浙江大学

应用运筹学基础

分枝定界法求解整数规划

第三组

分枝定界法求解整数规划

小组成员:

陈翰逸
林锦铿
黄文璨
赵竟霖

Contents

1 背景介绍 **1**

2 算法描述 **2**

2.1 单纯形法 2

2.1.1 标准形式 3

2.1.2 松弛形式 3

2.1.3 转轴变换 4

2.1.4 最优化过程 4

2.1.5 Bland 法则 5

2.2 分枝定界法 6

3 实现过程 **8**

3.1 Parse 8

3.1.1 预处理 9

3.1.2 优化 9

3.1.3 工具函数 10

3.2 单纯形法 11

3.2.1 源代码分析 11

3.2.2 其他函数分析 12

3.3 分枝定界法 13

4 测试结果 **15**

5 分析与评价 **16**

5.1 时间复杂度 16

5.1.1 单纯形法 16

5.1.2 分枝定界法 16

5.2 空间复杂度 17

5.3 评价 18

6 总结 **19**

Appendices **20**

A 源代码 **21**

B 声明与分工 **49**

Chapter 1

背景介绍

有很多规划问题，他们的决策变量都是连续的，并且约束和目标函数都是线性的，这种规划我们称为线性规划。线性规划是相对比较容易求解的，但是有许多实际问题，譬如人员、机器或者车辆的分配，它们是不可分割的整体，决策变量只有在它们具有整数值时才有意义。在线性规划中，增加决策变量的整数限制，这种我们称为线性整数规划，一般情况下，我们会简称为整数规划。

整数规划有很多求解方法，例如割平面发和分枝定界法。它们都是先将问题转化为线性规划求解，然后增加整数约束进行约化，直到最后找到整数解。而这两种方法都可以求纯或混合整数线性规划问题。我们组采用的是分枝定界的方法来求解整数规划。

在分支定界法中，我们将整数规划转换为线性规划后，是利用单纯形法进行求解。一言概之，我们组的大作业是利用单纯形法 + 分支定界发求解整数规划问题。

整数规划问题是 NP 困难问题，特别的，$0-1$ 规划是整数规划的特殊情况，它的决策变量要么取 0 要么取 1，这是 Karp 的 21 个 NP 完全问题之一。[3]

Chapter 2

算法描述

2.1 单纯形法

单纯形是 N 维中的 $N+1$ 个顶点的凸包，是一个多胞体，譬如是直线上的一个线段，平面上的一个三角形，三维空间中的一个四面体等等，这些都是单纯形。

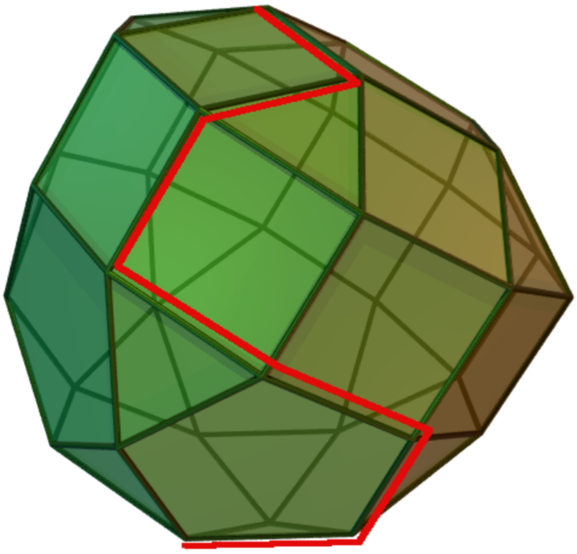

Figure 2.1: 单纯形

2.1.1 标准形式

在使用单纯形法之前，我们需要将线性规划转换为以下标准形式：

$$\begin{aligned} \max &: \sum_{1\le k\le n} c_k x_k \\ s.t. &: Ax \le b \\ & \quad x \ge 0 \end{aligned}$$

所有其他形式的线性规划方程组都可以转化称这个标准形式：

1 目标函数不是极大化：只需要将 c_k 取为原来的相反数，就可以从极小化问题转化为极大化问题。

2 约束条件中存在大于或等于约束：只需要将约束两边同乘 -1。

3 约束条件中存在等式：只需要将其转化为两个不等式，一个为大于等于，另一个为小于等于。

4 有的变量约束为大于等于 R：只需要做简单的仿射变换，用 $x' = x - R$ 代替原来的变量即可。

5 有的变量约束小于等于 0：只需要将与该变量有关的所有系数取相反数即可。

6 有的变量没有非负约束：加入新变量 x'，并用 $x - x'$ 替换原来的变量 x。

通过以上总总，我们就可以将一个一般的线性规划转换为标准形式。

2.1.2 松弛形式

在使用单纯形法进行变换之前，我们需要先计算出一个可行解。我们可以通过将标准形式的线性规划转化为松弛形式，这样能够快速得到线性规划的初始可行解。只需要在原来 n 个变量，m 个约束的线性规划中，加入 m 个新变量，就可以将原来的不等式化为等式：

$$\forall j \in \{1, 2, \ldots, m\}, \sum_{1\le k\le n} a_{j,k} x_k + x_{n+j} = b_j, x_{n+j} \ge 0$$

我们可以首先通过新加入的变量快速得到一组初始可行解：

$$x_{n+j} = b_j - \sum_{1\le k\le n} a_{j,k} x_k$$

我们现在称 $x_1, x_2, \ldots, x_n$ 这些变量为**非基变量**，而称 $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ 这些变量为**基变量**。非基变量能够由基变量唯一确定，也就是课上老师所说的**典则形式**。

我们通过两阶段法求得原标准形式的初始可行解：

1 第一阶段的目标函数为 $\min : \sum\limits_{n+1\le k\le n+m} x_{n+k}$，如果得到该目标函数值为 0，则通过转轴变换将基变量全部转换为原来的变量。如果目标函数值非 0，则表明原规划问题无解。

2 第一阶段结束以后，以第一阶段得到的可行解进行求解原始问题。

单纯形表则是将松弛形式（或者标准形式）的规划问题中的系数放入一个增广矩阵中，通过矩阵变换求得最终的最优解和最优值。

2.1.3 转轴变换

转轴变换是单纯形法中的核心操作，作用就是将一个基变量与一个非基变量进行互换。从几何的理解上就是从单纯形的一个极点走向另一个极点。设变量 x_{n+d} 是基变量，变量 x_e 是非基变量，那么转轴操作 **pivot(d, e)** 以后，x_{n+d} 将变为非基变量，相应的 x_e 变为基变量。将这些转化为用数学符号描述则如下：

$$\begin{array}{rcc}
\text{起初} & : & x_{n+d} = b_d - \sum\limits_{k\in N} a_{d,k}x_k \\
\text{移项} & : & a_{d,e}x_e = b_d - \sum\limits_{k\in N k\neq e} a_{d,k}x_k - x_{n+d} \\
\text{若}a_{d,e}\neq 0 & : & x_e = \dfrac{b_d}{a_{d,e}} - \sum\limits_{k\in N k\neq e} \dfrac{a_{d,k}}{a_{d,e}}x_k - \dfrac{1}{a_{d,e}}x_{n+d}
\end{array}$$

将这个式子代入其他的约束等式以及目标函数中，就实现了 x_{n+d} 和 x_e 的基变量与非基变量的转换。

这在增广矩阵中的操作则对应为第 i 行的基变量变为第 j 个变量，然后利用消元法将其他行中第 j 列的系数消去。我们称这个操作为转轴变换。

2.1.4 最优化过程

而我们挑选哪一个非基变量与基变量进行转轴变换则是最优化过程了，这个过程如下：

- 得到原规划问题的初始可行解（两阶段法）
- 任取一个非基变量 x_e，使得 $c_e > 0$
- 考虑基变量 x_d，$\min\limits_{a_{d,e}>0} \dfrac{b_d}{a_{d,e}}$
- 交换 x_e x_d，即转轴变换 **pivot(d, e)**
- 如果所有的非基变量的系数都是小于等于 0 时，我们已经得到最优解了。将基变量及其增广列对应值作为输出即可。如果只剩 $ce > 0$ 且 $\forall i \in \{1, 2, \ldots, m\}, a_{d,e} \le 0$ 则原规划问题没有有限最有解，目标函数值为正无穷。

2.1.5 Bland 法则

而我们选取非基变量入基的时候，不能够每次都选择检验数最大的入基，这样会导致单纯形法退化，进入搜索循环的 bug。根据 **Bland 法则**，我们可以每次选择下标最小的非基变量入基，就可以避免单纯形法退化。

2.2 分枝定界法

分枝定界法不只是解决整数规划的一种方法，它其实可以认为是一种组合优化问题以及数学优化算法设计的范式。分枝定界法由通过状态空间搜索的候选解决方案的系统枚举组成：候选解决方案集被认为是在根处形成具有全集的根树。该算法探索此树的分支，它代表解决方案集的子集。在枚举分支的候选解之前，针对最优解的**上下估计边界检查分支**，并且如果它不能产生比迄今为止由算法找到的最佳解决方案更好的解，则丢弃该分支（**称为剪枝**）。

在整数规划问题中，我们先将原问题放松成线性规划问题，解这个线性规划，就得到了整数规划最优解的上界。这是因为减少了约束，得到的目标函数值自然更大，所以是上界。然后我们检查最优解，如果最优解中有非整数变量，记为 x_i，$N < x_i < N + 1$，这时候就会有两种可能：$x_i \leq N$ 或者 $x_i \geq N + 1$。这时候我们分枝，一枝增加约束 $x_i \leq N$，另一枝增加约束 $x_i \geq N + 1$。然后递归进行搜索。如果中间过程得到的线性规划最优解也是整数规划最优解，就记其为下界。如果某一枝的上界比下界还小，则将这一枝剪去，称为剪枝，这一枝称为死枝。直到最后找到最优解。中间过程中需要反复降为线性规划以单纯形法进行求解。

这里我们分枝定界法是需要维护两个界的，一个是上界，一个是下界：

- 上界初始化为没有增加约束的原问题的线性规划最优解
  - 更新则在于从一个节点分成两个节点后，取两个节点中线性规划的最优解的最大值。
- 下界初始化为负无穷
  - 更新则在于每次求解出一个线性规划也正好为整数规划且比已知的下界大时，更新下界。
- 如果计算得到的线性规划最优解比已知的下界小，则进行剪枝。
- 如此计算，上界会不断减小，下界会不断提高，直到上界等于下界。

