运筹学报告——PROJ 1

谌奕同 PB22000250

January 5, 2025

1 背景介绍

在运筹学和优化领域，单纯形法是一种广泛应用于线性规划问题的算法。线性规划问题的目标是最大化或最小化一个线性目标函数，同时满足一组线性约束条件。单纯形法通过迭代地移动到相邻的基可行解来寻找最优解。

2 算法介绍

2.1 简介

在运筹学和优化领域，单纯形法是一种广泛应用于线性规划问题的算法。线性规划问题的目标是最大化或最小化一个线性目标函数，同时满足一组线性约束条件。单纯形法通过迭代地移动到相邻的基可行解来寻找最优解。

2.2 算法描述

2.2.1 单纯形法概述

单纯形法的目标是最大化或最小化一个线性目标函数，同时满足一组线性约束条件。算法通过以下步骤实现：

1. 初始化：选择一个初始基可行解。
2. 迭代：在每次迭代中，选择一个进入基的变量和一个离开基的变量，并更新解。
3. 终止条件：当所有检验数（reduced costs）均大于或等于零时，当前解为最优解；如果方向向量的所有分量均非正，则问题无界。

2.3 具体各个模块的实现和介绍

2.3.1 转化为标准形式

方便起见，我们假设最初的约束是 $Ax \leq b \quad x \geq 0$, 我们将其转化为 $A\_std\ x = b\_std \quad x\_std \geq 0$

To Standard Form

```
def to_standard_form(c, A, b):
"""
Convert the linear programming problem to standard form.
"""
m, n = A.shape
I = np.eye(m)
A_std = np.hstack((A, I))
c_std = np.concatenate((c, np.zeros(m)))
b_std = b.copy()
basis = list(range(n, n+m))
return basis, c_std, A_std, b_std
```

2.3.2 去除冗余约束

对于 $A\_std\ x = b\_std \quad x\_std \geq 0$ 等式约束问题，去除多余约束的想法是自然的，可以通过 SVD，QR，或者最基础的 pivot_based 来实现这一效果。下面是一个示例代码：

Remove Redundant Constraints by SVD

```python
def remove_redundant_constraints(A, b, tol=1e-9):
"""
Remove redundant constraints from the constraint matrix A and vector b.
"""
m, n = A.shape
rank = np.linalg.matrix_rank(A, tol)
if rank < m:
    U, s, Vt = np.linalg.svd(A)
    non_zero_singular_values = np.sum(s > tol)
    A = U[:, :non_zero_singular_values] @ np.diag(s[:non_zero_singular_values]) @
     ↪  Vt[:non_zero_singular_values, :]
    b = U[:, :non_zero_singular_values] @ b
return A, b
```

Remove Redundant Constraints by pivot_based method

```python
def remove_redundant_constraints(A, b, tol=1e-9):
"""
A pivot-based approach to remove redundant constraints.
This avoids using SVD for large problems and can be faster in practice.
"""
# Make copies to avoid modifying original arrays
A_work = A.copy().astype(float)
b_work = b.copy().astype(float)

m, n = A_work.shape
row = 0

# Perform a straightforward pivot-based row-reduction
for col in range(n):
    # 1) Find pivot
    pivot = row
    while pivot < m and abs(A_work[pivot, col]) < tol:
        pivot += 1
    if pivot == m:
        continue  # No pivot in this column

    # 2) Swap pivot row to current row if needed
    if pivot != row:
        A_work[[row, pivot]] = A_work[[pivot, row]]
        b_work[[row, pivot]] = b_work[[pivot, row]]

    # 3) Normalize pivot row
    pivot_val = A_work[row, col]
```

```
        if abs(pivot_val) < tol:
            continue
        A_work[row] /= pivot_val
        b_work[row] /= pivot_val

        # 4) Eliminate below pivot
        for r in range(row + 1, m):
            factor = A_work[r, col]
            A_work[r] -= factor * A_work[row]
            b_work[r] -= factor * b_work[row]

        row += 1
        if row == m:
            break

    # Identify nonzero rows
    nonzero_idx = []
    for i in range(m):
        # If row is effectively not zero
        if not np.all(np.abs(A_work[i]) < tol) or np.abs(b_work[i]) >= tol:
            nonzero_idx.append(i)

    # Extract the filtered constraints
    A_filtered = A_work[nonzero_idx, :]
    b_filtered = b_work[nonzero_idx]

    return A_filtered, b_filtered
```

