Honours Project

Geometric Incremental Linear Programming Solver

Irving Ou

School of Computer Science
Carleton University
Canada
2023-03-31

Honours Project Report:Geometric Incremental Linear Programming Solver

Irving Ou

March 31, 2023

Abstract

In this report, we present the implementation and evaluation of a geometric linear programming solver aimed at optimizing linear systems in low dimensions. This project was undertaken in response to the limitations of traditional approaches, such as the simplex method, which relies on algebraic identities. We detail the implementation of the proposed algorithm, capable of handling two-dimensional linear programming problems, and discuss the challenges encountered, including vertical constraints, parallel constraints, and the conversion of two-dimensional problems to one-dimensional ones. The main results of this project, including the performance comparison of our proposed method against the standard simplex method, will be presented in the Results section. Our findings contribute to the ongoing exploration of alternative approaches to linear programming, with potential applications in a variety of optimization problems.

Contents

1 Introduction

Linear programming is a mathematical technique used to optimize linear systems subject to certain constraints. The standard approach to solving linear programs is the simplex method, which has been well-studied for decades.

In this project, we implement a geometric incremental linear programming solver that aims to improve upon the limitations of the simplex method in lower dimensions. Our solver is designed to handle two-dimensional problems and is theoretically more efficient than the simplex method in lower dimensions.

The result does show that the geometric solver does better than the simplex solver on unbounded and infeasible problems, but not bounded ones.

In this report, we present the results of our implementation of the geometric linear programming solver. We describe the algorithm and its implementation, including the conversion of two-dimensional problems to one-dimensional problems and the handling of special cases. We evaluate the performance of our solver using test cases and compare it with the simplex method. We also discuss the limitations of our proposed method and potential future improvements and extension of two three dimensions.

2 Literature Review

Linear Programming is a study focusing on optimizing linear systems with certain constraints, and usually comes with the following form:

$$\begin{aligned} \text{minimize} \sum_{j=1}^{m} w_j * x_j = z \\ \text{subject to} \sum_{j=1}^{m} a_{i,j} x_j \leq c_i \ \text{ for i = 1,...,n} \end{aligned} \tag{1}$$

Linear Programming has been well-studied for decades, the primary tool we have today is called the simplex method, which relies on algebraic identities.

2.1 Simplex Method Overview

The simplex method is an algorithm used for solving linear programming problems. It begins with an initial feasible solution and then iteratively improves the objective function by moving to a neighbouring solution that maintains feasibility. The simplex method operates on a matrix representation of the problem, known as the simplex tableau, which is transformed through a series of pivot operations that eliminate coefficients of one variable in the constraints. The algorithm terminates when an optimal solution is found or when the problem is determined to be unbounded.

Although worst-case analysis indicates that the complexity of the simplex method is $O(2^d)$ where $\vec{d}$ is the dimension or the number of variables. The average-case complexity is polynomial [1, 2], and under certain probabilistic assumptions, the complexity is expected to be linear [1, p. 139]. Several theoretical results support the effectiveness of the simplex method for solving linear programming problems.

One popular implementation of the simplex method is GLOP (the Google Linear Optimization Package), which is developed by Google's Operation Research Team. GLOP uses a variation of the simplex method called the revised primal-dual simplex algorithm and is implemented in C++ [3].

While the simplex method has proven effective in practice, it has some limitations, such as being inefficient for certain types of problems and requiring an initial feasible solution. Other methods for solving linear programming problems, such as interior point methods, have also been developed and can sometimes provide better performance in certain scenarios. Nonetheless, the simplex method remains a key tool in the field of optimization and is widely used in industry and academia

2.2 Geometric Solver Overview

The geometric approach, also known as incremental linear programming, is a method for solving linear programming problems that differs from the simplex method in its approach to finding feasible solutions. The geometric approach begins by finding an initial optimal solution to the linear program. As new constraints are added, the solution is iteratively updated to incorporate these constraints. If the current solution is not feasible concerning the new constraints, the problem is transformed into a one-dimensional problem and solved to find the best new candidate solutions. This process is repeated until a feasible solution is found for all constraints.

