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

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。

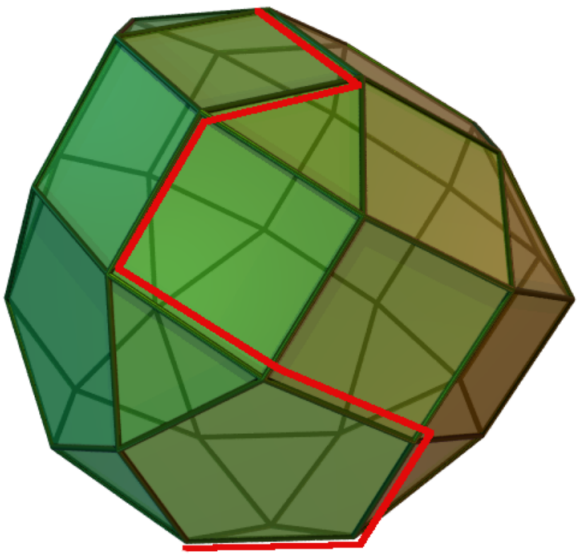

Figure 2.1: 单纯形

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：

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

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：$\sum_{n+1\le k\le n+m} x_{n+k}$，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：

$$\begin{aligned}
\text{起初} \quad &: \quad x_{n+d} = b_d - \sum_{k\in N} a_{d,k}x_k \\
\text{移项} \quad &: \quad a_{d,e}x_e = b_d - \sum_{k\in N k\neq e} a_{d,k}x_k - x_{n+d} \\
\text{若} a_{d,e}\neq 0 \quad &: \quad x_e = \frac{b_d}{a_{d,e}} - \sum_{k\in N k\neq e} \frac{a_{d,k}}{a_{d,e}}x_k - \frac{1}{a_{d,e}}x_{n+d}
\end{aligned}$$

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，$\min_{a_{d,e}>0} \frac{b_d}{a_{d,e}}$
- 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。