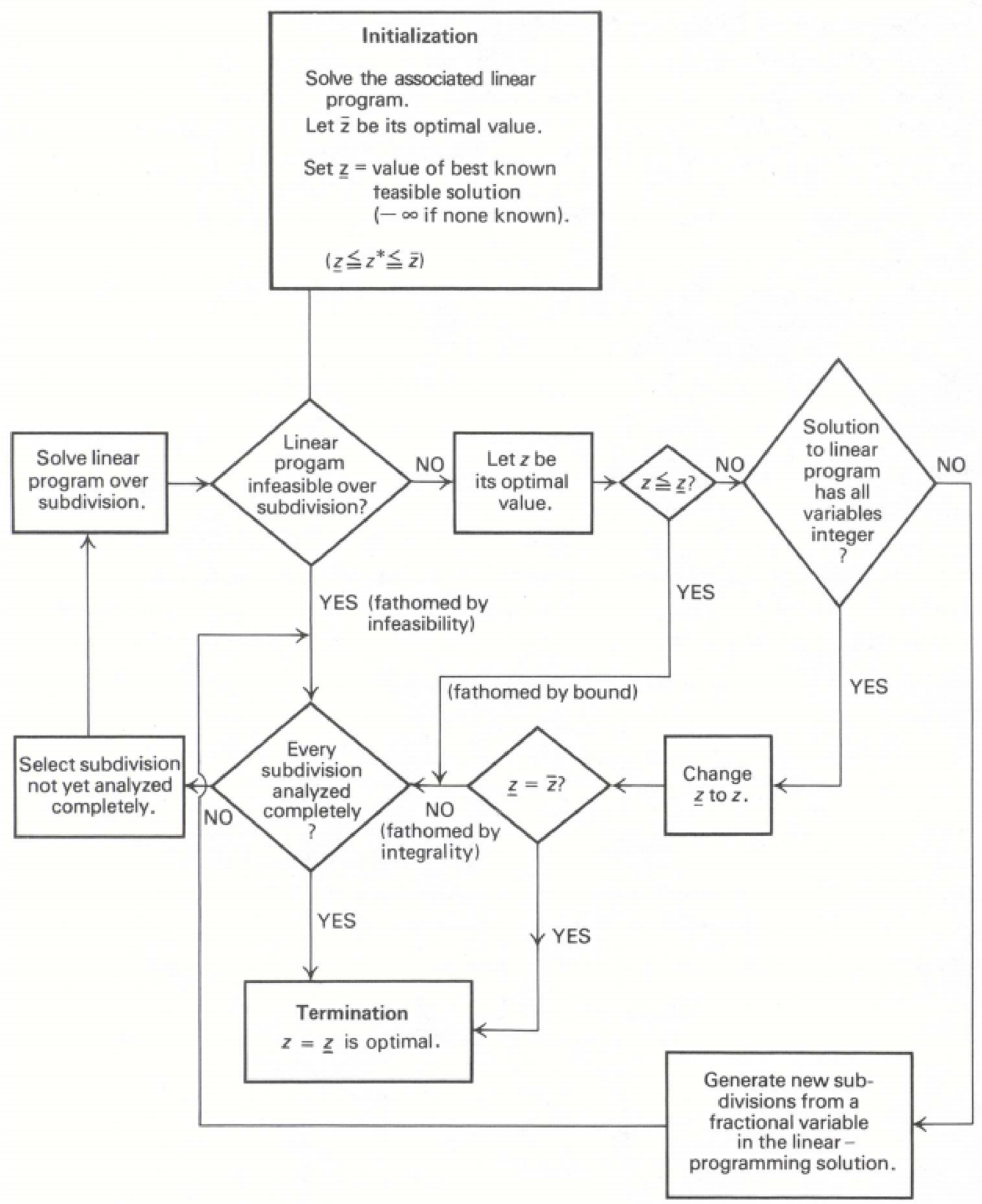
Initialization
Solve the associated linear program.
Let $\bar{z}$ be its optimal value.
Set $\underline{z}$ = value of best known feasible solution ($-\infty$ if none known).
($\underline{z} \leqq z^* \leqq \bar{z}$)
Solve linear program over subdivision.
Linear progam infeasible over subdivision?
NO
Let z be its optimal value.
$z \leqq \underline{z}$?
NO
Solution to linear program has all variables integer ?
NO
YES (fathomed by infeasibility)
YES
(fathomed by bound)
YES
Select subdivision not yet analyzed completely.
Every subdivision analyzed completely ?
NO
$\underline{z} = \bar{z}$?
NO (fathomed by integrality)
Change $\underline{z}$ to z.
YES
YES
Termination
$z = \underline{z}$ is optimal.
Generate new subdivisions from a fractional variable in the linear-programming solution.

Figure 2.2: 分枝定界算法流程图

Chapter 3

实现过程

3.1 Parse

程序接受的输入格式是**.lp** 文件，这种格式并没有办法很容易地给单纯形法作爲输入，因此需要一个转换工具。

首先需要定义一些类表示输入信息

异常，如果发先未知的语句会抛出异常，以及原因。

```
class ParseException
```

变量的上下界

```
class Bounds {
    public:
        // lower <= x && x <= upper
        int upper, lower;
};
```

在约束式中的变量，有变量的索引、🄵数

```
class Variable {
    public:
        int coefficient;
        size_t index;
};
```

约束，可以分 $\leq, \geq, =$ 三种情况，然后包含一个容器装约束中的变量，还有一个常数。左右顺序是: $C_1, C_2, ..., C_N \leq \text{Constant}$

```
class Condition {
    public:
        enum Type { eq, leq, geq };
        Type type;
        std::vector<Variable> variables;
        int constant;
};
```

Data 包含所有的数据:

- 约束: **conditions**
- 变量的上下界: **bounds**
- 变量的索引: **indices**
- 目标函数: **function**

```
class Data {
    private:
        std::vector<Condition> conditions;
        std::vector< std::pair<size_t, Bounds> > bounds;
        std::vector<size_t> indices;
        std::vector<Variable> function;
};
```

虽然输入类似 **C** 的风格，是个上下文无关文法，但是爲了方便就简化为正则语言 (应该不会出现非正则的情况)。

3.1.1 预处理

- 移除注释 $/\backslash\backslash*(.|\backslash n)*?\backslash\backslash*/$
- 移除多馀空白 $\backslash\backslash s*\$$
- 移除换行，用空白取代 $[\backslash n\backslash r]$

预处理之后，整个输入就可以当作一行，然后用; 符号当作真正的换行重新分行，一行一行处理

对于每一行，可以分成几种类型

- **int** 定义变量
- **max**, **min**
- 约束

3.1.2 优化

对于约束如果变量只有 1 个的情况，可以当作该变量的上下界，因爲我们实现的是分支定界法，所以这些信息可以对算法效率有帮助。

值得注意的事情是如果变量的$\boxed{F}$数是负数，大于、小于要交换。

3.1.3 工具函数

```
    std::vector<std::string> Data::Split(const std::string & input, char delim);
    std::string Data::Join(const std::vector<std::string> & input);
```

a;b; c 或 **a, b,\nc,d** 这类代表多个元素合在一个字符串上的形式，因爲比较复杂，需要合并、分割这两种功能来实现分开。

3.2 单纯形法

c++ 的版本使用了 Eigen/Dense 库, 编译运行之前请安装运行求解模块之前要求对输入进行部分预处理, 使得能够求解线性规划. 要给出满足条件的矩阵:C^T 和 $AB = [A|B]$, 使得约束条件可以表示为

$$\begin{aligned} max &: \quad C^T X \\ s.t. &: \quad AX \leq B \end{aligned}$$

在得到这样的输入后, 构造新矩阵 T 如下所示:

$$\begin{bmatrix} -C^T & 0 & 0 \\ A & I & B \end{bmatrix}$$

其中 I 是单位矩阵。

3.2.1 源代码分析

```cpp
bool SimplexSolver::simplexAlgorithm(int64_t variableNum) {
    MatrixXd::Index pivotColumn;
    int64_t pivotRow;

    while (true) {
        /*
            Find pivot column, check for halt condition
        */
        this->tableau.row(0).leftCols(variableNum).minCoeff(&pivotColumn);
        if (this->tableau(0, pivotColumn) >= 0) {
            //Found no negative coefficient
            break;
        }

        /*
            Find pivot row
        */
        pivotRow = this->findPivot_min(pivotColumn);
        if (pivotRow == -1) {
            //no solution
            return false;
        }

        /*
            Do pivot operation
        */
        this->tableau.row(pivotRow) /= this->tableau(pivotRow, pivotColumn);
        this->tableau(pivotRow, pivotColumn) = 1;    // For possible precision issues
        for (int i = 0; i < this->tableau.rows(); i++) {
            if (i == pivotRow) continue;

            this->tableau.row(i) -= this->tableau.row(pivotRow) * this->tableau(i, pivotColumn);
            this->tableau(i, pivotColumn) = 0;    // For possible precision issues
        }
    }

    return true;
}
```

然后对 T 不断进行如下操作:

1 找出 $-C^T$ 系数最小的一列, 设为 **j**.

2 当此列的元素大小 $>= 0$, 说明所有检验数的相反数都非负, 已找到最优解, 结束单纯形法.

3 找矩阵的最小枢轴量, 找不到的话说明线性规划无解, 结束单纯形法.

4 选择该列, 作为加入的新的基变量, 根据此基变量所在列选出基中不再作为基变量的变量对应的那一行, 设为 **i**(一个选择原则是 $\frac{B_{i1}}{A_{ij}} = max_{A_{kj}>0}(\frac{B_{k1}}{A_{kj}})$)

5 以第 **i** 行原有的基换出, 将第 **j** 个变量作为新的基.(消去使得其他行第 **j** 列元素为 0)

6 重复流程 1

原理说明: 在 **A** 所在行, 恰好每行都有一个元素值为 1, 且其他行在这一列的值为 0,(就是单位矩阵中的非 0 元素,) 说明新加入的松弛变量正好构成了一组基.