One advantage of the geometric approach is that it can be more efficient than the simplex method for lower-dimension problems. Additionally, it does not require an initial feasible solution, which can be an advantage when the problem is difficult to solve or the feasible region is unknown.

The expected time complexity of the geometric approach is $d * O(n)$, where d is the dimension and n is the number of variables [4].

3 Methodology

3.1 The Algorithm

Below is the core of the whole algorithm.

```
def solve(self, obj: ObjectiveFunction, cons: List[Constraints]) -> Point:
    bound_res = self.check_unbounded(obj, cons)
    if not bound_res.bounded:
        raise UnboundedException("The problem is unbounded",ray = bound_res.unbound_certificate)
    h1_idx = bound_res.bound_certificate[0]
    h2_idx = bound_res.bound_certificate[1]
    h1 = cons[h1_idx]
    h2 = cons[h2_idx]
    cons.remove(h1)
    cons.remove(h2)
    cons.insert(0, h1)
    cons.insert(1, h2)

    v = h1.find_intersection(h2)
    for idx, c in enumerate(cons):
        if not v.is_inside(c):
            one_d_constraints = to_1d_constraint(c, cons[:idx])
            if not c.is_vertical():
                x = solve_1d_linear_program(
                    one_d_constraints, get_one_d_optimize_direction(obj, c))
                v = c.find_point_with_x(x)
            else:
                y = solve_1d_linear_program(
                    one_d_constraints, get_one_d_optimize_direction(obj, c))
                v = c.find_point_with_y(y)

    return v
```

The algorithm consists of two main components: the objective function class and the constraint class. The objective function class contains two floats,c_x and c_y which represents the function $f_{\vec{c}]} = c_x x + c_y y$. The constraint class contains three floats, representing the coefficients of x, y, and a constant c, in the form of $ax + by <= c$.

All constraints are stored in the form of $ax + by <= c$. If a constraint is expressed in the form of $a'x + b'y >= c'$, they are converted to $-a'x - b'y <= -c'$ to simplify operations. This is achieved by multiplying both sides of the inequality by -1.

The linear programming algorithm involves two primary steps. The first is to check if the problem is unbounded. This is done by checking if there exists a feasible solution in which the objective function is unbounded. If bounded, a ray will be reported, if not, the unboundedness-checking algorithm will return a certificate which contains the index of two specific constraints that bounds the objective function.

The second step involves repeatedly checking whether each constraint h_i contains v or not, if v is not inside h_i, transforming the linear program into a one-dimensional problem with constraints $h_1, ..., h_i$ and solving it to obtain new v.

For further discussion, let's use $H = \{h_i\}$ to indicate the set of constraints and $f_{\vec{c}} = c_x p_x + c_y p_y$ to represent the objective function. Also, let's denote $\vec{\eta}(h_i)$ as the facing direction vector of constraint h_i.

3.2 Unboundedness

the unboundedness check aims to determine if there is an unbounded feasible region. Geometrically, this process requires finding a direction that is within 90 degrees of the objective function vector and not aligned with any two constraints. This direction should have an angle less than or equal to 90 degrees when compared to each facing direction vector of the constraints.

To perform the unboundedness check, one must first rotate the constraints in such a way that the objective function vector aligns with the y-axis. The angle of rotation needed can be calculated by determining the angle between the objective function vector and the vector $(0, 1)$. The formula to find this angle is $\text{atan2}(\vec{u} \times \vec{v}, \vec{u} \cdot \vec{v})$, where $\vec{u}$ and $\vec{v}$ are the two vectors whose angle we want to calculate.

In this context, the direction vector $\vec{d} = (d_x, 1)$ must satisfy the following conditions: $\vec{d} \cdot \vec{c} > 0$, where $\vec{c}$ is the objective function vector, and $\vec{d} \cdot \eta(h_i) \geq 0$ for all i, where $\boldsymbol{\eta}(h_i)$ is the facing direction vector of the i-th constraint h_i.

