实验报告

申长硕 @PB22020518
AI&DS
stephen_shen@mail.ustc.edu.cn

2025 年 1 月 9 日

摘要

本实验旨在实现一种线性规划的求解方法——单纯形法，并通过编程实现和实验测试，探讨不同问题规模下单纯形法的运行效率和数值稳定性。此外，我们还结合了现代数学规划求解器（如 SciPy 的 linprog）进行对比分析，以验证单纯形法的性能与局限性。本实验的目标是通过模块化设计，从线性规划问题的标准形式转化、冗余约束的处理、初始可行解的构造到单纯形法的迭代求解，完整实现线性规划问题的求解流程，并通过实验数据分析运行时间和方差随问题规模的变化规律。

1 背景与目标

线性规划（Linear Programming, LP）是运筹学和优化领域的重要研究方向，其目标是在线性约束条件下最小化或最大化目标函数值。线性规划在资源分配、生产调度、运输问题、金融投资等实际应用中发挥了重要作用。

单纯形法（Simplex Method）是最经典的线性规划求解算法之一，由 George Dantzig 于 1947 年提出。该算法通过迭代的方式，在可行解空间的顶点之间跳跃，最终找到最优解。尽管单纯形法在最坏情况下的时间复杂度是指数级，但其在实际问题中通常表现出极高的效率，因此被广泛应用于求解稀疏、大规模的线性规划问题。

近年来，随着计算能力的提升和优化算法的发展，基于内点法（Interior Point Method）的线性规划求解器逐渐成为主流。内点法通过在可行解空间的内部进行搜索，避免了单纯形法在退化问题中的循环现象。然而，单纯形法由于其易于实现和较低的单次迭代计算成本，仍然具有重要的研究意义和实际应用价值。

本实验的目标是：

- 实现线性规划问题的单纯形法求解流程，包括标准形式转化、冗余约束处理、初始可行解构造以及迭代求解。
- 测试单纯形法在不同规模问题上的运行时间和数值稳定性，并与现代优化求解器（如 SciPy 的 linprog）进行对比。
- 探究 Bland’s rule 等方法在避免退化解循环中的效果。
- 分析单纯形法在实际问题中的性能表现以及其适用场景。

通过本实验，我们希望对经典算法单纯形法的原理、实现过程以及在实际问题中的表现有更深入的理解，并结合实验数据分析其优缺点，为后续优化算法的学习和研究提供参考。

2 方法与实现

本实验实现了线性规划的单纯形法求解器，主要分为以下几个模块：

2.1 线性规划问题的标准形式转换

线性规划的基本形式为:

$$\min\ c^\top x \quad \text{s.t.} \quad Ax \leq b, \quad x \geq 0$$

为了简化问题，首先将其转化为标准形式:

$$\min\ c^\top x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0$$

标准形式的转换包括以下步骤: - 移除 A 中的全零行; - 如果 $b_i < 0$，将对应的 A_i 和 b_i 符号取反。

实现代码如下:

```python
@staticmethod
def convert_to_standard_form(c, A, b):
    zero_row_indices = [i for i in range(len(b)) if np.all(A[i, :] == 0)]
    A = np.delete(A, zero_row_indices, axis=0)
    b = np.delete(b, zero_row_indices, axis=0)
    for i in range(A.shape[0]):
        if b[i] < 0:
            b[i] = -b[i]
            A[i] = -A[i]
    return c, A, b
```

2.2 冗余约束的移除

为了提高计算效率，需移除 A 中的线性相关行。我们采用 QR 分解的方法:

$$B = A^\top, \quad B = QR$$

其中，Q 是正交矩阵，R 是上三角矩阵。通过检查 R 的对角元是否为零，移除线性相关的行。

实现代码如下:

```python
@staticmethod
def remove_redundant_constraints(A, b):
    A = np.column_stack((A, b))
    q, r = np.linalg.qr(A.T)
    valid_indices = np.abs(np.diag(r)) > 1e-6
    A_reduced = (A.T[:, valid_indices]).T
    return A_reduced[:, :-1], A_reduced[:, -1]
```

2.3 初始可行解的构造

为了保证单纯形法的初始解可行，本实验采用大 M 法（Big-M Method）。通过引入人工变量，构造如下目标函数:

$$\min\ c^\top x + M \sum_{i=1}^{m} y_i \tag{1}$$

其中 M 是一个非常大的常数。人工变量 y_i 确保初始解始终可行。

实现代码如下:

```
    @staticmethod
    def initialize_feasible_solution(A, b, c):
        m, n = A.shape
        M = max((np.abs(c)).max() * 1e5, 1e6)
        artificial_vars = np.eye(m)
        A_aug = np.hstack([A, artificial_vars])
        c_aug = np.hstack([c, M * np.ones(m)])
        initial_basic = np.concatenate([np.zeros(n), b])
        return A_aug, c_aug, initial_basic, M
```