在理解了单位矩阵的意义之后, 之后的算法流程就是按着单纯形法来的了。

3.2.2 其他函数分析

```
 /**
 * If the given column has only one coefficient with value 1 (except in topmost row), and all other
 * coefficients are zero, then returns the row of the non-zero value.
 * Otherwise return -1.
 * This method is used in the final step of maximization, when we read
 * the solution from the tableau.
 *
 * @param int64_t column
 * @returns int64_t
 */
int64_t SimplexSolver::getPivotRow(int64_t column){}
```

该函数用于找最大检验数所在列, 由于矩阵中存储的是每个检验数的相反数, 因此找最小值就是找最大检验数。

```
 /**
 * If the given column has only one coefficient with value 1 (except in topmost row), and all other
 * coefficients are zero, then returns the row of the non-zero value.
 * Otherwise return -1.
 * This method is used in the final step of maximization, when we read
 * the solution from the tableau.
 *
 * @param int64_t column
 * @returns int64_t
 */
int64_t SimplexSolver::findPivot_min(int64_t column);
```

该函数用于找最小枢轴量, 由于已经确定了哪个元素会入基, 找到一行使得以入基元素消去其他行元素后, 整个矩阵最后边的常数列向量依然保证非负. 显然, 只用找到 $\frac{B_{i,1}}{A_{i,column}} = min_{x>0}\{x|x = \frac{B_{j,1}}{A_{j,column}}\}$ 这样的一行, 再消去其他行就成功让 column 对应的元素入基, 而原本在该行的基就出基了。

3.3 分枝定界法

```
class IPsolver:
def __init__(self, c, Aub, bub, Aeq, beq, bounds, tol=1.0E-8):
    self.c = np.array(c)
    self.Aub = np.array(Aub) if Aub else None
    self.bub = np.array(bub) if bub else None
    self.Aeq = np.array(Aeq) if Aeq else None
    self.beq = np.array(beq) if beq else None
    self.bounds = np.array(bounds) if bounds else None
    self.tol = tol

    self.solution = np.zeros(self.c.shape)
    self.optimum = -np.inf
    self.isFoundSolution = False

    self.cur_sol = np.zeros(self.c.shape)
    self.cur_opt = -np.inf

    self.max_branch_num = 5
    self.cur_branch_num = 0
```

定义如上所示整数规划求解器类，以 c, Aub, bub, Aeq, beq, bounds 和 tol 为输入。默认求解最大化问题，其中 c 表示目标函数中各变量的系数；Aub 为小于等于约束 $Aub * x <= bub$ 中的 Aub 矩阵，bub 为其中右边的系数；Aeq 为等于约束 $Aeq * x = beq$ 中的 Aeq 矩阵，beq 为其中右边的系数；bounds 为各变量的上下界约束；tol 为整数容忍度。初始化中设置最大分支数为 max_branch_num，并初始化当前的解 cur_sol 和目标函数值 cur_opt。

```
def core_solve(self, c, Aub, bub, Aeq, beq, bounds):
    sol, opt = None, None
    res = linprog(-c, A_ub=Aub, b_ub=bub, A_eq=Aeq, b_eq=beq, bounds=bounds)
```

核心求解函数，输入与类构造函数相同，首先将问题松弛为一般的线性规划问题，调用线性规划求解器求解。注意，这里的线性规划求解器是求解最小化问题的，故将 c 反号。

```
if res.success:
    opt = -res.fun
    sol = res.x
    self.update_opt(sol, opt)
```

若该线性规划求解成功，则拿解和值更新当前的解和目标函数值；否则不用分支，直接退出。

```
if self.needBranch(sol, opt) and self.cur_branch_num < self.max_branch_num:
    index = self.getFirstNotInt(sol)
    to_round = sol[index]
    len_c = len(self.c)
```

进一步判断是否需要分支（该函数将在之后详细讲解），并判断 branch 次数是否超过阈值，若需要分支则获得第一个非整数的变量的索引，和解中的对应变量的值，并继续进行下述操作；否则，直接退出。

```
Con1 = np.zeros((len_c, ))
Con2 = np.zeros((len_c, ))
Con1[index] = -1.0
Con2[index] = 1.0
if Aub is None and bub is None:
    A1 = Con1.reshape(1, len_c)
    A2 = Con2.reshape(1, len_c)
    B1 = np.array([-math.ceil(to_round)])
    B2 = np.array([math.floor(to_round)])
else:
    A1 = np.vstack([Aub, Con1])
    A2 = np.vstack([Aub, Con2])
    B1 = np.hstack([bub, -math.ceil(to_round)])
    B2 = np.hstack([bub, math.floor(to_round)])
```

上述代码段将当前非整数索引进行划分,即分别添加 x_index<=-math.ceil(to_round) 和 x_index>=math.floor(to_round) 的约束到原来的小于等于约束中，变成两个不同的分支。

```
self.cur_branch_num += 1
self.core_solve(c, A1, B1, Aeq, beq, bounds) # right branch
self.core_solve(c, A2, B2, Aeq, beq, bounds) # left branch
```

将分好支的变量继续调用核心求解器函数，并将分支次数加 1。因为每次求解时，我们都保存了整数解，故不需要多余的代码来处理之后的结果。当各个分支运行完毕之后求解结束。

```
def allInteger(self, sol):
    tmp = np.array([abs(x-np.round(x)) for x in list(sol)])
    return all(tmp <= self.tol)
```

该函数判断解是否都是整数。

```
def getFirstNotInt(self, sol):
    tmp = np.array([abs(x-np.round(x)) for x in list(sol)])
    l = tmp > self.tol
    for i in range(len(l)):
        if l[i]: return i
    return -1
```

该函数取得解中第一个非整数的索引号。

```
def needBranch(self, sol, opt):
    if self.allInteger(sol): return False
    elif opt <= self.cur_opt: return False
    return True
```

该函数判断当前解和值的情况下，是否需要分支。并非都是整数或者，该解的目标函数值比当前函数值要大，则需要分支。

```
def update_opt(self, sol, opt):
    if self.allInteger(sol) and self.cur_opt<opt:
        self.cur_opt = opt
        self.cur_sol = sol.copy()
        self.isFoundSolution = True
```

该函数用于更新当前解和目标函数值，只有在解都是整数，并且当前函数值比该解的函数值小时，才更新。

```
def solve(self):
    self.core_solve(self.c, self.Aub, self.bub, self.Aeq, self.beq, self.bounds)
    if self.isFoundSolution:
        self.solution = np.array([int(np.round(x)) for x in list(self.cur_sol)])
        self.optimum  = self.cur_opt
        return True
    else:
        return False
```

该函数是对核心求解器函数的一个封装，用于将构造求解器类的各个参数，作为输入传入 core_solve 函数中求解，最后并将结果保存到最后结果 self.solution 和 self.optimum 中。

Chapter 4

测试结果

我们的测试数据除了使用助教在http://www.cs.zju.edu.cn/algo/teaching/2018/zgc_2018.html提供的中等规模及大规模测试外，另外根据需求构造了几组测试样例，我们称之为小型测试样例。以此来检验我们程序的运行情况。下表中详细描述了我们的测试样例和测试情况：

Table 4.1: 小型测试样例

测试样例	测试目的	测试结果
sample1	测试极小化优化问题	pass
sample2	所有约束都是小于等于的极大化问题	pass
sample3	常规例子	pass
sample4	约束包含大于等于，小于等于及等于	pass
sample5	测试无解的样例	pass
sample6	有无穷解的样例	pass
sample7	测试 bland 法则	pass

经过了小型的测试以后，我们对助教提供的例子进行测试。下面是测试结果：

Table 4.2: 大中规模测试

测试样例	测试目的	测试结果
case1	大规模测试 1	pending
case2	大规模测试 2	pending
case3	中等规模测试 1	pass
case4	中等规模测试 2	pass
case5	中等规模测试 3	pass

非常不幸的是，我们的程序无法接受大规模的测试，这可能归咎于我们的优化上没有做好，无法承受三千个变量的压力。这在后面分析的章节中，会稍微详细一点描述这个情况。

Chapter 5

分析与评价

5.1 时间复杂度

5.1.1 单纯形法

如果采用了 **Bland** 法则选择非基变量进行转轴变换, 我们时能够证明单纯形法在有限步内时一定能够终止的。单纯形法在实践中非常有效，并且比 Fourier–Motzkin 消去法 [2] 等早期方法有了很大的改进。然而，在 1972 年，Klee 和 Minty [1] 给出了一个例子，即 Klee-Minty 立方体，表明由 Dantzig 制定的单形方法的最坏情况复杂度是指数时间。

5.1.2 分枝定界法

求解整数规划的精确解是 NP 困难的，我们没有多项式时间复杂度的算法求解。分枝定界法中，我们可能需要遍历所有的枝，所以需要 $O(2^n)$ 次计算线性规划。而我们是使用单纯形法进行计算，所以这里我们的时间复杂度将是 $O(2^n) \times O(2^n) \approx O(2^n)$

5.2 空间复杂度

我们对于约束的存储是比较大开销的，使用的是密集的矩阵存储方式，即并没有使用稀疏矩阵，这在空间中是有极大的浪费，仅存储这个约束矩阵就需要 $O(n^2)$ 的空间了。

单纯形法额外使用的空间除了约束矩阵之外，没有更多的空间开销了。

而对于分枝定界法，则需要生枝。在测试过程中，随着变量数目的增多，分枝定界法会有明显的空间开销，每一枝都有自己的一个矩阵，如果不考虑优化问题，我们需要 $O(2^n) \times O(n^2) \approx O(2^n)$ 的空间。这个空间的开销是非常大的。

5.3 评价

我们实现的实现过程分为了三个模块，一个模块处理 IO，一个模块负责单纯形，一个模块负责分枝定界的演化。而这里，IO 的过程利用了正则匹配，对于将 **lp** 文件转换为我们需要的 **txt** 格式是需要一定的时间的。如果文件比较大，譬如助教提供的 case1 和 case2，这里的 IO 需要的时间就是几秒钟。

不过我们可以只去衡量我们的分枝定界的话，可以只从 **txt** 中进行文件读入。去考量我们的分枝定界法的性能。

不过由于我们的单纯形法和分枝定界法的实现都是基本实现，没有考虑更多的优化，导致运算比较慢。实际上，我们可以将一些已经定下结果的变量筛去，从一定程度上减少 n 这个维度的开销。另外，在分枝定界中，挑去与 $bound$ 比较接近的值进行分枝，这样可能会有一定的优化效果。

Chapter 6

总结

总的来说，我们的分枝定界法求解整数规划能对于中小型的数据进行求解是没有问题的，能够应对各种的约束的变化，具备一定的鲁棒性。但是对于两千个变量以上的大规模例子我们的程序跑起来就比较吃力了。这确实是我们优化的工作没有做足的问题导致的。这次的大作业凝聚了我们组的很多心血，也让我们组对于分枝定界和单纯形有了更加深刻的认识。相信这对于我们今后的学习是会有很大的帮助的。