Lemma 3.1. *The angles between two vector $\vec{v}$ and $\vec{u}$ is given by $atan2(\vec{u} \times \vec{v}, \vec{u} \cdot \vec{v})$.*

Proof. we use the following identity to prove the above statement, $\vec{u} \times \vec{v} = \|\vec{v}\|\|\vec{u}\| \sin\theta$ and $\vec{v} \cdot \vec{u} = \|\vec{v}\|\|\vec{u}\| \cos\theta$. One can easily see that

$$atan2(\vec{u} \times \vec{v}, \vec{u} \cdot \vec{v}) \tag{2}$$

$$= atan2(\|\vec{v}\|\|\vec{u}\| \sin\theta, \|\vec{v}\|\|\vec{u}\| \cos\theta)$$

denote $c = \|\vec{v}\|\|\vec{u}\|$

$$= 2arctan(\frac{c\sin\theta}{((c\cos\theta)^2 + (c\sin\theta)^2)^{\frac{1}{2}} + c\cos\theta})$$

$$= 2arctan(\frac{\sin\theta}{1 + \cos\theta})$$

By half angle formula $\tan\frac{x}{2} = \frac{\sin x}{1+\cos x}$m we have

$$= 2 * \frac{\theta}{2}$$

$$= \theta \quad \theta \in (-\pi, \pi)$$

Also if $\theta = \pi$, by the definition of atan2, it is equal to pi. □

now we can find the angle between two vectors, we now need to rotate constraints h_i correspondingly so that $rotate(\vec{c}) \cdot rotate(\eta(h_i)) = \vec{c} \cdot \eta(h_i)$. we perform the rotation through the rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

The final component we need to discuss is the computation of the facing direction vector $\boldsymbol{\eta}(h_i)$ of a constraint. Consider the constraint $ax + by \leq 0$, as the right-hand side does not contribute to the direction it is facing. Observe that we can choose a vector $\vec{u}$ in the same direction as the line of the function graph $ax + by = 0$ at $x = 1$, which gives us $y = \frac{a}{-b}$. Then, we simply insert $\vec{u} = (1, -\frac{a}{b})$ into the rotation matrix mentioned earlier.

Choose $\theta = -\frac{\pi}{2}$, as we want to rotate clockwise by 90 degrees. This rotation gives us $(-\frac{a}{b}, -1)$, which can be scaled by b to obtain the final result $(-a, -b)$. This vector represents the facing direction vector of the constraint $ax + by \leq 0$

Now that we have all the necessary components, let's put everything together to perform the unboundedness check for our linear programming problem.First, determine the angle θ between the objective function vector $\vec{c}$ and the vector $(0, 1)$. Then rotate all constraints using the rotation matrix R with the angle θ. Finally, compute the facing direction vectors $\eta(h_i)$ for each constraint h_i.

Indeed, the primary reason for rotating the constraints is to simplify the search for the direction vector $\vec{d}$. Once the constraints are rotated, the problem of finding $\vec{d}$ is transformed into a one-dimensional linear programming problem. The combination of rotating constraints and calculating facing direction vectors enables a more efficient determination of whether the feasible region is unbounded. This, in turn, aids in solving the linear programming problem.

Lemma 3.2. *A vector $\vec{d} = (d_x, d_y)$ such that $\vec{d} \cdot (0, 1) > 0$ if and only if $\vec{d}$ can be normalized to $(d'_x, 1)$.*

Proof.

$$\vec{d} \cdot (0, 1) = d_y > 0$$

which means the set $\{(\vec{v}) : v_y > 0\}$ contains all the possible vectors we are looking for. Let $d_y > 0$, obviously, there exist $\frac{1}{d_y}$ with $(d_x, d_y) * \frac{1}{d_y} = (\frac{d_x}{d_y}, 1)$. Pick $d'_x = \frac{d_x}{d_y}$, this statement holds. □

Now, all we are left to do is to solve the one-dimensional linear programming problem:

$$\vec{d}\vec{\eta}(h_i) = d_x \eta(h_i)_x + \eta(h_i)_y \geq 0 \quad \forall i = 1, ..., n$$

which is equivalent to

$$-d_x \eta(h_i)_x \leq \eta(h_i)_y \quad \forall i = 1, ..., n$$

One-dimensional linear programming is relatively simple. We keep track of the left and right boundaries on the real number axis and return the left or right boundary depending on the direction. If there is no solution, then the problem is bounded [4, Lemma 4.9 pno 80], and we return the bounded certificate, which consists of two constraints that bound the objective function. If there is a solution, we still have one more step to complete.