2.3.3 Big M method

Big_M_method

```
def big_m_method(c_std, A_std, b_std):
    m, n = A_std.shape
    A_art = np.hstack((A_std, np.eye(m)))
    big_M = 1e6
    c_art = np.concatenate((c_std, [big_M] * m))
    basis = list(range(n, n+m))
    x_opt, obj_val= simplex_iteration_straight(c_art, A_art, b_std, basis)
    return basis, x_opt, c_art, A_art
```

2.3.4 Simplex Iteration

Simplex iteration 的实现是 simplex method 能够成功的关键所在。本 proj 的 simplex iteration 实现几乎完全参照讲义。

Optimal Condition

Optimal Condition

```
for _ in range(max_iter):
# Compute reduced costs
cb = c[basis]
```

```python
B = A[:, basis]
try:
    invB = np.linalg.inv(B)
except np.linalg.LinAlgError:
    return None, "Infeasible domain (singular basis)"
lambd = cb @ invB
r = c - lambd @ A
```

LU decomposition 在后续实验中，随着系数矩阵 A 维数的增大，invB 造成的数值不稳定很容易造出我们的算法给出错误的判断（将有最优解的问题判断成无界解）。我们采用 LU 分解来缓解这一问题。

Inverse Replaced by LU decomposition

```python
for _ in range(max_iter):
# Compute reduced costs
cb = c[basis]
B = A[:, basis]
try:
    lu, piv = lu_factor(B)
except np.linalg.LinAlgError:
    return None, "Infeasible domain (singular basis)"
lu1, piv1 = lu_factor(B.T)
lambd = lu_solve((lu1, piv1), cb.T).T
r = c - lambd @ A
```

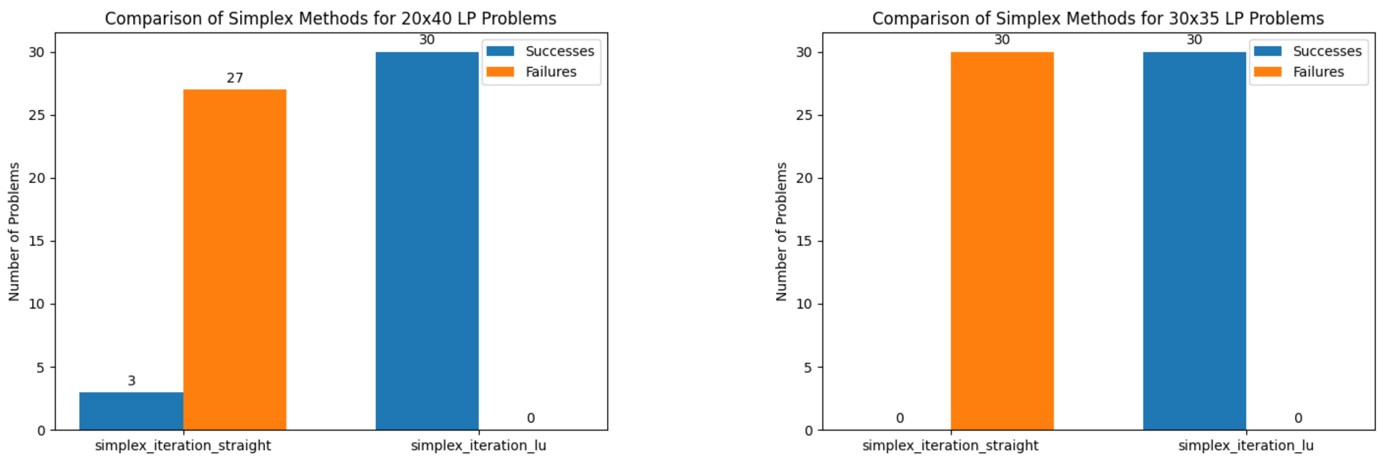

Figure 1: Stability of LU decomposition

右边的 simplex_iteration 将所有的 np.linalg.inv 和 np.linalg.solve 换成 lu_factor, lu_solve。生成随机线性规划问题的代码如下：