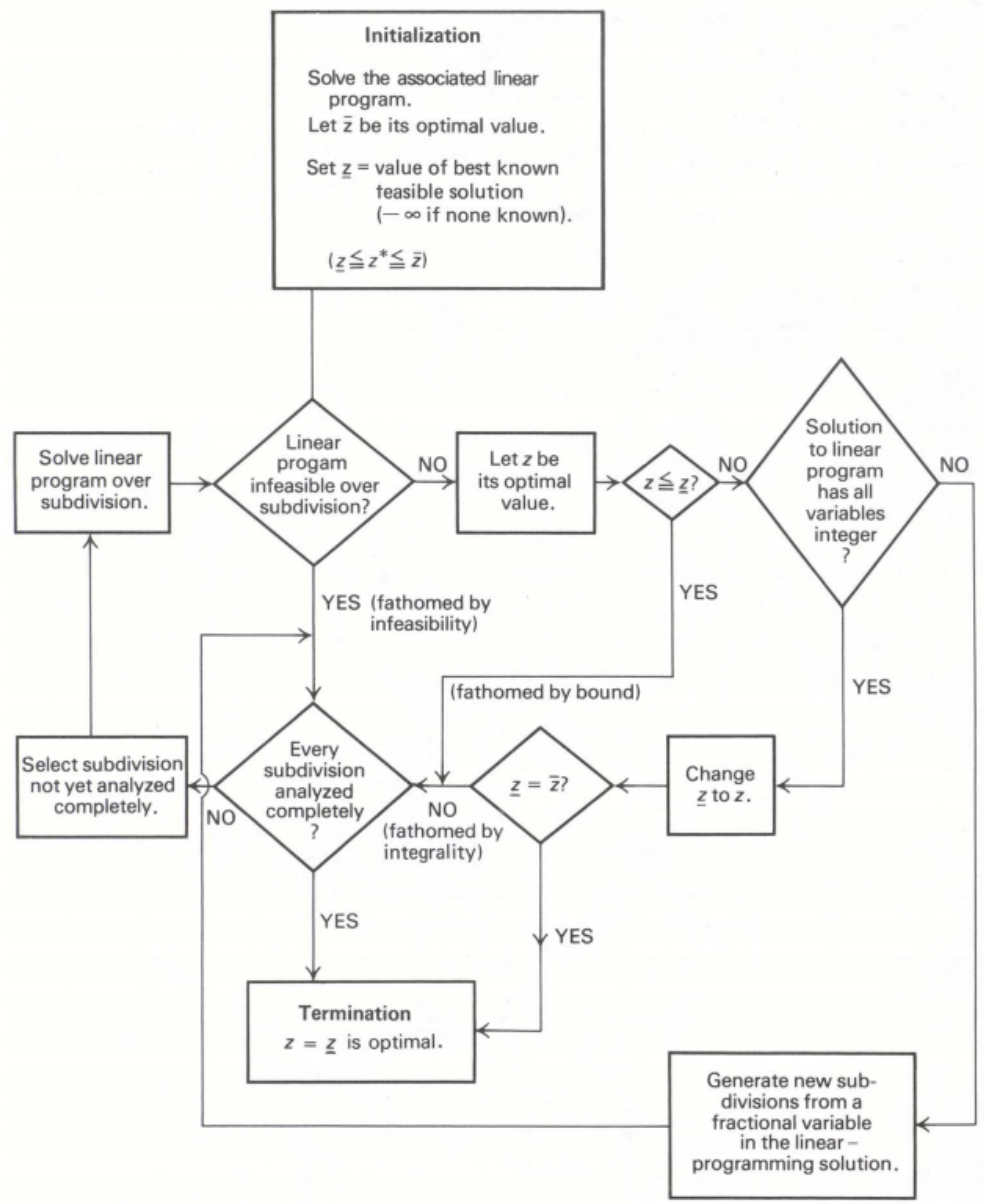

Figure 2.2: 分枝定界算法流程图

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□E数是负数，大于、小于要交换。

3.1.3 工具函数

```
1     std::vector<std::string> Data::Split(const std::string & input, char delim);
2     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 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

分析与评价

4.1 时间复杂度

4.1.1 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。

4.1.2 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 $O(2^n) \times O(2^n) \approx O(2^n)$

4.2 空间复杂度

Chapter 5

Conclusion

Appendices

Appendix A

源代码

Listing A.1: Header File for lpreader

```
#include "sparse.h"
#include <fstream>
#include <iostream>
#include <string>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;

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

    LPReader(const string& file){
        d.Parse(file);
        int64_t var_num = d.function.size();
        c = VectorXd::Zero(var_num);
        for(int64_t i=0;i<var_num;i++){
            c(d.function[i].index)=d.function[i].coefficient;
        }

    }

};
```

Listing A.2: 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;