Appendices

Appendix A

源代码

Listing A.1: C++ main top file

```cpp
#include <cinttypes>
#include <iostream>
#include <time.h>
#include <Eigen/Dense>
#include "Common/lpreader.h"
#include "SimplexSolver/SimplexSolver.h"
#include "SimplexSolver/exception.h"
#include "BranchBound/branch_bound.h"

using namespace std;
using namespace Eigen;

int main(int argc,char* argv[])
{
    if(argc<=2){printf("arg:␣input.txt␣output.txt\n"), exit(-1);}
    string s;
    char c;
    FILE* fp=fopen(argv[1],"r");
    while(!feof(fp)){
        c=fgetc(fp);
        if(c<=0)
            continue;
        s+=c;
    }
    fclose(fp);
    //cin>>s;
    //MatrixXd constraints(equ_m, x_m+1);
    //VectorXd objectiveFunction(x_m);

    try {
        LPReader lpr(s);
        /*
            Maximization problem
        */
        // VectorXd c(7);
        // c << -3, 4, 0, 0, 0, 0, 0;
        // MatrixXd Ab(5,8);
        // Ab<<4,2,-1,0,0,0,0,8,
        //  3,2,0,1,0,0,0,10,
        //  -1,3,0,0,1,0,0,1,
        //  1,0,0,0,0,-1,0,1,
        //  1,0,0,0,0,0,-1,1;

        // SimplexSolver solver1(SIMPLEX_MAXIMIZE, c, Ab);
        // cout<<"c"<<endl; cout<<lpr.c.transpose()<<endl;
        // cout<<"Ab"<<endl; cout<<lpr.Ab<<endl;

        BranchBound bbsolver(SIMPLEX_MAXIMIZE, lpr.c, lpr.Ab, 1E-8);
        clock_t start,end;
        start = clock();
        bbsolver.solve();
        end = clock();
        if(bbsolver.foundSolution){
            printf("result:\n");
            printf("solution:\n");
            cout<<bbsolver.solution.transpose()<<endl;
            printf("optimum:\n");
            cout<<bbsolver.optimum<<endl;
        }else{
            cout<<"failed"<<endl;
        }
        double dur = (double)(end - start);
        printf("used␣%.2e␣seconds\n", (dur/CLOCKS_PER_SEC));

        ofstream outf(argv[2]);
        outf<<bbsolver.solution<<endl;
        outf<<endl;
        outf<<"\noptimum:"<<endl;
        outf<<bbsolver.optimum<<endl;
        outf.close();
        printf("saved␣to␣%s", argv[2]);

        // if (solver1.hasSolution()) {
        //   cout << "The maximum is: " << solver1.getOptimum() << endl;
        //   cout << "The solution is: " << solver1.getSolution().transpose() << endl;
        // } else {
        //   cout << "The linear problem has no solution." << endl;
        // }
    } catch (const FException &ex) {
        ex.Print();
        return 1;
    }

    return 0;
}
```

Listing A.2: python test code

```
import math
from scipy.optimize import linprog
# from simplex import *
import sys
import numpy as np


def lpreader(path):
    f = open(path)
    is_max_problem = int(f.readline())
    avr_num = int(f.readline())
    l = np.array([int(s) for s in f.readline().split()]).reshape(-1, 2)
    c = [0]*avr_num
    for row in l:
        c[row[1]] = row[0]

    bound_num = int(f.readline())
    bound = [[0, None]]*avr_num
    for _ in range(bound_num):
        l = [int(s) for s in f.readline().split()]
        l[1] = None if l[1]==2147483647 or l[1]==-2147483648 else l[1]
        l[2] = None if l[2]==2147483647 or l[2]==-2147483648 else l[2]
        bound[l[0]] = [l[1], l[2]]

    eq_num = int(f.readline())
    A_eq, b_eq = [], []
    for _ in range(eq_num):
        l = [int(s) for s in f.readline().split()]
        A_eq.append(l[0:-1])
        b_eq.append(l[-1])
    if A_eq==[] and b_eq==[]:
        A_eq, b_eq = None, None
```

```python

    ub_num = int(f.readline())
    A_ub, b_ub = [], []
    for _ in range(ub_num):
        l = [int(s) for s in f.readline().split()]
        A_ub.append(l[0:-1])
        b_ub.append(l[-1])
    if A_ub==[] and b_ub == []:
        A_ub, b_ub = None, None

    f.close()
    return is_max_problem, c, A_ub, b_ub, A_eq, b_eq, bound


class IPsolver:
    def __init__(self, is_max_problem, c, Aub, bub, Aeq, beq, bounds, tol=1.0E-8):
        self.is_max_problem = is_max_problem
        self.c = np.array(c)
        self.Aub = np.array(Aub) if Aub else None
        self.bub = np.array(bub) if bub else None
        self.Aeq = np.array(Aeq) if Aeq else None
        self.beq = np.array(beq) if beq else None
        self.bounds = np.array(bounds) if bounds else None
        self.tol = tol

        self.solution = np.zeros(self.c.shape)
        self.optimum = -np.inf
        self.isFoundSolution = False

        self.cur_sol = np.zeros(self.c.shape)
        self.cur_opt = -np.inf

        self.max_branch_num = 5
        self.cur_branch_num = 0

    def allInteger(self, sol):
        tmp = np.array([abs(x-np.round(x)) for x in list(sol)])
        return all(tmp <= self.tol)

    def getFirstNotInt(self, sol):
        tmp = np.array([abs(x-np.round(x)) for x in list(sol)])
        l = tmp > self.tol
        for i in range(len(l)):
            if l[i]: return i
        return -1

    def needBranch(self, sol, opt):
        if self.allInteger(sol): return False
        elif opt <= self.cur_opt: return False
        return True

    def update_opt(self, sol, opt):
        if self.allInteger(sol) and self.cur_opt<opt:
            self.cur_opt = opt
            self.cur_sol = sol.copy()
            self.isFoundSolution = True

    def core_solve(self, c, Aub, bub, Aeq, beq, bounds):
        sol, opt = None, None
        res = linprog(-c, A_ub=Aub, b_ub=bub, A_eq=Aeq, b_eq=beq, bounds=bounds) # for min
        # if Aub is not None:
        #     Aub_ = Aub.tolist()
        #     bub_ = bub.tolist()
        # else:
        #     Aub_, bub_ = [],[]

        # if Aeq is not None:
        #     Aeq_ = Aeq.tolist()
        #     beq_ = beq.tolist()
        # else:
        #     Aeq_, beq_ = [],[]

        # if bounds is not None:
        #     bounds_ = bounds.tolist()
        # else:
        #     bounds_ = []
        # c_ = c.tolist()
        # res = my_simplex_solver(c_, Aub_, bub_, Aeq_, beq_, bounds_)
        if res.success:
            opt = -res.fun # for min
            # opt = res.fun
            sol = res.x
            # print(opt, sol)
            print("current␣branch␣number:", self.cur_branch_num)
            print("cur_opt:", opt)
            # print(self.allInteger(sol))
            self.update_opt(sol, opt)
            if self.needBranch(sol, opt) and self.cur_branch_num < self.max_branch_num:
                index = self.getFirstNotInt(sol)
                # print("index =", index)
                to_round = sol[index]
                len_c = len(self.c)
                Con1 = np.zeros((len_c, ))
                Con2 = np.zeros((len_c, ))
                Con1[index] = -1.0
                Con2[index] = 1.0
                if Aub is None and bub is None:
                    A1 = Con1.reshape(1, len_c)
                    A2 = Con2.reshape(1, len_c)
                    B1 = np.array([-math.ceil(to_round)])
                    B2 = np.array([math.floor(to_round)])
                else:
                    A1 = np.vstack([Aub, Con1])
                    A2 = np.vstack([Aub, Con2])
                    B1 = np.hstack([bub, -math.ceil(to_round)])
                    B2 = np.hstack([bub, math.floor(to_round)])

                self.cur_branch_num += 1
                self.core_solve(c, A1, B1, Aeq, beq, bounds) # right branch
                self.core_solve(c, A2, B2, Aeq, beq, bounds) # left branch

    def solve(self):
        self.core_solve(self.c, self.Aub, self.bub, self.Aeq, self.beq, self.bounds)
        if self.isFoundSolution:
            self.solution = np.array([int(np.round(x)) for x in list(self.cur_sol)])
            if self.is_max_problem: self.optimum  = self.cur_opt
            else: self.optimum  = -self.cur_opt
            return True
        else:
            return False


# def test():
#     c, A_ub, b_ub, A_eq, b_eq, bound = lpreader(sys.argv[1])
#     print(c, A_ub, b_ub, A_eq, b_eq, bound)
#     c = [-x for x in c]
#     res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bound)
#     print(res)


# def test1():
#     c, A_ub, b_ub, A_eq, b_eq, bound = lpreader(sys.argv[1])
#     print(c, A_ub, b_ub, A_eq, b_eq, bound)
#     res = my_simplex_solver(c, A_ub, b_ub, A_eq, b_eq, bound)
#     print(res.x, res.fun, res.success)


def test2():
    if len(sys.argv)<=2:
        print("arg:␣input.txt␣output.txt")
        return
    is_max_problem, c, A_ub, b_ub, A_eq, b_eq, bound = lpreader(sys.argv[1])
    import time
    time_start=time.time()
```

```python
    solver = IPsolver(is_max_problem, c, A_ub, b_ub, A_eq, b_eq, bound)
    solver.solve()
    time_end=time.time()
    print("result:")
    print("solution:")
    print(solver.solution)
    print("optimum:")
    print(solver.optimum)
    print('time␣cost',time_end-time_start,'s')
    np.savetxt(sys.argv[2], solver.solution, fmt='%d')
    f = open(sys.argv[2], "a")
    f.write('\noptimum:'+'\n')
    f.write(str(int(np.round(solver.optimum)))+'\n')
    f.close()
    print("saved␣to", sys.argv[2])


if __name__=="__main__":
    # test()
    # test1()
    test2()
```