Generate_solvable_lp_linprog

```python
def generate_random_lp(m, n):
    """
    Generate a random LP problem.
    """
    c = np.random.randint(-10, 11, n)
```

```python
    A = np.random.randint(-10, 11, (m, n))
    b = np.random.randint(0, 10, m)
    return c, A, b

def generate_solvable_lp_linprog(m, n):
    while True:
        c, A, b = generate_random_lp(m, n)
        res = linprog(c, A_ub=A, b_ub=b, method='simplex')
        if res.success:
            return c, A, b
```

Bland's Rule 为了避免单纯形法中的循环问题，我们实现了 Bland's 规则，选择索引最小的负检验数作为进入基的变量。

Bland's Rule

```python
def first_negative(r, basis):
    for j in range(len(r)):
        if j not in basis and r[j] < 0:
            return j
    return None

r = c - lambd @ A

# If all non-basis reduced costs >= 0, we have an optimal solution
if all_non_basis_non_negative(r, basis):
    return x, "Optimal solution = " + str(c @ x)

# Choose entering variable -> change to bland's rule
entering = first_negative(r, basis)
# entering = np.argmin(r) # you will see the exceeding of max_iter
```

2.4 增加代码稳定性

2.4.1 无界解还是无可行域？

另一个难点是正确判断无解的具体情况。

Unbounded Solution 我们通过检查方向向量 d 的所有分量是否均非正来实现判断无界解

Check the Unbounded Solution

```python
if np.all(d <= 0):
    return None, "Unbounded"

def check_slack_variables(x_opt, n):
    """
    Check if the slack variables are all non-negative to determine feasibility.
    """
    slack_variables = x_opt[n:]
    if all(slack_variables >= 0):
        return True, "Feasible domain"
```

```
    else:
        return False, "Infeasible domain"
```

Infeasible domain 如何判断给定的约束条件是否是 infeasible domain?

本 proj 采取的方法是正常求解转化为标准形式的 LP，再通过变量的正负来判断是否是 Infeasible Domain.

Check the Infeasible domain

```
# 检查可行性
feasible, status = check_slack_variables(x_opt, len(c))
if not feasible:
    return None, status

return x_original, obj_val
```

2.4.2 输入验证与标准化

为了确保算法的鲁棒性，我们在 `solve_lp` 函数中添加了输入验证和标准化步骤，确保输入数据的维度和类型正确。

Capture the LinAlgError

```
def solve\_lp(c, A, b):
    if not isinstance(A, np.ndarray) or not isinstance(b, np.ndarray) or not isinstance(c,
    ↪ np.ndarray):
        raise TypeError("Inputs must be numpy arrays")

    if A.shape[0] != len(b):
        raise ValueError("Inconsistent dimensions between A and b")

    if A.shape[1] != len(c):
        raise ValueError("Inconsistent dimensions between A and c")

    basis, c_std, A_std, b_std = to_standard_form(c, A, b)
    try:
        x_opt, obj_val = simplex_iteration(A=A_std, b=b_std, c=c_std, basis=basis)
    except np.linalg.LinAlgError:
        return None, "Numerical instability encountered"
```

3 实验结果

3.1 简单的测试用例

Normal Case

```
# 正常求解
c1 = np.array([-2, -3, 5], dtype=float)
A1 = np.array([[2, -1, 1],
               [1, 1, 3],
               [1, -1, 4],
               [3, 1, 2]], dtype=float)
b1 = np.array([2, 5, 6, 8], dtype=float)
x_opt, obj_val = solve_lp_lu(c1, A1, b1)
```