```

```
20          }
21          virtual const char* what() const throw () {
22                  return msg_.c_str();
23          }
24  };
25
26  class Bounds {
27  public:
28          // lower <= x && x <= upper
29          int upper, lower;
30          Bounds()
31          {
32                  this->upper = std::numeric_limits<int>::max();
33                  this->lower = std::numeric_limits<int>::min();
34          }
35
36          bool operator < (const Bounds & b) const {
37                  return false;
38          }
39  };
40
41  class Variable {
42  public:
43          int coefficient;
44          size_t index;
45          Variable(int coefficient, size_t index) :
46                  coefficient(coefficient), index(index)
47          {
48          }
49
50          bool operator < (const Variable &b) const {
51                  return index < b.index;
52          }
53  };
54
55  class Condition {
56  public:
57          enum Type { eq, leq, geq };
58          Type type;
59          std::vector<Variable> variables;
60          int constant;
61          Condition(Type type, const std::vector<Variable> & variables, int constant):
62                  type(type), variables(variables), constant(constant)
63          {
64          }
65  };
66
67  class Data {
68  public:
69          std::vector<Condition> conditions;
70          std::vector< std::pair<size_t, Bounds> > bounds;
71          std::vector<size_t> indices;
72          std::vector<Variable> function;
73
74          //static std::string Trim(const std::string &s);
75          static std::vector<std::string> Split(const std::string & input, char delim);
76          static std::string Join(const std::vector<std::string> & input);
77          static std::vector<Variable> ParseVariables(const std::vector<std::string> &
                tokens, bool opposite = false);
78          static size_t ParseVariable(std::string variable);
79          static Condition ParseExpression(const std::vector<std::string> & tokens);
80  public:
81          void Parse(const std::string & input);
82          std::string Print();
83  };
84
85  //std::string Data::Trim(const std::string &s)
86  //{
87  //      auto wsfront = std::find_if_not(s.begin(), s.end(), [](int c) {return std::
        isspace(c); });
88  //      auto wsback = std::find_if_not(s.rbegin(), s.rend(), [](int c) {return std::
        isspace(c); }).base();
89  //      return (wsback <= wsfront ? std::string() : std::string(wsfront, wsback));
```

```
90  //}
91
92  std::vector<std::string> Data::Split(const std::string & input, char delim)
93  {
94          std::vector<std::string> result;
95          std::stringstream buffer(input);
96          for (std::string line; std::getline(buffer, line, delim); ) {
97                  if (!line.empty()) {
98                          result.push_back(line);
99                  }
100         }
101         return result;
102 }
103
104
105 std::string Data::Join(const std::vector<std::string> & input)
106 {
107         std::stringstream buffer;
108         for (const std::string &s : input) {
109                 buffer << s;
110         }
111         return buffer.str();
112 }
113
114
115 std::vector<Variable> Data::ParseVariables(const std::vector<std::string> & tokens,
        bool opposite)
116 {
117         using std::vector;
118         using std::string;
119         using std::regex;
120
121         vector<Variable> variables;
122
123         for (const string & token : tokens) {
124
125                 bool parseFail = false;
126                 int sign, coefficient, index;
127
128                 std::smatch sm;
129                 bool result = std::regex_search(token, sm, regex("([\\+\\-]?)([0-9]*)C
                        ([0-9]+)"));
130                 if (result && sm.size() == 4) {
131                         // sign
132                         if (sm[1] == "+" || sm[1] == "") {
133                                 sign = 1;
134                         } else if (sm[1] == "-") {
135                                 sign = -1;
136                         } else {
137                                 parseFail = true;
138                         }
139
140                         // check whether opposite
141                         if (opposite){
142                                 sign *= -1;
143                         }
144
145                         // coefficient
146                         try {
147                                 if (sm[2] == "") {
148                                         coefficient = 1;
149                                 } else {
150                                         coefficient = std::stoi(sm[2]);
151                                 }
152                         } catch (std::exception e) {
153                                 parseFail = true;
154                         }
155
156                         // index
157                         try {
158                                 index = std::stoi(sm[3]);
159                         } catch (std::exception e) {
160                                 parseFail = true;
```

```
161                         }
162                 } else {
163                         parseFail = true;
164                 }
165
166                 if (parseFail) {
167                         throw new ParseException("'" + token + "' is not a valid
                            variable");
168                 }
169
170                 variables.push_back(Variable(sign * coefficient, index));
171         }
172
173         return variables;
174 }
175
176
177 Condition Data::ParseExpression(const std::vector<std::string> & tokens_)
178 {
179         using std::vector;
180         using std::string;
181
182         string tokensString = Data::Join(tokens_);
183         tokensString = std::regex_replace(tokensString, std::regex("([\\+\\-])"), " $1
                ");
184         tokensString = std::regex_replace(tokensString, std::regex("
                (\\<\\=|\\>\\=|\\=)"), " $1 ");
185         vector<string> tokens = Data::Split(tokensString, ' ');
186
187         if (tokens.size() < 3) {
188                 throw new ParseException("'" + Data::Join(tokens) + "' is not a valid
                    expression");
189         }
190
191         Condition::Type conditionType;
192         vector<Variable> variables;
193         int constant;
194
195         const string & operatorString = tokens[tokens.size() - 2];
196         const string & constantString = tokens[tokens.size() - 1];
197
198         // variables
199         variables = Data::ParseVariables(vector<string>(tokens.begin(), tokens.begin()
                + tokens.size() - 2));
200
201         // =, <=, >=
202         if (operatorString == "=") {
203                 conditionType = Condition::Type::eq;
204         } else if (operatorString == "<=") {
205                 conditionType = Condition::Type::leq;
206         } else if (operatorString == ">=") {
207                 conditionType = Condition::Type::geq;
208         } else {
209                 throw new ParseException("'" + operatorString + "' is not a valid
                    operator");
210         }
211
212         // constant
213         try {
214                 constant = std::stoi(constantString);
215         } catch (std::exception e) {
216                 throw new ParseException("'" + constantString + "' is not a valid
                    integer");
217         }
218
219         std::sort(variables.begin(), variables.end());
220         return Condition(conditionType, variables, constant);
221 }
222
223 size_t Data::ParseVariable(std::string variable)
224 {
225         using std::string;
226         using std::regex;
```

```
227
228         std::smatch sm;
229         bool result = std::regex_search(variable, sm, regex("C([0-9]+)"));
230         bool parseFail = false;
231
232         int index;
233
234         if (result && sm.size() == 2) {
235                 try {
236                         index = std::stoi(sm[1]);
237                 }
238                 catch (std::exception e) {
239                         parseFail = true;
240                 }
241         }
242         else {
243                 parseFail = true;
244         }
245
246         if (parseFail) {
247                 throw new ParseException("'" + variable + "' is not a valid variable")
                        ;
248         }
249
250         return index;
251 }
252
253 void Data::Parse(const std::string & input_)
254 {
255         using std::vector;
256         using std::string;
257         using std::regex;
258         using std::stringstream;
259
260         string input = input_;
261         input = std::regex_replace(input, regex("/\\*(.|\n)*?\\*/"), ""); // remove
                comment
262         input = std::regex_replace(input, regex("^\\s*$"), ""); // remove blank
263         input = std::regex_replace(input, regex("[\n\r]"), " "); // remove line
264
265         stringstream buffer(input);
266         for (string line; std::getline(buffer, line, ';'); ) {
267                 vector<string> tokens = Data::Split(line, ' ');
268                 if (!tokens.empty()) {
269                         if (tokens[0].size() >= 3 && tokens[0].substr(0, 3) == "int")
                                {
270                                 tokens.erase(tokens.begin());