Listing A.3: Header File for lpreader

```cpp
#include "sparse.h"
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
#include <Eigen/Dense>

#define pb push_back
using namespace std;
using namespace Eigen;

class LPReader{
public:
    VectorXd c;
    MatrixXd Ab;
    Data d;

    //void transgte(vector<int>,int,int &,vector<vector<int> >&,vector<int>&);
    //void transequ(vector<int>,int,int &,vector<vector<int> >&,vector<int>&);
    //void translte(vector<int>,int,int &,vector<vector<int> >&,vector<int>&);
    void transgte(vector<int>equ,int b,int &freex,vector<vector<int> >&A,vector<int>&B){
        if(b>0){
            equ.pb(-1);
            freex++;
            transequ(equ,b,freex,A,B);
            return ;
        }
        for(int i=0;i<equ.size();i++)
            equ[i]=-equ[i];
        translte(equ,-b,freex,A,B);
        return ;
    }
    void transequ(vector<int>equ,int b,int &freex,vector<vector<int> >&A,vector<int>&B){
        if(b>=0){
            equ.pb(1);
            freex++;
            translte(equ,b,freex,A,B);
            equ[equ.size()-1]=-1;
            translte(equ,b,freex,A,B);
            return ;
        }
        for(int i=0;i<equ.size();i++)
            equ[i]=-equ[i];
        transequ(equ,-b,freex,A,B);
        return ;
    }
    void translte(vector<int>equ,int b,int &freex,vector<vector<int> >&A,vector<int>&B
        ){
        int i,j;
        if(b>=0){
            for(i=0;i<A.size();i++)
                for(j=A[i].size();j<equ.size();j++)
                    A[i].pb(0);
            A.pb(equ);
            B.pb(b);
            return ;
        }
        for(i=0;i<equ.size();i++)
            equ[i]=-equ[i];
        transgte(equ,-b,freex,A,B);
    }
    void print(vector<vector<int> >&A,vector<int>&B){
        int i,j;
        for(i=0;i<A.size();i++){
            for(j=0;j<A[i].size();j++){
                printf("%3d␣",A[i][j]);
            }
            printf("%3d\n",B[i]);
        }
        return ;
    }
    LPReader(const string& file){
        puts("succeed␣in␣reading");
        d.Parse(file);
        unsigned int i,j;
        int freex=d.indices.size();

        vector<int>tempv;
        vector<int>B;
        vector<vector<int> >A;/*
        for(i=0;i<var_num;i++){
            c(d.function[i].index)=d.function[i].coefficient;
        }*/
        for(i=0;i<d.bounds.size();i++){
            if(d.bounds[i].second.lower>0){
                tempv.resize(freex);
                for(j=0;j<freex;j++)
                    tempv[j]=0;
                tempv[d.bounds[i].first-1]=1;//l<=x  x>=l
                transgte(tempv,d.bounds[i].second.lower,freex,A,B);
            }
            if(d.bounds[i].second.upper<INT_MAX){
                tempv.resize(freex);
                for(j=0;j<freex;j++)
                    tempv[j]=0;
                tempv[d.bounds[i].first-1]=1;//x<=r
                translte(tempv,d.bounds[i].second.upper,freex,A,B);
            }
            //print(A,B);
        }
        for(i=0;i<d.conditions.size();i++){
            tempv.resize(freex);
            for(j=0;j<freex;j++)
                tempv[j]=0;
            for(j=0;j<d.conditions[i].variables.size();j++)
                tempv[d.conditions[i].variables[j].index-1]=d.conditions[i].variables[
                    j].coefficient;
            switch(d.conditions[i].type){
                case Condition::Type::eq:transequ(tempv,d.conditions[i].constant,freex
                    ,A,B);break;
                case Condition::Type::leq:translte(tempv,d.conditions[i].constant,
                    freex,A,B);break;
                case Condition::Type::geq:transgte(tempv,d.conditions[i].constant,
                    freex,A,B);break;
            }
        }
        c = VectorXd::Zero(freex);
        for(i=0;i<d.function.size();i++)
            c(d.function[i].index-1)=d.function[i].coefficient;
        //for(i=0;i<freex;i++)
        //    printf("%02d ",c(i));
        //putchar('\n');
        assert(A.size());
        for(i=1;i<A.size();i++)
            assert(A[i-1].size()==A[i].size());
        Ab.resize(A.size(),A[0].size()+1);
        for(i=0;i<A.size();i++){
            for(j=0;j<A[i].size();j++){
                Ab(i,j)=A[i][j];
                //printf("%02d ",A[i][j]);
            }
            Ab(i,j)=B[i];
            //printf("%02d\n",B[i]);
        }
        puts("succeed␣in␣constructing");
        return ;
    }

};
```

Listing A.4: Header File for sparse

```
#include <fstream>
#include <sstream>
#include <vector>
#include <regex>
#include <cctype>
#include <string>
#include <algorithm>
#include <exception>
#include <map>
#include <cmath>


class ParseException : public std::exception {
protected:
    std::string msg_;
public:
    ParseException(std::string message) {
        this->msg_ = message;

    }
    virtual const char* what() const throw () {
        return msg_.c_str();
    }
};

class Bounds {
public:
    // lower <= x && x <= upper
    int upper, lower;
    Bounds()
    {
        this->upper = std::numeric_limits<int>::max();
        this->lower = std::numeric_limits<int>::min();
    }

    bool operator < (const Bounds & b) const {
        return false;
    }
};

class Variable {
public:
    int coefficient;
    size_t index;
    Variable(int coefficient, size_t index) :
        coefficient(coefficient), index(index)
    {
    }

    bool operator < (const Variable &b) const {
        return index < b.index;
    }
};

class Condition {
public:
    enum Type { eq, leq, geq };
    Type type;
    std::vector<Variable> variables;
    int constant;
    Condition(Type type, const std::vector<Variable> & variables, int constant):
        type(type), variables(variables), constant(constant)
    {
    }
};

class Data {
public:
    std::vector<Condition> conditions;
    std::vector< std::pair<size_t, Bounds> > bounds;
    std::vector<size_t> indices;
    std::vector<Variable> function;

    //static std::string Trim(const std::string &s);
    static std::vector<std::string> Split(const std::string & input, char delim);
    static std::string Join(const std::vector<std::string> & input);
    static std::vector<Variable> ParseVariables(const std::vector<std::string> & tokens, bool opposite = false);
    static size_t ParseVariable(std::string variable);
    static Condition ParseExpression(const std::vector<std::string> & tokens);
public:
    void Parse(const std::string & input);
    std::string Print();
};

//std::string Data::Trim(const std::string &s)
//{
//  auto wsfront = std::find_if_not(s.begin(), s.end(), [](int c) {return std::isspace(c); });
//  auto wsback = std::find_if_not(s.rbegin(), s.rend(), [](int c) {return std::isspace(c); }).base();
//  return (wsback <= wsfront ? std::string() : std::string(wsfront, wsback));
//}

std::vector<std::string> Data::Split(const std::string & input, char delim)
{
    std::vector<std::string> result;
    std::stringstream buffer(input);
    for (std::string line; std::getline(buffer, line, delim); ) {
        if (!line.empty()) {
            result.push_back(line);
        }
    }
    return result;
}


std::string Data::Join(const std::vector<std::string> & input)
{
    std::stringstream buffer;
    for (const std::string &s : input) {
        buffer << s;
    }
    return buffer.str();
}


std::vector<Variable> Data::ParseVariables(const std::vector<std::string> & tokens, bool opposite)
```

```
{
    using std::vector;
    using std::string;
    using std::regex;

    vector<Variable> variables;

    for (const string & token : tokens) {

        bool parseFail = false;
        int sign, coefficient, index;

        std::smatch sm;
        bool result = std::regex_search(token, sm, regex("([\\+\\-]?)([0-9]*)C([0-9]+)"));
        if (result && sm.size() == 4) {
            // sign
            if (sm[1] == "+" || sm[1] == "") {
                sign = 1;
            } else if (sm[1] == "-") {
                sign = -1;
            } else {
                parseFail = true;
            }

            // check whether opposite
            if (opposite){
                sign *= -1;
            }

            // coefficient
            try {
                if (sm[2] == "") {
                    coefficient = 1;
                } else {
                    coefficient = std::stoi(sm[2]);
                }
            } catch (std::exception e) {
                parseFail = true;
            }

            // index
            try {
                index = std::stoi(sm[3]);
            } catch (std::exception e) {
                parseFail = true;
            }
        } else {
            parseFail = true;
        }

        if (parseFail) {
            throw new ParseException("'" + token + "'␣is␣not␣a␣valid␣variable");
        }

        variables.push_back(Variable(sign * coefficient, index));
    }

    return variables;
}


Condition Data::ParseExpression(const std::vector<std::string> & tokens_)
{
    using std::vector;
    using std::string;

    string tokensString = Data::Join(tokens_);
    tokensString = std::regex_replace(tokensString, std::regex("([\\+\\-])"), "␣$1");
    tokensString = std::regex_replace(tokensString, std::regex("(\\<\\=|\\>\\=|\\=)"), "␣$1␣");
    vector<string> tokens = Data::Split(tokensString, '␣');

