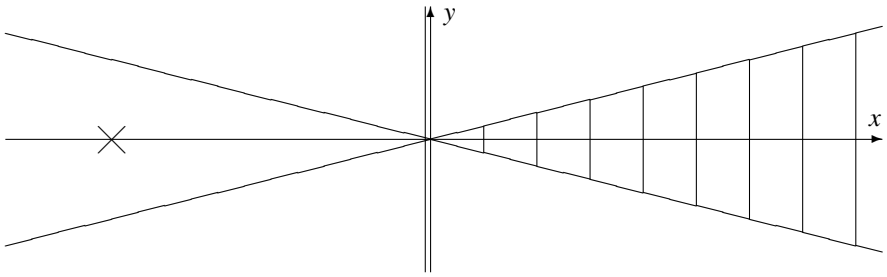
This configuration is invariant with respect to $y \to -y$. However, in the traditional constraint satisfaction algorithm, this symmetry is violated:

- we either start with the first constraint,
- or we start with the second constraint.

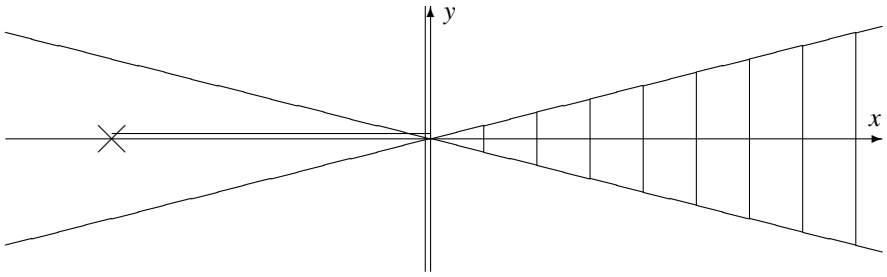
In Chubanov's algorithm, instead, we find $\alpha_i \ge 0$ to form a symmetric derivative constraint:

$$\alpha_1 \cdot y + \alpha_2 \cdot (-y) \le \alpha_1 \cdot \varepsilon \cdot x + \alpha_2 \cdot \varepsilon \cdot x.$$

One can easily check that this constraint is invariant w.r.t. $y \to -y$ if and only if $\alpha_1 = \alpha_2$. For these values α_i, the derivative constraint takes the form $0 \le 2\alpha_i \cdot \varepsilon \cdot x$, i.e., $0 \le x$.



The closest point satisfying this derivative constraint is $(0, 0)$ – so Chubanov's algorithm is indeed symmetric!



Acknowledgments

This work was supported by Chiang Mai University. It was also partially supported by the US National Science Foundation via grant HRD-1242122.

References

1. S. Chubanov, "A polynomial relaxation-type algorithm for linear programming", *Optimization Online*, February 2011.
2. S. Chubanov, "A strongly polynomial algorithm for linear systems having a binary solution", *Mathematical Programming*, 2012, Vol. 134, No. 2, pp. 533–570.
3. Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.
4. N. Karmarkar, "A new polynomial-time algorithm for linear programming", *Combinatorica*, 1984, Vol. 4, pp. 373–395.
5. L. Khachiyan, "Polynomial Algorithm for Linear Programming", *Soviet Math. Doklady*, 1979, Vol. 224, No. 5, pp. 1093–1096.
6. D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, Springer, Cham, Switzerland, 2016.
7. C. H. Papadimitriou, *Computational Complexity*, Pearson, Boston, Massachusetts, 1993.