271                                 vector<string> vars = Data::Split(Data::Join(tokens),
                                        ',');
272                                 for (const string & var : vars) {
273                                         int index = ParseVariable(var);
274                                         this->indices.push_back(index);
275                                 }
276                         } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) ==
                                "max:") {
277                                 function = Data::ParseVariables(vector<string>(tokens.
                                        begin() + 1, tokens.end()));
278                         } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) ==
                                "min:") {
279                                 function = Data::ParseVariables(vector<string>(tokens.
                                        begin() + 1, tokens.end()), true);
280                         } else {
281                                 if (tokens[0].back() == ':') {
282                                         tokens.erase(tokens.begin());
283                                 }
284                                 Condition condition = ParseExpression(tokens);
285                                 if (condition.variables.size() == 1) {
286                                         Variable variable = condition.variables.front
                                                ();
287                                         Bounds bounds;
288
289                                         /** coefficient * variable [= // <= // >=]
                                                constant
```

```
290                                  * if operator is =
291                                  *              upper = lower = constant /
                                         coefficient
292                                  * if operator is >=
293                                  *              if sign is +
294                                  *                      upper = +infinity
295                                  *                      lower = ceil(constant
                                         / coefficient)
296                                  *              if sign is -
297                                  *                      upper = floor(constant
                                         / coefficient)
298                                  *                      lower = -infinity
299                                  * if operator is <=
300                                  *              if sign is +
301                                  *                      upper = floor(constant
                                         / coefficient)
302                                  *                      lower = -infinity
303                                  *              if sign is -
304                                  *                      lower = ceil(constant
                                         / coefficient)
305                                  *                      upper = +infinity
306                                  */
307                                 int ceil_ = ceil(condition.constant*1.0 /
                                        variable.coefficient);
308                                 int floor_ = floor(condition.constant*1.0 /
                                        variable.coefficient);
309
310                                 switch (condition.type){
311                                         case Condition::Type::eq:
312                                                 if (ceil_ == floor_) {
313                                                         bounds.upper = bounds.
                                                             lower = ceil_;
314                                                 } else {
315                                                         // if ceil_ is not
                                                             equal to floor_
316                                                         // the variable should
                                                              be unsolvable.
317                                                         // simply let upper <
                                                             lower
318                                                         bounds.upper = floor_;
319                                                         bounds.lower = ceil_;
320                                                 }
321                                                 break;
322
323                                         case Condition::Type::leq:
324                                                 if (variable.coefficient < 0)
                                                     {
325                                                         bounds.lower = ceil_;
326                                                 } else {
327                                                         bounds.upper = floor_;
328                                                 }
329                                                 break;
330
331                                         case Condition::Type::geq:
332                                                 if (variable.coefficient < 0)
                                                     {
333                                                         bounds.upper = floor_;
334                                                 } else {
335                                                         bounds.lower = ceil_;
336                                                 }
337                                                 break;
338
339                                         default:
340                                                 break;
341                                 }
342
343                                 this->bounds.push_back(std::pair<size_t,
                                        Bounds>(variable.index, bounds));
344                         } else {
345                                 this->conditions.push_back(condition);
346                         }
347                 }
348         }
```

```
349         }
350
351         std::sort(this->indices.begin(), this->indices.end());
352         std::sort(this->bounds.begin(), this->bounds.end());
353 }
354
355
356 std::string Data::Print()
357 {
358         using std::stringstream;
359         using std::vector;
360
361         stringstream output;
362
363         vector<vector<int> > eq;
364         vector<vector<int> > leq;
365
366         // give a new index
367         const size_t indexSize = this->indices.size();
368         std::map<size_t, size_t> mapIndex;
369         {
370                 int count = 0;
371                 for (size_t index : this->indices) {
372                         mapIndex[index] = count++;
373                 }
374         }
375
376         for (const Condition & condition : this->conditions) {
377                 vector<int> vec;
378                 vec.resize(indexSize, 0);
379
380                 for (const Variable & variable : condition.variables) {
381                         size_t index = mapIndex[variable.index];
382                         int coe = variable.coefficient;
383                         vec[index] = coe;
384                 }
385                 vec.push_back(condition.constant);
386
387                 switch (condition.type) {
388                         case Condition::Type::eq:
389                                 eq.push_back(vec);
390                                 break;
391
392                         case Condition::Type::leq:
393                                 leq.push_back(vec);
394                                 break;
395
396                         case Condition::Type::geq:
397                                 for (int &e : vec) {
398                                         e *= -1;
399                                 }
400                                 leq.push_back(vec);
401                                 break;
402
403                         default:
404                                 break;
405                 }
406         }
407
408         output << indexSize << std::endl;
409
410         for (const auto & var : function){
411                 output << var.coefficient << "␣" << var.index << "␣";
412         }
413         output << std::endl;
414
415         output << this->bounds.size() << std::endl;
416         for (const auto p : this->bounds) {
417                 size_t index = p.first;
418                 const Bounds & bounds = p.second;
419                 output << index << "␣" << bounds.lower << "␣" << bounds.upper << std::
                    endl;
420         }
```

```
421
422         output << eq.size() << std::endl;
423         for (const auto & vec : eq) {
424                 for (const int e : vec) {
425                         output << e << "␣";
426                 }
427                 output << std::endl;
428         }
429
430         output << leq.size() << std::endl;
431         for (const auto & vec : leq) {
432                 for (const int e : vec) {
433                         output << e << "␣";
434                 }
435                 output << std::endl;
436         }
437
438         return output.str();
439 }
```