```

```cpp
    if (tokens.size() < 3) {
        throw new ParseException("'" + Data::Join(tokens) + "'␣is␣not␣a␣valid␣expression");
    }

    Condition::Type conditionType;
    vector<Variable> variables;
    int constant;

    const string & operatorString = tokens[tokens.size() - 2];
    const string & constantString = tokens[tokens.size() - 1];

    // variables
    variables = Data::ParseVariables(vector<string>(tokens.begin(), tokens.begin() + tokens.size() - 2));

    // =, <=, >=
    if (operatorString == "=") {
        conditionType = Condition::Type::eq;
    } else if (operatorString == "<=") {
        conditionType = Condition::Type::leq;
    } else if (operatorString == ">=") {
        conditionType = Condition::Type::geq;
    } else {
        throw new ParseException("'" + operatorString + "'␣is␣not␣a␣valid␣operator");
    }

    // constant
    try {
        constant = std::stoi(constantString);
    } catch (std::exception e) {
        throw new ParseException("'" + constantString + "'␣is␣not␣a␣valid␣integer");
    }

    std::sort(variables.begin(), variables.end());
    return Condition(conditionType, variables, constant);
}

size_t Data::ParseVariable(std::string variable)
{
    using std::string;
    using std::regex;

    std::smatch sm;
    bool result = std::regex_search(variable, sm, regex("C([0-9]+)"));
    bool parseFail = false;

    int index;

    if (result && sm.size() == 2) {
        try {
            index = std::stoi(sm[1]);
        }
        catch (std::exception e) {
            parseFail = true;
        }
    }
    else {
        parseFail = true;
    }

    if (parseFail) {
        throw new ParseException("'" + variable + "'␣is␣not␣a␣valid␣variable");
    }

    return index;
}

void Data::Parse(const std::string & input_)
{
    using std::vector;
    using std::string;
    using std::regex;
    using std::stringstream;

    string input = input_;
    //std::cout<<input<<std::endl;
    input = std::regex_replace(input, regex("/\\*(.|\n)*?\\*/"), ""); // remove
        comment
    input = std::regex_replace(input, regex("^\\s*$"), ""); // remove blank
    input = std::regex_replace(input, regex("[\n\r]"), "␣"); // remove line
    //std::cout<<input<<std::endl;
    stringstream buffer(input);
    for (string line; std::getline(buffer, line, ';'); ) {
        vector<string> tokens = Data::Split(line, '␣');
        if (!tokens.empty()) {
            if (tokens[0].size() >= 3 && tokens[0].substr(0, 3) == "int") {
                tokens.erase(tokens.begin());
                vector<string> vars = Data::Split(Data::Join(tokens), ',');
                for (const string & var : vars) {
                    int index = ParseVariable(var);
                    this->indices.push_back(index);
                }
            } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) == "max:") {
                function = Data::ParseVariables(vector<string>(tokens.begin() + 1,
                    tokens.end()));
            } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) == "min:") {
                function = Data::ParseVariables(vector<string>(tokens.begin() + 1,
                    tokens.end()), true);
            } else {
                if (tokens[0].back() == ':') {
                    tokens.erase(tokens.begin());
                }
                Condition condition = ParseExpression(tokens);
                if (condition.variables.size() == 1) {
                    Variable variable = condition.variables.front();
                    Bounds bounds;

                    /** coefficient * variable [= // <= // >=] constant
                     * if operator is =
                     *      upper = lower = constant / coefficient
                     * if operator is >=
                     *      if sign is +
                     *          upper = +infinity
                     *          lower = ceil(constant / coefficient)
                     *      if sign is -
                     *          upper = floor(constant / coefficient)
                     *          lower = -infinity
                     * if operator is <=
                     *      if sign is +
                     *          upper = floor(constant / coefficient)
                     *          lower = -infinity
                     *      if sign is -
                     *          lower = ceil(constant / coefficient)
                     *          upper = +infinity
                     */
                    int ceil_ = ceil(condition.constant*1.0 / variable.coefficient);
                    int floor_ = floor(condition.constant*1.0 / variable.coefficient);

                    switch (condition.type){
                        case Condition::Type::eq:
                            if (ceil_ == floor_) {
                                bounds.upper = bounds.lower = ceil_;
                            } else {
                                // if ceil_ is not equal to floor_
                                // the variable should be unsolvable.
                                // simply let upper < lower
                                bounds.upper = floor_;
                                bounds.lower = ceil_;
                            }
                            break;

                        case Condition::Type::leq:
                            if (variable.coefficient < 0) {
                                bounds.lower = ceil_;
                            } else {
                                bounds.upper = floor_;
                            }
                            break;

                        case Condition::Type::geq:
                            if (variable.coefficient < 0) {
                                bounds.upper = floor_;
                            } else {
                                bounds.lower = ceil_;
                            }
                            break;

                        default:
                            break;
                    }

                    this->bounds.push_back(std::pair<size_t, Bounds>(variable.index,
                        bounds));
                } else {
                    this->conditions.push_back(condition);
                }
            }
        }
    }

    std::sort(this->indices.begin(), this->indices.end());
    std::sort(this->bounds.begin(), this->bounds.end());
}


std::string Data::Print()
{
    using std::stringstream;
    using std::vector;

    stringstream output;

    vector<vector<int> > eq;
    vector<vector<int> > leq;

    // give a new index
    const size_t indexSize = this->indices.size();
    std::map<size_t, size_t> mapIndex;
    {
        int count = 0;
        for (size_t index : this->indices) {
            mapIndex[index] = count++;
        }
    }

    for (const Condition & condition : this->conditions) {
        vector<int> vec;
        vec.resize(indexSize, 0);

        for (const Variable & variable : condition.variables) {
            size_t index = mapIndex[variable.index];
            int coe = variable.coefficient;
            vec[index] = coe;
        }
        vec.push_back(condition.constant);

        switch (condition.type) {
            case Condition::Type::eq:
                eq.push_back(vec);
                break;

            case Condition::Type::leq:
                leq.push_back(vec);
                break;

            case Condition::Type::geq:
                for (int &e : vec) {
                    e *= -1;
                }
                leq.push_back(vec);
                break;

            default:
                break;
        }
    }

    output << indexSize << std::endl;

    for (const auto & var : function){
        output << var.coefficient << "␣" << var.index << "␣";
    }
    output << std::endl;

    output << this->bounds.size() << std::endl;
    for (const auto p : this->bounds) {
        size_t index = p.first;
        const Bounds & bounds = p.second;
        output << index << "␣" << bounds.lower << "␣" << bounds.upper << std::endl;
    }

    output << eq.size() << std::endl;
    for (const auto & vec : eq) {
        for (const int e : vec) {
            output << e << "␣";
        }
        output << std::endl;
    }

    output << leq.size() << std::endl;
    for (const auto & vec : leq) {
        for (const int e : vec) {
            output << e << "␣";
        }
        output << std::endl;
    }

    return output.str();
}
```

Listing A.5: Implementation File for parsing

```
#include <fstream>
#include <sstream>
#include <vector>
#include <regex>
#include <cctype>
#include <string>
#include <algorithm>
#include <exception>
#include <map>
#include <cmath>


class ParseException : public std::exception {
protected:
    std::string msg_;
public:
    ParseException(std::string message) {
        this->msg_ = message;

    }
    virtual const char* what() const throw () {
        return msg_.c_str();
    }
};

class Bounds {
public:
    // lower <= x && x <= upper
    int upper, lower;
    Bounds()
    {
        this->upper = std::numeric_limits<int>::max();
        this->lower = 0;
    }

    bool operator < (const Bounds & b) const {
        return false;
    }
};

class Variable {
public:
    int coefficient;
    size_t index;
    Variable(int coefficient, size_t index) :
        coefficient(coefficient), index(index)
    {
    }

    bool operator < (const Variable &b) const {
        return index < b.index;
    }
};

class Condition {
public:
    enum Type { eq, leq, geq };
    Type type;
    std::vector<Variable> variables;
    int constant;
    Condition(Type type, const std::vector<Variable> & variables, int constant):
        type(type), variables(variables), constant(constant)
    {
    }
};

class Data {
private:
    std::vector<Condition> conditions;
    std::vector< std::pair<size_t, Bounds> > bounds;
    std::vector<size_t> indices;
    std::vector<Variable> function;
    bool isMaxProblem;

    //static std::string Trim(const std::string &s);
    static std::vector<std::string> Split(const std::string & input, char delim);
    static std::string Join(const std::vector<std::string> & input);
    static std::vector<Variable> ParseVariables(const std::vector<std::string> & tokens, bool opposite = false);
    static size_t ParseVariable(std::string variable);
    static Condition ParseExpression(const std::vector<std::string> & tokens);
public:
    void Parse(const std::string & input);
    std::string Print();
};

//std::string Data::Trim(const std::string &s)
//{
//  auto wsfront = std::find_if_not(s.begin(), s.end(), [](int c) {return std::isspace(c); });
//  auto wsback = std::find_if_not(s.rbegin(), s.rend(), [](int c) {return std::isspace(c); }).base();
//  return (wsback <= wsfront ? std::string() : std::string(wsfront, wsback));
//}

std::vector<std::string> Data::Split(const std::string & input, char delim)
{
    std::vector<std::string> result;
    std::stringstream buffer(input);
    for (std::string line; std::getline(buffer, line, delim); ) {
        if (!line.empty()) {
            result.push_back(line);
        }
    }
    return result;
}


std::string Data::Join(const std::vector<std::string> & input)
{
    std::stringstream buffer;
    for (const std::string &s : input) {
        buffer << s;
    }
    return buffer.str();
}


std::vector<Variable> Data::ParseVariables(const std::vector<std::string> & tokens, bool opposite)
{
    using std::vector;
    using std::string;
    using std::regex;

    vector<Variable> variables;

    for (const string & token : tokens) {

        bool parseFail = false;
        int sign, coefficient, index;

        std::smatch sm;
        bool result = std::regex_search(token, sm, regex("([\\+\\-]?)([0-9]*)C([0-9]+)"));
        if (result && sm.size() == 4) {
            // sign
            if (sm[1] == "+" || sm[1] == "") {
                sign = 1;
            } else if (sm[1] == "-") {
                sign = -1;
            } else {
                parseFail = true;
            }

            // check whether opposite
            if (opposite){
                sign *= -1;
            }

            // coefficient
            try {
                if (sm[2] == "") {
                    coefficient = 1;
                } else {
                    coefficient = std::stoi(sm[2]);
                }
            } catch (std::exception e) {
                parseFail = true;
            }

            // index
            try {
                index = std::stoi(sm[3]);
            } catch (std::exception e) {
                parseFail = true;
            }
        } else {
            parseFail = true;
        }

        if (parseFail) {
            throw new ParseException("'" + token + "'␣is␣not␣a␣valid␣variable");
        }

        variables.push_back(Variable(sign * coefficient, index));
    }

    return variables;
}


Condition Data::ParseExpression(const std::vector<std::string> & tokens_)
{
    using std::vector;
    using std::string;

    string tokensString = Data::Join(tokens_);
    tokensString = std::regex_replace(tokensString, std::regex("([\\+\\-])"), "␣$1");
    tokensString = std::regex_replace(tokensString, std::regex("(\\<\\=|\\>\\=|\\=)"), "␣$1␣");
    vector<string> tokens = Data::Split(tokensString, '␣');

    if (tokens.size() < 3) {
        throw new ParseException("'" + Data::Join(tokens) + "'␣is␣not␣a␣valid␣expression");
    }

    Condition::Type conditionType;
    vector<Variable> variables;
    int constant;

    const string & operatorString = tokens[tokens.size() - 2];
    const string & constantString = tokens[tokens.size() - 1];

    // variables
    variables = Data::ParseVariables(vector<string>(tokens.begin(), tokens.begin() + tokens.size() - 2));

    // =, <=, >=
    if (operatorString == "=") {
        conditionType = Condition::Type::eq;
    } else if (operatorString == "<=") {
        conditionType = Condition::Type::leq;
    } else if (operatorString == ">=") {
        conditionType = Condition::Type::geq;
    } else {
        throw new ParseException("'" + operatorString + "'␣is␣not␣a␣valid␣operator");
    }

    // constant
    try {
        constant = std::stoi(constantString);
    } catch (std::exception e) {
        throw new ParseException("'" + constantString + "'␣is␣not␣a␣valid␣integer");
    }

    std::sort(variables.begin(), variables.end());
    return Condition(conditionType, variables, constant);
}

size_t Data::ParseVariable(std::string variable)
{
    using std::string;
    using std::regex;

    std::smatch sm;
    bool result = std::regex_search(variable, sm, regex("C([0-9]+)"));
    bool parseFail = false;

    int index;

    if (result && sm.size() == 2) {
        try {
            index = std::stoi(sm[1]);
        }
        catch (std::exception e) {
            parseFail = true;
        }
    }
    else {
        parseFail = true;
    }

    if (parseFail) {
        throw new ParseException("'" + variable + "'␣is␣not␣a␣valid␣variable");
    }

    return index;
}

void Data::Parse(const std::string & input_)
{
    using std::vector;
    using std::string;
    using std::regex;
    using std::stringstream;

    string input = input_;
    input = std::regex_replace(input, regex("/\\*(.|\n)*?\\*/"), ""); // remove comment
    input = std::regex_replace(input, regex("^\\s*$"), ""); // remove blank
    input = std::regex_replace(input, regex("[\n\r]"), "␣"); // remove line

    stringstream buffer(input);
    for (string line; std::getline(buffer, line, ';'); ) {
        vector<string> tokens = Data::Split(line, '␣');
        if (!tokens.empty()) {
            if (tokens[0].size() >= 3 && tokens[0].substr(0, 3) == "int") {
                tokens.erase(tokens.begin());
                vector<string> vars = Data::Split(Data::Join(tokens), ',');
                for (const string & var : vars) {
                    int index = ParseVariable(var);
                    this->indices.push_back(index);
                }
            } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) == "max:") {
                isMaxProblem = true;
                function = Data::ParseVariables(vector<string>(tokens.begin() + 1, tokens.end()));
            } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) == "min:") {
                isMaxProblem = false;
                function = Data::ParseVariables(vector<string>(tokens.begin() + 1, tokens.end()), true);
            } else {
                if (tokens[0].back() == ':') {
                    tokens.erase(tokens.begin());
                }
                Condition condition = ParseExpression(tokens);
                if (condition.variables.size() == 1) {
                    Variable variable = condition.variables.front();
                    Bounds bounds;

                    /** coefficient * variable [= // <= // >=] constant
                     * if operator is =
                     *      upper = lower = constant / coefficient
                     * if operator is >=
                     *      if sign is +
                     *          upper = +infinity
                     *          lower = ceil(constant / coefficient)
                     *      if sign is -
                     *          upper = floor(constant / coefficient)
                     *          lower = -infinity
                     * if operator is <=
                     *      if sign is +
                     *          upper = floor(constant / coefficient)
                     *          lower = -infinity
                     *      if sign is -
                     *          lower = ceil(constant / coefficient)
                     *          upper = +infinity
                     */
                    int ceil_ = ceil(condition.constant*1.0 / variable.coefficient);
                    int floor_ = floor(condition.constant*1.0 / variable.coefficient);

                    switch (condition.type){
                        case Condition::Type::eq:
                            if (ceil_ == floor_) {
                                bounds.upper = bounds.lower = ceil_;
                            } else {
                                // if ceil_ is not equal to floor_
                                // the variable should be unsolvable.
                                // simply let upper < lower
                                bounds.upper = floor_;
                                bounds.lower = ceil_;
                            }
                            break;

                        case Condition::Type::leq:
                            if (variable.coefficient < 0) {
                                bounds.lower = ceil_;
                            } else {
                                bounds.upper = floor_;
                            }
                            break;

                        case Condition::Type::geq:
                            if (variable.coefficient < 0) {
                                bounds.upper = floor_;
                            } else {
                                bounds.lower = ceil_;
                            }
                            break;

                        default:
                            break;
                    }

                    this->bounds.push_back(std::pair<size_t, Bounds>(variable.index,
                        bounds));
                } else {
                    this->conditions.push_back(condition);
                }
            }
        }
    }

    std::sort(this->indices.begin(), this->indices.end());
    std::sort(this->bounds.begin(), this->bounds.end());
}


std::string Data::Print()
{
    using std::stringstream;
    using std::vector;

    stringstream output;

    vector<vector<int> > eq;
    vector<vector<int> > leq;

    // give a new index
    const size_t indexSize = this->indices.size();
    std::map<size_t, size_t> mapIndex;
    {
        int count = 0;
        for (size_t index : this->indices) {
            mapIndex[index] = count++;
        }
    }

    for (const Condition & condition : this->conditions) {
        vector<int> vec;
        vec.resize(indexSize, 0);

        for (const Variable & variable : condition.variables) {
            size_t index = mapIndex[variable.index];
            int coe = variable.coefficient;
            vec[index] = coe;
        }
        vec.push_back(condition.constant);

        switch (condition.type) {
            case Condition::Type::eq:
                eq.push_back(vec);
                break;

            case Condition::Type::leq:
                leq.push_back(vec);
                break;

            case Condition::Type::geq:
                for (int &e : vec) {
                    e *= -1;
                }
                leq.push_back(vec);
                break;

            default:
                break;
        }
    }

    output << isMaxProblem << std::endl;
    output << indexSize << std::endl;

    for (const auto & var : function){
        output << var.coefficient << "␣" << mapIndex[var.index] << "␣";
    }
    output << std::endl;

    output << this->bounds.size() << std::endl;
    for (const auto p : this->bounds) {
        size_t index = p.first;
        const Bounds & bounds = p.second;
        output << mapIndex[index] << "␣" << bounds.lower << "␣" << bounds.upper << std::endl;
    }

    output << eq.size() << std::endl;
    for (const auto & vec : eq) {
        for (const int e : vec) {
            output << e << "␣";
        }
        output << std::endl;
    }

    output << leq.size() << std::endl;
    for (const auto & vec : leq) {
        for (const int e : vec) {
            output << e << "␣";
        }
        output << std::endl;
    }

    return output.str();
}


int main(int argc, char *argv[]) {
    Data *data = new Data;
    std::fstream fin;
    //fin.open("case3.lp", std::fstream::in);

    std::string file_name;
    if (argc == 2){
        file_name = argv[1];
    }
    else{
        printf("argc␣error:␣input.lp\n");
        return -1;
    }

    std::ifstream t(file_name);
    t.seekg(0, std::ios::end);
    size_t size = t.tellg();
    std::string buffer(size, '␣');
    t.seekg(0);
    t.read(&buffer[0], size);
    fin.close();

    try {
        data->Parse(buffer);
    }
    catch (ParseException e) {
        printf("%s\n", e.what());
    }

    std::fstream fout;
    fout.open(file_name.substr(0, file_name.find_last_of(".") + 1) + "txt", std::
        fstream::out);
    std::string result = data->Print();
    fout << result;
    fout.close();

    // check if there is a variable not been declared
    return 0;
}
```

