Linear Programming: Exhaustive Search vs. Simplex Method

Derryl Sayo

```python
from random import randint
import numpy as np
import itertools
import scipy
import time
import matplotlib.pyplot as plt
```

What is Linear Programming?

Linear programming is defined by the standard form:

$$\begin{aligned} \text{Maximize:} \quad & \mathbf{c}^T\mathbf{x} \\ \text{Subject To:} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^n$, A is a $m \times n$ matrix, $\mathbf{b} \in \mathbb{R}^m$.

The x_i are refered to as **decision variables** whose values are to be decided and are restricted by the set of m **constraints** of the form:

$$a_{i,0}x_{i,0} + \cdots + a_{i,n-1}x_{i,n-1} \begin{Bmatrix} \geq \\ = \\ \leq \end{Bmatrix} b_i$$

The **objective function** is a linear combination of the n decision variables with weights c_i written as:

$$c_0x_0 + c_1x_1 + \cdots + c_{n-1}x_{n-1}$$

In other words, linear programming is a way to model optimization problems with a linear objective function restricted by linear equality and inequality constraints.

Solving an LP problem entails finding the set of decision variables that will lead to the maximum/minimum value of the objective function while satisfying all constraints.

Different Ways to Solve Linear Programming Problems

The Fundamental Theorem of Linear Programming (see Wikipedia contributors 2022) states that the optimal value of a bounded linear programming problem will occur at the intersection of n constraint hyperplanes (ie. a vertex of the convex polytope enclosing the feasible set of solutions).

At first glance, a naive approach would be to enumerate every single vertex, check each vertex's feasibility, and then calculate the max objective value over all vertices. However, as we will see later, this approach will run into complications as the scale of the problems increase.

During the mid 1900s, linear programming became vital in the efforts of WW2 and ecomonics post war with many economists and mathematicians studying this class of problems independently (George B. Dantzig 1990). The **simplex method**, developed by George Bernard Dantzig and detailed in *Linear Programming and Extensions* (George Bernard Dantzig 1963), arose as an efficient method to solve linear programming problems by iterating through adjacent feasible vertices until the optimal solution is reached.

Other methods to solve linear programming problems include a family of interior-point methods which are not covered here.

Below we will detail the first two methods for solving linear programming problems and compare the runtimes of each.

Geometry of Linear Programming

Before we can iterate through the vertices, we first establish how we can programmatically find these vertices.

LP problems have linear constraints which can be represented as hyperplanes of dimension $n-1$ which make up a bounded polytope in $\mathbb{R}^n$. A vertex of a polytope in $\mathbb{R}^n$ is a point on its boundary at the intersection of n hyperplanes.

The optimal value of a bounded linear programming problem will be at one of these vertices. So, for each combination of n constraints from the total pool of $n + m$ constraints (ie. m constraints defined explicitly in A and n decision variable non-negativity constraints defined implicitly) we pick n rows from A and b and solve $Ax = b$ for the intersection. If we do this for each $\binom{n+m}{n}$ constraint intersections, we will have found every single vertex possible (feasible and infeasible) which we can then check for feasibility and then optimality.

Exhaustive Search by Enumerating All Vertices

To be able to use linear programming solvers, we must create random LP problems. Below is a simple generator that returns the A, b, c matrices/vectors with randomly selected dimensions m and n:

```python
def generate_random_LPP():
        """
        Generate a random linear programming problem with `n` decision
 ↪  variables and `m` constraints.

        Returns:
        - A matrix `A` of LP constraints
        - A vector `b` of LP constraint values
        - The number of decision variables `n`
        """

        # Number of constraints in the A matrix
        m = randint(1, 12)

        # Number of decision variables
        n = randint(1, 12)

        A = np.rint(np.random.rand(m, n) * 100)
        b = np.rint(np.random.rand(m)  * 100)
        c = np.rint(np.random.rand(n) * 100)

        return A, b, c
```