2.4 单纯形法的迭代求解

单纯形法的核心是基于以下步骤的迭代：

1. 计算检验数：

$$\bar{c}_N = c_N - c_B^\top B^{-1} A_N$$

若 $\bar{c}_N \geq 0$，则当前解最优。

2. 确定入基变量和出基变量，更新基解。

为避免循环退化问题，实验中采用 Bland’s rule，即每次入基变量选择下标最小的变量。实现代码如下：

```
    def simplex_method(self, A, b, c, initial_basic, M):
        m, n = A.shape
        x = initial_basic
        var_index = np.arange(n - m, n)
        table = np.column_stack((A, b))
        zs = np.zeros(n + 1)
        zs[:-1] = c - np.dot(c[var_index], table[:, :-1])
        zs[-1] = -np.dot(c[var_index], table[:, -1])

        while True:
            in_index = next((i for i in range(n) if zs[i] < 0), -1)
            if in_index == -1:
                x = np.zeros(n)
                x[var_index] = table[:, -1]
                return x, -zs[-1]

            theta = np.inf
            out_index = -1
            for i in range(m):
                if table[i, in_index] > 0:
                    ratio = table[i, -1] / table[i, in_index]
                    if ratio < theta:
                        theta = ratio
                        out_index = i

            if out_index == -1:
                self.No_unbounded_solution += 1
                raise ValueError("Linear programming problem is unbounded.")

            var_index[out_index] = in_index
            pivot = table[out_index, in_index]
```

```
            table[out_index, :] /= pivot
            for i in range(m):
                if i != out_index:
                    table[i, :] -= table[i, in_index] * table[out_index, :]
            zs[:-1] = c - np.dot(c[var_index], table[:, :-1])
            zs[-1] = -np.dot(c[var_index], table[:, -1])
```

3 结果与分析

3.1 实验设置

本实验包括两部分测试：

1. 固定测试案例：按照实验要求，设计了 4 个具有代表性的线性规划问题，用于验证算法在不同情况下的正确性，包括有可行解、冗余约束、无解（无可行域）和无解（无界）的情形。
2. 随机测试案例：对不同规模的线性规划问题（变量数量 m 从 10 到 200，步长为 10）随机生成 20 个测试用例，分别使用单纯形法和 SciPy 的 **linprog** 求解。记录每个方法的运行时间，并计算其平均值和方差。

3.2 固定测试案例分析

固定测试案例的结果如图 1 所示。可以观察到，单纯形法和 SciPy 的 **linprog** 在有解的情况下能够得到相同的最优解和目标值；在无解（无可行域）和无解（无界）的情况下，算法能够正确检测问题的属性。

```
● (base) shenc@tk:~/Desktop/study/OperationResearch_Labs/lab1$ python lab1.py
Initializing data generator and simplex solver...
Solving predefined test cases...

Test Case 1: Feasible solution
Matrix A:
[[1 2 2 1 0 0]
 [2 1 2 0 1 0]
 [2 2 1 0 0 1]]
Vector b: [20 20 20]
Vector c: [-10 -12 -12   0   0   0]
Simplex Method Solution: [4. 4. 4. 0. 0. 0.]
Simplex Method Objective: -136.0
SciPy Linprog Solution: [4. 4. 4. 0. 0. 0.]
SciPy Linprog Objective: -136.0

Test Case 2: Redundant constraints
Matrix A:
[[1 2 3]
 [2 4 6]
 [1 1 1]]
Vector b: [ 6 12  3]
Vector c: [1 2 3]
Simplex Method Solution: [0. 3. 0.]
Simplex Method Objective: 6.0
SciPy Linprog Solution: [1.5 0.  1.5]
SciPy Linprog Objective: 6.0

Test Case 3: Infeasible solution (no feasible region)
Matrix A:
[[1 0 0]
 [0 1 0]
 [0 0 1]
 [1 0 1]]
Vector b: [2 2 2 2]
Vector c: [1 1 1]
Error: Linear programming problem is infeasible.
Simplex Method Solution: None
Simplex Method Objective: None
SciPy Linprog Solution: Failed
SciPy Linprog Objective: Failed

Test Case 4: Unbounded solution
Matrix A:
[[ 1 -1]
 [-1  1]]
Vector b: [0 0]
Vector c: [-1  0]
Error: Linear programming problem is unbounded.
Simplex Method Solution: None
Simplex Method Objective: None
SciPy Linprog Solution: Failed
SciPy Linprog Objective: Failed
```