Listing A.6: Header File for Branch Bound

```
#include <Eigen/Dense>

#define INF 1E100

using namespace Eigen;
using namespace std;

class BranchBound{
public:
    VectorXd solution;
    bool foundSolution;
    double optimum;
    int64_t numberOfVariables;

    BranchBound(int mode, const VectorXd &objectiveFunction,
                          const MatrixXd &constraints, double tol=1E-8)
    : mode(mode), c(objectiveFunction), Ab(constraints), tol(tol) {
        numberOfVariables = objectiveFunction.rows();
        current_opt = optimum = -INF;
        solution.resize(numberOfVariables);
        current_solution.resize(numberOfVariables);
        foundSolution = false;
    }

    bool solve();

private:
    int mode;
    VectorXd c;
    MatrixXd Ab;
    double current_opt;
    VectorXd current_solution;
    double tol;

    bool allInteger(const VectorXd &solution){
        for(int64_t i=0;i<numberOfVariables;i++){
            if(abs(solution(i) - int(solution(i))) > tol)return false;
        }
        return true;
    }

    int64_t getFirstNotInt(const VectorXd &solution){
        for(int64_t i=0;i<numberOfVariables;i++){
            if(abs(solution(i) - int(solution(i))) > tol)return i;
        }
        return -1;
    }

    bool needBranch(const VectorXd &solution, double value){
        bool ai = allInteger(solution);
        if(ai)return false;
        if(value <= current_opt)return false;
        return true;
    }

    void update_opt(const VectorXd &solution, double value){
        bool ai = allInteger(solution);
        if(ai && current_opt < value){
            current_opt = value;
            current_solution = solution;
        }
        foundSolution = true;
    }

    void branch(int64_t index,
                double to_round,
                const MatrixXd& Ab,
                MatrixXd& new_Ab,
                bool is_left)
    {
        new_Ab.resizeLike(Ab);
        new_Ab.conservativeResize(Ab.rows()+1, Ab.cols());

        VectorXd to_append;
        to_append.resize(Ab.cols());
        for(int64_t i=0;i<Ab.cols();i++){to_append(i) = 0;}
        to_append(index) = is_left ? 1 : -1;
        to_append(Ab.cols()-1) = is_left ? int(to_round) : (int(to_round) + 1);

        new_Ab.row(Ab.rows()) = to_append;
    }

    void core_solve(const VectorXd& c, const MatrixXd& Ab, VectorXd& sol, double& opt)
        ;
};
```

Listing A.7: Implementation File for Branch Bound

```
#include <iostream>
#include "branch_bound.h"
#include "../SimplexSolver/SimplexSolver.h"
using namespace Eigen;
using namespace std;

bool BranchBound::solve()
{
    VectorXd tmp_c(c);
    VectorXd sol;
    double   opt;
    if(mode==SIMPLEX_MINIMIZE){
        tmp_c = c * -1.0;
    }

    core_solve(tmp_c, Ab, sol, opt);

    if(foundSolution){
        optimum  = (mode==SIMPLEX_MINIMIZE) ? -current_opt : current_opt;
        solution = current_solution;
        return true;
    }
    else return false;
}

void BranchBound::core_solve(const VectorXd& c,
                             const MatrixXd& Ab,
                             VectorXd& sol,
                             double& opt)
{
    SimplexSolver ssolver(SIMPLEX_MAXIMIZE, c, Ab);
    if(ssolver.hasSolution()){
        opt = ssolver.getOptimum();
        cout<<"DEBUG␣:␣current␣opt␣=␣"<<opt<<endl;
        sol = ssolver.getSolution();
        update_opt(sol, opt);
        if(needBranch(sol, opt)){
            int64_t index = getFirstNotInt(sol);
            double to_round = sol(index);
            MatrixXd left_Ab,  right_Ab;
            VectorXd left_sol, right_sol;
            double   left_opt, right_opt;
            branch(index, to_round, Ab, left_Ab,  true);
            branch(index, to_round, Ab, right_Ab, false);

            core_solve(c, left_Ab, left_sol, left_opt);
            core_solve(c, right_Ab, right_sol, right_opt);
        }
    }
}
```

Listing A.8: Header File for Exception

```
#pragma once

#include <cinttypes>

#define FE_MESSAGE_BUFFER_SIZE 1024

/**
 * Exception class with printf like text formatting
 * capabilities, using variable argument list.
 */
class FException {
private:
    char error_msg[FE_MESSAGE_BUFFER_SIZE];
    int64_t error_code;

protected:

public:
    FException(const char* error_msg, ...);
    FException(int64_t error_code, const char* error_msg, ...);

    void Print() const;
    int64_t getErrorCode() const;
    const char *getMessage() const;
};
```

Listing A.9: Implementation File for Exception

```
#include <cinttypes>
#include <iostream>
#include <cstdarg>
#include "exception.h"

/**
 * Constructor
 * Use the same parameters as for the printf() function.
 *
 * @param error_msg
 * @param ... Variable argument list.
 * @returns FException
 */
FException::FException(const char *error_msg, ...) {
    va_list arg_list;

    va_start(arg_list, error_msg);
    vsnprintf(this->error_msg, (size_t)FE_MESSAGE_BUFFER_SIZE, error_msg, arg_list);
    va_end(arg_list);

    this->error_code = 0;
}

/**
 * Constructor
 * After the error_code use the same parameters as for the printf() function.
 *
 * @param error_code
 * @param error_msg
 * @param ... Variable argument list.
 * @returns FException
 */
FException::FException(int64_t error_code, const char *error_msg, ...) {
    va_list arg_list;

    va_start(arg_list, error_msg);
    vsnprintf(this->error_msg, (size_t)FE_MESSAGE_BUFFER_SIZE, error_msg, arg_list);
    va_end(arg_list);

    this->error_code = error_code;
}

/**
 * Prints out the error message text to the console.
 */
void FException::Print() const {
    std::cout << this->error_msg;
}

/**
 * Returns the error code if it's set. Otherwise returns 0.
 *
 * @returns int64_t
 */
int64_t FException::getErrorCode() const {
    return this->error_code;
}

/**
 * Returns the error message text of the exception.
 *
 * @returns char*
 */
const char* FException::getMessage() const {
    return this->error_msg;
}
```