Listing A.3: Implementation File for parsing

```
1  #include <fstream>
2  #include <sstream>
3  #include <vector>
4  #include <regex>
5  #include <cctype>
6  #include <string>
7  #include <algorithm>
8  #include <exception>
9  #include <map>
10 #include <cmath>
11
12
13 class ParseException : public std::exception {
14 protected:
15         std::string msg_;
16 public:
17         ParseException(std::string message) {
18                 this->msg_ = message;
19
20         }
21         virtual const char* what() const throw () {
22                 return msg_.c_str();
23         }
24 };
25
26 class Bounds {
27 public:
28         // lower <= x && x <= upper
29         int upper, lower;
30         Bounds()
31         {
32                 this->upper = std::numeric_limits<int>::max();
33                 this->lower = std::numeric_limits<int>::min();
34         }
35
36         bool operator < (const Bounds & b) const {
37                 return false;
38         }
39 };
40
41 class Variable {
42 public:
43         int coefficient;
44         size_t index;
45         Variable(int coefficient, size_t index) :
46                 coefficient(coefficient), index(index)
47         {
48         }
49
```

```
50          bool operator < (const Variable &b) const {
51                  return index < b.index;
52          }
53  };
54
55  class Condition {
56  public:
57          enum Type { eq, leq, geq };
58          Type type;
59          std::vector<Variable> variables;
60          int constant;
61          Condition(Type type, const std::vector<Variable> & variables, int constant):
62                  type(type), variables(variables), constant(constant)
63          {
64          }
65  };
66
67  class Data {
68  private:
69          std::vector<Condition> conditions;
70          std::vector< std::pair<size_t, Bounds> > bounds;
71          std::vector<size_t> indices;
72          std::vector<Variable> function;
73
74          //static std::string Trim(const std::string &s);
75          static std::vector<std::string> Split(const std::string & input, char delim);
76          static std::string Join(const std::vector<std::string> & input);
77          static std::vector<Variable> ParseVariables(const std::vector<std::string> &
                tokens, bool opposite = false);
78          static size_t ParseVariable(std::string variable);
79          static Condition ParseExpression(const std::vector<std::string> & tokens);
80  public:
81          void Parse(const std::string & input);
82          std::string Print();
83  };
84
85  //std::string Data::Trim(const std::string &s)
86  //{
87  //      auto wsfront = std::find_if_not(s.begin(), s.end(), [](int c) {return std::
        isspace(c); });
88  //      auto wsback = std::find_if_not(s.rbegin(), s.rend(), [](int c) {return std::
        isspace(c); }).base();
89  //      return (wsback <= wsfront ? std::string() : std::string(wsfront, wsback));
90  //}
91
92  std::vector<std::string> Data::Split(const std::string & input, char delim)
93  {
94          std::vector<std::string> result;
95          std::stringstream buffer(input);
96          for (std::string line; std::getline(buffer, line, delim); ) {
97                  if (!line.empty()) {
98                          result.push_back(line);
99                  }
100         }
101         return result;
102 }
103
104
105 std::string Data::Join(const std::vector<std::string> & input)
106 {
107         std::stringstream buffer;
108         for (const std::string &s : input) {
109                 buffer << s;
110         }
111         return buffer.str();
112 }
113
114
115 std::vector<Variable> Data::ParseVariables(const std::vector<std::string> & tokens,
        bool opposite)
116 {
117         using std::vector;
118         using std::string;
```

```
119         using std::regex;
120
121         vector<Variable> variables;
122
123         for (const string & token : tokens) {
124
125                 bool parseFail = false;
126                 int sign, coefficient, index;
127
128                 std::smatch sm;
129                 bool result = std::regex_search(token, sm, regex("([\\+\\-]?)([0-9]*)C
                        ([0-9]+)"));
130                 if (result && sm.size() == 4) {
131                         // sign
132                         if (sm[1] == "+" || sm[1] == "") {
133                                 sign = 1;
134                         } else if (sm[1] == "-") {
135                                 sign = -1;
136                         } else {
137                                 parseFail = true;
138                         }
139
140                         // check whether opposite
141                         if (opposite){
142                                 sign *= -1;
143                         }
144
145                         // coefficient
146                         try {
147                                 if (sm[2] == "") {
148                                         coefficient = 1;
149                                 } else {
150                                         coefficient = std::stoi(sm[2]);
151                                 }
152                         } catch (std::exception e) {
153                                 parseFail = true;
154                         }
155
156                         // index
157                         try {
158                                 index = std::stoi(sm[3]);
159                         } catch (std::exception e) {
160                                 parseFail = true;
161                         }
162                 } else {
163                         parseFail = true;
164                 }
165
166                 if (parseFail) {
167                         throw new ParseException("'" + token + "'␣is␣not␣a␣valid␣
                                variable");
168                 }
169
170                 variables.push_back(Variable(sign * coefficient, index));
171         }
172
173         return variables;
174 }
175
176
177 Condition Data::ParseExpression(const std::vector<std::string> & tokens_)
178 {
179         using std::vector;
180         using std::string;
181
182         string tokensString = Data::Join(tokens_);
183         tokensString = std::regex_replace(tokensString, std::regex("([\\+\\-])"), "␣$1
                ");
184         tokensString = std::regex_replace(tokensString, std::regex("
                (\\<\\=|\\>\\=|\\=)"), "␣$1␣");
185         vector<string> tokens = Data::Split(tokensString, '␣');
186
187         if (tokens.size() < 3) {
```

```
188                 throw new ParseException("'" + Data::Join(tokens) + "' is not a valid
                        expression");
189         }
190
191         Condition::Type conditionType;
192         vector<Variable> variables;
193         int constant;
194
195         const string & operatorString = tokens[tokens.size() - 2];
196         const string & constantString = tokens[tokens.size() - 1];
197
198         // variables
199         variables = Data::ParseVariables(vector<string>(tokens.begin(), tokens.begin()
                + tokens.size() - 2));
200
201         // =, <=, >=
202         if (operatorString == "=") {
203                 conditionType = Condition::Type::eq;
204         } else if (operatorString == "<=") {
205                 conditionType = Condition::Type::leq;
206         } else if (operatorString == ">=") {
207                 conditionType = Condition::Type::geq;
208         } else {
209                 throw new ParseException("'" + operatorString + "' is not a valid
                        operator");
210         }
211
212         // constant
213         try {
214                 constant = std::stoi(constantString);
215         } catch (std::exception e) {
216                 throw new ParseException("'" + constantString + "' is not a valid
                        integer");
217         }
218
219         std::sort(variables.begin(), variables.end());
220         return Condition(conditionType, variables, constant);
221 }
222
223 size_t Data::ParseVariable(std::string variable)
224 {
225         using std::string;
226         using std::regex;
227
228         std::smatch sm;
229         bool result = std::regex_search(variable, sm, regex("C([0-9]+)"));
230         bool parseFail = false;
231
232         int index;
233
234         if (result && sm.size() == 2) {
235                 try {
236                         index = std::stoi(sm[1]);
237                 }
238                 catch (std::exception e) {
239                         parseFail = true;
240                 }
241         }
242         else {
243                 parseFail = true;
244         }
245
246         if (parseFail) {
247                 throw new ParseException("'" + variable + "' is not a valid variable")
                        ;
248         }
249
250         return index;
251 }
252
253 void Data::Parse(const std::string & input_)
254 {
255         using std::vector;
```

```
256         using std::string;
257         using std::regex;
258         using std::stringstream;
259 
260         string input = input_;
261         input = std::regex_replace(input, regex("/\\*(.|\n)*?\\*/"), ""); // remove
                comment
262         input = std::regex_replace(input, regex("^\\s*$"), ""); // remove blank
263         input = std::regex_replace(input, regex("[\n\r]"), "␣"); // remove line
264 
265         stringstream buffer(input);
266         for (string line; std::getline(buffer, line, ';'); ) {
267                 vector<string> tokens = Data::Split(line, '␣');
268                 if (!tokens.empty()) {
269                         if (tokens[0].size() >= 3 && tokens[0].substr(0, 3) == "int")
                                {
270                                 tokens.erase(tokens.begin());
271                                 vector<string> vars = Data::Split(Data::Join(tokens),
                                        ',');
272                                 for (const string & var : vars) {
273                                         int index = ParseVariable(var);
274                                         this->indices.push_back(index);
275                                 }
276                         } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) ==
                                "max:") {