图 1: 实验要求中的 test cases

固定测试案例验证了单纯形法的正确性，并说明了其能够有效处理冗余约束问题，避免退化解循环，同时在无解和无界的情况下给出准确的判定结果。

3.3 随机测试案例分析

随机测试案例的结果如图 3 所示，展示了运行时间和运行时间方差随问题规模的变化情况。

图 2: 随机 test case 测试

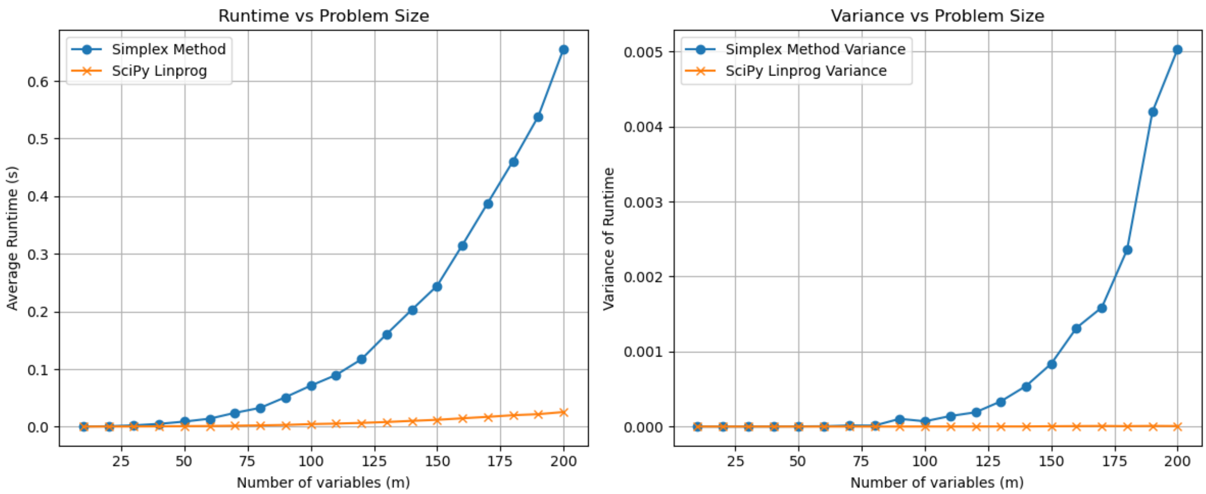

图 3: 运行时间与方差随问题规模的变化

从图中可以观察到：

- **运行时间分析**：单纯形法的运行时间随问题规模的增大呈指数增长。这是因为单纯形法在每次迭代中需要执行矩阵计算，且迭代次数与问题规模成正相关。而 SciPy 的 **linprog** 基于内点法，其运行时间随问题规模增长表现平稳，效率更高。
- **运行时间方差分析**：单纯形法的运行时间方差也随着问题规模的增大而增大，这表明其对问题结构较为敏感，尤其是在退化问题中可能需要执行更多的迭代。而 SciPy 的 **linprog** 始终保持稳定的运行时间方差，说明其性能在不同问题中更为一致。

- -

3.4 实验结果的进一步分析

结合固定和随机测试案例，可以得出以下结论：

1. 单纯形法在小规模问题上表现较好，能够快速给出准确的最优解。
2. 随着问题规模的增大，单纯形法的运行时间增长迅速，难以应对大规模问题。
3. SciPy 的 **linprog** 基于内点法，在运行效率和稳定性上均优于单纯形法，适用于大规模问题。
4. 固定测试案例表明，单纯形法能够正确处理冗余约束问题，避免退化解循环，且在无解和无界情况下给出准确判定。

4 总结

通过本实验，我们验证了单纯形法的正确性和适用性。总结如下：

1. 单纯形法在小规模问题上运行效率较高，但随着问题规模增大，其运行时间迅速增长。
2. SciPy 的 **linprog** 在运行效率和稳定性上表现更优，尤其在大规模问题和复杂问题中更具优势。
3. 单纯形法能够有效处理冗余约束问题，并通过 Bland’s Rule 避免退化解循环。
4. 随机测试案例表明，单纯形法的性能对问题结构较为敏感，而内点法的表现更为稳定。

实验结果表明，尽管单纯形法是经典的线性规划算法，但在现代优化问题中，基于内点法的求解器更为高效，适用于更复杂的应用场景。