Redundant constraint case

```
# 有冗余约束
c2 = np.array([1, 1], dtype=float)
A2 = np.array([[1, 2], [2, 4]], dtype=float)
b2 = np.array([5, 10], dtype=float)
A2_filtered, b2_filtered = remove_redundant_constraints(A2, b2)
print(" 使用 linprog 求解")
print(linprog(c2, A_ub=A2, b_ub=b2))
x_opt, obj_val = solve_lp_lu(c2, A2_filtered, b2_filtered)
print(" 有冗余约束:")
```

Infeasible Domain

```
# 无可行域
c3 = np.array([1, 1], dtype=float)
A3 = np.array([[1, 1], [-1, -1]], dtype=float)
b3 = np.array([1, -3], dtype=float)
x_opt, obj_val = solve_lp_lu(c3, A3, b3)
```

Unbounded

```
# 无界
c4 = np.array([-1, 0], dtype=float)
A4 = np.array([[1, -1], [-1, 1]], dtype=float)
b4 = np.array([0, 0], dtype=float)
x_opt, obj_val = solve_lp_lu(c4, A4, b4)
print(" 无界:")
```

我们的程序均给出了正确的判断和求解，具体结果可以运行 test_random_specific.py 文件得到。

3.2 Custom Simplex Iteration v.s. Scipy.optimize.linprog

3.2.1 Success Rates

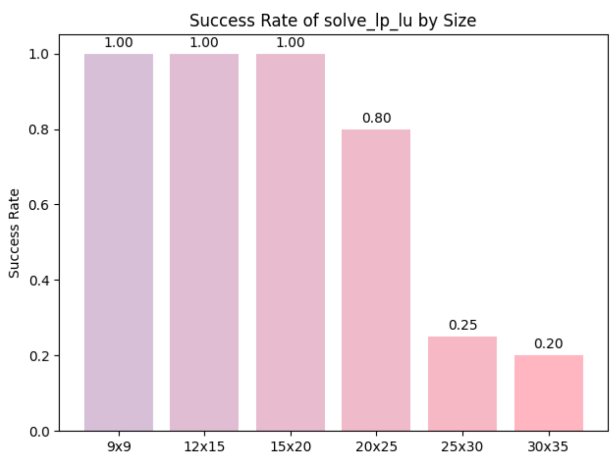

Figure 2: slack variables are chosen to be initial basis

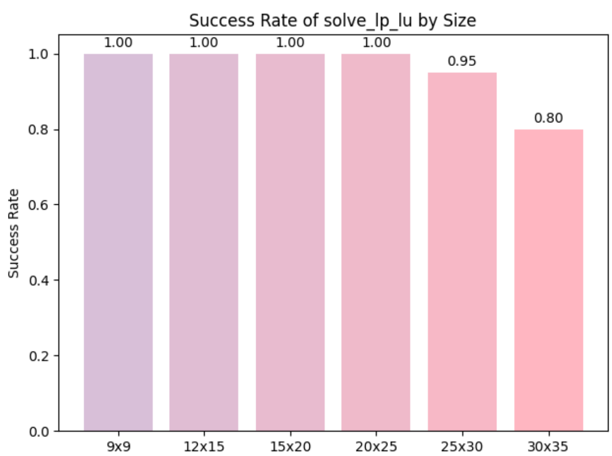

Figure 3: Big_M_method select the Right basis

用松弛变量作为最初的基向量，可以观察到求解正确的概率随着 problem size 增大而迅速降低。下面我们通过大 M 方法，挑选正确的初始基变量。

虽然正确率已经提升了很多，可还是不能让人满意。通过调整 simplex_iteration 中的 max_iter 为 500 后，可以得

到正确的结果。

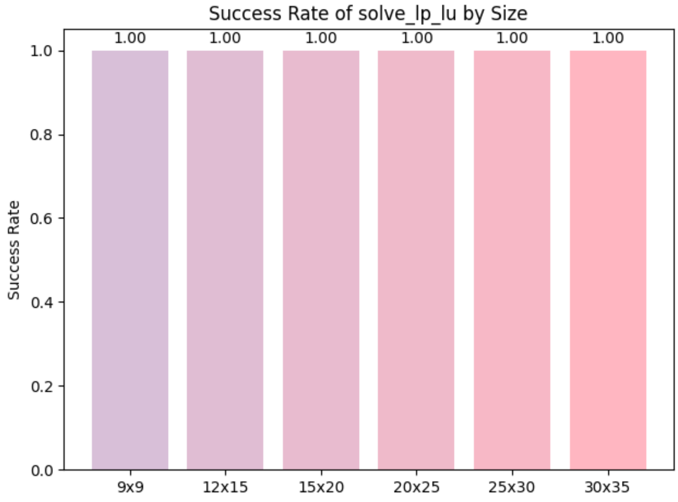

Figure 4: make max_iter bigger

3.2.2 Effiency

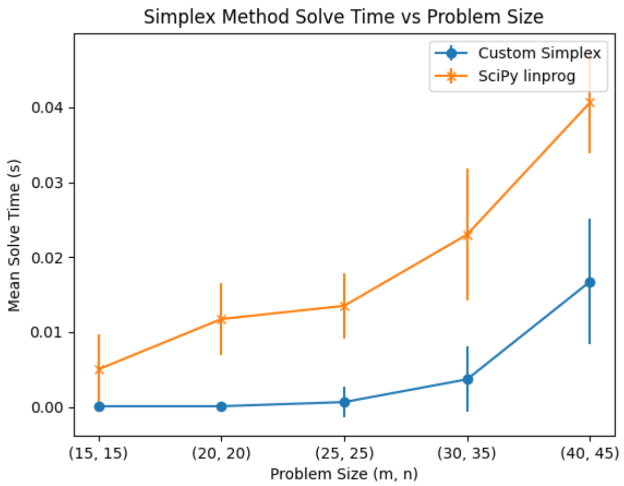