277                                 function = Data::ParseVariables(vector<string>(tokens.
                                        begin() + 1, tokens.end()));
278                         } else if (tokens[0].size() >= 4 && tokens[0].substr(0, 4) ==
                                "min:") {
279                                 function = Data::ParseVariables(vector<string>(tokens.
                                        begin() + 1, tokens.end()), true);
280                         } else {
281                                 if (tokens[0].back() == ':') {
282                                         tokens.erase(tokens.begin());
283                                 }
284                                 Condition condition = ParseExpression(tokens);
285                                 if (condition.variables.size() == 1) {
286                                         Variable variable = condition.variables.front
                                                ();
287                                         Bounds bounds;
288 
289                                         /** coefficient * variable [= // <= // >=]
                                                constant
290                                          * if operator is =
291                                          *              upper = lower = constant /
                                                 coefficient
292                                          * if operator is >=
293                                          *              if sign is +
294                                          *                      upper = +infinity
295                                          *                      lower = ceil(constant
                                                 / coefficient)
296                                          *              if sign is -
297                                          *                      upper = floor(constant
                                                 / coefficient)
298                                          *                      lower = -infinity
299                                          * if operator is <=
300                                          *              if sign is +
301                                          *                      upper = floor(constant
                                                 / coefficient)
302                                          *                      lower = -infinity
303                                          *              if sign is -
304                                          *                      lower = ceil(constant
                                                 / coefficient)
305                                          *                      upper = +infinity
306                                          */
307                                         int ceil_ = ceil(condition.constant*1.0 /
                                                variable.coefficient);
308                                         int floor_ = floor(condition.constant*1.0 /
                                                variable.coefficient);
309 
310                                         switch (condition.type){
311                                                 case Condition::Type::eq:
312                                                         if (ceil_ == floor_) {
```

```
313                                                             bounds.upper = bounds.
                                                                    lower = ceil_;
314                                                     } else {
315                                                             // if ceil_ is not
                                                                    equal to floor_
316                                                             // the variable should
                                                                    be unsolvable.
317                                                             // simply let upper <
                                                                    lower
318                                                             bounds.upper = floor_;
319                                                             bounds.lower = ceil_;
320                                                     }
321                                                     break;
322
323                                             case Condition::Type::leq:
324                                                     if (variable.coefficient < 0)
                                                            {
325                                                             bounds.lower = ceil_;
326                                                     } else {
327                                                             bounds.upper = floor_;
328                                                     }
329                                                     break;
330
331                                             case Condition::Type::geq:
332                                                     if (variable.coefficient < 0)
                                                            {
333                                                             bounds.upper = floor_;
334                                                     } else {
335                                                             bounds.lower = ceil_;
336                                                     }
337                                                     break;
338
339                                             default:
340                                                     break;
341                                     }
342
343                                     this->bounds.push_back(std::pair<size_t,
                                            Bounds>(variable.index, bounds));
344                             } else {
345                                     this->conditions.push_back(condition);
346                             }
347                     }
348             }
349     }
350
351     std::sort(this->indices.begin(), this->indices.end());
352     std::sort(this->bounds.begin(), this->bounds.end());
353 }
354
355
356 std::string Data::Print()
357 {
358     using std::stringstream;
359     using std::vector;
360
361     stringstream output;
362
363     vector<vector<int> > eq;
364     vector<vector<int> > leq;
365
366     // give a new index
367     const size_t indexSize = this->indices.size();
368     std::map<size_t, size_t> mapIndex;
369     {
370             int count = 0;
371             for (size_t index : this->indices) {
372                     mapIndex[index] = count++;
373             }
374     }
375
376     for (const Condition & condition : this->conditions) {
377             vector<int> vec;
378             vec.resize(indexSize, 0);
```

```
379
380                 for (const Variable & variable : condition.variables) {
381                         size_t index = mapIndex[variable.index];
382                         int coe = variable.coefficient;
383                         vec[index] = coe;
384                 }
385                 vec.push_back(condition.constant);
386
387                 switch (condition.type) {
388                         case Condition::Type::eq:
389                                 eq.push_back(vec);
390                                 break;
391
392                         case Condition::Type::leq:
393                                 leq.push_back(vec);
394                                 break;
395
396                         case Condition::Type::geq:
397                                 for (int &e : vec) {
398                                         e *= -1;
399                                 }
400                                 leq.push_back(vec);
401                                 break;
402
403                         default:
404                                 break;
405                 }
406         }
407
408         output << indexSize << std::endl;
409
410         for (const auto & var : function){
411                 output << var.coefficient << "␣" << mapIndex[var.index] << "␣";
412         }
413         output << std::endl;
414
415         output << this->bounds.size() << std::endl;
416         for (const auto p : this->bounds) {
417                 size_t index = p.first;
418                 const Bounds & bounds = p.second;
419                 output << mapIndex[index] << "␣" << bounds.lower << "␣" << bounds.
                    upper << std::endl;
420         }
421
422         output << eq.size() << std::endl;
423         for (const auto & vec : eq) {
424                 for (const int e : vec) {
425                         output << e << "␣";
426                 }
427                 output << std::endl;
428         }
429
430         output << leq.size() << std::endl;
431         for (const auto & vec : leq) {
432                 for (const int e : vec) {
433                         output << e << "␣";
434                 }
435                 output << std::endl;
436         }
437
438         return output.str();
439 }
440
441
442 int main(int argc, char *argv[]) {
443         Data *data = new Data;
444         std::fstream fin;
445         //fin.open("case3.lp", std::fstream::in);
446
447         std::string file_name;
448         if (argc == 2){
449                 file_name = argv[1];
450         }
```

```
451         else{
452                 return -1;
453         }
454 
455         std::ifstream t(file_name);
456         t.seekg(0, std::ios::end);
457         size_t size = t.tellg();
458         std::string buffer(size, '␣');
459         t.seekg(0);
460         t.read(&buffer[0], size);
461         fin.close();
462 
463         try {
464                 data->Parse(buffer);
465         }
466         catch (ParseException e) {
467                 printf("%s\n", e.what());
468         }
469 
470         std::fstream fout;
471         fout.open(file_name.substr(0, file_name.find_last_of(".") + 1) + "txt", std::
                fstream::out);
472         std::string result = data->Print();
473         fout << result;
474         fout.close();
475 
476         // check if there is a variable not been declared
477         return 0;
478 }
```