Then, we create a python function that will iterate through all feasible vertices and find the maximal objective value for the optimal solution:

```python
def get_vertices(A, b):
        """
        Find all vertices for an LP problem defined by A and b.

        Params:
```

```
        - A: m x n matrix of constraints (LHS of constraint inequalities)
        - b: m x 1 matrix of constraint values (ie. RHS of constraint
↪   inequalities)

        Returns:
        A list of feasible vertices
        """

        m = np.shape(A)[0]
        n = np.shape(A)[1]

        # Use itertools to get all n-tuple combinations from the m + n
         ↪  constraints
        row_index_combinations = itertools.combinations(range(m + n), n)

        # Since LP in standard form implicitly defines the x >= 0 constraint
         ↪  which corresponds to the x_i = 0 hyperplane
        # So, we vstack the identity matrix onto A and hstack a vector of 0s
         ↪  to b to get
        # all m + n constraint hyperplanes
        A_stacked = np.vstack((A, np.eye(n)))
        b_stacked = np.hstack((b, np.zeros(n)))

        feasible_vertices = []

        for row_combo in row_index_combinations:
                # Select n constraints
                A_constraints = A_stacked[list(row_combo), :]
                b_constraints = b_stacked[list(row_combo)]

                x = None

                try:
                  # Solve for the intersection of n constraint hyperplanes.
                   ↪  If
                  # we cannot solve due to a singular matrix, catch the error
                  # and continue
                  x = np.linalg.solve(A_constraints[:, :n], b_constraints)

                  # Set too small values to 0 (account for floating point
                   ↪  errors)
                  x[(np.abs(x) < 1e-14)] = 0
                # Check that the vertex satisfies non-negative constraints
                 ↪  and all inequality constraints
                # within a set tolerance
                if np.all(x >= 0) and np.all((A @ x) <= (b + 1e-10)):
                  feasible_vertices.append(x)
              except:
                pass

        return feasible_vertices


def LP_exhaustive_search(A, b, c):
        """
        Solve the given linear programming problem by exhaustive search
 ↪  through all vertices.

        Params:
        - A: m x n matrix of constraints
        - b: m x 1 vector of constraint inequality values
        - c: 1 x n vector of objective function coefficients

        Returns:
        - Optimal objective value of min -c * x
        """
        feasible_vertices = get_vertices(A, b)

        objective_values = []

        # Check the objective value for each of the found vertices
        for vertex in feasible_vertices:
                vertex_obj_val = np.dot(-c, vertex)
                objective_values.append(vertex_obj_val)

        return np.min(objective_values)
```

Solving LP Problems by Simplex Method Using SciPy

Detailed below is a python function which solves the given linear programming problem using `scipy.optimize.linprog` (SciPy Developers 2024) using the simplex method:

```
def LP_simplex(A, b, c):
        """
        Wrapper to call scipy.optimize.linprog with method="simplex" for
 ↪  consistency
        """
        return scipy.optimize.linprog(-c, A_ub=A, b_ub=b, method="simplex")
```

Comparing Exhaustive Search vs. Simplex Method

We can now run compare these two routines. Below we run 150 iterations of randomized LP problems of varying sizes and measure the runtimes of both the exhaustive search and the SciPy solver:

```
num_iterations = 150

scipy_test_results = []
exhaustive_test_results = []

for _ in range(1, num_iterations):
    A, b, c = generate_random_LPP()
    m, n = np.shape(A)

    t_scipy = time.time()
    scipy_res = LP_simplex(A, b, c)
    duration_scipy = time.time() - t_scipy
    scipy_test_results.append((m + n, duration_scipy))

    t_exhaustive = time.time()
    exhaustive_res = LP_exhaustive_search(A, b, c)
    duration_exhaustive = time.time() - t_exhaustive
    exhaustive_test_results.append((m + n, duration_exhaustive))

    # Check that the results are the same to 5 decimal places
    assert(np.round(scipy_res.fun, 5) == np.round(exhaustive_res, 5))
```

Visualizing Exhaustive Search vs. Simplex Method

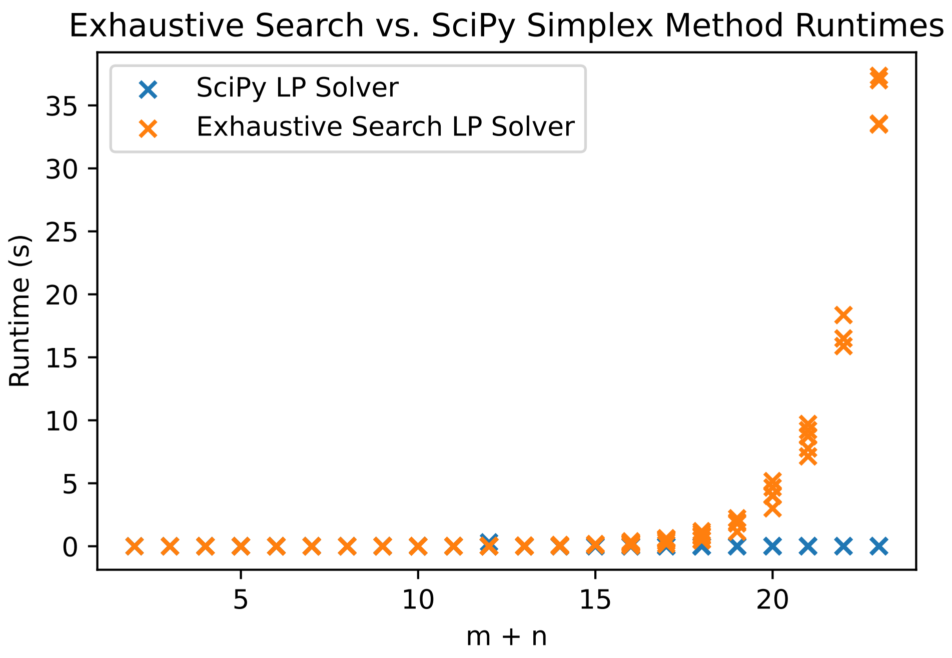
Below is a figure comparing the runtimes between exhaustive search and the simplex method for problems of size $m + n$:

```
x1, y1 = zip(*scipy_test_results)
x2, y2 = zip(*exhaustive_test_results)

plt.scatter(x1, y1, label="SciPy LP Solver", marker="x")
plt.scatter(x2, y2, label="Exhaustive Search LP Solver", marker="x")

plt.xlabel("m + n")
plt.ylabel("Runtime (s)")
plt.title("Exhaustive Search vs. SciPy Simplex Method Runtimes")
plt.legend()

plt.show()
```