The final step is to ensure that all constraints parallel to $\vec{d}$ are feasible. That is, the set $H' = h \in H : \boldsymbol{\eta}(h) \cdot \vec{d} = 0$ forms a feasible area. This condition can also be converted into a one-dimensional problem by checking that $rotate(h_j).a \leq rotate(h_j).c \quad \forall h_j \in H'$. If a solution exists for this condition, then we report the unbounded certificate $\vec{d} = (d_x, d_y)$. Otherwise, the problem is infeasible.

3.3 Conversion to One-Dimensional

Since the problem has shown to be bounded with bounded certificate h_1, h_2, we can begin our iteration by first finding the intersection of h_1 and h_2, which provides our initial point v. For each newly added constraint h_i, we first check if it contains v. If not, that means v must be updated, or possibly, the problem becomes infeasible, as h_i shares no common area with the constraints encountered before [4, Lemma 4.5]. If h_i does contain v, we proceed to the next constraint h_{i+1}.

We search for the new v by converting the two-dimensional problem into one dimension. Let k be the index of the constraint we have encountered so far. The approach is to find all intersections of h_k with constraints $h_1, \ldots, h_{k-1}$. We then use those points and project them onto the x-axis, which provides a new set of one-dimensional constraints.

There are two edge cases, as previously mentioned in the midterm report. Here's a quick recap. The original textbook on linear programming does not cover two special cases: the vertical constraint and the parallel constraint.

The vertical constraint occurs when the coefficients for y are zero. To handle this case, the entire system can be rotated 90 degrees using the rotation matrix mentioned earlier.

The second special case arises when some constraints are parallel to each other, which can cause the algorithm to fail in detecting infeasible problems. For example, a system with only two constraints, $x + y \leq 1$ and $x + y \geq 10$, would miss the necessary constraints to detect infeasibility when converted to a one-dimensional problem, since it requires finding intersections of two constraints that do not intersect at all in this case. To solve this problem, we can compare C_i with every constraint found so far and report infeasibility if they are parallel and do not share common areas.

Another aspect not mentioned in the mid-term report is that determining the direction in which the one-dimensional constraint faces is heavily subject to floating-point errors. In particular, two nearly parallel constraints perform the worst. Since we are looking for intersections of two constraints, two almost parallel constraints could easily have an intersection point with $x > 10^{15}$. This is a serious issue for large differences in the constant value c. Consider two lines with $a_1x+b_1y=c_1$ and $a_2x+b_2y=c_2$, where if they intersect, the x and y will be the same, which gives:

$$\frac{c_1-b_1y}{a_1}=\frac{c_2-b_2y}{a_2}$$

with a bit of algebraic manipulation, one can show that

$$y=\frac{a_2c_1-a_1c_2}{a_2*b_1-a_1*b_2} \quad \text{and} \quad x=\frac{c_1-b_1*y}{a_1}$$

Notice that the more parallel the two lines are, the smaller the value of $a_2*b_1-a_1*b_2$, which in turn makes y and x grow very large.

This becomes a real problem when trying to decide the facing direction of a one-dimensional constraint that is almost parallel, as it makes the detection of the facing side much more difficult. Currently, finding the facing direction is done by sampling two points from the left and right-hand sides of the intersection on the bounding line of h_k. If the left-hand side point is contained by h_i where $i<k$, then h_i when projected onto one dimension, should face left and vice versa. However, if we have two almost parallel constraints, this method may fail to detect the direction correctly, where one possible case is that both the left and right points are not contained or both are contained by h_i.

The solution is to extend the distance between the left and right points as much as possible and treat almost parallel constraints as parallel. If none of the above works, throw a precision error if we can't avoid it, which is also a standard approach in most linear programming solvers.