Listing A.4: Header File for Branch Bound

```
1  #include <Eigen/Dense>
2  
3  #define INF 1E100
4  
5  using namespace Eigen;
6  using namespace std;
7  
8  class BranchBound{
9  public:
10     VectorXd solution;
11     bool foundSolution;
12     double optimum;
13     int64_t numberOfVariables;
14 
15     BranchBound(int mode, const VectorXd &objectiveFunction,
16                           const MatrixXd &constraints, double tol=1E-8)
17     : mode(mode), c(objectiveFunction), Ab(constraints), tol(tol) {
18         numberOfVariables = objectiveFunction.rows();
19         current_opt = optimum = -INF;
20         solution.resize(numberOfVariables);
21         current_solution.resize(numberOfVariables);
22         foundSolution = false;
23     }
24 
25     bool solve();
26 
27 private:
28     int mode;
29     VectorXd c;
30     MatrixXd Ab;
31     double current_opt;
32     VectorXd current_solution;
33     double tol;
34 
35     bool allInteger(const VectorXd &solution){
36         for(int64_t i=0;i<numberOfVariables;i++){
37             if(abs(solution(i) - int(solution(i))) > tol)return false;
38         }
39         return true;
```

```
40      }
41
42      int64_t getFirstNotInt(const VectorXd &solution){
43          for(int64_t i=0;i<numberOfVariables;i++){
44              if(abs(solution(i) - int(solution(i))) > tol)return i;
45          }
46          return -1;
47      }
48
49      bool needBranch(const VectorXd &solution, double value){
50          bool ai = allInteger(solution);
51          if(ai)return false;
52          if(value <= current_opt)return false;
53          return true;
54      }
55
56      void update_opt(const VectorXd &solution, double value){
57          bool ai = allInteger(solution);
58          if(ai && current_opt < value){
59              current_opt = value;
60              current_solution = solution;
61          }
62          foundSolution = true;
63      }
64
65      void branch(int64_t index,
66                  double to_round,
67                  const MatrixXd& Ab,
68                  MatrixXd& new_Ab,
69                  bool is_left)
70      {
71          new_Ab.resizeLike(Ab);
72          new_Ab.conservativeResize(Ab.rows()+1, Ab.cols());
73
74          VectorXd to_append;
75          to_append.resize(Ab.cols());
76          for(int64_t i=0;i<Ab.cols();i++){to_append(i) = 0;}
77          to_append(index) = is_left ? 1 : -1;
78          to_append(Ab.cols()-1) = is_left ? int(to_round) : (int(to_round) + 1);
79
80          new_Ab.row(Ab.rows()) = to_append;
81      }
82
83      void core_solve(const VectorXd& c, const MatrixXd& Ab, VectorXd& sol, double& opt)
            ;
84  };
```