With the current implementation, the exhaustive search method's runtime grows exponentially as the problem size grows and already takes a non-trivial amount of time for small problems of size $n+m \approx 25$. Meanwhile, the simplex method remains efficient and doesn't grow rapidly.

Runtime Analysis

In order to compare the efficiency of these two algorithms we first need to establish some notion of quantifiable measures and distinguish between worst case and average case runtimes. Oftentimes, we use some function on the number of operations required in a given routine.

In the case of linear programming solvers, we know that the optimal value exists at a vertex of the feasible region and that the basis of the simplex algorithm is iterating through adjacent vertices with increasing objection function value so a sensible measure would be to see how much many vertices we must check before arriving at the optimal solution.

Worst Case Time

In regards to the simplex method/other forms of vertex based solvers, the "worst case" runtime can be thought of as "if I take the longest route possible, how many vertices must I pass through"? For a problem with m constraints and n decision variables, each vertex exists as the intersection of n constraint hyperplanes. So, the number of vertices is $n+m$ choose n, or:

$$\binom{n+m}{n} = \frac{(n+m)!}{n!m!}$$

Written as the fraction of factorials, it becomes clear that this grows exponentially relative to problem size. For example, a problem with 12 constraints and 10 decision variables in the worst-case needs to check $\binom{22}{10} = 646646$ vertices. For non-trivial LP problems, this number quickly becomes too large to viably search through every vertex until optimality is reached.

Average Case Time

As seen in the figure above, practical LP solvers' runtime does not grow exponentially even though the theoretical worst-case runtime does grow exponentially. Explanations may include that the exhaustive search implementation detailed above requires solving a linear system of equations for every $n+m$ choose n vertices without extra heurstics to trim vertices outside the feasible regions compared to the actual geometry of the problem allowing for very few pivots to reach optimality.

Further research has gone into why the simplex algorithm performs well under most circumstances. In "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time" (Spielman and Teng 2003), Spielman and Teng have shown that under "smoothed analysis", the simplex method has polynomial-time complexity rather than exponential.

Conclusion

Above, we have detailed code to generate and solve linear programming problems using exhaustive search through all vertices and have seen that the runtime grows exponentially when compared to the more stable simplex algorithm. Althrough both algorithms theoretically have the same worst case, recent analysis has shown why the simplex algorithm usually only takes polynomial time.

When reading further into this topic, future topics of interest include a more in-depth look into the smoothed analysis paper on the practical efficiency of the simplex algorithm, interior point LP solvers, and the Klee-Minty cube for a problem designed to demonstrate worst-case performance of the simplex algorithm.

References

Dantzig, George B. 1990. "Origins of the Simplex Method." In *A History of Scientific Computing*, 141–51. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/87252.88081.

Dantzig, George Bernard. 1963. *Linear Programming and Extensions*. Santa Monica, CA: RAND Corporation. https://doi.org/10.7249/R366.

SciPy Developers. 2024. "SciPy Optimize Module - Linprog." https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html.

Spielman, Daniel A., and Shang-Hua Teng. 2003. "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time." https://arxiv.org/abs/cs/0111050.

Wikipedia contributors. 2022. "Fundamental Theorem of Linear Programming — Wikipedia, the Free Encyclopedia." 2022. https://en.wikipedia.org/w/index.php?title=Fundamental_theorem_of_linear_programming&oldid=1113019829.