Listing A.10: Header File for Simplex

```
#pragma once

#include <cinttypes>
#include <Eigen/Dense>

using namespace Eigen;

#define SIMPLEX_MINIMIZE 1
#define SIMPLEX_MAXIMIZE 2
```

```

class SimplexSolver {
private:
    MatrixXd tableau;
    bool foundSolution;
    double optimum;
    VectorXd solution;
    int64_t numberOfVariables;

    int64_t findPivot_min(int64_t column);
    bool simplexAlgorithm(int64_t variableNum);
    int64_t getPivotRow(int64_t column);

protected:

public:
    SimplexSolver(int mode, const VectorXd &objectiveFunction, const MatrixXd &constraints);
    bool hasSolution();
    double getOptimum();
    VectorXd getSolution();
};
```

Listing A.11: Implementation File for Simplex

```
#include <cinttypes>
#include <Eigen/Dense>
#include "SimplexSolver.h"
#include "exception.h"

using namespace Eigen;

/**
 * Constructor
 *
 * @param int mode This can be one of these: SIMPLEX_MINIMIZE, SIMPLEX_MAXIMIZE
 * @param const VectorXd &objectiveFunction The coefficients of the objective function.
 * @param const MatrixXd &constraints Full matrix for the constraints. Contains also the righthand-side values.
 * @returns SimplexSolver
 */
SimplexSolver::SimplexSolver(int mode, const VectorXd &objectiveFunction, const MatrixXd &constraints) {
    int64_t constantColumn, temp;
    this->foundSolution = false;
    this->optimum = 0;
    this->numberOfVariables = objectiveFunction.rows();

    /*
        Validate input parameters
    */
    if (mode != SIMPLEX_MINIMIZE && mode != SIMPLEX_MAXIMIZE) {
        throw FException("SimplexSolver:␣invalid␣value␣for␣the␣'mode'␣parameter.");
    }

    if (objectiveFunction.rows() < 1) {
        throw FException("SimplexSolver:␣The␣coefficient␣vector␣of␣the␣objective␣function␣must␣contain␣at␣least␣one␣row.");
    }

    if (constraints.rows() < 1) {
        throw FException("SimplexSolver:␣The␣constraint␣matrix␣must␣contain␣at␣least␣one␣row.");
    }

    if (constraints.cols() != objectiveFunction.rows() + 1) {
        throw FException("SimplexSolver:␣The␣constraint␣matrix␣has␣%d␣columns,␣but␣should␣have␣%d,␣because␣the␣coefficient␣vector␣of␣the␣objective␣function␣has␣%d␣rows.",
          constraints.cols(), objectiveFunction.rows() + 1, objectiveFunction.rows());
    }
    /*
    for (int i = 0; i < this->numberOfVariables; i++) {
        if (objectiveFunction(i) == 0) {
            throw FException("SimplexSolver: One of the coefficients of the objective function is zero.");
        }
    }
    */

    temp = constraints.cols() - 1;
    for (int i = 0; i < constraints.rows(); i++) {
        if (constraints(i, temp) < 0) {
            throw FException("SimplexSolver:␣All␣righthand-side␣coefficients␣of␣the␣constraint␣matrix␣must␣be␣non-negative.");
        }
    }

    /*
        Build tableau
    */
    if (mode == SIMPLEX_MAXIMIZE) {
        // Maximize
        this->tableau.resize(constraints.rows() + 1, this->numberOfVariables + constraints.rows() + 1);

        this->tableau <<    -objectiveFunction.transpose(),                    MatrixXd::Zero(1, constraints.rows() + 1),
                            constraints.leftCols(this->numberOfVariables),    MatrixXd::Identity(constraints.rows(), constraints.rows()), constraints.rightCols(1);
    } else {
        // Minimize: construct the Dual problem
        this->tableau.resize(this->numberOfVariables + 1, this->numberOfVariables + constraints.rows() + 1);

        this->tableau <<    -constraints.rightCols(1).transpose(),        MatrixXd::Zero(1, this->numberOfVariables + 1),
                            constraints.leftCols(this->numberOfVariables).transpose(),    MatrixXd::Identity(this->numberOfVariables, this->numberOfVariables), objectiveFunction;
    }

    /*
        Simplex algorithm
    */
    if (mode == SIMPLEX_MAXIMIZE) {
        // Maximize original problem
        if (!this->simplexAlgorithm(this->numberOfVariables)) {
            return;    // No solution
        }
    } else {
        // Maximize the dual of the minimization problem
        if (!this->simplexAlgorithm(constraints.rows())) {
            return;    // No solution
        }
    }

    /*
        Fetch solution
    */
    constantColumn = this->tableau.cols() - 1;
    this->solution.resize(this->numberOfVariables);

    if (mode == SIMPLEX_MAXIMIZE) {
        // Maximize
        for (int i = 0; i < this->numberOfVariables; i++) {
            temp = this->getPivotRow(i);

            if (temp > 0) {
                // Basic variable
                this->solution(i) = this->tableau(temp, constantColumn);
            } else {
                // Non-basic variable
                this->solution(i) = 0;
            }
        }
        this->foundSolution = true;
        this->optimum = this->tableau(0, constantColumn);
    } else {
        // Minimize
        for (int i = 0; i < this->numberOfVariables; i++) {
            this->solution(i) = this->tableau(0, constraints.rows() + i);
        }
        this->foundSolution = true;
        this->optimum = this->tableau(0, constantColumn);
    }
}

/**
 * Returns true if a solution has been found.
 * Return false otherwise.
 *
 * @returns boolean
 */
bool SimplexSolver::hasSolution() {
    return this->foundSolution;
}

/**
 * Returns the maximum/minimum value of
 * the objective function.
 *
 * @returns double
 */
double SimplexSolver::getOptimum() {
    return this->optimum;
}

/**
 * Returns a vector with the variable
 * values for the solution.
 *
 * return VectorXd
 */
VectorXd SimplexSolver::getSolution() {
    return this->solution;
}

/**
 * Searches for the pivot row in the given column, by calculating the ratios.
 * Tries to find smallest non-negative ratio.
 * Returns -1 if all possible pivots are 0 or if the ratios are negative.
 * Deals with cases like this:  0/negative < 0/positive
 *
 * @param int64_t column
 * @returns int64_t Returns the number of the pivot row, or -1 if found none.
 */
int64_t SimplexSolver::findPivot_min(int64_t column) {
    int64_t minIndex = -1;
    int64_t constantColumn = this->tableau.cols() - 1;
    double minRatio = 0;
    double minConstant = 0;    // For the "0/negative < 0/positive" difference the constants have to be tracked also.
    double ratio;
    int64_t rowNum = this->tableau.rows();

    for (int i = 1; i < rowNum; i++) {
        if (this->tableau(i, column) == 0) {
            continue;
        }

        ratio = this->tableau(i, constantColumn) / this->tableau(i, column);
        if (ratio < 0) {
            //The ratio must be non-negative
            continue;
        }

        if (minIndex == -1) {
            // First pivot candidate
            minIndex = i;
            minRatio = ratio;
            minConstant = this->tableau(i, constantColumn);
        } else {
            if (ratio == 0 && ratio == minRatio) {
                // 0/negative < 0/positive
                if (this->tableau(i, constantColumn) < minConstant) {
                    minIndex = i;
                    minRatio = ratio;
                    minConstant = this->tableau(i, constantColumn);
                }
            } else if (ratio < minRatio) {
                minIndex = i;
                minRatio = ratio;
                minConstant = this->tableau(i, constantColumn);
            }
        }
    }

    return minIndex;
}

/**
 * Iterates through the this->tableau matrix to solve the problem.
 *
 * @param int64_t variableNum The number of variables (dimensions). (different for the minimization problem)
 * @returns bool Returns true if a solution has been found. Returns false otherwise.
 */
bool SimplexSolver::simplexAlgorithm(int64_t variableNum) {
    MatrixXd::Index pivotColumn;
    int64_t pivotRow;

    while (true) {
        /*
            Find pivot column, check for halt condition
        */
        this->tableau.row(0).leftCols(variableNum).minCoeff(&pivotColumn);
        if (this->tableau(0, pivotColumn) >= 0) {
            //Found no negative coefficient
            break;
        }

        /*
            Find pivot row
        */
        pivotRow = this->findPivot_min(pivotColumn);
        if (pivotRow == -1) {
            //no solution
            return false;
        }

        /*
            Do pivot operation
        */
        this->tableau.row(pivotRow) /= this->tableau(pivotRow, pivotColumn);
        this->tableau(pivotRow, pivotColumn) = 1;    // For possible precision issues
        for (int i = 0; i < this->tableau.rows(); i++) {
            if (i == pivotRow) continue;

            this->tableau.row(i) -= this->tableau.row(pivotRow) * this->tableau(i, pivotColumn);
            this->tableau(i, pivotColumn) = 0;    // For possible precision issues
        }
    }

    return true;
}

```

```
/**
 * If the given column has only one coefficient with value 1 (except in topmost row),
     and all other
 * coefficients are zero, then returns the row of the non-zero value.
 * Otherwise return -1.
 * This method is used in the final step of maximization, when we read
 * the solution from the tableau.
 *
 * @param int64_t column
 * @returns int64_t
 */
int64_t SimplexSolver::getPivotRow(int64_t column) {
    int64_t one_row = -1;

    for (int i = 1; i < this->tableau.rows(); i++) {
        if (this->tableau(i, column) == 1) {
            if (one_row >= 0) {
                return -1;
            } else {
                one_row = i;
                continue;
            }
        } else if (this->tableau(i, column) != 0) {
            return -1;
        }
    }

    return one_row;
}
```

Appendix B

声明与分工

声明

我们在这里声明，这份工程《分枝定界法求解整数规划》是我们组独立完成的工作。

分工

陈翰逸	:	负责处理 I/O
林锦铿	:	负责文档、测试及评估
黄文璨	:	负责分枝定界法实现
赵竞霖	:	负责单纯形法实现

Bibliography

[1] Victor Klee and George J. Minty. Worst case for simplex method. http://mathscinet.ams.org/mathscinet-getitem?mr=0332165, 1972. Retrieved Jan 13rd, 2018.

[2] Wikipedia. Fourier–motzkin elimination. https://en.wikipedia.org/wiki/Fourier%E2%80%93Motzkin_elimination, 2018. Retrieved Jan 13rd, 2018.

[3] Wikipedia. Integer programming. https://en.wikipedia.org/wiki/Integer_programming, 2018. Retrieved Jan 13rd, 2018.