Listing A.5: Implementation File for Branch Bound

```
1   #include <iostream>
2   #include "branch_bound.h"
3   #include "../SimplexSolver/SimplexSolver.h"
4   using namespace Eigen;
5   using namespace std;
6
7   bool BranchBound::solve()
8   {
9       VectorXd tmp_c(c);
10      VectorXd sol;
11      double   opt;
12      if(mode==SIMPLEX_MINIMIZE){
13          tmp_c = c * -1.0;
14      }
15
16      core_solve(tmp_c, Ab, sol, opt);
17
18      if(foundSolution){
19          optimum  = (mode==SIMPLEX_MINIMIZE) ? -current_opt : current_opt;
20          solution = current_solution;
21          return true;
22      }
```

```
23      else return false;
24  }
25
26  void BranchBound::core_solve(const VectorXd& c,
27                               const MatrixXd& Ab,
28                               VectorXd& sol,
29                               double& opt)
30  {
31      SimplexSolver ssolver(SIMPLEX_MAXIMIZE, c, Ab);
32      if(ssolver.hasSolution()){
33          opt = ssolver.getOptimum();
34          cout<<"DEBUG : current opt = "<<opt<<endl;
35          sol = ssolver.getSolution();
36          update_opt(sol, opt);
37          if(needBranch(sol, opt)){
38              int64_t index = getFirstNotInt(sol);
39              double to_round = sol(index);
40              MatrixXd left_Ab,  right_Ab;
41              VectorXd left_sol, right_sol;
42              double   left_opt, right_opt;
43              branch(index, to_round, Ab, left_Ab,  true);
44              branch(index, to_round, Ab, right_Ab, false);
45
46              core_solve(c, left_Ab, left_sol, left_opt);
47              core_solve(c, right_Ab, right_sol, right_opt);
48          }
49      }
50  }
```

Appendix B

声明与分工

Declaration

We hereby declare that all the work done in this project titled "Linux Kernel Debugging" is of our independent effort as a group.

Duty Assignments:

Bibliography

[1] Victor Klee and George J. Minty. Integer programming. 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.