Figure 5: Efficiency between Custom Simplex Iteration and linprog from scipy

Custom Simplex Iteration 的求解效率在较小的 problem size 上优于 scipy.optimize.linprog，可能是因为 Custom Simplex Iteration 的预处理比较简单

4 总结

Simplex method simplex method 求解线性规划问题展现了非凡的效率，但数值稳定问题对此 proj 依然非常严重。采取单纯形表或许是更好是的实现单纯形法的方法。碍于时间紧张，作者并没有实现单纯形表。用两阶段法替代大 M 方法，以及对代码的进一步整理和简化都是未来的改进方向。

感谢陈老师和各位助教的辛勤付出。

5 伪代码

以下是单纯形法的伪代码：

Algorithm 1 单纯形法迭代

```
输入: 目标函数系数 c，约束矩阵 A，约束向量 b
输出: 最优解 x_opt 和目标函数值 obj_val
function SIMPLEX_ITERATION(c, A, b, basis)
    初始化最大迭代次数 max_iter
    初始化解向量 x 为零向量
    if 基变量数量超过约束条件数量 then
        return "Infeasible domain (basis size mismatch)"
    end if
    计算初始解：
      提取基变量对应的矩阵 B
      通过求解 Bx_b = b 计算初始基变量解 x_b
    if B 是奇异矩阵 then
        return "Infeasible domain (singular basis)"
    end if
    将 x_b 赋值给 x 中对应的基变量位置
    进入迭代过程：
    for 每次迭代 do
        计算检验数 r
        if 所有检验数均大于或等于零 then
            return 当前解和目标函数值
        end if
        选择进入基的变量 entering
        计算方向向量 d
        if 所有 d 的分量均非正 then
            return "Unbounded"
        end if
        进行比例测试，选择离开基的变量 leaving
        更新基变量和解
    end for
    if 超过最大迭代次数 then
        return "Infeasible or iteration limit reached"
    end if
end function
```

Algorithm 2 线性规划求解函数

```
function SOLVE_LP( c, A, b )
    验证输入
    将问题转换为标准形式
    调用 simplex_iteration 函数求解
    return 最优解和目标函数值
end function
```