The final part is to determine the optimization direction of the one-dimensional problem. Consider the objective function $f_{\vec{c}}(x,y)=c_xx+c_yy$, which returns the value of a point based on $\vec{c}$. The current constraint h_k, which can be expressed as $a_kx+b_ky\leq c$, has the bounding line $a_kx+b_ky=c$, and we know that the next v must exist on the bounding line [4, Lemma 4.5]. The equation $a_kx+b_ky=c$ can also be written as $y=\frac{c-a_kx}{b_k}$. Then, we substitute it into $f_{\vec{c}}$ to get a new function:

$$f_{h_k}(x)=c_x*x+c_y\frac{c-a_kx}{b_k}$$

By taking its derivative, we obtain:

$$f'_{h_k}(x)=c_x-\frac{c_ya_k}{b_k}$$

If this is positive, it means the value of the function is increasing as x is increasing, and we should optimize in the positive direction, and vice versa.

4 Results

Overall, the algorithm performs linearly as stated in theory. For bounded problems, it performs worse than GLOP, but it does a much better job when the problem is infeasible or unbounded. Considering that GLOP is implemented in C++, based on this metric, I personally believe it has performed better than GLOP.

4.1 The Data Set

Because the algorithm is specifically designed for two-dimensional linear programming problems, it is difficult to find a dataset that caters to our needs. Hence, all problems are randomly generated.

The bounded problems are randomly generated by ensuring that there are always three points contained by a randomly generated constraint. If not, we flip it and try again; if it still doesn't work, we ignore it and move on to the next one. Of course, this only guarantees that the problem is feasible, not necessarily bounded. However, as the number of constraints (n) becomes very large, the likelihood of the problem being unbounded is quite low. Infeasible problems, on the other hand, are generated using the same procedure, but with the guarantee that there exists a point never inside any constraint.

The unbounded problems are generated based on Lemma 4.9 in Computational Geometry, which states that a linear programming problem is unbounded if and only if there exists a $\vec{d}$ with $\vec{d}\cdot\vec{c}>0$ and $\vec{d}\cdot\vec{\eta}(h_i)\geq 0 \quad \forall i=1,...,n$. We fix a randomly generated $\vec{d}$ and, for every generated constraint, check if $\vec{\eta}(h)\cdot\vec{d}\geq 0$. If not, we flip it and append it to the set.

4.2 Analysis

We conducted thirty tests for each problem class, where each test computes 200 problems with sizes ranging approximately from 1 to 20,000. We then calculated the average for each test with the same size and analyzed it using linear regression.

The data can be found in the repository under the time data directory. The score indicated in the graphs below is the R-square value, which represents the correlation between the dataset and the linear model. The x-axis shows the number of constraints, and the y-axis displays the time it takes to solve the problems in seconds.

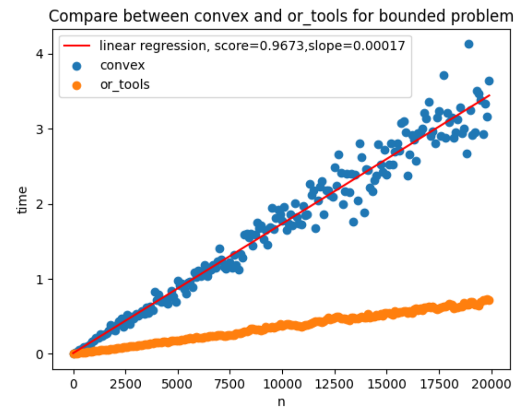

Figure 1: bounded problem comparison with linear regression analysis

The bounded problem has a score of 0.9673 which is considered highly correlated in the context of statistics. the convex solver is noticeably unstable compare to the GLOP solver, which will be further discussed in the next section.

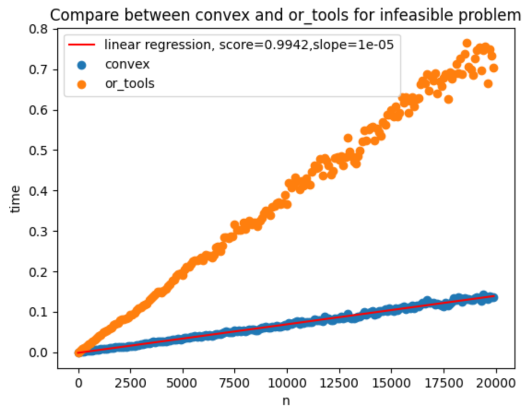

Figure 2: infeasible problem comparison with linear regression analysis

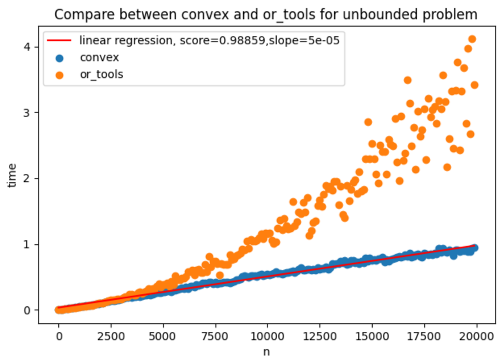

Figure 3: unbounded problem comparison with linear regression analysis

The unbounded and infeasible problems both perform much better in terms of speed and stability. In particular, both have a very high score which indicates the linear complexity of the algorithm aligns with the theory. Notice that in the unbounded graph, the GLOP solver tends to have polynomial complexity.

4.3 Discussion

Overall, the performance analysis of the convex solver indicates that it performs better in unbounded and infeasible problems, but worse when the problem is bounded. This observation is consistent with the algorithm's design, as the solver iterates over every constraint to find the best solution, and it exits early when the problem is unbounded or infeasible.

The instability of the bounded problem is mainly due to the significant impact of the order of constraints on performance. In the worst case, the solver may update the solution vector v every time, resulting in a time complexity of $O(n^2)$. However, as the data set is randomly generated, the expected time complexity is $O(n)$ [4, Lemma 4.8 pno 78].

The development of the convex solver involved addressing several challenges, particularly related to debugging edge cases and floating-point errors from thousands of randomly generated constraints. To address these challenges, I developed a set of methods for debugging large linear programs. For instance, shuffling the constraints repeatedly and reporting the number of constraints used so far from within the solver helped identify the minimum size of problematic cases that can be managed.

In conclusion, while the current implementation of the convex solver has limitations, particularly in the bounded problem, it has shown promise in solving unbounded and infeasible problems. As future work, I am already starting to explore the implementation of the three-dimensional case. With two weeks remaining until the deadline, I am hopeful that we can solve it as well.

5 Conclusion

In summary, the experimental results obtained confirm that the implementation supports the theoretical time complexity claim. Moreover, the proposed algorithm demonstrates better performance than the Simplex method in bounded and infeasible problems, although it is still inferior to GLOP in bounded problems. However, considering the differences in programming languages used, the proposed algorithm may be preferable over the Simplex method.

The project's main objective, which was the implementation and performance comparison of the algorithm, has been successfully completed. As for future work, it is suggested that the algorithm be extended to three-dimensional cases, which is a more challenging task due to the complexity of geometric properties and rotations. Despite the limited time remaining, the researcher hopes to make progress in this direction. Moreover, there is still room for further optimization of the implementation, which may reduce the slope of the linear regression model and potentially outperform the GLOP library at bounded problems with Python implementation.

6 Acknowledgements

I wish to express my heartfelt gratitude to my supervisor, Dr. Prosenjit Bose, for his invaluable guidance, support, and encouragement throughout this project. As a novice researcher, I initially struggled to find a suitable topic, but with Dr. Bose's assistance and expertise, I was able to select a relevant and captivating subject matter. I am grateful for the opportunity to participate in COMP 4804 and undertake this honours project, both of which have helped me to develop my skills and knowledge in computer science. Thank you, Dr. Bose, for your mentorship and unwavering commitment to my academic growth.

References

[1] A. Schrijver. *Theory of Linear and Integer programming.* Wiley-Interscience, 1986.

[2] Jonathan A. Kelner and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*, 2006.

[3] Google Developers. Linear Programming: Advanced Topics. https://developers.google.com/optimization/lp/lp_advanced, n.d. Accessed: March 29, 2023.

[4] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry.* Springer, Berlin, Germany, 2 edition, February 2000.